Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Contents

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Linux Standard Base Development Kit for application building/porting

Rajesh Banginwar
Intel Corporation
rajesh.banginwar@intel.com

Nilesh Jain
Intel Corporation
nilesh.jain@intel.com

Abstract

The Linux Standard Base (LSB) specifies the binary interface between an application and a runtime environment. This paper discusses the LSB Development Kit (LDK) consisting of a build environment and associated tools to assist software developers in building/porting their applications to the LSB interface. Developers will be able to use the build environment on their development machines, catching the LSB porting issues early in the development cycle and reducing overall LSB conformance testing time and cost. Associated tools include application and package checkers to test for LSB conformance of application binaries and RPM packages.

This paper starts with the discussion of advantages the build environment provides by showing how it simplifies application development/porting for LSB conformance. With the availability of this additional build environment from LSB working group, the application developers will find the task of porting applications to LSB much easier. We use the standard Linux/Unix `chroot` utility to create a controlled environment to keep check of the API usage by the application during the build to ensure LSB conformance. After discussing the build environment implementation details, the paper briefly talks about the associated tools for validating binaries and RPM packages for LSB conformance. We conclude with a couple of case studies that demonstrate usage of the build environment as well as the associated tools described in the paper.

1 Linux Standard Base Overview

The Linux* Standard Base (LSB)[1] specifies the binary interface between an application and a runtime environment. The LSB Specification consists of a generic portion, gLSB, and an architecture-specific portion, archLSB. As the names suggest, gLSB contains everything that is common across all architectures, and archLSBs contain the things that are specific to each processor architecture, such as the machine instruction set and C library symbol versions.

As much as possible, the LSB builds on existing standards, including the Single UNIX Specification (SUS), which has evolved from POSIX, the System V Interface Definition (SVID), Itanium C++ ABI, and the System V Application Binary Interface (ABI). LSB adds the formal listing of what interfaces are available in which library as well as the data structures and constants associated with them.

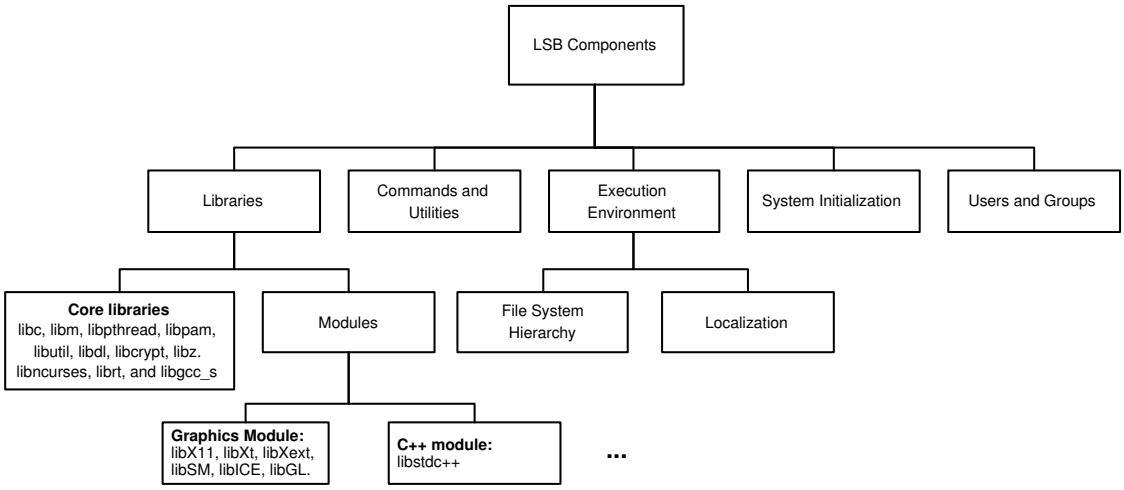


Figure 1: LSB Components

1.1 Components of LSB 3.0

Figure 1 shows the components of LSB 3.0 including the set of libraries covered in the specification. For applications to be LSB compliant, they are allowed to import only the specified symbols from these libraries. If application needs additional libraries, they either need to be statically linked or bundled as part of the application.

As the LSB expands its scope, future specification versions will include more libraries.

In addition to the Application Binary Interface (ABI) portion, the LSB specification also specifies a set of commands that may be used in scripts associated with the application. It also requires that applications follow the filesystem hierarchy standard (FHS)[7].

Another component of the LSB is the packaging format specification. The LSB specifies the package file format to be a subset of the RPM file format. While LSB does not specify that the operating system (OS) distribution has to be based on RPM, it needs to have a way to process a file in RPM format correctly.

All LSB compliant applications use a special program interpreter: `/lib/ld-lsb.so.3` for LSB version 3.0 instead of the traditional `/lib/ld-linux.so.2` for IA32 platforms. This program interpreter is executed first when an application is started, and is responsible for loading the rest of the program and shared libraries into the process address space. This provides the OS with a hook early in the process execution in case something special needs to be done for LSB to provide the correct runtime environment to the application. Generally, `/lib/ld-`*`arch`*`-lsb.so.3` or `/lib64/ld-`*`arch`*`-lsb.so.3` is used for other 32– or 64–bit architectures.

The next section discusses issues involved in porting/developing applications to LSB conformance along with the basic requirements for the same. The section ends with the overview of LSB development kit to help with the task. The subsequent sections discuss alternate standalone build environments and case studies showing real applications ported to LSB.

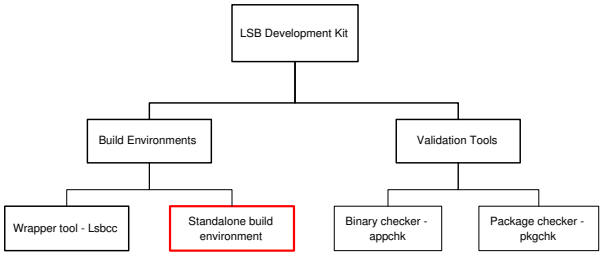


Figure 2: LSB Development Kit

2 Porting/Developing applications to LSB

This section starts the discussion with requirements for porting or developing applications to LSB. The application binaries will include executables and Dynamic Shared Object (DSO) files.

- Limit usage of DSOs to only LSB-specified libraries. Applications are also limited to import only LSB-specified symbols from those libraries.
- Use LSB-specific program interpreter `/lib/ld-lsb.so.3` for IA32 and `/lib/ld-`*`arch`*`-lsb.so.3` for other LSB-supported architectures.
- Use ELF as specified by LSB for created binaries.
- Use LSB-specified subset of RPM for application package.

For many application developers it may be a non-trivial task to port or develop applications to LSB. The LSB WG provides a development kit shown in Figure 2 to assist application developers in this task.

The LDK mainly consists of build environments to assist application developers with porting/development of applications to LSB and validation tools to verify for LSB conformance of application binaries and packages. LSB WG today has `lsbcc/lsbc++`—a `gcc/g++` wrapper tool which serves as a build environment as discussed in a subsection below. The second build environment which we are calling a standalone build environment is the topic of discussion for this paper. Before we discuss that build environment in detail, let's talk about the validation tools and existing build tools briefly.

2.1 Validation Tools in LDK

There are two validation tools delivered as part of LDK. These tools are to be used as part of LSB compliance testing for application binaries and packages.

1. `appchk`: This tool is used to validate ELF binaries (executables and DSOs) for their LSB conformance. This tool will work hand-in-hand with the build environment as discussed in the later sections of this paper. The LDK Case Studies section details the usage of this tool.
2. `pkgchk`: This tool new for LSB 3.0 is used for validating application packages. The tool makes sure that the package uses the LSB specified RPM file format. It also validates the installation aspect of the package for FHS conformance.

2.2 `lsbcc/lsbc++` – Existing build tool

In the last few years, the LSB WG has been providing a compiler wrapper, called `lsbcc` and `lsbc++`, as a build tool for application porting. `lsbcc` or `lsbc++` is used wherever build scripts use `gcc` or `g++` respectively. The wrapper tool parses all of the command line options

passed to it and rearranges them, inserting a few extra options to cause the LSB-supplied headers and libraries to be used ahead of the normal system libraries[6]. This tool also recognizes non-LSB libraries and forces them to be linked statically. Because the LSB-supplied headers and libraries are inserted into the head of the search paths, it is generally safe to use things not in the LSB.

With these simple steps many of the applications can be ported to LSB by simply replacing `gcc` with `lsbcc` and `g++` with `lsbc++`. In this method, the host environment is used for the build process; hence sometimes it may be difficult to reproduce the results on multiple platforms due to environment differences. This issue is not specific to the `lsbcc` wrapper build environment, but a common problem for many build systems. The build environment discussed in this paper addresses this issue by creating a standalone environment. Another shortcoming of the `lsbcc` approach is that the wrapper tools rely on the usage of `gcc` as compiler and configure-make process for application building. If the application relies on tools like `libtool` which modify the compiler command lines, `lsbcc` may not work correctly without additional configuration changes to produce LSB-compliant results. Similarly, usage of other compilers may not be possible as the wrapper tool relies on the command line option format used by `gcc`. For similar reasons, the tool may require additional configuration in certain customized build processes which may not rely on traditional config-make like build scripts.

3 LDK Standalone build environment

The standalone build environment is created using the standard Linux utility `chroot`. The isolated directory hierarchy is built from source packages and is completely independent of its host environment. With the development of this tool application developers will have a choice between the wrapper tool discussed above and the standalone build environment discussed here. From now on we refer to this standalone build environment as simply the build environment unless otherwise explicitly noted.

The concept of this build environment is derived from the Automated Linux from Scratch (ALFS)[2] project to create an isolated environment. The build environment comes with basic build tools and packages required for common application building. These tools are preconfigured so that the applications built produce LSB-conformant results. The application developer may add more tools/packages to this build environment as discussed later.

Since the application build happens in an isolated environment, except for some minor changes to Makefiles, the application developers do not need to change the build process. Since the whole mechanism is independent of the compiler as well as build scripts used, this build environment will work for most application development situations.

The build environment provides a set of clean headers and stub libraries for all the symbols included in the LSB specification. Applications are restricted to use only these symbols to achieve LSB conformance.

The build environment when used as documented will help produce the LSB-conformant application binaries. We recommend using the build environment from the beginning of application development cycle which will help catch any LSB porting issues early, reducing overall cost of LSB conformance testing.

The remainder of this section discusses the

build environment implementation in details. In addition to providing information on how it is used and accessed, the section also describes how the build tools are configured and/or updated.

3.1 Build environment Structure

Like a typical Linux distribution, the build environment has a directory hierarchy with `/bin`, `/lib`, `/usr`, and other related directories. Some of the differences between this build environment and a Linux distribution are the lack of Linux Kernel, most daemons, and an X server, etc. To start this build environment the developer will need root privileges on the host machine. The `lsb-buildenv` command used for starting the build environment behaves as follows:

```
Usage: lsb-buildenv -m [lsb|
nonlsb] -p [port] start|stop|
status
```

By default when used with no options, the environment will be configured for LSB-compliant building. The option `non-lsb` will force it to remain in normal build mode. This option typically is used for updating the build environment itself with additional packages/tools. The default sshd-port is set at 8989.

The `lsb-buildenv` command starts the `sshd` daemon at the specified port number. To access and use the build environment the user will need to `ssh` into the started build environment. By default, only the `root` account is created; the password is set to `lsbbuild123`. Once the user is logged into the build environment as `root`, he/she can add/update the user accounts needed for regular build processes.

```
$ ssh -p 8989 root@localhost
```

The build environment comes with the LSB WG-provided headers and stub libraries for all the LSB 3.0-specified libraries. These headers and stub libraries are located in the `/opt/lsb/include` and `/opt/lsb/lib` directories respectively. It is strongly recommended against modifying these directories.

X11 and OpenGL headers are exceptions to this and are located in `/usr/X11R6/include` although they are soft-linked in `/opt/lsb/include/X11`. These headers are taken from the Release 6 packages from X.org. The stub libraries related to all X libraries specified in LSB are located in `/opt/lsb/lib`.

3.1.1 Tools and Configuration updates

As discussed earlier the build environment is equipped with all the standard C/C++ build tools like `gcc` compiler suite, `binutils` package, etc. The goal for this build environment is to minimize the changes the application developer needs to make in the build scripts for the build to produce LSB-compliant results. The build tools are modified/configured to help produce LSB-conformant results as discussed below:

- **Compile time changes:** As discussed above, LSB provides a clean set of header files in the `/opt/lsb/include` directory. The `gcc` specs file is updated so that the compiler looks for this directory before continuing looking for other system locations. The string `-I /opt/lsb/include` is appended to the `*cpp_options` and `*cc1_options` sections in the `gcc` specs file.
- **Link time changes:**
  - By default the link editor (`ld` on most systems) is configured to look

in `/lib`, `/usr/lib`, and some other directories for DSO files. For the build to produce LSB-compliant results, we need to make sure the linking happens only with the LSB-provided stub libraries. For this, the default search path link editor uses to search for DSOs is changed to `/opt/lsb/lib` by configuring the `ld` build process at the time of creating/building this build environment. The `ld` is built with the following command: `./configure -with-lib-path=/opt/lsb/lib`

- Add `-L /opt/lsb/lib` to `*link` section of the `gcc` specs file to restrict the first directory accessed for libraries
- Remove `%D` from `*link_libgcc` section of `gcc` specs file. This will disallow `gcc` to add `-L` option for startup files.
- Set dynamic linker to `ld-lsb.so.3` by updating the `gcc` specs file by appending `*link` section with `%{!dynamic-linker: -dynamic-linker /lib/ld-lsb.so.3}`.

3.2 Packaging structure

The build environment comes with the most commonly needed packages pre-installed. Commonly used development (devel) packages are also pre-installed. As it is not possible to guess exactly what each application developer will need (since each build process is unique in requirements), the build environment comes with a populated RPM database to help the user add new packages as needed. This RPM database is built from scratch during the building of all the packages installed in the build environment. As no binary RPM is used for creating the build environment, Linux distribution-specific dependencies are avoided.

We use the CheckInstall [3] tool for populating RPM database in the build environment. This tool works by monitoring the steps taken by `make install` process and creates an RPM package which can then be installed. Please refer to the relevant reference listed in the Reference section for further documentation regarding this tool.

This RPM database may be used by the application developer if he/she needs to add/update a package required for a given build process. If for some reason (like dependency issues) a binary RPM cannot be installed, we suggest building and installing the package from source code by starting the build environment in non-lsb mode. Although not recommended, the user can always copy the relevant files manually into the build environment from the host machine.

4 Typical LSB porting process

This section discusses the process involved in porting the application to LSB. The subsection below discusses how LDK can be used during active development of application. Figure 3 shows the porting process in the form of a flow diagram.

- The first step is to run the existing application binaries through `appchk`. This will identify all the DSOs and symbols used by the application binaries that are not specified by LSB.
- The next step is to remove any unnecessary library dependencies where possible. Review all the makefiles (or similar scripts) to make sure the application is not

linking with any libraries that it does not need.

- If `appchk` is reporting that the application binary is dependent on a DSO not specified in LSB, there are two options to fix that:
  - The first option is to use static version of the library. This way the application will not depend on the concerned DSO.
  - If for some reason (licensing issues, etc.) that is not possible, the required functions will need to be implemented by the application developer avoiding the usage of that library or creating an application-specific DSO with those functions. When an application-specific DSO is created, it needs to be certified along with the application binary.
- For changing the usage of DSO to static library the Makefiles need to be updated manually. Remove `-l` options used during the linking phase for the concerned library. Include the corresponding static library in the linker command line.
- The next step is to perform `configure` and `make` (or similar scripts) as required by the application. Since the build environment is configured to use LSB-provided headers by default, the user may see some compilation errors. Typically these errors result due to usage of internal (although exported) or deprecated symbols. The developer will need to fix these by using the appropriate symbols for the given situation. The case study below shows one such situation. Another type of error occurs when a used symbol is not part of LSB although the concerned library is partially specified in LSB. The application developer needs to find alternatives to

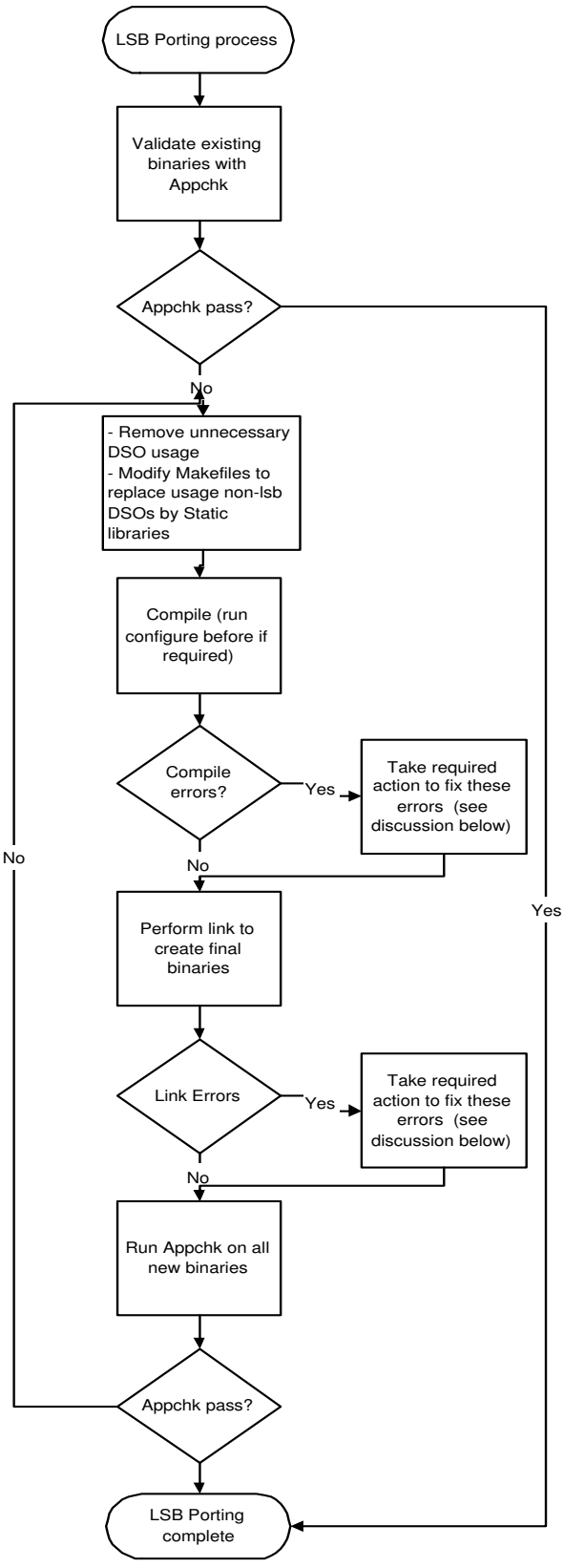


Figure 3: LSB porting process

such symbols that are covered by LSB or implement them as part of the application.

- The next step is linking to create final binaries for the application. If the Makefiles are correctly modified as discussed above, there should be minimal errors at this stage. The common error about "Symbol not defined" needs to be handled if certain deprecated or unspecified LSB symbols are used by the application and not caught in the compilation phase. Again the case studies below show couple of such examples.

4.1 LDK usage during application development

Other than porting the existing Linux applications to LSB, the build environment and the tools in LDK can be used by developers during the application development cycle. Regular or periodic usage of the build environment during the development cycle will help catch the LSB porting issues early in the development cycle, reducing overall LSB conformance testing time and cost. Such usage is highly recommended.

5 LDK Case Studies

This section discusses the real-life example of how LSB porting will work using this build environment. We consider two examples here to show different aspects of application porting. Since these examples are from the Open Source Software (OSS) projects they follow the optional `configure`, `make`, and `make install` model of building and installing software.

5.1 Example 1: ghostview 1.5

Ghostview[4] uses `xmkmf` to create the Makefile. When the application is built on a regular Linux machine, the `ldd` output for the ghostview binary is as follows:

```
$ ldd ghostview
libXaw.so.7 => /usr/X11R6/lib/libXaw.so.7
(0x00751000)
libXmu.so.6 => /usr/X11R6/lib/libXmu.so.6
(0x00b68000)
libXt.so.6 => /usr/X11R6/lib/libXt.so.6
(0x00af6000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6
(0x00ade000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6
(0x0024f000)
libXpm.so.4 => /usr/X11R6/lib/libXpm.so.4
(0x03c80000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6
(0x00522000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6
(0x00459000)
libm.so.6 => /lib/tls/libm.so.6 (0x0042e000)
libc.so.6 => /lib/tls/libc.so.6 (0x00303000)
libdl.so.2 => /lib/libdl.so.2 (0x00453000)
/lib/ld-linux.so.2 (0x002ea000)
```

Several of these libraries are not part of LSB yet and hence the application will not be LSB-compliant. To confirm that, run the `appchk` tool from LDK to find out exactly what is being used that is outside LSB's current specification:

```
$appchk -A ghostview
Incorrect program interpreter: /lib/ld-linux.so.2
Header[1] PT_INTERP    Failed
Found wrong intepreter in .interp section: /lib/ld-linux.so.2
 instead of: /lib/ld-lsb.so.3
DT_NEEDED: libXaw.so.7 is used, but not part of the LSB
DT_NEEDED: libXmu.so.6 is used, but not part of the LSB
DT_NEEDED: libXpm.so.4 is used, but not part of the LSB
section .got.plt is not in the LSB
appchk for LSB Specification
Checking symbols in all modules
Checking binary ghostview
Symbol XawTextSetInsertionPoint used, but not part of LSB
Symbol XawTextReplace used, but not part of LSB
Symbol XmuInternAtom used, but not part of LSB
Symbol XawTextUnsetSelection used, but not part of LSB
Symbol XawScrollbarSetThumb used, but not part of LSB
Symbol XmuCopyISOLatin1Lowered used, but not part of LSB
Symbol XawTextDisableRedisplay used, but not part of LSB
Symbol XawFormDoLayout used, but not part of LSB
Symbol XawTextEnableRedisplay used, but not part of LSB
Symbol XmuMakeAtom used, but not part of LSB
Symbol XawTextGetSelectionPos used, but not part of LSB
Symbol XawTextInvalidate used, but not part of LSB
Symbol XawTextGetInsertionPoint used, but not part of LSB
```

The first message indicates the usage of `ld-linux.so` instead of `ld-lsb.so.3` as

dynamic linker. `DT_NEEDED` messages indicate the libraries which are not part of LSB specification but used by the application. The rest of the messages indicate symbols imported by the application but not specified in LSB.

Let's now look at how the build environment will help with porting this application to LSB and the steps users will need to go through in this process.

Step 1: Modify `Makefile` so that it does not use DSOs for the non-LSB libraries. Replace them with the static version of the libraries.

Step 2: Fix the compilation errors. In this case the errors included usage of symbols `sys_nerr` and `sys_errlist`. These are deprecated symbols and hence not part of LSB headers. The usage of these symbols is replaced by function `strerror`.

Step 3: Fix the link-time errors. In this case since the application uses three X libraries outside of LSB scope, we need to replace them with the corresponding static libraries.

After compilation and linking, we use `appchk` to check for LSB conformance for the created binary `ghostview`:

```
$ appchk -A ghostview
appchk for LSB Specification
Checking symbols in all modules
Checking binary ghostview
```

If we run ldd on this binary we will see:

```
$ ldd ghostview
libXt.so.6 => /usr/X11R6/lib/libXt.so.6
(0x00af6000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6
(0x00ade000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6
(0x0024f000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6
(0x00522000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6
(0x00459000)
libm.so.6 => /lib/tls/libm.so.6 (0x0042e000)
libc.so.6 => /lib/tls/libc.so.6 (0x00303000)
libdl.so.2 => /lib/libdl.so.2 (0x00453000)
/lib/ld-lsb.so.3 (0x002ea000)
```

All these libraries are part of LSB and the `appchk` confirms that the symbols imported by the binary ghostview are specified in LSB. This shows the successful porting of the application to LSB.

5.2 Example 2: lesstif package

Lesstif[5] is an implementation of OSF/Motif producing following binaries:

```
bin/mwm
bin/uil
bin/xmbind
lib/libDt.so*
lib/libDtPrint.so*
lib/libMrm.so*
lib/libUil.so*
lib/libXm.so*
```

By default none of these binaries is LSB-compatible. On a regular Linux machine, we get the following output when we run `ldd` and `appchk` on `mwm`.

```
$ ldd clients/Motif-2.1/mwm/.libs/mwm
libXm.so.2 => not found
libXp.so.6 => /usr/X11R6/lib/libXp.so.6
(0x0042e000)
libXt.so.6 => /usr/X11R6/lib/libXt.so.6
(0x00af6000)
libSM.so.6 => /usr/X11R6/lib/libSM.so.6
(0x00ade000)
libICE.so.6 => /usr/X11R6/lib/libICE.so.6
(0x0024f000)
libXext.so.6 => /usr/X11R6/lib/libXext.so.6
(0x00522000)
libX11.so.6 => /usr/X11R6/lib/libX11.so.6
(0x00459000)
libXft.so.2 => /usr/X11R6/lib/libXft.so.2
(0x00705000)
libXrender.so.1 =>
/usr/X11R6/lib/libXrender.so.1 (0x00747000)
libc.so.6 => /lib/tls/libc.so.6 (0x00303000)
libdl.so.2 => /lib/libdl.so.2 (0x00453000)
libfontconfig.so.1 =>
/usr/lib/libfontconfig.so.1 (0x006ad000)
libexpat.so.0 => /usr/lib/libexpat.so.0
(0x006e4000)
libfreetype.so.6 => /usr/lib/libfreetype.so.6
(0x0598c000)
/lib/ld-linux.so.2 (0x002ea000)
libz.so.1 => /usr/lib/libz.so.1 (0x00532000)
```

```
$ appchk -A clients/Motif-2.1/mwm/.libs/mwm
Incorrect program interpreter: /lib/ld-linux.so.2
Header[ 1] PT_INTERP    Failed
Found wrong intepreter in .interp section: /lib/ld-linux.so.2
 instead of: /lib/ld-lsb.so.3
DT_NEEDED: libXm.so.2 is used, but not part of the LSB
DT_NEEDED: libXp.so.6 is used, but not part of the LSB
DT_NEEDED: libXft.so.2 is used, but not part of the LSB
DT_NEEDED: libXrender.so.1 is used, but not part of the LSB
section .got.plt is not in the LSB
appchk for LSB Specification
Checking symbols in all modules
Checking binary clients/Motif-2.1/mwm/.libs/mwm
Symbol XmGetXmDisplay used, but not part of LSB
Symbol XmGetPixmapByDepth used, but not part of LSB
Symbol _XmMicroSleep used, but not part of LSB
Symbol XpmReadFileToImage used, but not part of LSB
Symbol _XmFontListCreateDefault used, but not part of LSB
Symbol XmeWarning used, but not part of LSB
Symbol XmRegisterConverters used, but not part of LSB
Symbol XmStringCreateSimple used, but not part of LSB
Symbol _XmAddBackgroundToColorCache used, but not part of LSB
Symbol _XmGetColors used, but not part of LSB
Symbol _XmSleep used, but not part of LSB
Symbol _XmBackgroundColorDefault used, but not part of LSB
Symbol _XmFontListGetDefaultFont used, but not part of LSB
Symbol XmStringFree used, but not part of LSB
Symbol XmCreateQuestionDialog used, but not part of LSB
Symbol XmMessageBoxGetChild used, but not part of LSB
Symbol _XmAccessColorData used, but not part of LSB
```

As explained in the previous case study, these messages indicate the usage of libraries and symbols not specified in LSB.

This package follows the typical OSS build process of `configure`, `make`, and `make install`. All the makefiles are generated at the end of configure step. What makes this package an interesting exercise is the usage of `libtool`. This tool is used for portability in the usage and creation of DSO and static libraries.

Let's now walk through the process of building this package for LSB conformance.

Step 1: Modify `Makefile` so that it does not use DSOs for the non-LSB libraries. Replace them with the static version of the libraries.

Step 2: There are no compilation errors observed for this package.

Step 3: The first linktime error we see is about the undefined reference to some of the `_Xt` functions. These functions exported from `libXt.so` are not part of the LSB specification even though most of the other functions coming from the same library are covered. In this case the reason for this exclusion happens to be the nature of these functions. Most of these are internal functions and not really meant to be used by applications. The workaround for this will be to use a static version of the library instead of DSO. All the makefiles using `libXt.so` are modified for this.

The next error we see is the usage of function `_XInitImageFuncPtrs`. This function is deprecated and private (although exported). The suggested function in this case is `XImageInit`. Make the required change in file `ImageCache.c`.

After the compilation and linking we use `appchk` to check for LSB conformance for the created binaries. The output is shown below:

```
$ appchk -A -L lib/Xm-2.1/.libs/libXm.so.2 -L
lib/Mrm-2.1 /.libs/\ libMrm.so.2 ?L lib/Uil-2
.1/.libs/libUil.so.2 clients/Motif-2.1/mwm/\
 .libs/mwm
appchk for LSB Specification
Checking symbols in all modules
Adding symbols for library lib/Xm-2.1/.libs/
libXm.so.2
Adding symbols for library lib/Mrm-2.1/.libs/
libMrm.so.2
Adding symbols for library lib/Uil-2.1/.libs/
libUil.so.2
Checking binary clients/Motif-2.1/mwm/.libs/mwm
```

This shows the successful porting of lesstif to LSB.

6 Future Directions for LDK

For the LSB Development Kit, we will continue to make the tools better and easier to use for application developers. As the LDK is maintained actively through the LSB Working Group, ongoing feedback will be included in the future development and active participation in the tools development is strongly encouraged.

One of the features we are actively considering is the integration of the LDK with Eclipse or similar IDE. Another area under consideration is a tool to help develop/create LSB conformance packages.

We would like to take this opportunity to encourage all application developers to use the tools discussed in this paper and provide feedback and feature requests to the LSB mailing lists. We strongly encourage ISV participation in this process and solicit their feedback on the available tools as well as LSB in general.

7 Acknowledgments

We sincerely thank Free Standards Group and its members for providing support to LSB project. We would also like to extend our thanks to a core group of LSB developers including Stuart Anderson, Marvin Heffler, Gordon McFadden, and especially Mats Wichmann for their patience and support during the development of the LDK project.

References

[1] Linux Standard Base at `http://www.linuxbase.org/`

[2] Automated Linux From Scratch project at `http://www.linuxfromscratch.org/alfs/`

[3] CheckInstall utility at `http://asic-linux.com.mx/~izto/checkinstall`

[4] ghostview at `http://www.gnu.org/software/ghostview/ghostview.html`

[5] lesstif at `http://www.lesstif.org/`

[6] `lsbcc` usage and details at `http://www.linuxjournal.com/article/7067`

[7] File hierarchy standard at `http://www.pathname.com/fhs/`

8 Legal

Building Murphy-compatible embedded Linux systems

Gilad Ben-Yossef
Codefidence Ltd.
gilad@codefidence.com

> "If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."
>
> — *Murphy's Law of Technology #5* [Murphy]

Abstract

It's 2:00 a.m. An embedded Linux system in the ladies' room of an Albuquerque gas station is being updated remotely. Just as the last bytes hit the flash, disaster strikes—the power fails. Now what? The regular way of updating the configuration or performing software upgrade of Linux systems is a *nonsequitur* in the embedded space. Still, many developers use these methods, or worse, for lack of a better alternative. This talk introduces a better alternative—a framework for safe remote configuration and software upgrade of a Linux system that supports atomic transactions, parallel, interactive and programmed updates, and multiple software versions with rollback and all using using such "novel" concepts as POSIX `rename(2)`, Linux `pivot_root(2)`, and the initrd/initramfs mechanism.

1 Introduction: When bad things happen to good machines

Building embedded systems, Linux-based or otherwise, involves a lot of effort. Thought must be given to designing important aspects of the system as its performance, real time constraints, hardware interfaces, and cost.

All too often, the issue of system survivability in face of Murphy's Law is not addressed as part of the overall design. Alternatively, it may be delegated to the implementor of specific parts of the overall system as "implementation details."

To understand what we mean by "system survivability in face of Murphy's law," let us consider the common warning often encountered when one updates the firmware of an embedded system:

> "*Whatever happens, DO NOT pull the plug or reboot this system until the firmware update has been completed or you risk turning this system into a brick.*"

If there is something we can guarantee with certainty, while reading such a sincere warning, it is that somewhere and some when the power will indeed falter or the machine reboot just as

those last precious bits are written to flash, rendering the system completely unusable.

It is important to note that this eventuality, although sure to happen, is not common. Indeed, the system can undergo thousands of firmware upgrades in the QA lab without an incident; there seems to be some magical quality to the confinements of QA labs that stops this sort of thing from happening.

Indeed, any upgrade of a a non-critical piece of equipment in an idle Tuesday afternoon is considered quite safe in the eyes of the authors, with relation to the phenomena that we are discussing.

However, any critical system upgrade, performed on a late Friday afternoon is almost guaranteed to trigger a complex chain of events involving power failures, stray cats, or the odd meteorite or two, all leading to the some sad (yet expected) outcome—a $3k or $50 irreplaceable brick.

In essence therefore, system survivability in face of Murphy's Law is defined as the chances of a given system to function in face of failure in the "worst possible time."

Despite the humorous tone chosen above, this characteristic of embedded system has a very serious and direct consequence on the bottom line: a 0.1% RMA[1] rate for a wireless router device, or a single melt down of a critical core router in a strategic customer site can spell the difference between a successful project or a failed one. Despite this, all too often design requirements and QA processes do not take Murphy's Law into account, leading to a serious issue which is only detected in the most painful way by a customer, after the product has been shipped.

[1]Return Materials Authorization, frequently used to refer to all returned product, whether authorized or not.

If there is a way therefore, to build Murphy-compliant systems, as it were, that will survive the worse possible scenario without costing the implementor too much money or time, it will be a great boon to society, not to mention embedded system developers.

As always, a trade off is at work here: for example, we can coat the system developed with a thick layer of lead, thus protecting it from damage by cosmic rays. This, however, is not very logical to do—the price-to-added-protection ratio is simply not attractive enough.

We must therefore pick our battles wisely.

In the course of a 7-year career working on building GNU/Linux-based embedded systems, we have identified two points of failure which we believe based on anecdotal evidence to be responsible for a significant number of embedded system failures, and that are easily addressable with no more then a little premeditative thought and the GNU/Linux feature set. In this paper we describe those points and suggest an efficient way to address them when developing GNU/Linux-based embedded systems. Those points are:

- Embedded system configuration
- Embedded system software upgrade

The lessons we talk about were learned the hard way: three different products sold in the market today (by Juniper Networks Inc., Finjan Software Inc., and BeyondSecurity Ltd.) already make use of ideas or whole parts of the system we're about to introduce here, and more products are on the way. In addition, as we will later disclose—we are not the first going down this road, but more on that later.

The rest of this paper is outlined as follows: In Section 2 we present current approaches, their

weaknesses and strengths. In Section 3 we present the requirements from a system which will have all the strengths of the current approaches but none of their weaknesses. In Section 4 we present our approach to solving the problem of embedded system configuration: `cfgsh`, the configuration shell. In Section 5 we present our approach to solving the problem of embedded system upgrade: `sysup`: the system upgrade utility. In Section 6 we discuss future directions, and we conclude in Section 7.

2 Current approaches: The good, the bad, and the ugly

In this section we will present two of the more common approaches: the "naïve" approach and the RTOS approach. We will discuss each approach as to its merits and faults.

2.1 The "naïve" approach: tar balls and rc files

When a developer familiar with the Unix way is faced with the task of building a GNU/Linux-based embedded system, his or her tendency when it comes to handling configuration files and software update is to mimic the way such tasks are traditionally handled in Unix based workstation or servers [Embedded Linux Systems]. The flash device is used in the same way a hard disk is used in a traditional GNU/Linux workstation or server.

System configuration state, such as IP addresses, host name, or the like, is stored in small text files which are read by scripts being run by the `init(8)` process at system startup. Updating the configuration calls for editing the text files and possibly re-running the scripts.

In a similar fashion, a software upgrade is done by downloading and opening tar files of binaries which replace the system binaries and restarting the relevant processes. The more "advanced" developers forgo tar files in favor of plain cpio archives, RPM, deb files, ipkg or proprietary formats which are essentially file archives as well.

2.1.1 The good: the Unix way

The strengths of this approach are self evident: this approach makes use of the Unix "Everything is a file" paradigm, configuration files are written in the universal interface of plain text, and since the system behaves like a regular GNU/Linux workstation or server installation, it's easy to build and debug.

In addition, because all the components of a software version are just files in a file system, one can replace individual files during system operation, offering an easy "patch" facility. In the development and QA labs, this is a helpful feature.

2.1.2 The bad: no atomic transactions

A power loss during a configuration or software update may result in a system at an inconsistent state. Since the operations being performed in either case are non atomic replacements of files, a power loss in the middle of a configuration change or a system upgrade can leave some of the files in a pre-changed status while the rest of the files have already been updated and the system is no longer in a consistent state.

Inconsistent here really can mean anything at all: from a system that boots with the wrong IP, through a system which behaves strangely

or fails in various ways due to incompatible library versions, and all the way up to a system that will not boot at all.

Considering that many embedded devices are being upgraded and managed via a network, a system with the wrong (or no) IP address may be as useless as a system which does not boot, when you are on the wrong side of the continent or even on a different continent altogether.

In addition, the ease of replacing single files, which is considered a boon in the development and QA labs, is a software-versions nightmare at the customer site. The ability to patch single files at a customer site gives rise to a multitude of unofficial mini-versions of the software. Thus, when a bug report comes in, how can one tell if the software really is "version 1.6" as the report says and not "version 1.6 with that patch we sent to this one customer to debug the problem but that the guys from professional services decided to put on each installation since"? The sad answer is: you can't.

2.1.3 The ugly: user interface

Editing configuration files and scripts or opening tar files is not an acceptable interface for the user of a embedded device. A tool has to be written to supply a decent interface for the user.

Given the lack of any such standard tool, every GNU/Linux-based embedded system developer seems to write one of its own. Sometimes, when there is a need for a configuration solution that spans telnet and serial CLI, web,and SNMP interfaces, three different configuration tools are written.

2.2 The RTOS approach: what we did in that other project

The RTOS[2] approach is favored by people experienced with legacy RTOS systems, which seldom have a file system at their disposal, because it costs extra.

The basic idea is that both configuration information and software versions are kept as blobs of data directly on the system flash.

Configuration is changed by mapping the flash disk memory and modifying the configuration parameters in place.

Software update is performed by overwriting whole images of the system software, comprised of the kernel, initrd or initramfs images to the flash. Some developers utilize an Ext2 [ext2] ram disk image, which leaves the system running from a read/write, but volatile environment.

Other developers prefer to use Cramfs [cramfs] or Squashfs [Squashfs] file systems, in which the root file system is read-only.

2.2.1 The good

The RTOS approach enjoys two advantages: atomic system upgrade (under certain conditions) and manageability of system software versions while possible retaining the ability to update single files at the lab.

Because system software is a single blob of data, we can achieve a sort of atomic update ability by having two separate partitions to store two versions of the system software. In

[2] For marketing reasons, most embedded OS vendors call their offering a Real Time OS, even if most of the projects using them have negligible real time requirements, if any.

this scenario, software update is performed by writing the new version firmware to the partition we *didn't* boot from, verify the write and marking in the configuration part of the flash the partition we just wrote to as the active one and booting from it.

In addition, because software updates are performed on entire file systems images, we need not worry about the software version nightmare stemming from the ability to update single files as we described in the Section 2.1.2 previously.

Furthermore, if we happen to be using a read/write but volatile root file system (such as a ram disk), we allow the developer the freedom to patch single files at run time, while having the safety guard of having all these changes rolled back automatically in the next reboot.

2.2.2 The bad

However, utilizing this advanced method requires additional flash space and a customized boot loader that can switch boot partition based on configuration information stored on the flash. Even then, we are required to prepare in advance a partition for each possible software version, which in practice leads only supporting two versions at a time.

In addition, booting into a verified firmware version with a major bug might still turn the machine into a brick.

As for the configuration information—it is kept as binary data on a flash, which is an inflexible and unforgiving format, hard to debug, and hard to backup.

2.2.3 The ugly

This approach suffers from the same need for a user interface as the naïve approach. While the approach based on standard Unix configuration files can at least rely on some common infrastructure to read and update its files, the RTOS approach dictates the creation of a proprietary tool to read the binary format in which the configuration is written on the flash.

Moreover, if different user interfaces are required to handle the configuration of the system (for example: telnet and serial CLI, web and SNMP interfaces) three different tools will have to be written or at least some common library that allows all three interfaces to cooperate in managing the configuration.

3 Requirements: building a better solution

In this section we present the requirements from a solution for updating and configuring an embedded system. These requirements are derived from the merits of existing approaches, while leaving out the limitations.

The selected approach should follow the following guidelines:

1. Allow atomic update of configuration and software versions.
2. Not require any special boot loader software.
3. Allow an update of individual files of the system software, but in a controlled fashion.
4. Everything that can be represented as a file, should.
5. Configuration files should be human readable and editable.

6. Offer a standard unified tools to deal with configuration and version management.

As we have seen, the naïve approach follows guidelines 2, 4, and 5 but fails to meet guidelines 1, 3, and 6. On the other hand the RTOS approach follows guidelines 1 and 3, although both of them optionally, and fails to meet guidelines 2, 4, 5, and 6.

It should be pointed out that both the approaches we introduced are only examples. One can think of many other approaches that follow some of the 6 guidelines but not all of them. Looking at the two approaches described above we can understand why—choosing one or the other of them is a trade off: it mandates choosing which of the guidelines you are willing to give up for the others.

Another thing worth mentioning is that there is no tool currently known to the authors which will be a good candidate to satisfy guideline 6. This is surprising, since the embedded GNU/Linux field is not short of such embedded space infrastructure (or framework): the busybox meta-utility maintained by Eric Anderson and friends or the crosstool script by Dan Kegel are two prime examples of such software which most (if not all) embedded GNU/Linux systems are built upon[3].

Still, no common framework exists today that deals with configuration and software upgrade of embedded systems in the same way that Busybox deals with system utilities and crosstool with building cross tool chains and which allows the embedded developer to build upon to create his or her respective systems.

[3] And which the authors of this article will gladly sacrifice a goat or two in order to show their gratitude to their maintainers if not for the very real fear of scaring them off from doing any additional work on their respective tools...

Can there really exist a solution which will allow us to follow all 6 guidelines with no compromises or do embedded systems are too tied up to their unique hardware platforms to give rise to such a unified tool? And if such a tool is made, will it need to be a complex and costly-to-implement solution requiring changes in the kernel, or a simple straightforward solution requiring no more than some knowledge in C?

Since you're reading this paper, you're probably assuming that we did came up with something in the end and you're perfectly right. But before we are going to tell you all about it we need to get something off of our chest first: we didn't really invent this solution at all.

Rather, when faced with the daunting task of building the perfect embedded configuration and upgrade tool(s) we chose to "stand on the shoulders of giants" and simply went off and found the best example we could lay our hands on and imitated it.

Our victim was the Cisco family of routers and its IOS operating system. Since we have observed that this specific product of embedded devices does seem to follow all of these guidelines, we naturally asked ourselves, "How did they do that?"

Cisco embedded products, however, do not run on GNU/Linux, our embedded OS of choice, nor does Cisco shares the code to its OS with the world [4]. What we are about to describe in the next chapters is therefore, how to get the same useful feature set of the Cisco line of embedded devices when using GNU/Linux—all implemented as Free Software.

[4] At least not willingly...

4 cfgsh – an embedded GNU / Linux configuration shell

cfgsh is an embedded GNU/Linux system configuration shell. It is a small C utility which aims to provides a unified standard way of handling the configuration of a GNU/Linux-based embedded system.

cfgsh was independently implemented from scratch, though it is influenced by the Cisco IOS shell. cfgsh supports three modes: an interactive mode, a setup mode, and a silent mode. Those modes will be described in the following subsections.

4.1 Interactive mode

Interactive mode gives a user an easy text-based user interface to manage the configuration, complete with menus, context sensitive help and command line completion. This is the default mode.

Upon entering the program, the user is presented with a prompt of the host name of the machine. The user can then manage the system configuration by entering commands. On-line help is available for all menus.

The GNU readline library [GNU Readline] is used to implement all the interaction with the user.

Figure 4.1 shows cfgsh main help menu.

The user may enter a sub-menu by entering the proper command. Upon doing so, the prompt changes to reflect the menu level the user is at that moment.

Figure 2 shows how the network menu is entered.

```
linbox>help
      role  Display or set system role: role [role].
  timezone  Display or set time zone: timezone [time zone].
   network  Enter network configuration mode: network.
      ping  Ping destination: ping <hostname | address>.
  hostname  Displays or set the host name: hostname [name].
      halt  Shutdown.
    reboot  Reboot.
      show  Display settings: show [config | interfaces | routes | resolver].
      save  Save configuration.
      exit  Logout.
      quit  Logout.
      help  Display this text.
linbox>
```

Figure 1: cfgsh main menu help

At any stage the user may utilize the online context-sensitive line help by simply pressing the [TAB] key. If the user is entering a command, the result is simple command completion. If the user has already specified a command and she is requesting help with the parameters, she will get either a short help text on the command parameters or parameter completion, where appropriate.

Figure 3 shows the command-line completion for the “timezone” command[5]

Every change of configuration requested by the user is attempted immediately. If the attempt to reconfigure the system is successful, it is also stored in the cfgsh internal configuration “database.”

The user can ask to view cfgsh internal configuration database, which reflects (barring

[5] As can be guessed, the source for the suggested values for the timezone command are the list of files found in /usr/share/zoneinfo/. These are dynamically generated and are a good example of how cfgsh utilizes the GNU readline library to create a friendly user interface.

```
linbox>network
linbox (network)>help
interface     Enter interface configuration mode:
              interface [interface].
    route     Enter route configuration mode:
              route [priority].
  default     Display or set default gateway ad-
              dress: gateway [address].
 resolver     Enter domain name resolution con-
              figuration mode: resolver.
     exit     Return to root mode.
     quit     Logout.
     help     Display this text.
linbox (network)>
```

Figure 2: `cfgsh` network sub-menu

```
linbox>timezone
  timezone Display or set time zone: timezone [time zone].
Africa          Cuba            GMT+0           Kwajalein       Pacific         W-SU
America         EET             GMT-0           Libya           Poland          WET
Antarctica      EST             GMT0            MET             Portugal        Zulu
Arctic          EST5EDT         Greenwich       MST             ROC             iso3166.tab
Asia            Egypt           HST             MST7MDT         ROK             posix
Atlantic        Eire            Hongkong        Mexico          Singapore       posixrules
Australia       Etc             Iceland         Mideast         SystemV         right
Brazil          Europe          Indian          NZ              Turkey          zone.tab
CET             Factory         Iran            NZ-CHAT         UCT
CST6CDT         GB              Israel          Navajo          US
Canada          GB-Eire         Jamaica         PRC             UTC
Chile           GMT             Japan           PST8PDT         Universal
linbox>timezone Africa/Lu
  timezone Display or set time zone: timezone [time zone].
Luanda          Lubumbashi      Lusaka
linbox>timezone Africa/Lusaka
```

Figure 3: `cfgsh` timezone context sensitive help

bugs: see below on loosing sync with the system **??**) the system status using "show config" command. When used, the "show config" command will display the list of `cfgsh` commands that, once fed into `cfgsh`, will re-create the current configuration state.

Figure 4 shows an example of such a report.

In order to save the current system information for the next system boot, the user enters the command "save," which stores the configuration as a text file comprised of `cfgsh` commands. If issued, those commands will bring the system to the exact current state. This config text file looks exactly like the output of the "show config" commands (and is in fact generated from the same code).

```
linbox>show config
# Configuration Shell config file
hostname linbox
timezone Israel/IDT
network
    interface eth0
        dhcp off
        ip 192.168.1.1
        netmask 255.255.255.0
        broadcast 2192.168.1.255
        exit
    default none
    route 0
        set none
        exit
    route 1
        set none
        exit
    route 2
        set none
        exit
    resolver
        primary 194.90.1.5
        secondary 194.90.1.7
        search codefidence.com
        exit
    exit
role none
linbox>
```

Figure 4: cfgsh show config command output

Unless the user has issued the "save" command, all the changes to the system configuration are in effect only until the next system reboot, at which point the previous configuration will be used.

4.2 Setup mode

The purpose of setup mode is to allow `cfgsh` to set up the system as a replacement for system rc files. This mode is entered by running the program with the "setup" argument. Normally, this will be done once when the system boots, on every boot, by calling the program from the system `inittab(5)` file.

During setup mode, `cfgsh` reads the text config file saved using the "save" command in interactive mode and executes all of the command in the file in order to automatically set up the embedded system while also initializing the run time configuration data base in the

shared memory segment for future instances of `cfgsh` running in interactive or silent mode.

After the file has been read and all the commands executed, `cfgsh` exists. When running in this mode, the normal prompt and some output is suppressed but normal messages are printed to stdout (e.g. "the system IP is now 192.168.1.1").

4.3 Silent mode

Silent mode is `cfgsh` way of supporting a simple programmable interface to manage the configuration to other programs, such as web management consoles and the like. This mode is entered by supplying the argument "silent" when running the program.

In this mode `cfgsh` runs exactly like in interactive mode, except that the prompt and some verbose output is suppressed. A program wishing to change the system configuration can simply run an instance of `cfgsh` running in silent mode and feed it via a Unix pipe `cfgsh` command for it to execute.

4.4 Internal configuration database

The internal configuration database is kept in a POSIX shared memory object obtained via `shm_open(3)` which is shared between all instances of `cfgsh`[6] and which stays resident even when no instance of `cfgsh` is running.

Thanks to this design decision, `cfgsh` does not need to re-read configuration files or query system interfaces when an instance of it is being run, since the information is available in the shared memory object.

```
typedef struct {
  char ip[NUMIF][IPQUADSIZ];
  char nmask[NUMIF][IPQUADSIZ];
  char bcast[NUMIF][IPQUADSIZ];
  char gw[IPQUADSIZ];
  char ns_search[HOST_NAME_MAX];
  char ns1[IPQUADSIZ];
  char ns2[IPQUADSIZ];
  char role[PATH_MAX];
  char tz[PATH_MAX];
  char dhcp[NUMIF][DHCP_OPT];
  char dhcp_is_on[NUMIF];
  char hostname[HOST_NAME_MAX];
  char route[ROUTE_NUM][MAX_ROUTE_SIZE];
  char num_ifs;
} CONF;
```

Figure 5: Internal configuration database structure

This design also suffers from at least one downside: since most of the information in the configuration database is already present in the system in some form (the Linux kernel for IP addresses or /etc/resolv.conf for resolver address for example), there is always a risk of losing sync with the real state of the system. Despite this down side we believe that the central database which holds all the configuration information in a unified format is a design win (for embedded systems) despite the replication of information.

Figure 5 shows the structure of this internal database.

4.5 Command structure

`cfgsh` menus are comprised from arrays of commands. The program maintain a pointer to the current menu which is initialized in program start to the array of the main menu. Each choice of a sub-menu simply replaces the current menu pointer with the address of the ap-

[6] At the time of writing this paper, cfgsh still misses *correct* code that will prevent race conditions when accessing the shared memory area by multiple instances at the same time. This is however on the TODO list...

```
typedef struct {
  char *name;
  rl_icpfunc_t *func;
  char *doc;
  complete_func_t *complete_func;
  char * complete_param;
} COMMAND;
```

Figure 6: `cfgsh` commands structure

propriate command array. It also updates the prompt.

Each command entry in the command array is a command structure which holds a pointer to the function to be called to perform the command, a description line to be shown as part of the help, a GNU readline library command competition function to perform context sensitive help for the command and a parameter to pass to the completer function to enable re-use of common functions (like directory competition).

Figure 6 shows the structure used to hold a single command entry.

4.6 Atomic configuration update

As have been described previously, `cfgsh` keeps the configuration database in memory and only commits it to disk (as a text file containing `cfgsh` commands) at the user requests via the "save" command. The same file is then used during the next boot to initialize booth the system and `cfgsh` own configuration database.

As can be understood, writing this configuration file correctly so that in to point on time we will not have a corrupt (or empty) configuration, is very important part of what `cfgsh` is meant to accomplish.

The method used is a very simple and well know one, which is based on the fact that the

```
int commit_file(char *tmp_file, char *file)
      int ret = 0;
      int fd = open(tmp_file, O_RDWR);
      if(fd == -1) return errno;
      if((ret = fsync(fd)) == -1) {
        close(fd);
        goto error;
      }
      if((ret = close(fd)) == -1) goto error;
      if ((ret = rename(tmp_file, file)) != 0)
            goto error;
      return 0;
error:
      unlink(tmp_file);
      return ret;
}
```

Figure 7: The commit_file() procedure

POSIX standard mandates that the if the second argument to the `rename(2)` system call already exists, the call will atomically replace the old file for the new file such that there is not point at which another process attempting to access the original name will find it missing.

To utilize this fact, we simply first created a full configuration file at a temporary location, sync its content to disk using `fsync(2)` and then `rename(2)` the new file over the old file.

Figure 7 shows the code of the `commit_file()` procedure that does the actual heavy lifting.

One thing which is perhaps missing from the procedure is a sync to the directory which holds the configuration file after the rename is over. Without this extra sync a sudden power failure after the rename may result in the directory entry never being written to permanent storage and the old configuration file used after reboot.

We believe that this is a valid scenario, as our purpose is to guarantee that the operation either fails as a whole or succeed as a whole but people who consider (quite rightfully) a system which boots with the previous IP address and

network parameters after a configuration save a failure can simple add an fsync to the directory where the configuration file resides.

This brings up another issue to consider - the atomicity of this method is really only valid if and only if the underlying file system saves a directory entry as part of an atomic transaction. Since file systems that do exactly this are not rare (e.g. Ext3 [ext3]) this is considered a reasonable requirement by the authors, but it is worth noting by would be implementors.

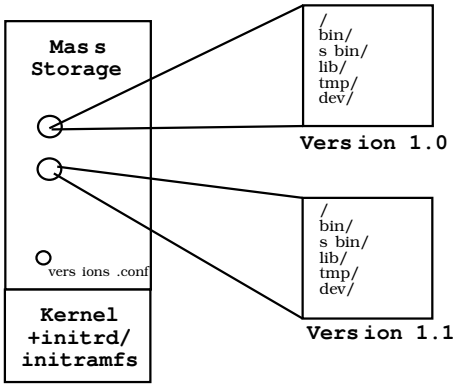


Figure 8: File systems layout with sysup

5 sysup – embedded GNU/Linux system software update utility

sysup—an embedded GNU/Linux system software update utility—is a very simple tool that is meant to run at boot time from initrd/initramfs of a Linux-based embedded system in order to mount the root file system. Its objective is allow for an easily and atomically update-able embedded system software versions.

5.1 File system structure

To utilize sysup, the system file system layout must be done in a certain specific way, which is a little different from the normal layout used in embedded systems.

We shall define several different file system:

Main storage This is the file system on the main storage of the system—usually the flash disk. JFFS2 [JFFS2] or Ext3 appropriate file system types. This file system will contain configuration files and images of versions file system but not system software or libraries.

Version image This is a file system that contains the files for a specific version of the system software. It is meant to be used as the root file system of the system and contains all binaries, static configuration files, device files and software. Version images are (possibly compressed) loop back images and reside as files on the Main storage file system. Cramfs or Squashfs are the prime candidate as the type of these file system, although an Ext2 file system can be used as well if it is mount read-only.

initrd/initramfs This is a file system image or cpio archive which are used to host the files of the sysup mechanism. These file system are mounted during boot and discarded once the boot sequence is complete. Again, Cramfs, Squashfs, or Ext2 are good choices for the kind of this file system.

Figure 8 shows a schematic of the various file systems layout in relation to each other.

5.2 The boot process with sysup

What sysup does can be best explained by describing the basic boot process on a sysup enabled system:

1. System boots.
2. Boot loader loads kernel and initrd/initramfs images into memory.
3. Kernel runs and mounts initrd or initramfs content
4. sysup is run.
5. sysup mounts the main storage.
6. sysup locates on the main storage the versions.conf file.
7. sysup locates on the main storage a version image.
8. sysup takes an MD5 signature of the version image and compares it to the one stored in the versions.conf file.
9. If the signatures do not match or in response to any other failure, sysup rolls back to the previous version by moving on to the next entry in the versions.conf file and branching back to stage 7[7].
10. If the signatures match, sysup will loop back mount the version image in a temporary mount point.
11. sysup will move the mount point of the main storage device into a mount point in temporary mount point of the version image. This is done using the "new" (2.5.1, but back ported to 2.4) MS_MOVE mount flag to `mount(2)`[8].
12. sysup will then `pivot_root(2)` into the temporary mount point of the mounted version image, thus turning it to the new root file system.

[7] At the time of the writing of this paper only 2 versions.conf entries are supported, but changing this is very easy should the need ever arise.

[8] Used by the `-move` options to `mount(8)`.

```
7e90f657aaa0f4256923b43e900d2351 \
/boot/version-1.5.img
2c9d55454605787d5eff486b99055dba \
/boot/versions-1.6.img
```

Figure 9: versions.conf

13. The boot is completed by un-mounting the initrd or initramfs file systems and execing into `/sbin/init`.

An example version.conf file is shown in figure 9. A very simple implementation of sysup as a shell script is in figure 10.

5.3 System software update

The above description of a sysup boot sequence sounds more complicated then usual. On the other hand, the system software upgrade procedure is quite simple:

1. Download a new version image to the main storage storage.
2. Calculate its MD5sum and do any other sanity checks on the image.
3. Create a new versions.conf file under a temporary name, with the MD5 and path of the new version image as the first entry and the old version image and its MD5 sum (taken from the current version.conf file) as the second entry.
4. fsync the new versions.conf under its temporary name.
5. rename(2) the new version.conf file over the old one.
6. Reboot.

Once again, just like with `cfgsh` configuration file, the POSIX assured atomicity of the rename(2) system call, guarantees that at no point in time will a loss of power lead to a system that cannot boot.

```
#!/bin/sh
# Name and path to file with filename and MD5s
VERSIONS=versions
# How to get the first line of the file
LINE=‘tail -n 1 versions‘
# File name if MD5 is correct, empty otherwise
FILE=‘./md5check $LINE‘
# How to get the second line of the file
ALTLINE=‘head -n 1 versions‘
# File name if MD5 is correct, empty otherwise
ALTFILE=‘./md5check $LINE‘
# File system type of images
FSTYPE=ext2
# Mount point for new root
MNT=/mnt
# File system type for data parition
# (which holds the image files)
DATAFSTYPE=ext3
# Mount point of data partition
DATA=/data
# Name of directory inside the images
# where the original root mount point
# will be moved to
OLDROOT=initrd
# device of data parition
DATADEV=/dev/hda1
# Emergency shell
EMR_SHELL=/bin/sh
boot() {
      mount -t $FSTYPE $FILE $MNT && \
      cd $MNT && \
      pivot_root . $OLDROOT &&
      mount $OLDROOT/$DATA $DATA -o move && \
      umount $OLDROOT && \
      exec /sbin/init
}
mount -t proc /proc && \
mount -t $DATAFSTYPE $DATADEV && \
if test -z "$FILE"; then \
  echo "Attempting to boot 1st choice" && boot(); \
fi && \
if test -z "$ALTFILE"; then \
  echo "Attempting to boot 2nd choice" && boot(); \
fi
echo "Boot failure." && exec $EMR_SHELL
```

Figure 10: sysup shell script

5.4 Variations on a theme

To this basic scheme we can add some more advanced features as described in this section. None of these implemented in the current version of sysup, but they are on our TODO list for future versions.

5.4.1 Upgrade watchdog

A version image might have good checksum and mounted correctly, but the software in it might be broken in such a way as to get the machine stuck or reboot before allowing the user to reach a stage that he or she can roll back to the previous version.

To resolve this issue, or at least to mitigate its effect to some extent, the following addition can be made to the sysup boot process:

- During boot, before mounting a version image file, sysup should look on the main storage file system for a “boot in progress” indicator file. If the file is there, it should roll back and boot the next entry of versions.conf file.
- If the file is not there and before sysup mounts the new version image file, it will create a “boot in progress” indicator file on the main storage file system.
- After a version image finished its boot successfully to such a degree that the user can request a software version upgrade or

downgrade, the software on the version image will delete this "boot in progress" indicator from the main storage file system.

This addition to the boot process allows detect errors that would otherwise lead to a system that reboots into a broken version in an infinite loop.

5.4.2 Network software updates

Another possible extension to the sysup boot model is to extend sysup to look for newer version to boot in a network directory of sorts, in addition to the `versions.conf` file.

If a newer version is found, it is downloaded and Incorporated into the regular version repository on the main storage (perhaps deleting an older version to accommodate).

If the newest version on the network directory is the same as the version stored on the mass storage, boot continues as before.

5.4.3 Safe mode

Sometime, despite our best efforts, the version images on the flash can become corrupted. In such an event, it can be very useful to allow the sysup code in the initrd/initramfs image, when it detects such an occurrence, to enter a "safe mode" which will allow the minimal configuration of the system (e.g. network settings) and download of a fresh version image to flash.

5.5 The Achilles heel of sysup: kernel upgrades

The reason sysup is able to supply atomic upgrade of software version is exactly because, thank to the ability of the Linux kernel to loop back file system images, all the system software can be squeezed into a single file. Unfortunately, the kernel image itself cannot be included in this image for obvious reasons, which leads to a multitude of problems

As long as we are willing to treat the kernel and the initrd/initramfs images with it, as a sort of a boot ROM, only update-able in special circumstances by re-writing in a non atomic fashion the appropriate flash partition, we have no problem.

Unfortunately, this is not always enough. Bugs in the kernel, support for new features and the need of kernel modules to be loaded into the same kernel version for which they were compiled may require kernel upgrades, not to mentions bugs in sysup code itself...

There are two ways to overcome this limitation, each with its own set of problems:

5.5.1 Two kernel Monte

Under this scheme, we add another field to the `versions.conf` file—the version of the kernel required by that version image. sysup then needs to check whether the currently running kernel is the right one. If it is, we proceed as usual. If it is not we utilize a Linux based Linux loader such as kexec or similar in-kernel loaders [kexec] [9] and boot into the correct kernel. This time we will be in the right kernel version and boot will continue as normal.

This method works quite well, however it has two major drawbacks:

- At the time of the writing of this paper, neither kexec, two kernel Monte or lobos

[9] Our tests with this mode of operation were done with the Two kernel Monte module from Scyld Computing.

are integrated into the vanilla kernel, requiring a patch.

- Those solutions that do exist seems to cover x86 and to some degree ppc32 architecture only.
- Using this approach lengthens boot time.

5.5.2 Cooperating with the boot loader

As an alternative to booting a Linux kernel from Linux, we can use the very same mechanism discussed before of marking each version with the required kernel version to run it and simply change the boot loader configuration to boot the right kernel version next time and then reboot the machine. If all goes well, when we next run, we will already be running under the right kernel.

The drawback of this method is of course that we are now tied to the specific feature set of a specific boot loader and not all boot loader lend themselves easily to this sort of cooperation with regard to choosing the Linux kernel image to boot.

6 Read, Copy, Update

One of the readers of the early draft of this paper remarked how much our approach to create atomic update of complex data by creating a copy and then switching pointers to this data is similar to the well known IBM patented RCU method utilized in the latest Linux kernel versions.

While we happily agree that the mechanism is the basically the same, we would like to point out that the purpose of applying the technique (which we of course do not claim to have invented) is different: the RCU implementation in the Linux kernel is done to avoid locking when accessing data structure as an way to speed up access to these data structures, where as our use of the technique is done because it is *impossible* to lock the data structure we want to access, barring the use of a battery attached to each embedded device.

It is interesting though, to note the usefulness of the same technique to solve different, but related problems.

7 Future directions

Apart from implementing our lengthy TODO list, some of which has been discussed in this paper, there are some “blue skies” areas of interest for additional research with `cfgsh` and sysup.

The most interesting one, in the humble opinion of the authors, is the possibility that the techniques outlined here and implemented in the two projects can be useful outside the scope of embedded systems design, especially with regard to “stateless Linux,” virtual machine settings and GNU/Linux-based clusters.

Because the approach presented here essentially locks all the information about software versions and configuration in a couple of easily controlled files, and supports transactional management of these resources it is hoped that developers and researches working in those fields would be able to utilize the easy ability to change and save the state of a machine with regard to software version and configuration to create mechanism to automatically and safely control their systems, virtual instances or cluster node in the same way that we demonstrated can be done with embedded systems.

8 Thanks

The authors wish to thank the following people:

To Orna Agmon, Oleg Goldshmidt and Muli Ben-Yehuda for their support and advice during the writing of this paper.

To Aviram Jenik and Noam Rathaus from BeyondSecurity Ltd. for sponsoring the original creation of `cfgsh` as Free Software licensed under the GPL (which is why I forgive them for forgetting to send me their patches to the program.)

Last but not least, to my wife Limor, just for being herself.

References

[Squashfs] Artemiy I. Pavlov, *SquashFS HOWTO*, The Linux Documentation Project, `http://www.artemio.net/projects/linuxdoc/squashfs/`

[ext2] Remy Card, Theodore Ts'o, Stephen Tweedie *Design and Implementation of the Second Extended Filesystem*, The Proceedings of the First Dutch International Symposium on Linux, ISBN 90-367-0385-9, `http://www.artemio.net/projects/linuxdoc/squashfs/`

[Murphy] `http://dmawww.epfl.ch/roso.mosaic/dm/murphy.html#technology`

[GNU Readline] Chet Ramey and others, `http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html`

[cramfs] Linus Torvalds and others,

[ext3] Stephen Tweedie, *EXT3, Journaling Filesystem*, The Proceedings of Ottawa Linux Symposium 2000, `http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`

[JFFS2] David Woodhouse, *JFFS: The Journalling Flash File System*, `http://sources.redhat.com/jffs2/jffs2.pdf`

[Embedded Linux Systems] Karim Yaghmour, *Building Embedded Linux Systems*, O'Reilly Press, ISBN: 0-596-00222-X

[kexec] Andy Pfiffer, *Reducing System Reboot Time With kexec*, `http://developer.osdl.org/rddunlap/kexec/whitepaper/kexec.html`

Can you handle the pressure? Making Linux bulletproof under load

Martin J. Bligh, mbligh@mbligh.org
Badari Pulavarty, pbadari@us.ibm.com
Andy Whitcroft, apw@shadowen.org
Darren Hart, dvhltc@us.ibm.com
IBM Linux Technology Center

Abstract

Operating under memory pressure has been a persistent problem for Linux customers. Despite significant work done in the 2.6 kernel to improve its handling of memory, it is still easy to make the Linux kernel slow to a crawl or lock up completely under load.

One of the fundamental sources for memory pressure is the filesystem pagecache usage, along with the `buffer_head` entries that control them. Another problem area is inode and dentry cache entries in the slab cache. Linux struggles to keep either of these under control. Userspace processes provide another obvious source of memory usage, which are partially handled by the OOM killer subsystem, which has often been accused of making poor decisions on which process to kill.

This paper takes a closer look at various scenerios causing of memory pressure and the way VM handles it currently, what we have done to keep the system for falling apart. This paper also discusses the future work that needs to be done to improve further, which may require careful re-design of subsystems.

This paper will try to describe the basics of memory reclaim in a way that is comprensible. In order to achieve that, some minor details have been glossed over; for the full gore, see the code. The intent is to give an overview first to give the reader some hope of understanding basic concepts and precepts.

As with any complex system, it is critical to have a high-level broad overview of how the system works before attempting to change anything within. Hopefully this paper will provide that skeleton understanding, and allow the reader to proceed to the code details themselves. This paper covers Linux® 2.6.11.

1 What is memory pressure?

The Linux VM code tries to use up spare memory for cache, thus there is normally little free memory on a running system. The intent is to use memory as efficiently as possible, and that cache should be easily recoverable when needed. We try to keep only a small number of pages free for each zone—usually between two watermarks: `zone->pages_low` and `zone->pages_high`. In practice, the interactions between zones make it a little more complex, but that is the basic intent. When the

system needs a page and there are insufficient available the system will trigger reclaim, that is it will start the process of identifying and releasing currently in-use pages.

Memory reclaim falls into two basic types:

- per-zone general page reclaim: `shrink_zone()`
- slab reclaim: `shrink_slab()`

both of which are invoked from each of 2 places:

- kswapd—background reclaim daemon; tries to keep a small number of free pages available at all times.
- direct reclaim—processes freeing memory for their own use. Triggered when a process is unable to allocate memory and is willing to wait.

2 When do we try to free pages?

The normal steady state of a running system is for most pages to be in-use, with just the minimum of pages actually free. The aim is to maintain the maximum working set in memory whilst maintaining sufficient truly empty pages to ensure critical operations will not block. The only thing that will cause us to have to reclaim pages is if we need to allocate new ones. In the diagram below are the watermarks that trigger reclaim activities.

Caption: For highmem zones, `pages_min` is normally 512KB. For lowmem, it is about `4*sqrt(low_kb)`, but spread across all low zones in the system. For an ia32 machine with 1GB or more of memory, that works out at about 3.8MB.

The memory page allocator (`__alloc_pages`) iterates over all the allowable zones for a given allocation (the zonelist) and tries to find a zone with enough free memory to take from. If we are below `pages_low`, it will wake up kswapd to try to reclaim more. If kswapd is failing to keep up with demand, and we fall below `pages_min`, each allocating process can drop into direct reclaim via `try_to_free_pages` ... searching for memory itself.

3 What pages do we try to free?

The basic plan is to target the least useful pages on the system. In broad terms the least recently used pages (LRU). However, in practice it is rather more complex than this, as we want to apply some bias over in which pages we keep, and which we discard (e.g. keeping a balance of anonymous (application) memory vs filebacked (pagecache) memory).

Some pages (e.g. slabcache and other kernel control structures) are not kept on the LRU lists at all. Either they are not reclaimable, or require special handling before release (we will cover these separately below).

3.1 The LRU lists

We keep memory on two lists (per zone)—the active and inactive lists. The basic premise is that pages on the active list are in active use, and pages on the inactive lists are not. We monitor the hardware pagetables (on most architectures) to detect whether the the page is being actively referenced or not, and copy that information down into the struct page in the form of the `PG_referenced` flag.

When attempting to reclaim pages, we scan the LRU lists; pages that are found to be active will

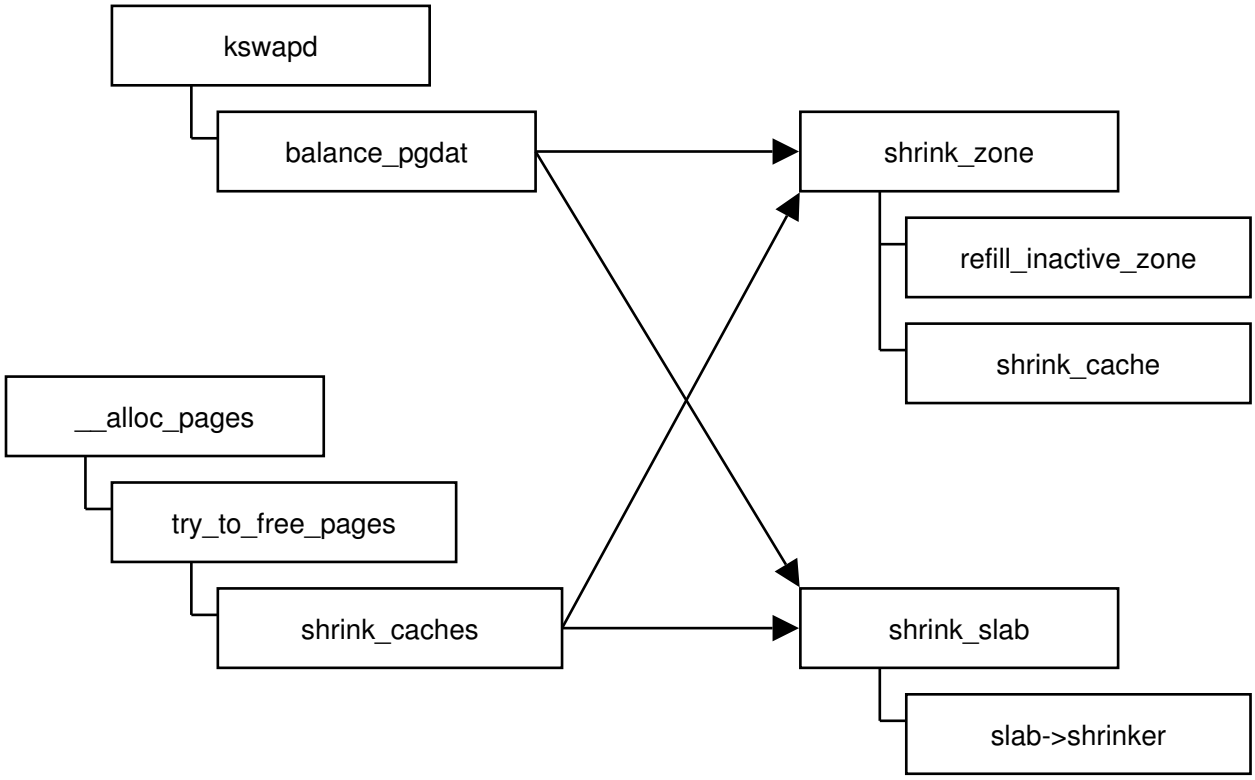


Figure 1: Memory Reclaimers

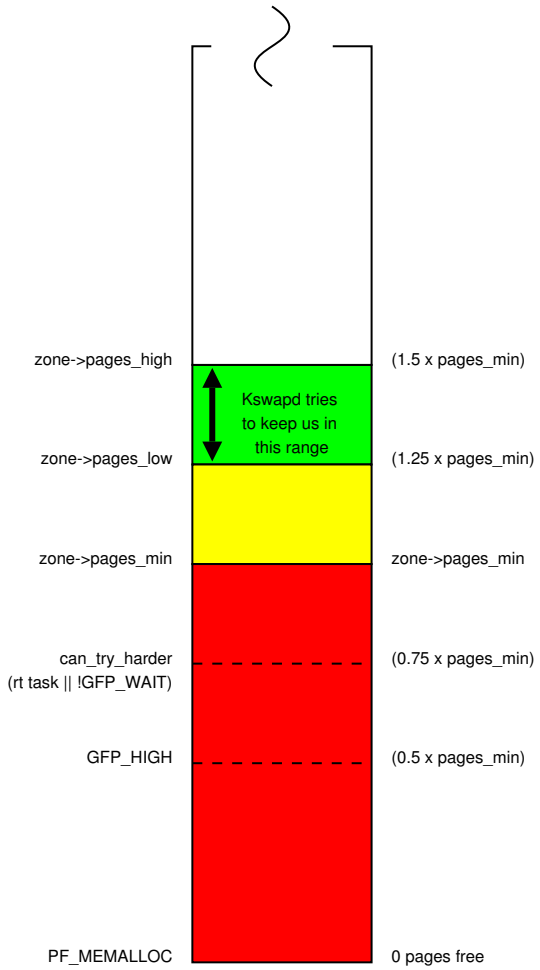


Figure 2: Zone Reclaim Watermarks

be moved to the head of the active list, pages that are found to be inactive will be demoted:

- If they were on the active list, they will be moved to the inactive list.
- If they were on the inactive list, we will try to discard them

3.2 Discarding pages

Reclaiming an in-use page from the system involves 5 basic steps:

- free the pagetable mappings (`try_to_unmap()`)
- clean the page if it is dirty (i.e. sync it to disk)
- release any `buffer_heads` associated with the page (explained in section below)
- Remove it from the pagecache
- free the page

3.3 Freeing the pagetable mappings

Freeing the pagetable mappings uses the rmap (reverse mapping) mechanism to go from a

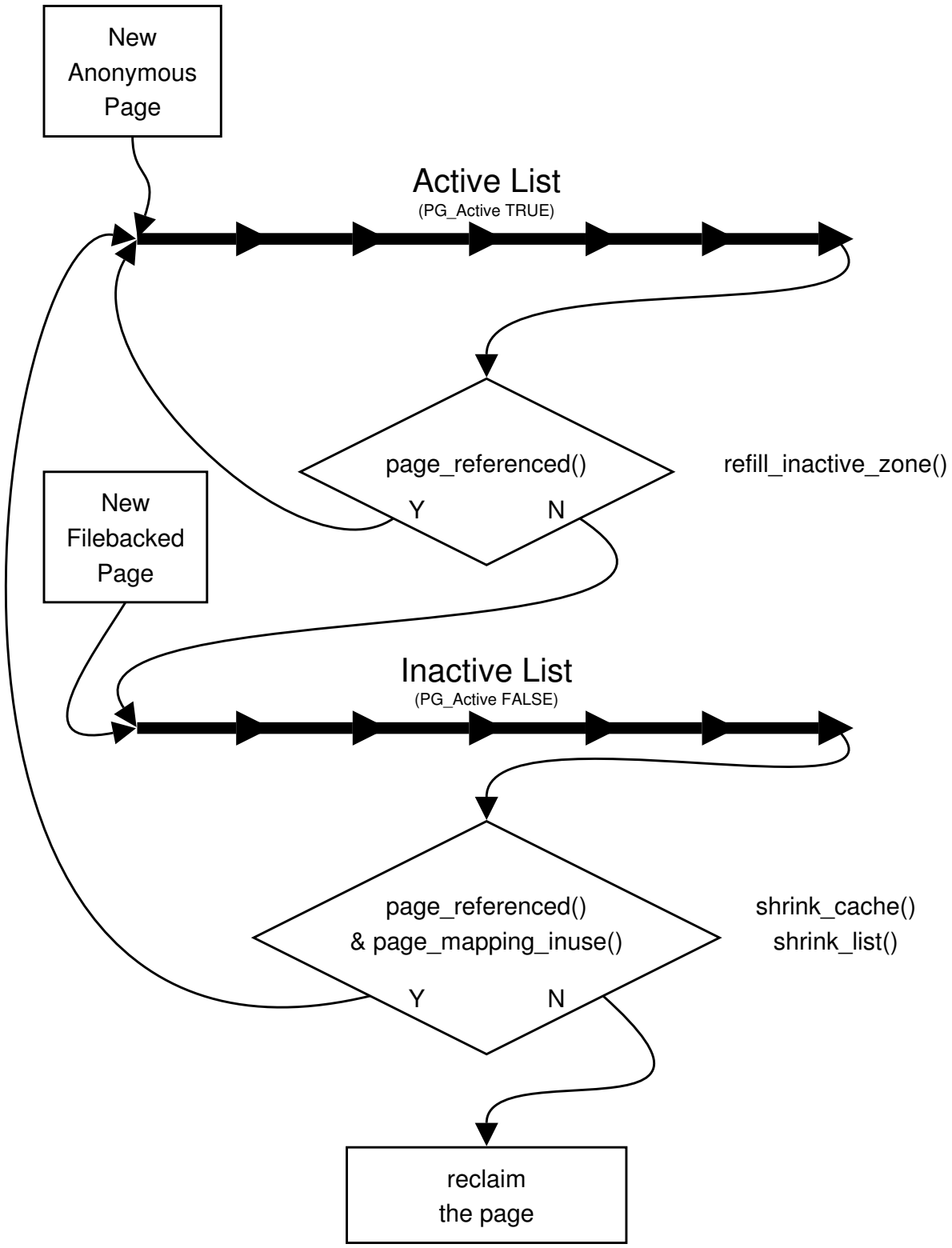


Figure 3: LRU lists

physical page to a list of the pagetable entries mapping it. The mechanism for how this works depends on whether the page is anonymous, or filebacked

- Anonymous page (`try_to_unmap_anon()`) use the `anon_vma` structure to retrieve the list of `vma`s mapping the page
- Filebacked page (`try_to_unmap_file()`) Go via the `address_space` structure (the file's controlling object) to retrieve a list of `vma`s mapping the page.

From the `vma`, combined with the offset information in the struct page, we can find the virtual address within the process, and walk the pagetables to the PTE entry.

4 Buffer heads

A `buffer_head` is a control structure for a page in the pagecache, but are not required for all pages. Their basic usage is to cache the disk mapping information for that pagecache page.

4.1 Why are bufferheads used?

- to provide support for filesystem blocksizes not matching system pagesize. If the filesystem blocksize is smaller than the system pagesize, each page may end up belonging to multiple physical blocks on the disk. Buffer heads provide a convenient way to map multiple blocks to a single page.
- To cache the page to disk block mapping information. All the pages belong to a file/inode are attached to that inode using the logical offset in the file and they are represented by a radix tree. This will significantly reduce the search/traversal times to map from a given file offset to the backing pagecache page. However, the filesystem can map these pages on the disk whatever way it wants. So every time, we need disk block mapping, we need to ask the filesystem to give us physical block number for the given page. Bufferheads provide a way to cache this information and there by eliminates an extra call to filesystem to figure out the disk block mapping. Note that figuring out the disk block mapping could involve reading the disk, depending on the filesystem.
- In order to provide ordering guarantees in case of a transaction commit. Ext3 ordered mode guarantees that the file data gets written to the disk before the metadata gets commited to the journal. In order to provide this guarantee, bufferheads are used as mechanism to link the pages belong to a transaction. If the transaction is getting commited to the journal, the `buffer_head` makes sure that all the pages attached to the transaction using the bufferhead are written to the disk.
- as meta data cache. All the meta data (superblock, directory, inode data, indirect blocks) are read into the buffer cache for a quick reference. Bufferheads provide a way to access the data.

4.2 What is the problem with bufferheads?

- Lowmem consumption: All bufferheads come from `buffer_head` slab cache (see later section on slab cache). Since all the slabs come from `ZONE_NORMAL`, they all consume lowmem (in the case of ia32). Since there is one or more `buffer_head` structures for each filesystem pagecache page, the `buffer_head` slab

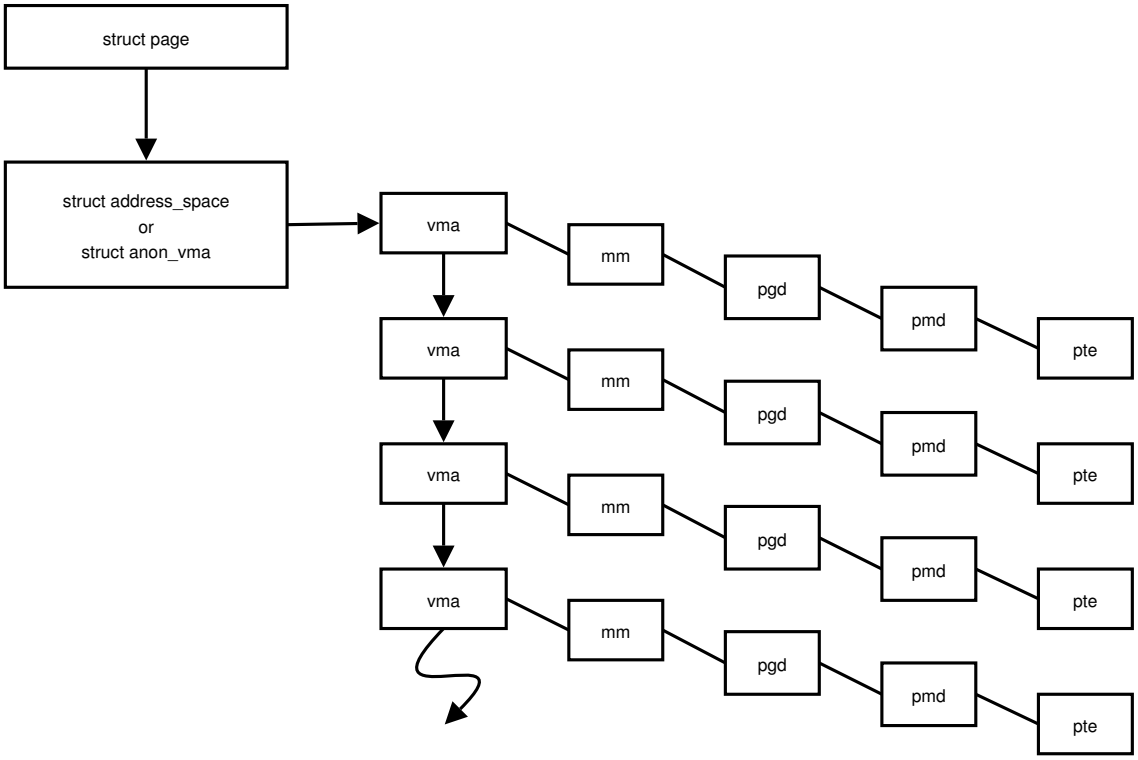


Figure 4: Object based rmap

grows very quickly and consumes lots of lowmem. In an attempt to address the problem, there is a limit on how much memory "bh" can occupy, which has been set to 10% of `ZONE_NORMAL`.

- page reference handling: When buffer-heads get attached to a page, they take a reference on the page, which is held until the VM tries to release the page. Typically, once the page gets flushed to disk it is acceptable to release the bufferhead. However, there is no clean way to drop the `buffer_head`, since the completion of the page being flushed is done in interrupt context. Thus we leave the buffer-heads around attached to the page and release them as and when VM decides to reuse the page. So, its normal to see lots of bufferheads floating around in the system. The `buffer_head` structures are allocated via `page_cache_get()`, and freed in `try_to_free_buffers()`.

- TLB/SLB/cache efficiency: Everytime we reference the `buffer_head`'s attached to page, it might cause a TLB/SLB miss. We have observed this problem with a large NFS workload, where ext3 `kjournald()` goes through all the transactions, all the journal heads and all the bufferheads looking for things to flush/clean. Eliminating bufferheads completely would be the best solution.

5 Non-reclaimable pages

Memory allocated to user processes are generally reclaimable. A user process can almost always be stopped and its memory image pushed out onto swap. Not all memory in the system can be so easily reclaimed: for example, pages allocated to the kernel text, pagetable pages, or those allocated to non-cooperative slab caches (as we will see later) may not be readily freed. Such memory is termed non-reclaimable—the ultimate owner of the allocation may not even be traceable.

5.1 Kernel Pages

By far the largest source of non-reclaimable allocations come from the kernel itself. The kernel text, and all pagetables are non-reclaimable. Any allocation where the owner is not easily determined will fall into this category. Often this occurs because the cost of maintaining the ownership information for each and every allocation would dominate the cost of those allocations.

5.2 Locked User Pages

The mlock system call provides a mechanism for a userspace process to request that a section of memory be held in RAM. mlock operates on the processes' reference to the page (e.g. the `vma` and pagetables), not the physical page controlling structure (e.g. the struct page). Thus the lock is indicated within the vma by the `VM_LOCKED` flag.

Whilst it would be useful to track this state within the struct page, this would require another reference count there, for something that is not often used. The `PG_locked` flag is sometimes confused with mlock functionality, but is not related to this at all; `PG_LOCKED` is held whilst the page is in use by the VM (e.g. whilst being written out).

5.3 Why are locked pages such an issue?

Locked pages in and of themselves are not a huge issue. There will always be information

which must remain in memory and cannot be allowed to be 'moved' out to secondary storage. It is when we are in need of higher order allocations (physically contiguous groups of pages) or are attempting to hotplug a specific area of physical memory that such 'unmovable' memory becomes an issue.

Taking a pathological example (on an ia32 system), we have a process allocating large areas of anonymous memory. For each 1024 4k pages we will need to allocate a page table page to map it, which is non-reclaimable. As allocations proceed we end up with a non-reclaimable page every 1025 pages, or one per `MAX_ORDER` allocation. As those unreclaimable pages are interspersed with the reclaimable pages, if we now need to free a large physically contiguous region we will find no fully reclaimable area.

6 Slab reclaim

The slab poses special problems. The slab is a typed memory allocator and as such takes system pages and carves them up for allocation. Each system page in the slab is potentially split into a number of separate allocations and owned by different parts of the operating system. In order to reclaim any slab page we have to first reclaim each and every one of the contained allocations.

In order to be reclaimable a slab must register a reclaim method—each slab can register a callback function to ask it to shrink itself, known as a "shrinker" routine. These are registered with `set_shrinker()` and unregistered with `remove_shrinker()`, held on `shrinker_list`, and called from `shrink_slab()`. Note that most slabs do NOT register a shrinker, and are thus non-reclaimable, the only ones that currently do are:

- directory entry cache (dentry)
- disk quota subsystem (dquot)
- inode cache (icache)
- filesystem meta information block cache (mbcache)

6.1 The Blunderbuss Effect

Taking the dentry cache as an example, we walk an LRU-type list of dentries—but note this is *entries*, not of *pages*. The problem with this is that whilst it will get rid of the best dcache entries it may not get rid of any whole pages at all. Imagine the following situation, for example:

Each row represents a page of dentrys, each box represents in individual dentry. Whilst many entries have been freed, no pages are reclaimable as a result—I call this the blunderbuss effect. We are suffering from internal fragmentation, which is made worse by the fact that some of the dentries (e.g. directories) may be locked. We actually have a fairly high likelihood of blowing away a very significant portion of the cache before freeing any pages at all. So whilst the shrink routine is good for keeping dcache size in check, it is not effective at shrinking it.

Dentry holds a reference to the inode as well. When we decrement the reference count to the dentry, the inode entry count is decremented as well. If the inode refcount is decremented to 0, we will call `truncate_inode_pages()` which will write back the pages for that file to disk. That could take a *very* long time to complete. This means that slab reclaim can cause very high latencies in order to allocate a page.

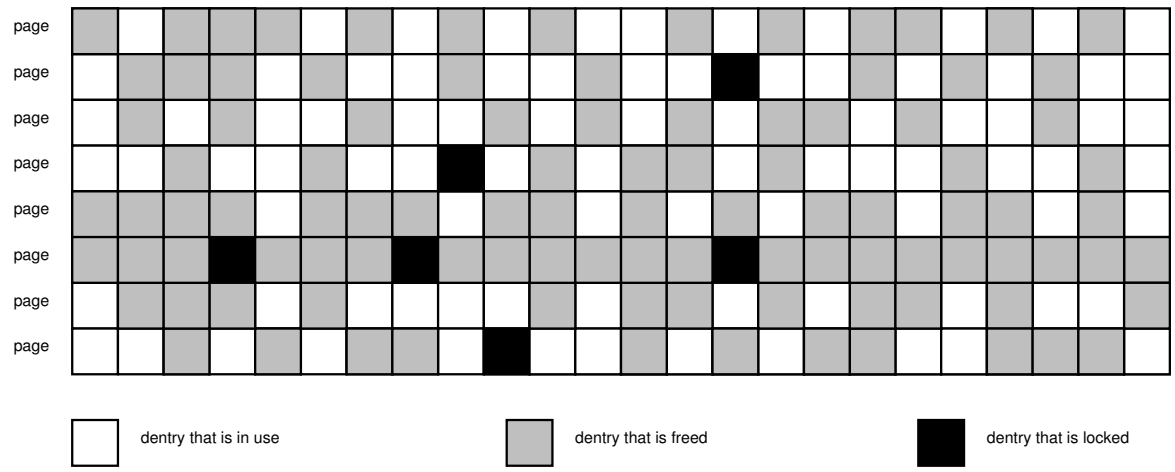


Figure 5: dentry slab

7 Diagnosing OOM situations

When the system runs out of memory, you will typically see messages either from the OOM killer, or "page allocation failures." These are typically symptoms that either:

- Your workload is unreasonable for the machine
- Something is wrong

If the workload does not fit into RAM + SWAP, then you are *going* to run out of memory. If it does not fit into RAM, then it will probably perform badly, but should still work.

7.1 Examining error messages

When `__alloc_pages` can not allocate you the memory you asked for, it prints something like this:

`%s: page allocation failure.`
`order:%d, mode:0x%x`

If the order was 0, the system could not allocate you 1 single page of memory. Examine the flags for the allocation carefully, and match them up to the `GFP_` ones in `include/linux/gfp.h`. Things like `GFP_HIGH`, and not having `GFP_WAIT` and/or `GFP_IO` set are brutal on the allocator. If you do such things at a high rate then, yes, you will exhaust the system of memory. Play nice!

If it was a normal alloc (e.g. `__GFP_WAIT` | `__GFP_IO` | `__GFP_FS`), then you have no memory, and we could free no memory to satisfy your request. Your system is in deep trouble.

If the order was say 3 (or even larger) you probably have a slightly different problem. Order n means trying to allocate 2^n pages. For example, order 3 means $2^3 = 8$ pages. Worse, these cannot be any old 8 pages, but 8 physically contiguous pages, aligned on an boundary of 8 pages. Systems that have been running for a while inevitably get fragmented to the point

where large allocs are inevitably going to fail (i.e. we have lots of smaller blocks free, but none big enough for that). Possible fixes are:

- Wait for the VM to get better at dealing with fragmentation (do not hold your breath).
- See if the caller can do without physically contiguous blocks of RAM.
- Make the caller operate out of a reserved mempool

Use the printed stack trace to find the associated caller requesting the memory. CIFS, NFS, and gigabit ethernet with jumbo frames are known offenders. /proc/buddyinfo will give you more stats on fragmentation. Adding a `show_mem()` to `__alloc_pages` just after the `printk` of the failure is often helpful.

7.2 So who ate all my memory then?

There are two basic answers, either the kernel ate it, or userspace ate it. If the userspace ate it, then hopefully the OOM killer will blow them away. If it was kernel memory, we need two basic things to diagnose it, */proc/meminfo* and */proc/slabinfo*.

If your system has already hung, Alt+Sysrq+M may give you some some of the same information.

If your system has already OOM killed a bunch of stuff, then it is hard to get any accurate output. Your best bet is to reproduce it, and do something like this:

```
while true
do
  date
  cat /proc/meminfo
  cat /proc/slabinfo
  ps ef -o user,pid,rss,command
  echo ------------------------
  sleep 10
done
```

From a script, preferably running that from a remote machine and logging it, i.e.:

```
script log
ssh theserverthatkeepsbreaking
./thatstupidloggingscript
```

Examination of the logs from the time the machine got into trouble will often reveal the source of the problem.

8 Future Direction

Memory reclaim is sure to be a focus area going forward—the difference in access latencies between disk and memory make the decisions about which pages we select to reclaim critical. We are seeing ever increasing complexity at the architectural level: SMP systems are becoming increasingly large at the high end and increasingly common at the desktop, SMT (symmetric multi-threading) and multi-core CPUs are entering the market at ever lower prices. All of these place new constraints on memory in relation to memory contention and locality which has a knock on effect on memory allocation and thereby memory reclaim. There is already much work in progress looking at these issues.

Promoting Reclaimabilty: work in the allocator to try and group the reclaimable and non-reclaimable allocations with allocations of the same type at various levels. This increases the chance of finding contigious allocations and

when they are not available greatly improves the likelihood of being able to reclaim an appropriate area.

Promoting Locality: work is ongoing to better target allocations in NUMA systems when under memory pressure. On much NUMA hardware the cost of using non-local memory for long running tasks is severe both for the performance of the affected process and for the system as a whole. Promoting some reclaim for local allocations even when remote memory is available is being added.

Hotplug: hot-removal of memory requires that we be able to force reclaim the memory which is about to be removed. Work is ongoing to both increase the likelyhood of being able to reclaim the memory and how to handle the case where it cannot be reclaimed thorough remapping and relocation.

Targeted Reclaim: memory reclaim currently only comes in the form of general pressure on the memory system. The requirements of hotplug and others brings a new kind of pressure, pressure over a specific address range. Work is ongoing to see how we can apply address specific pressure both to the normal memory allocator and the slab allocators.

Active Defragmentation: as a last resort, we can re-order pages within the system in order to free up physically contiguous segments to use.

9 Conclusion

As we have shown memory reclaim is a complex subject, something of a black art. The current memory reclaim system is extremely complex, one huge heuristic guess. Moreover, it is under pressure from new requirements from big and small iron alike. NUMA architectures are moving to the desktop. Hotplug memory is becoming the norm for larger machines, and is increasingly important for virtualization. Each of these requirements brings its own issues to what already is a difficult, complex subsystem.

Block Devices and Transport Classes: Where are we going?

James E.J. Bottomley
SteelEye Technology, Inc.
jejb@steeleye.com

Abstract

A transport class is quite simply a device driver helper library with an associated sysfs component. Although this sounds deceptively simple, in practise it allows fairly large simplifications in device driver code. Up until recently, transport classes were restricted to be SCSI only but now they can be made to apply to any device driver at all (including ones with no actual transports).

Subsystems that drive sets of different devices derive the most utility from transport classes. SCSI is a really good example of this: We have a core set of APIs which are needed by every SCSI driver (whether Parallel SCSI, Fibre Channel or something even more exotic) to do command queueing and interpret status codes. However, there were a large number of ancillary services which don't apply to the whole of SCSI, like Domain Validation for Parallel SCSI or target disconnection/reconnection for Fibre Channel. Exposing parameters (like period and offset, for parallel SCSI) via `sysfs` gives the user a well known way to control them without having to develop a core SCSI API. Since a transport class has only a `sysfs` interface and a driver API it is completely independent of the SCSI core. This makes the classes arbitrarily extensible and imposes no limit on how many may be simultaneously present.

This paper will examine the evolution of the transport class in SCSI, covering its current uses in Parallel SCSI (SPI), Fibre Channel (FC) and other transports (iSCSI and SAS), contrasting it with previous approaches, like CAM, and follow with a description of how the concept was freed from the SCSI subsystem and how it could be applied in other aspects of kernel development, particularly block devices.

1 Introduction

Back in 1986, when the T10 committee first came out with the Small Computer Systems Interconnect (SCSI) protocol, it was designed to run on a single 8 bit parallel bus. A later protocol revision: SCSI-2 [1] was released in 1993 which added the ability to double the bus width and do synchronous data transfers at speeds up to 10MHz. Finally, in 1995, the next generation SCSI-3 architecture [5] was introduced. This latest standard is a constantly evolving system which includes different transports (like serial attached SCSI and Fibre Channel) and enhances the existing parallel SCSI infrastructure up to Ultra360.

2 Overview of SCSI

From its earliest days, SCSI has obeyed a command model, which means that every device attached to a SCSI controller has a command driven state mode; however, this state model tends to differ radically by device type. This means that most Operating System's SCSI subsystem implementations tend to consist of device drivers (which understand the device command model) sitting on top of a more generic command handling mechanism which understands how to send commands to devices. This split was also reflected in the first standard for operating system interfaces to SCSI: CAM [6].

2.1 SCSI CAM

The object of CAM, as the name implies was to provide a set of common access methods that would be identical across all operating systems. Looking at figure 1 one can see how the CAM infrastructure was laid out.

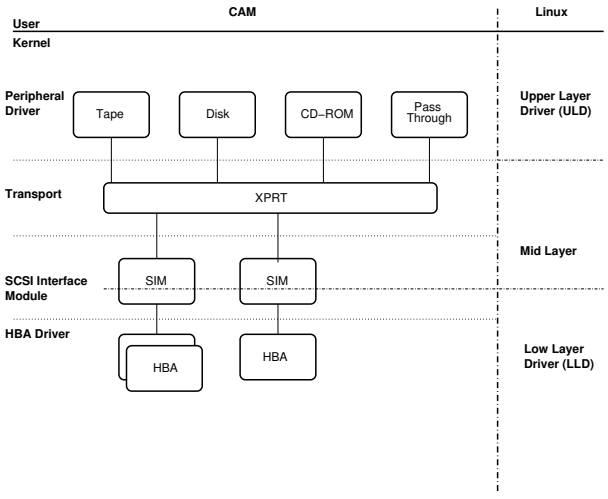


Figure 1: Illustration of CAM methods with a comparison to the current Linux SCSI subsystem

The CAM four level infrastructure on the left is shown against the current Linux three level infrastructure. The object of the comparison isn't to describe the layers in detail but to show that they map identically at the peripheral driver layer and then disconnect over the remaining ones.

Although CAM provided a good model to follow in the SCSI-2 days, it was very definitely tied to the parallel SCSI transport that SCSI-2 was based on and didn't address very well the needs of the new transport infrastructures like Fibre Channel. There was an attempt to produce a new specification taking these into account (CAM-3) but it never actually managed to produce a specification.

2.2 SCSI-3 The Next Generation

From about 1995 onwards, there was a movement to revolutionise the SCSI standard [9]. The basic thrust was a new Architecture Model (called SAM) whereby the documents were split up into Peripheral Driver command, a primary core and transport specific standards. The basic idea was to unbind SCSI from the concept of a parallel bus and make it much more extensible in terms of transport architectures.

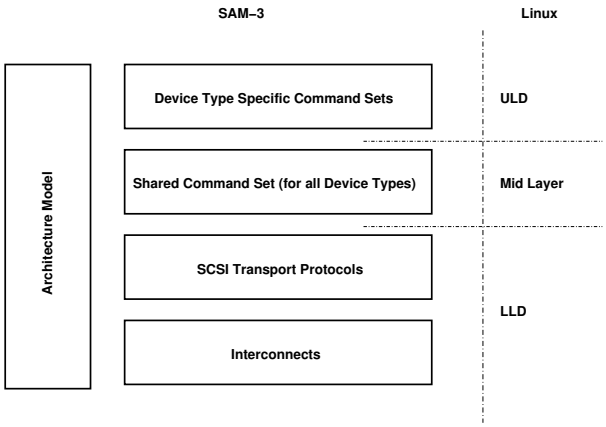


Figure 2: SAM-3 with its corresponding mapping to Linux on the right

The actual standard [8] describes the layout as depicted in figure 2 which compares almost

exactly to the layout of the Linux SCSI subsystem. Unfortunately, the picture isn't quite as rosy as this and there are certain places in the mid-layer, most notably in error handling, where we still make transport dependent assumptions.

3 Linux SCSI Subsystem

From the preceding it can be seen that the original SCSI Subsystem didn't follow either CAM or SAM exactly (although the implementation is much closer to SAM). Although the SCSI mid layer (modulo error handling) is pretty efficient now in the way it handles commands, it still lacks fine grained multi-level control of devices that CAM allows. However, in spite of this the property users most want to know about their devices (what is the maximum speed this device is communicating to the system) was lacking even from CAM.

3.1 Things Linux Learned from CAM

The basic thing CAM got right was splitting the lower layers (see figure 1) into XPRT (generic command transport) SIM (HBA specific processing) and HBA (HBA driver) was heading in the right direction. However, there were several basic faults in the design:

1. Even the XPRT which is supposed to be a generic command transport had knowledge of parallel SCSI specific parameters.
2. The User wasn't given a prescribed method for either reading or altering parameters they're interested in (like bus speed).
3. The SIM part allowed for there being one unique SIM per HBA driver.

Point 3 looks to be an advantage because it allows greater flexibility for controlling groups of HBAs according to their capabilities. However, its disadvantage is failing to prescribe precisely where the dividing line lies (i.e. since it permits one SIM per HBA, most driver writers wrote for exactly that, their own unique SIM).

A second issue for Linux is that the XPRT layer is actually split between the generic block layer and the SCSI mid-layer. Obviously, other block drivers are interested in certain features (like tags and command queueing) whereas some (like bus scanning or device identification) are clearly SCSI specific. Thus, the preferred implementation should also split the XPRT into a block generic and a SCSI specific piece, with heavy preference on keeping the SCSI specific piece as small as possible.

3.2 Recent Evolution

The policy of slimming down SCSI was first articulated at the Kernel Summit in 2002 [3] and later refined in 2003 [4]. The idea was to slim down SCSI as far as possible by moving as much of its functionality that could be held in common up to the block layer (the exemplar of this at the time being tag command queueing). and to make the mid-layer a small compact generic command processing layer with plug in helper libraries to assist the device drivers with transport and other issues. However, as is the usual course, things didn't quite go according to plan. Another infrastructure was seeping into SCSI: generic devices and `sysfs`.

3.3 `sysfs`

SCSI was the first device driver subsystem to try to embrace `sysfs` fully. This was done purely out of selfish reasons: Users were requesting extra information which we could export via `sysfs` and also, moving to the `sysfs`

infrastructure promised to greatly facilitate the Augean scale cleaning task of converting SCSI to be hotplug compliant. The way this was done was to embed a generic device into each of the SCSI device components (host, target and device) along with defining a special SCSI bus type to which the ULDs now attach as `sysfs` drivers.

However, once the initial infrastructure was in place, with extra additions that allowed drivers to export special driver specific parameters, it was noticed that certain vendor requirements were causing them to push patches into drivers that were actually exporting information that was specific to the actual transport rather than the driver [11].

Since this export of information fitted the general pattern of the "helper libraries" described above, discussion ensued about how best to achieve this export in a manner that could be utilised by all drivers acting for the given transport [12]. And thus, the concept of a Transport Class was born.

4 Transport Classes

The original concept of a transport class was that it was an entity which attached to the SCSI device at three levels (host, target and LUN) and that it exported properties from these devices straight to the user via the `sysfs` class interface. A further refinement was that the transport class (although it had support from the mid-layer) had no API that it exported to (or via) the mid layer (this is essential for allowing HBA's that aren't transport class compliant to continue to operate; however, it also has the extremely advantageous property of ensuring that the transport class services aren't bounded by any API of the mid-layer and thus makes them truly extensible). Figure 3 illustrates the relationships between transport classes and the rest of the Linux system.

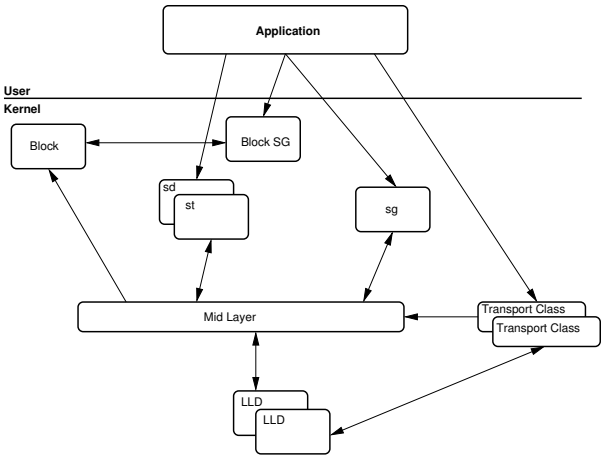


Figure 3: SCSI transport classes under Linux

4.1 Implementation

This section describes historical implementation only, so if you want to know how the classes function now[1] see section 5.3. The original implementation was designed to export transport specific parameters, so the code in the SCSI subsystem was geared around defining the class and initialising its attribute files at the correct point in the `sysfs` tree. However, once this was done, it was fairly easy to export an API from the transport class itself that could make use of these parameters (like Domain Validation for SPI, see below).

The key point was that the interaction between the mid-layer and the transport class was restricted to the mid-layer providing an API to get all the `sysfs` properties initialised and exported correctly.

[1] or rather, at the time of writing, which corresponds to the 2.6.12 kernel

4.2 Case Study: the SPI transport Class

SPI means SCSI Parallel Interface and is the new SCSI-3 terminology for the old parallel bus. In order to ascertain and control the speed of the bus, there are three essential characteristics: period, offset and width (plus a large number of minor characteristics that were added as the SPI standard evolved).

Once the ability to fetch and set these characteristics had been added, it was natural to add a domain validation [7] capability to the transport class. What domain validation (DV) does is to verify that the chosen transport characteristics match the capability of the SCSI transport by attempting to send and receive a set of prescribed patterns over the bus from the device and adjust the transport parameters if the message is garbled As the parallel bus becomes faster and faster, this sort of line clearing becomes essential since just a small kink in the cable may produce a large number of errors at the highest transfer speed.

Since the performance of Domain Validation depends on nothing more than the setting of SPI transfer parameters, it is an ideal candidate service to be performed purely within the SPI transport class. Although domain validation is most important in the high speed controllers, it is still useful to the lower speed ones. Further, certain high speed controllers themselves contain Domain Validation internally adding code bloat at best and huge potential for incorrectness at worst (the internal Domain Validation code has proved to be a significant source of bugs in certain drivers). As an illustration of the benefit, the conversion of the `aic7xxx` driver to the transport class domain validation resulted in the removal of 1,700 lines of code [2].

4.3 The Fibre Channel Transport Class

Of all the SCSI transport classes in flux at the moment, the FC class is doing the most to revolutionise the way the operating system sees the transport. Following a fairly huge program of modification, the FC transport class is able to make use of expanded mid-layer interfaces to cause even non-SCSI ports of the fabric to appear under the SCSI device tree—even the usual SCSI device structure is perturbed since the tree now appears as host/rport/target/device.

The object of this transport class is twofold:

1. Consolidate all services for Fibre Channel devices which can be held in common (things like cable pull timers, port scanning), thus slimming down the actual Fibre Channel drivers.

2. Implement a consistent API via `sysfs` which all drivers make use of, thus in theory meaning a single SAN management tool can be used regardless of underlying HBA hardware.

5 Transport Class Evolution

Looking at what's happening in the SCSI world today, it's clear that the next nascent transport to hit the Linux Kernel will be the Serial Attached SCSI (SAS) one. Its cousin, Serial ATA (SATA) is already present in both the 2.4 and 2.6 kernels. One of the interesting points about SAS and SATA is that at the lowest level, they both share the same bus and packet transport mechanism (the PHY layer, which basically represent a physical point to point connection which may be only part of a broader logical point to point connection).

The clear direction here is that SAS should have *two* separate transport classes: one for SAS itself and one for the PHY, and further that the PHY transport class (which would control the physical characteristics of the PHY interface) should be common between SAS and SATA.

5.1 Multiple Transport Classes per Device

In the old transport class paradigm, each transport class requires an "anchor" in the enveloping device structure (for SCSI we put these into `struct Scsi_Host`, `struct scsi_target`, and `struct scsi_device`). However, to attach multiple transport classes under this paradigm, we'd have to have multiple such anchors in the enveloping device which is starting to look rather inefficient.

The basic anchor that is required is a pointer to the class and also a list of attributes which appear as files in `sysfs`, so the solution is to remove the need for this anchor altogether: the generic attribute container.

5.2 Generic Attribute Containers

The idea here is to dispense entirely with the necessity for an anchor within some enveloping structure. Instead, all the necessary components and attribute files are allocated separately and then matched up to the corresponding generic device (which currently always sits inside the enveloping structure). The mechanism by which attribute containers operate is firstly by the pre-registration of a structure that contains three elements:

1. A pointer to the class,
2. a pointer to the set of class device attributes
3. and a match callback which may be coded to use subsystem specific knowledge to determine if a given generic device should have the class associated with it.

Once this is registered, a set of event triggers on the generic device must be coded into the subsystem (of necessity, some of these triggers are device creation and destruction, which are used to add and remove the container, but additional triggers of any type whatever may also be included). The benefit of these triggers is enormous: the trigger function will be called for all devices to whom the given class is registered, so this can be used, for instance, to begin device configuration. Once the generic attribute container was in place, it was extremely simple to build a generic transport class on top of it.

5.3 Generic Transport Classes

Looking at the old SCSI transport classes in the light of the new attribute containers, it was easily seen that there are five trigger points:

1. setup (mandatory), where the class device is created but not yet made visible to the system.
2. add (mandatory), where the created class device and its associated attributes are now made visible in `sysfs`
3. configure (optional), which is possibly more SCSI-centric; the above two operations (setup and add) probe the device using the lowest common transport settings. Configure means that the device has been found and identified and is now ready to be brought up to its maximum capabilities.
4. remove (mandatory), where the class device should be removed from the `sysfs` export preparatory to being destroyed.

5. destroy (mandatory), called on final last put of the device to cause the attribute container to be deallocated.

All of these apart from configure are essentially standard events that all generic devices go through. Basically then, a generic transport class is a structure containing three of the five trigger points (add, configure and remove; setup and destroy being purely internally concerned with allocation and deallocation of the transport class, with no external callback visibility). To make use of the generic transport container, all the subsystem has to do is to register the structure with the three callbacks (which is usually done in the transport class initialisation routine) and embed the mandatory trigger points into the subsystem structure creation routines as `transport_`*`event`*`_device()`.

As a demonstration of the utility of the generic transport class, the entire SCSI transport infrastructure was converted over to the generic transport class code with no loss of functionality and a significant reduction in lines of code and virtually no alteration (except for initialisations) within the three existing SCSI transport classes.

Finally, because the generic transport class is built upon the generic attribute containers, which depend only on the `sysfs` generic device, any subsystem or driver which has been converted to use generic devices may also make use of generic transport classes.

6 So Where Are We Going?

Although the creation of the generic transport classes was done for fairly selfish reasons (to get SAS to fit correctly in the transport framework with two attached classes), the potential utility of a generic transport infrastructure extends well beyond SCSI.

6.1 IDE and `hdparm`

As the ATA standards have evolved [10], the transport speed and feature support (like Tag Command Queueing) has also been evolving.

Additionally, with the addition of SATA and AoE (ATA over Ethernet), IDE is evolving in the same direction that SCSI did many years ago (acquiring additional transports), so it begins to make sense to regroup the currently monolithic IDE subsystem around a core command subsystem which interacts with multiple transports.

Currently if you want to see what the transfer settings of your drive are, you use the `hdparm` program, which manipulates those settings via special ioctls. This same information would be an ideal candidate for exporting through `sysfs` via the generic transport classes.

6.2 Hardware RAID

The kernel today has quite a plethora of hardware RAID drivers; some, like `cciss` are present in the block subsystem but the majority are actually presented to the system as SCSI devices. Almost every one of these has a slew of special ioctls for configuration, maintenance and monitoring of the arrays, and almost all of them comes with their own special packages to interface to these private ioctls. There has recently been a movement in the standards committees to unify the management approach (and even the data format) of RAID arrays, so it would appear that the time is becoming ripe for constructing a raid management transport class that would act as the interface between a generic management tool and all of the hardware RAID drivers.

6.3 SAS

As has been mentioned before, the need to have both a SAS and a PHY class for the same device was one of the driving reasons for the creation of the generic transport class. We are also hoping that SAS will be the first SCSI transport to enter the kernel with a fully fledged transport class system (both SPI and FC had their transport classes grafted on to them after drivers for each had been accepted into the kernel, and not all FC or SPI drivers currently make use of the capabilities afforded by the transport classes).

Hopefully, the vastly improved functionality provided to FC drivers by the FC transport class, with the addition of the concept of the remote port and transport class driven domain enumeration will at least have convinced the major SAS protagonists of the benefits of the approach. However, the current statement of the SCSI maintainers has been that a working SAS transport class is a necessary prerequisite for inclusion of any SAS driver.

6.4 SCSI Error Handling

One of the last major (and incredibly necessary) re-organisations of SCSI involves cleaning up the error handler. Currently, the SCSI error handler is completely monolithic (i.e. it applies to every driver) and its philosophy of operation is still deeply rooted in the old parallel bus, which makes it pretty inappropriate for a large number of modern transports. Clearly, the error handler should be transport specific, and thus it would make a natural candidate for being in a transport class. However, previously transport classes took services from the mid-layer but didn't provide any services to it (the provide services only to the LLD). However, an error handler primarily provides services to the Mid Layer and an API for handling errors to the LLD, so it doesn't quite fit in with the original vision for the SCSI transport classes. However, it does seem that it can be made to conform more closely with the generic transport class, where the error handler classes become separate from the actual "transport" transport classes.

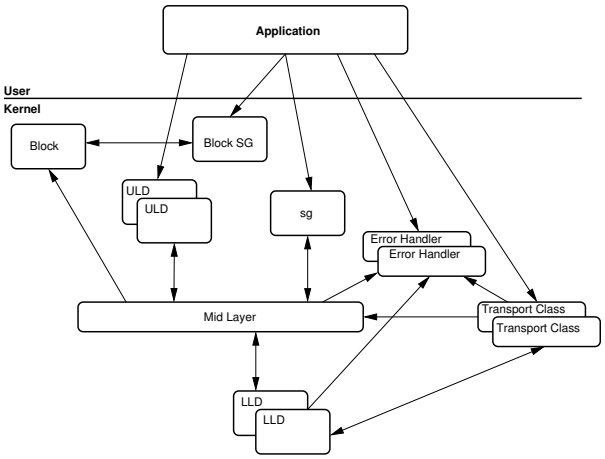


Figure 4: An illustration of how the error handlers would work as generic transport classes

How this would work is illustrated in figure 4 The arrows represent the concept of "uses the services of." The idea essentially is that the error handler classes would be built on the generic transport classes but would provide a service to the mid-layer based on a transport dependent API. The error handler parameters would, by virtue of the `sysfs` component, be accessible to the user to tweak.

7 Conclusions

The SCSI transport classes began life as helper libraries to slim down the SCSI subsystem. However, they subsequently became well defined transport class entities and went on to spawn generic transport classes which have utility far beyond the scope of the original requirement.

Two basic things remain to be done, though:

1. Retool SCSI error handling to be modular using generic transport classes.
2. Actually persuade someone outside of the SCSI subsystem to make use of them.

References

[1] Secretariat: Computer & Business Equipment Manufacturers Association. Small computer system interface - 2, 1993. http://www.t10.org/ftp/t10/drafts/s2/s2-r10l.pdf.

[2] James Bottomley. [PATCH] convert aic7xxx to the generic Domain Validation, April 2005. http://marc.theaimsgroup.com/?l=linux-scsi&m=111325605403216.

[3] James E.J. Bottomley. Fixing SCSI. USENIX Kernel Summit, July 2002. http://www.steeleye.com/support/papers/scsi_kernel_summit.pdf.

[4] James E.J. Bottomley. SCSI. USENIX Kernel Summit, July 2003. http://www.steeleye.com/support/papers/scsi_kernel_summit_2003.pdf.

[5] Secretariat: Information Technology Industry Council. SCSI-3 architecture model, 1995. http://www.t10.org/ftp/t10/drafts/sam/sam-r18.pdf.

[6] ASC X3T10 Technical Editor. SCSI-2 common access method transport and scsi interface module, 1995. http://www.t10.org/ftp/t10/drafts/cam/cam-r12b.pdf.

[7] NCITS T10 SDV Technical Editor. SCSI domain validation (SDV), 2001. http://www.t10.org/ftp/t10/drafts/sdv/sdv-r08b.pdf.

[8] T10 Technical Editor. SCSI architecture model - 3, 2004. http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf.

[9] T10 Technical Editors. Collection of SCSI-3 protocol specifications. There are too many to list individually, but they can all be found at: http://www.t10.org/drafts.htm.

[10] T13 Technical Editors. Collection of ATA protocol specifications. There are too many to list individually, but they can all be found at: http://www.t13.org.

[11] Martin Hicks. [PATCH] Transport Attributes Export API, December 1993. http://marc.theaimsgroup.com/?l=linux-scsi&m=107289789102940.

[12] Martin Hicks. Transport attributes – attempt#4, January 1994. http://marc.theaimsgroup.com/?l=linux-scsi&m=107463606609790.

ACPI in Linux

Architecture, Advances, and Challenges

Len Brown Anil Keshavamurthy David Shaohua Li

Robert Moore Venkatesh Pallipadi Luming Yu

Intel Open Source Technology Center

{len.brown, anil.s.keshavamurthy, shaohua.li}@intel.com

{robert.moore, venkatesh.pallipadi, luming.yu}@intel.com

Abstract

ACPI (Advanced Configuration and Power Interface) is an open industry specification establishing industry-standard interfaces for OS-directed configuration and power management on laptops, desktops, and servers.

ACPI enables new power management technology to evolve independently in operating systems and hardware while ensuring that they continue to work together.

This paper starts with an overview of the ACPICA architecture. Next a section describes the implementation architecture in Linux.

Later sections detail recent advances and current challenges in Linux/ACPI processor power management, CPU and memory hot-plug, legacy plug-and-play configuration, and hot-keys.

1 ACPI Component Architecture

The purpose of ACPICA, the ACPI Component Architecture, is to simplify ACPI implementations for operating system vendors (OSVs) by providing major portions of an ACPI implementation in OS-independent ACPI modules that can be integrated into any operating system.

The ACPICA software can be hosted on any operating system by writing a small and relatively simple OS Services Layer (OSL) between the ACPI subsystem and the host operating system.

The ACPICA source code is dual-licensed such that Linux can share it with other operating systems, such as FreeBSD.

1.1 ACPICA Overview

ACPICA defines and implements a group of software components that together create an implementation of the ACPI specification. A major goal of the architecture is to isolate all operating system dependencies to a relatively small translation or conversion layer (the OS Services Layer) so that the bulk of the ACPICA code is independent of any individual operating system. Therefore, hosting the ACPICA code on new operating systems requires no source code modifications within the CA code

itself. The components of the architecture include (from the top down):

- A user interface to the power management and configuration features.
- A power management and power policy component (OSPM). [1]
- A configuration management component.
- ACPI-related device drivers (for example, drivers for the Embedded Controller, SMBus, Smart Battery, and Control Method Battery).
- An ACPI Core Subsystem component that provides the fundamental ACPI services (such as the AML[2] interpreter and namespace[3] management).
- An OS Services Layer for each host operating system.

1.2 The ACPI Subsystem

The ACPI Subsystem implements the low level or fundamental aspects of the ACPI specification. It includes an AML parser/interpreter, ACPI namespace management, ACPI table and device support, and event handling. Since the ACPICA core provides low-level system services, it also requires low-level operating system services such as memory management, synchronization, scheduling, and I/O. To allow the Core Subsystem to easily interface to any operating system that provides such services, the OSL translates OS requests into the native calls provided by the host operating system.

[1]OSPM, Operating System directed Power Management.

[2]AML, ACPI Machine Language exported by the BIOS in ACPI tables, interpreted by the OS.

[3]The ACPI namespace tracks devices, objects, and methods accessed by the interpreter.

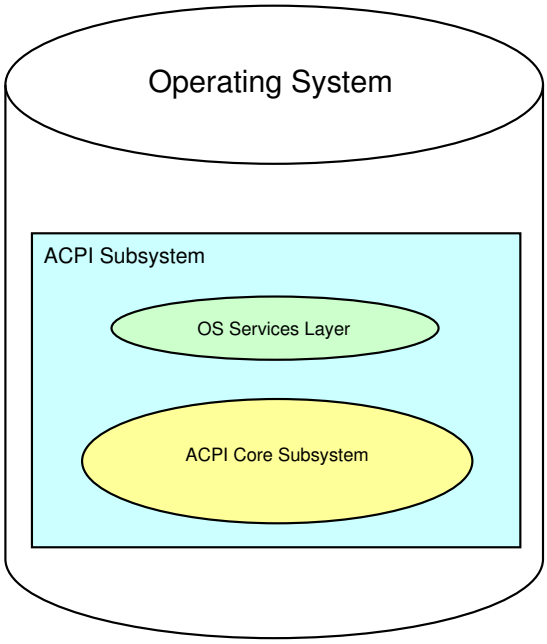


Figure 1: The ACPI Subsystem Architecture

The OS Services Layer is the only component of the ACPICA that contains code that is specific to a host operating system. Figure 1 illustrates the ACPI Subsystem is composed of the OSL and the Core.

The ACPI Core Subsystem supplies the major building blocks or subcomponents that are required for all ACPI implementations including an AML interpreter, a namespace manager, ACPI event and resource management, and ACPI hardware support.

One of the goals of the Core Subsystem is to provide an abstraction level high enough such that the host OS does not need to understand or know about the very low-level ACPI details. For example, all AML code is hidden from the OSL and host operating system. Also, the details of the ACPI hardware are abstracted to higher-level software interfaces.

The Core Subsystem implementation makes no assumptions about the host operating system or environment. The only way it can request

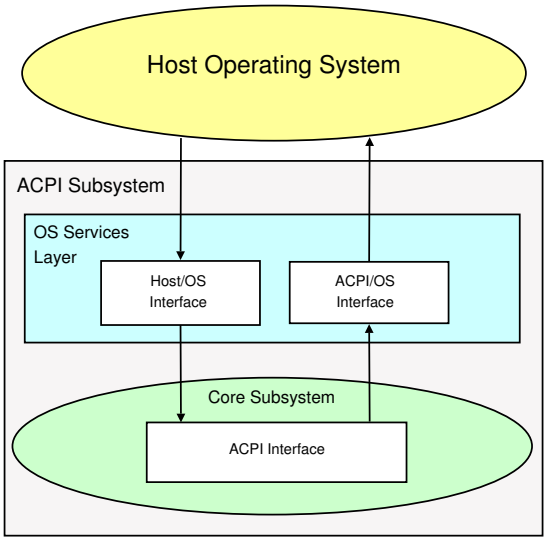


Figure 2: Interaction between the Architectural Components

operating system services is via interfaces provided by the OSL. Figure 2 shows that the OSL component "calls up" to the host operating system whenever operating system services are required, either for the OSL itself, or on behalf of the Core Subsystem component. All native calls directly to the host are confined to the OS Services Layer, allowing the core to remain OS independent.

1.3 ACPI Core Subsystem

The Core Subsystem is divided into several logical modules or sub-components. Each module implements a service or group of related services. This section describes each subcomponent and identifies the classes of external interfaces to the components, the mapping of these classes to the individual components, and the interface names. Figure 3 shows the internal modules of the ACPI Core Subsystem and their relationship to each other. The AML interpreter forms the foundation of the component, with additional services built upon this foundation.

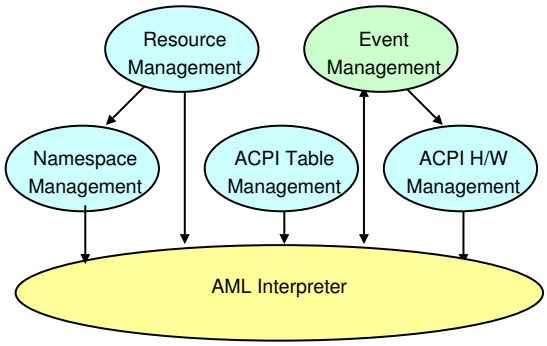


Figure 3: Internal Modules of the ACPI Core Subsystem

1.4 AML Interpreter

The AML interpreter is responsible for the parsing and execution of the AML byte code that is provided by the computer system vendor. The services that the interpreter provides include:

- AML Control Method Execution
- Evaluation of Namespace Objects

1.5 ACPI Table Management

This component manages the ACPI tables. The tables may be loaded from the firmware or directly from a buffer provided by the host operating system. Services include:

- ACPI Table Parsing
- ACPI Table Verification
- ACPI Table installation and removal

1.6 Namespace Management

The Namespace component provides ACPI namespace services on top of the AML interpreter. It builds and manages the internal ACPI namespace. Services include:

- Namespace Initialization from either the BIOS or a file
- Device Enumeration
- Namespace Access
- Access to ACPI data and tables

1.7 Resource Management

The Resource component provides resource query and configuration services on top of the Namespace manager and AML interpreter. Services include:

- Getting and Setting Current Resources
- Getting Possible Resources
- Getting IRQ Routing Tables
- Getting Power Dependencies

1.8 ACPI Hardware Management

The hardware manager controls access to the ACPI registers, timers, and other ACPI–related hardware. Services include:

- ACPI Status register and Enable register access
- ACPI Register access (generic read and write)
- Power Management Timer access
- Legacy Mode support
- Global Lock support
- Sleep Transitions support (S-states)
- Processor Power State support (C-states)
- Other hardware integration: Throttling, Processor Performance, etc.

1.9 Event Handling

The Event Handling component manages the ACPI System Control Interrupt (SCI). The single SCI multiplexes the ACPI timer, Fixed Events, and General Purpose Events (GPEs). This component also manages dispatch of notification and Address Space/Operation Region events. Services include:

- ACPI mode enable/disable
- ACPI event enable/disable
- Fixed Event Handlers (Installation, removal, and dispatch)
- General Purpose Event (GPE) Handlers (Installation, removal, and dispatch)
- Notify Handlers (Installation, removal, and dispatch)
- Address Space and Operation Region Handlers (Installation, removal, and dispatch)

2 ACPICA OS Services Layer (OSL)

The OS Services Layer component of the architecture enables the re-hosting or re-targeting of the other components to execute under different operating systems, or to even execute in environments where there is no host operating system. In other words, the OSL component provides the glue that joins the other components to a particular operating system and/or environment. The OSL implements interfaces and services using native calls to host OS. Therefore, an OS Services Layer must be written for each target operating system.

The OS Services Layer has several roles.

1. It acts as the front-end for some OS-to-ACPI requests. It translates OS requests that are received in the native OS format (such as a system call interface, an I/O request/result segment interface, or a device driver interface) into calls to Core Subsystem interfaces.

2. It exposes a set of OS-specific application interfaces. These interfaces translate application requests to calls to the ACPI interfaces.

3. The OSL component implements a standard set of interfaces that perform OS dependent functions (such as memory allocation and hardware access) on behalf of the Core Subsystem component. These interfaces are themselves OS-independent because they are constant across all OSL implementations. It is the implementations of these interfaces that are OS-dependent, because they must use the native services and interfaces of the host operating system.

2.1 Functional Service Groups

The services provided by the OS Services Layer can be categorized into several distinct groups, mostly based upon when each of the services in the group are required. There are boot time functions, device load time functions, run time functions, and asynchronous functions.

Although it is the OS Services Layer that exposes these services to the rest of the operating system, it is very important to note that the OS Services Layer makes use of the services of the lower-level ACPI Core Subsystem to implement its services.

2.1.1 OS Boot-load-Time Services

Boot services are those functions that must be executed very early in the OS load process, before most of the rest of the OS initializes. These services include the ACPI subsystem initialization, ACPI hardware initialization, and execution of the _INI control methods for various devices within the ACPI namespace.

2.1.2 Device Driver Load-Time Services

For the devices that appear in the ACPI namespace, the operating system must have a mechanism to detect them and load device drivers for them. The Device driver load services provide this mechanism. The ACPI subsystem provides services to assist with device and bus enumeration, resource detection, and setting device resources.

2.1.3 OS Run-Time Services

The runtime services include most if not all of the external interfaces to the ACPI subsystem. These services also include event logging and power management functions.

2.1.4 Asynchronous Services

The asynchronous functions include interrupt servicing (System Control Interrupt), Event handling and dispatch (Fixed events, General Purpose Events, Notification events, and Operation Region access events), and error handling.

2.2 OSL Required Functionality

There are three basic functions of the OS Services Layer:

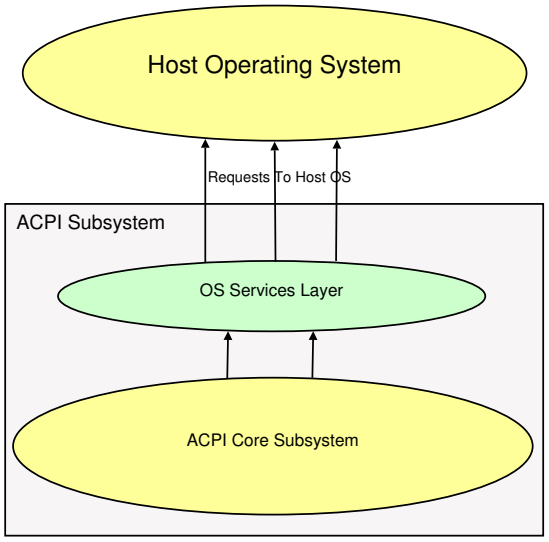


Figure 4: ACPI Subsystem to Operating System Request Flow

1. Manage the initialization of the entire ACPI subsystem, including both the OSL and ACPI Core Subsystems.

2. Translate requests for ACPI services from the host operating system (and its applications) into calls to the Core Subsystem component. This is not necessarily a one-to-one mapping. Very often, a single operating system request may be translated into many calls into the ACPI Core Subsystem.

3. Implement an interface layer that the Core Subsystem component uses to obtain operating system services. These standard interfaces (referred to in this document as the ACPI OS interfaces) include functions such as memory management and thread scheduling, and must be implemented using the available services of the host operating system.

2.2.1 Requests from ACPI Subsystem to OS

The ACPI subsystem requests OS services via the OSL shown in Figure 4. These requests must be serviced (and therefore implemented) in a manner that is appropriate to the host operating system. These requests include calls for OS dependent functions such as I/O, resource allocation, error logging, and user interaction. The ACPI Component Architecture defines interfaces to the OS Services Layer for this purpose. These interfaces are constant (i.e., they are OS-independent), but they must be implemented uniquely for each target OS.

2.3 ACPICA—more details

The ACPICA APIs are documented in detail in the *ACPICA Component Architecture Programmer Reference* available on `http://www.intel.com`.

The ACPI header files in `linux/include/acpi/` can also be used as a reference, as can the ACPICA source code in the directories under `linux/drivers/acpi/`.

3 ACPI in Linux

The ACPI specification describes platform registers, ACPI tables, and operation of the ACPI BIOS. It also specifies AML (ACPI Machine Language), which the BIOS exports via ACPI tables to abstract the hardware. AML is executed by an interpreter in the ACPI OS.[4]

In some cases the ACPI specification describes the sequence of operations required by the

[4] ACPI OS: an ACPI-enabled OS, such as Linux.

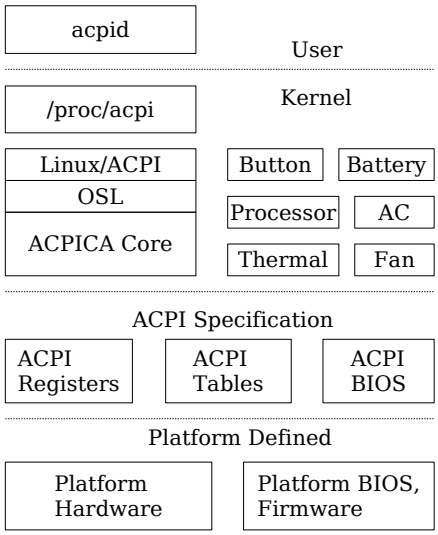


Figure 5: Implementation Architecture

ACPI OS—but generally the OS implementation is left as an exercise to the reader.

There is no platform ACPI compliance test to assure that platforms and platform BIOS' are compliant to the ACPI specification. System manufacturers assume compliance when available ACPI-enabled operating systems boot and function properly on their systems.

Figure 5 shows these ACPI components logically as a layer above the platform specific hardware and firmware.

The ACPI kernel support centers around the ACPICA core. ACPICA implements the AML interpreter as well as other OS-agnostic parts of the ACPI specification. The ACPICA code does not implement any policy, that is the realm of the Linux-specific code. A single file, `osl.c`, glues ACPICA to the Linux-specific functions it requires.

The box in Figure 5 labeled "Linux/ACPI" represents the Linux-specific ACPI code, including boot-time configuration.

Optional "ACPI drivers," such as Button, Battery, Processor, etc. are (optionally loadable) modules that implement policy related to those specific features and devices.

There are about 200 ACPI-related files in the Linux kernel source tree—about 130 of them are from ACPICA, and the rest are specific to Linux.

4 Processor Power management

Processor power management is a key ingredient in system power management. Managing processor speed and voltage based on utilization is effective in increasing battery life on laptops, reducing fan noise on desktops, and lowing power and cooling costs on servers. This section covers recent and upcoming Linux changes related to Processor Power Management.

4.1 Overview of Processor Power States

But first refer to Figure 6 for this overview of processor power management states.

1. G0—System Working State. Processor power management states have meaning only in the context of a running system—not when the system is in one of its various sleep or off-states.

2. Processor C-state: C0 is the executing CPU power state. C1–Cn are idle CPU power states used by the Linux idle loop; no instructions are executed in C1–Cn. The deeper the C-state, the more power is saved, but at the cost of higher latency to enter and exit the C-state.

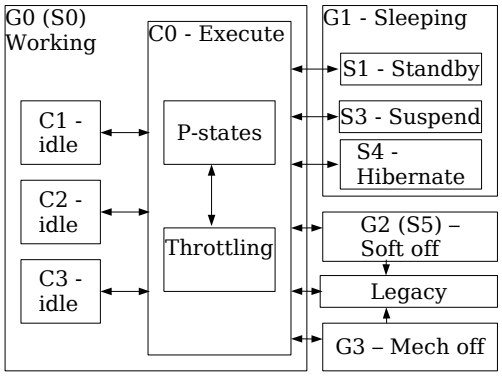


Figure 6: ACPI Global, CPU, and Sleep states

3. Processor P-state: Performance states consist of states representing different processor frequencies and voltages. This provides an opportunity to OS to dynamically change the CPU frequency to match the CPU workload.

   As power varies with the square of voltage, the voltage-lowing aspect of p-states is extremely effective at saving power.

4. Processor T-state: Throttle states change the processor frequency only, leaving the voltage unchanged.

   As power varies directly with frequency, T-states are less effective than P-states for saving processor power. On a system with both P-states and T-states, Linux uses T-states only for thermal (emergency) throttling.

4.2 Processor Power Saving Example

Table 1 illustrates that high-volume hardware offers dramatic power saving opportunities to the OS through these mechanisms.[5] Note that these numbers reflect processor power, and do not include other system components, such as the LCD chip-set, or disk drive. Note also that on this particular model, the savings in the C1, C2, and C3 states depend on the P-state the processor was running in when it became idle. This is because the P-states carry with them reduced voltage.

C-State	P-State	MHz	Volts	Watts
C0	P0	1600	1.484	24.5
	P1	1300	1.388	22
	P2	1100	1.180	12
	P3	600	0.956	6
C1, C2	from P0	0	1.484	7.3
	from P3	0	0.956	1.8
C3	from P0	0	1.484	5.1
	from P3	0	0.956	1.1
C4	(any)	0	0.748	0.55

Table 1: C-State and P-State Processor Power

4.3 Recent Changes

4.3.1 P-state driver

The Linux kernel cpufreq infrastructure has evolved a lot in past few years, becoming a modular interface which can connect various vendor specific CPU frequency changing drivers and CPU frequency governors which handle the policy part of CPU frequency changes. Recently different vendors have different technologies, that change the

[5] Ref: 1600MHz Pentium M processor Data-sheet.

CPU frequency and the CPU voltage, bringing with it much higher power savings than simple frequency changes used to bring before. This combined with reduced CPU frequency-changing latency (10uS–100uS) provides a opportunity for Linux to do more aggressive power savings by doing a frequent CPU frequency change and monitoring the CPU utilization closely.

The P-state feature which was common in laptops is now becoming common on servers as well. `acpi-cpufreq` and `speedstep-centrino` drivers have been changed to support SMP systems. These drivers can run with i386 and x86-64 kernel on processors supporting Enhanced Intel Speedstep Technology.

4.3.2 Ondemand governor

One of the major advantages that recent CPU frequency changing technologies (like Enhanced Intel SpeedStep Technology) brings is lower latency associated with P-state changes of the order of 10mS. In order to reap maximum benefit, Linux must perform more-frequent P-state transitions to match the current processor utilization. Doing frequent transitions with a user-level daemon will involve more kernel-to-user transitions, as well as a substantial amount of kernel-to-user data transfer. An in-kernel P-state governor, which dynamically monitors the CPU usage and makes P-state decisions based on that information, takes full advantage of low-latency P-state transitions. The ondemand policy governor is one such in-kernel P-state governor. The basic algorithm employed with the ondemand (as in Linux-2.6.11) governor is as follows:

```
Every X milliseconds
  Get the current CPU utilization
  If (utilization > UP_THRESHOLD %)
    Increase the P-state
    to the maximum frequency

Every Y milliseconds
  Get the current CPU utilization
  If (utilization < DOWN_THRESHOLD %)
    Decrease P-state
    to next available lower frequency
```

The ondemand governor, when supported by the kernel, will be listed in the `/sys` interface under `scaling_available_governors`. Users can start using the ondemand governor as the P-state policy governor by writing onto `scaling_governor`:

```
# cat scaling_available_governors
ondemand user-space performance
# echo ondemand > scaling_governor
# cat scaling_governor
ondemand
```

This sequence must be repeated on all the CPUs present in the system. Once this is done, the ondemand governor will take care of adjusting the CPU frequency automatically, based on the current CPU usage. CPU usage is based on the `idle_ticks` statistics. Note: On systems that do not support low latency P-state transitions, `scaling_governor` will not change to "ondemand" above. A single policy governor cannot satisfy all of the needs of applications in various usage scenarios, the ondemand governor supports a number of tuning parameters. More details about this can be found on Intel's web site.[6]

4.3.3 cpufreq stats

Another addition to cpufreq infrastructure is the cpufreq stats interface. This interface

[6] Enhanced Intel Speedstep Technology for the Pentium M Processor.

appears in `/sys/devices/system/cpu/cpuX/cpufreq/stats`, whenever cpufreq is active. This interface provides the statistics about frequency of a particular CPU over time. It provides

- Total number of P-state transitions on this CPU.
- Amount of time (in jiffies) spent in each P-state on this CPU.
- And a two-dimensional (n x n) matrix with value count(i,j) indicating the number of transitions from Pi to Pj.
- A `top`-like tool can be built over this interface to show the system wide P-state statistics.

4.3.4 C-states and SMP

Deeper C-states (C2 and higher) are mostly used on laptops. And in today's kernel, C-states are only supported on UP systems. But, soon laptop CPUs will be becoming Dual-Core. That means we need to support C2 and higher states on SMP systems as well. Support for C2 and above on SMP is in the base kernel now ready for future generation of processors and platforms.

4.4 Upcoming Changes

4.4.1 C4, C5, . . .

In future, one can expect more deeper C states with higher latencies. But, with Linux kernel jiffies running at 1mS, CPU may not stay long enough in a C-state to justify entering C4, C5 states. This is where we can use the existing variable HZ solution and can make use of more deeper C-states. The idea is to reduce the rate of timer interrupts (and local APIC interrupts) when the CPU is idle. That way a CPU can stay in a low power idle state longer when they are idle.

4.4.2 ACPI 3.0 based Software coordination for P and C states

ACPI 3.0 supports having P-state and C-state domains defined across CPUs. A domain will include all the CPUs that share P-state and/or C-state. Using these information from ACPI and doing the software coordination of P-states and C-states across their domains, OS can have much more control over the actual P-states and C-states and optimize the policies on systems running with different configuration.

Consider for example a 2-CPU package system, with 2 cores on each CPU. Assume the two cores on the same package share the P-states (means both cores in the same package change the frequency at the same time). If OS has this information, then if there are only 2 threads running, OS, can schedule them on different cores of same package and move the other package to lower P-state thereby saving power without loosing significant performance.

This is a work in progress, to support software coordination of P-states and C-states, whenever CPUs share the corresponding P and C states.

5 ACPI support for CPU and Memory Hot-Plug

Platforms supporting physical hot-add and hot remove of CPU/Memory devices are entering the systems market. This section covers a variety of recent changes that went into kernel

specifically to enable ACPI based platform to support the CPU and Memory hot-plug technology.

5.1 ACPI-based Hot-Plug Introduction

The hot-plug implementation can be viewed as two blocks, one implementing the ACPI specific portion of the hot-plug and the other non ACPI specific portion of the hot-plug.

The non-ACPI specific portion of CPU/Memory hot-plug, which is being actively worked by the Linux community, supports what is know as Logical hot-plug. Logical hot-plug is just removing or adding the device from the operating system perspective, but physically the device still stays in the system. In the CPU or Memory case, the device can be made to disappear or appear from the OS perspective by echoing either 0 or 1 to the respective online file. Refer to respective hot-plug paper to learn more about the logical online/off-lining support of these devices. The ACPI specific portion of the hot-plug is what bridges the gap between the platforms having the physical hot-plug capability to take advantage of the logical hot-plug in the kernel to provide true physical hot-plug. ACPI is not involved in the logical part of on-lining or off-lining the device.

5.2 ACPI Hot-Plug Architecture

At the module init time we search the ACPI device namespace. We register a system notify handler callback on each of the interested devices. In case of CPU hot-plug support we look for `ACPI_TYPE_PROCESSOR_DEVICE` and in case of Memory hot-plug support we look for PNP0C80 HID[7] and in case of container[8] we look for ACPI004 or PNP0A06 or PNP0A05 devices.

When a device is hot-plugged, the core chipset or the platform raises the SCI,[9] the SCI handler within the ACPI core clears the GPE event and runs _Lxx[10] method associated with the GPE. This _Lxx method in turn executes Notify(XXXX, 0) and notifies the ACPI core, which in turn notifies the hot-plug modules callback which was registered during the module init time.

When the module gets notified, the module notify callback handler looks for the event code and takes appropriate action based on the event. See the module Design section for more details.

5.3 ACPI Hot-Plug support Changes

The following enhancements were made to support physical Memory and/or CPU device hot-plug.

- A new `acpi_memhotplug.c` module was introduced into the drives/acpi directory for memory hot-plug.
- The existing ACPI processor driver was enhanced to support the ACPI hot-plug notification for the physical insertion/removal of the processor.
- A new container module was introduced to support hot-plug notification on a ACPI

[7] HID, Hardware ID.

[8] A container device captures hardware dependencies, such as a Processor and Memory sharing a single removable board.

[9] SCI, ACPI's System Control Interrupt, appears as "acpi" in `/proc/interrupts`.

[10] _Lxx - L stands for level-sensitive, xx is the GPE number, e.g. GPE 42 would use _L42 handler.

container device. The ACPI container device can contain multiple devices, including another container device.

5.4 Memory module

A new `acpi_memhotplug.c` driver was introduced which adds support for the ACPI based Memory hot-plug. This driver provides support for fielding notifications on ACPI memory device (PNP0C80) which represents memory ranges that may be hot-added or hot removed during run time. This driver is enabled by enabling `CONFIG_ACPI_HOTPLUG_MEMORY` in the con fig file and is required on ACPI platforms supporting physical Memory hot plug of the Memory DIMMs (at some platform granularity).

Design: The memory hot-plug module's device notify callback gets called when the memory device is hot-plug plugged. This handler checks for the event code and for hot-add case, first checks the device for physical presence and reads the memory range reported by the _CRS method and tells the VM about the new device. The VM which resides outside of ACPI is responsible for actual addition of this range to the running kernel. The ACPI memory hotplug module does not yet implement the hot-remove case.

5.5 Processor module

The ACPI processor module can now support physical CPU hot-plug by enabling `CONFIG_ACPI_HOTPLUG_CPU` under `CONFIG_ACPI_PROCESSOR`.

Design: The processor hot-plug module's device notify callback gets called when the processor device is hot plugged. This handler checks for the event code and for the hot-add case, it first creates the ACPI device by calling `acpi_bus_add()` and `acpi_bus_scan()` and then notifies the user mode agent by invoking `kobject_hotplug()` using the kobj of the ACPI device that got hot-plugged. The user mode agent in turn on-lines the corresponding CPU devices by echoing on to the online file. The `acpi_bus_add()` would invoke the `.add` method of the processor module which in turn sets up the `apic_id` to `logical_id` required for logical online.

For the remove case, the notify callback handler in turn notifies the event to the user mode agent by invoking `kobject_hotplug()` using the kobj of the ACPI device that got hot-plugged. The user mode first off-lines the device and then echoes 1 on to the eject file under the corresponding ACPI namespace device file to remove the device physically. This action leads to call into the kernel mode routine called `acpi_bus_trim()` which in turn calls the `.remove` method of the processor driver which will tear the ACPI id to logical id mappings and releases the ACPI device.

5.6 Container module

ACPI defines a Container device with the HID being ACPI004 or PNP0A06 or PNP0A05. This device can in turn contain other devices. For example, a container device can contain multiple CPU devices and/or multiple Memory devices. On a platform which supports hotplug notify on Container device, this driver needs to be enabled in addition to the above device specific hotplug drivers. This driver is enabled by enabling `CONFIG_ACPI_CONTAINER` in the config file.

Design: The module init is pretty much the same as the other driver where in we register for the system notify callback on to every

container device with in the ACPI root namespace scope. The `container_notify_cb()` gets called when the container device is hot-plugged. For the hot-add case it first creates an ACPI device by calling `acpi_bus_add()` and `acpi_bus_scan()`. The `acpi_bus_scan()` which is a recursive call in turns calls the `.add` method of the respective hotplug devices. When the `acpi_bus_scan()` returns the container driver notifies the user mode agent by invoking `kobject_hotplug()` using kobj of the container device. The user mode agent is responsible to bring the devices to online by echoing on to the online file of each of those devices.

5.7 Future work

ACPI-based, NUMA-node hotplug support (although there are a little here and there patches to support this feature from different hardware vendors). Memory hot-remove support and handling physical hot-add of memory devices. This should be done in a manner consistent with the CPU hotplug—first kernel mode does setup and notifies user mode, then user mode brings the device on-line.

6 PNPACPI

The ACPI specification replaces the PnP BIOS specification. As of this year, on a platform that supports both specifications, the Linux PNP ACPI code supersedes the Linux PNP BIOS code. The ACPI compatible BIOS defines all PNP devices in its ACPI DSDT. [11] Every ACPI PNP device defines a PNP ID, so the OS can enumerate this kind of device through the PNP

[11] DSDT, Differentiated System Description Table, the primary ACPI table containing AML

```
pnpacpi_get_resources()
  pnpacpi_parse_allocated_resource() /* _CRS */

pnpacpi_disable_resources()
  acpi_evaluate_object (_DIS)        /* _DIS */

pnpacpi_set_resources()
  pnpacpi_build_resource_template()  /* _CRS */
  pnpacpi_encode_resources()         /* _PRS */
  acpi_set_current_resources()       /* _SRS */
```

Figure 7: ACPI PNP protocol callback routines

ID. ACPI PNP devices also define some methods for the OS to manipulate device resources. These methods include _CRS (return current resources), _PRS (return all possible resources) and _SRS (set resources).

The generic Linux PNP layer abstracts PNPBIOS and ISAPNP, and some drivers use the interface. A natural thought to add ACPI PNP support is to provide a PNPBIOS-like driver to hook ACPI with PNP layer, which is what the current PNPACPI driver does. Figure 7 shows three callback routines required for PNP layer and their implementation overview. In this way, existing PNP drivers transparently support ACPI PNP. Currently there are still some systems whose PNPACPI does not work, such as the ES7000 system. Boot option `pnpacpi=off` can disable PNPACPI.

Compared with PNPBIOS, PNPACPI does not need to call into 16-bit BIOS. Rather it directly utilizes the ACPICA APIs, so it is faster and more OS friendly. Furthermore, PNPACPI works even under IA64. In the past on IA64, ACPI-specific drivers such as 8250_acpi driver were written. But since ACPI PNP works on all platforms with ACPI enabled, existing PNP drivers can work under IA64 now, and so the specific ACPI drivers can be removed. We did not remove all the drivers yet for the reason of stabilization (PNPACPI driver must be widely tested)

Another advantage of ACPI PNP is that it sup-

ports device hotplug. A PNP device can define some methods (_DIS, _STA, _EJ0) to support hotplug. The OS evaluates a device's _STA method to determine the device's status. Every time the device's status changes, the device will receive a notification. Then the OS registered device notification handler can hot add/remove the device. An example of PNP hotplug is a docking station, which generally includes some PNP devices and/or PCI devices.

In the initial implementation of ACPI PNP, we register a default ACPI driver for all PNP devices, and the driver will hook the ACPI PNP device to PNP layer. With this implementation, adding an ACPI PNP device will automatically put the PNP device into Linux PNP layer, so the driver is hot-pluggable. Unfortunately, the feature conflicted with some specific ACPI drivers (such as 8250_acpi), so we removed it. We will reintroduce the feature after the specific ACPI drivers are removed.

7 Hot-Keys

Keys and buttons on ACPI-enabled systems come in three flavors:

1. Keyboard keys, handled by the keyboard driver/Linux input sub-system/X-window system. Some platforms add additional keys to the keyboard hardware, and the input sub-system needs to be augmented to understand them through utilities to map scan-codes to characters, or though model-specific keyboard drivers.

2. Power, Sleep, and Lid buttons. These three buttons are fully described by the ACPI specification. The kernel's ACPI button.c driver sends these events to user-space via `/proc/acpi/event`. A user-space utility such as `acpid(8)` is responsible for deciding what to do with them. Typically shutdown is invoked on power button events, and suspend is invoked for sleep or lid button events.

3. The "other" keys are generally called "hot-keys," and have icons on them describing various functions such as display output switching, LCD brightness control, WiFi radio, audio volume control etc.

Hot-keys may be implemented in a variety of ways, even within the same platform.

- Full BIOS control: Here hot-keys trigger an SMI, and the SMM BIOS[12] will handle everything. Using this method, the hot-key is invisible to the kernel—to the OS they are effectively done "in hardware."

  The advantage is that the buttons will do their functions successfully, even in the presence of an ignorant or broken OS.

  The disadvantage is that the OS is completely un-aware that these functions are occurring and thus has no opportunity to optimize its policies. Also, as the SMI/SMM is shipped by the OEM in the BIOS, users are unable to either fix it when it is broken, or customize it in any way.

  Some systems include this SMI-based hot-key mechanism, but disable it when an ACPI-enabled OS boots and puts the system into ACPI-mode.

- Self-contained AML methods: from a user's—even a kernel programmer's—point of view, method is analogous to the full-BIOS control method above. The OS is is un-aware that the button is pressed and what the button does. However, the OS actually supplies the mechanics for

[12] SMI, System Management Interrupt; SMM, System Management Mode—an interrupt that sends the processor directly into BIOS firmware.

this kind of button to work, It would not work if the OS's interrupts and ACPI AML interpreter were not available.

Here a GPE[13] causes an ACPI interrupt. The ACPI sub-system responds to the interrupt, decodes which GPE caused it, and vectors to the associated BIOS-supplied GPE handler (_Lxx/_Exx/_Qxx). The handler is supplied by the BIOS in AML, and the kernel's AML interpreter make it run, but the OS is not informed about what the handler does. The handler in this scenario is hard-coded to tickle whatever hardware is necessary to to implement the button's function.

- Event based: This is a platform-specific method. Each hot-key event triggers a corresponding hot-key event from `/proc/acpi/event` to notify user space daemon, such as `acpid(8)`. Then, acpid must execute corresponding AML methods for hot-key function.
- Polling based: Another non-standard implementation. Each hot-key pressing will trigger a polling event from `/proc/acpi/event` to notify user space daemon acpid to query the hot-key status. Then acpid should call related AML methods.

Today there are several platform specific "ACPI" drivers in the kernel tree such as `asus_acpi.c`, `ibm_acpi.c`, and `toshiba_acpi.c`, and there are even more of this group out-of-tree. The problem with these drivers is that they work only for the platforms they're designed for. If you don't have that platform, it doesn't help you. Also, the different drivers perform largely the same functions.

There are many different platform vendors, and so producing and supporting a platform-specific driver for every possible vendor is not a good strategy. So this year several efforts have been made to unify some of this code, with the goal that the kernel contain less code that works on more platforms.

[13] GPE, General Purpose Event.

7.1 ACPI Video Control Driver

The ACPI specification includes an appendix describing ACPI Extensions for Display Adapters. This year, Bruno Ducrot created the initial `acpi/video.c` driver to implement it.

This driver registers notify handlers on the ACPI video device to handle events. It also exports files in `/proc` for manual control.

The notify handlers in the video driver are sufficient on many machines to make the display control hot-keys work. This is because the AML GPE handlers associated with these buttons simply issue a Notify() event on the display device, and if the `video.c` driver is loaded and registered on that device, it receives the event and invokes the AML methods associated with the request via the ACPI interpreter.

7.2 Generic Hot-Key Driver

More recently, Luming Yu created a generic hot-key driver with the goal to factor the common code out of the platform-specific drivers. This driver is intended to support two non-standard hot-key implementations—event-based and polling-based.

The idea is that configurable interfaces can be used to register mappings between event number and GPEs associated with hot-keys, and mappings between event number and AML methods, then we don't need the platform-specific drivers.

Here the user-space daemon, acpid, needs to issue a request to an interface for the execution of those AML methods, upon receiving a specific hot-key GPE. So, the generic hot-key driver implements the following interfaces to meet the requirements of non-standard hot-key.

- Event based configure interface, `/proc/acpi/hotkey/event_config`.
  - Register mappings of event number to hot-key GPE.
  - Register ACPI handle to install notify handler for hot-key GPE.
  - Register AML methods associated with hot-key GPE.
- Polling based configure interface, `/proc/acpi/hotkey/poll_config`.
  - Register mappings of event number to hot-key polling GPE.
  - Register ACPI handle to install notify handler for hot-key polling GPE.
  - Register AML methods associated with polling GPE
  - Register AML methods associated with hot-key event.
- Action interface, `/proc/acpi/hotkey/action`.
  - Once acpid knows which event is triggered, it can issue a request to the action interface with arguments to call corresponding AML methods.
  - For polling based hot-key, once acpid knows the polling event triggered, it can issue a request to the action interface to call polling method, then it can get hot-key event number according to the results from polling methods. Then, acpid can issue another request to action interface to invoke right AML methods for that hot-key function.

The current usage model for this driver requires some hacking—okay for programmers, but not okay for distributors. Before using the generic hot-key driver for a specific platform, you need to figure out how vendor implemented hot-key for it. If it just belongs to the first two standard classes, the generic hot-key driver is useless. Because, the hot-key function can work without any hot-key driver including this generic one. Otherwise, you need to flow these steps.

- Disassemble DSDT.
- Figure out the AML method of hot-key initialization.
- Observing `/proc/acpi/event` to find out the corresponding GPE associated with each hot-key.
- Figure out the specific AML methods associated with each hot-key GPE.
- After collecting sufficient information, you can configure them through interfaces of `event_config`, `poll_config`.
- Adjust scripts for acpid to issue right command to action interface.

The hope is that this code will evolve into something that consolidates, or at least mitigates a potential explosion in platform-specific drivers. But to reach that goal, it will need to be supportable without the complicated administrator incantations that it requires today. The current thinking is that the additions of a quirks table for configuration may take this driver from prototype to something that “just works” on many platforms.

8 Acknowledgments

It has been a busy and productive year on Linux/ACPI. This progress would not have been possible without the efforts of the many developers and testers in the open source community. Thank you all for your efforts and your support—keep up the good work!

9 References

Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba *Advanced Configuration & Power Specification*, Revision 3.0, September 2, 2004. `http://www.acpi.info`

ACPICA Component Architecture Programmer Reference, Intel Corporation.

Linux/ACPI Project Home page: `http://acpi.sourceforge.net`

State of the Art: Where we are with the Ext3 filesystem

Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya
IBM Linux Technology Center
{cmm, theotso, pbadari}@us.ibm.com, suparna@in.ibm.com
Andreas Dilger, Alex Tomas,
Cluster Filesystem Inc.
adilger@clusterfs.com, alex@clusterfs.com

Abstract

The ext2 and ext3 filesystems on Linux® are used by a very large number of users. This is due to its reputation of dependability, robustness, backwards and forwards compatibility, rather than that of being the state of the art in filesystem technology. Over the last few years, however, there has been a significant amount of development effort towards making ext3 an outstanding filesystem, while retaining these crucial advantages. In this paper, we discuss those features that have been accepted in the mainline Linux 2.6 kernel, including directory indexing, block reservation, and online resizing. We also discuss those features that have been implemented but are yet to be incorporated into the mainline kernel: extent maps, delayed allocation, and multiple block allocation. We will then examine the performance improvements from Linux 2.4 ext3 filesystem to Linux 2.6 ext3 filesystem using industry-standard benchmarks features. Finally, we will touch upon some potential future work which is still under discussion by the ext2/3 developers.

1 Introduction

Although the ext2 filesystem[4] was not the first filesystem used by Linux and while other filesystems have attempted to lay claim to being the native Linux filesystem (for example, when Frank Xia attempted to rename xiafs to linuxfs), nevertheless most would consider the ext2/3 filesystem as most deserving of this distinction. Why is this? Why have so many system administrations and users put their trust in the ext2/3 filesystem?

There are many possible explanations, including the fact that the filesystem has a large and diverse developer community. However, in our opinion, robustness (even in the face of hardware-induced corruption) and backwards compatibility are among the most important reasons why the ext2/3 filesystem has a large and loyal user community. Many filesystems have the unfortunate attribute of being *fragile*. That is, the corruption of a single, unlucky, block can be magnified to cause a loss of far larger amounts of data than might be expected. A fundamental design principle of the ext2/3 filesystem is to avoid fragile data structures by limiting the damage that could be caused by the loss of a single critical block.

This has sometimes led to the ext2/3 filesystem's reputation of being a little boring, and perhaps not the fastest or the most scalable filesystem on the block, but which is one of the most dependable. Part of this reputation can be attributed to the extremely conservative design of the ext2 filesystem [4], which had been extended to add journaling support in 1998, but which otherwise had very few other modern filesystem features. Despite its age, ext3 is actually growing in popularity among enterprise users/vendors because of its robustness, good recoverability, and expansion characteristics. The fact that `e2fsck` is able to recover from very severe data corruption scenarios is also very important to ext3's success.

However, in the last few years, the ext2/3 development community has been working hard to demolish the first part of this common wisdom. The initial outline of plans to "modernize" the ext2/3 filesystem was documented in a 2002 Freenix Paper [15]. Three years later, it is time to revisit those plans, see what has been accomplished, what still remains to be done, and what further extensions are now under consideration by the ext 2/3 development community.

This paper is organized into the following sections. First, we describe about those features which have already been implemented and which have been integrated into the mainline kernel in Section 2. Second, we discuss those features which have been implemented, but which have not yet been integrated in mainline in Section 3 and Section 4. Next, we examine the performance improvements on ext3 filesystem during the last few years in Section 5. Finally, we will discuss some potential future work in Section 6.

2 Features found in Linux 2.6

The past three years have seen many discussions of ext2/3 development. Some of the planned features [15] have been implemented and integrated into the mainline kernel during these three years, including directory indexing, reservation based block allocation, online resizing, extended attributes, large inode support, and extended attributes in large inode. In this section, we will give an overview of the design and the implementation for each feature.

2.1 Directory indexing

Historically, ext2/3 directories have used a simple linked list, much like the BSD Fast Filesystem. While it might be expected that the O(n) lookup times would be a significant performance issue, the Linux VFS-level directory cache mitigated the O(n) lookup times for many common workloads. However, ext2's linear directory structure did cause significant performance problems for certain applications, such as web caches and mail systems using the Maildir format.

To address this problem, various ext2 developers, including Daniel Phillips, Theodore Ts'o, and Stephen Tweedie, discussed using a B-tree data structure for directories. However, standard B-trees had numerous characteristics that were at odds with the ext2 design philosophy of simplicity and robustness. For example, XFS's B-tree implementation was larger than all of ext2 or ext3's source files combined. In addition, users of other filesystems using B-trees had reported significantly increased potential for data loss caused by the corruption of a high-level node in the filesystem's B-tree.

To address these concerns, we designed a radically simplified tree structure that was specifically optimized for filesystem directories[10].

This is in contrast to the approach used by many other filesystems, including JFS, Reiserfs, XFS, and HFS, which use a general-purpose B-tree. Ext2's scheme, which we dubbed "HTree," uses 32-bit hashes for keys, where each hash key references a range of entries stored in a leaf block. Since internal nodes are only 8 bytes, HTrees have a very high fanout factor (over 500 blocks can be referenced using a 4K index block), two levels of index nodes are sufficient to support over 16 million 52-character filenames. To further simplify the implementation, HTrees are constant depth (either one or two levels). The combination of the high fanout factor and the use of a hash of the filename, plus a filesystem-specific secret to serve as the search key for the HTree, avoids the need for the implementation to do balancing operations.

We maintain forwards compatibility in old kernels by clearing the `EXT3_INDEX_FL` whenever we modify a directory entry. In order to preserve backwards compatibility, leaf blocks in HTree are identical to old-style linear directory blocks, and index blocks are prefixed with an 8-byte data structure that makes them appear to non-HTree kernels as deleted directory entries. An additional advantage of this extremely aggressive attention towards backwards compatibility is that HTree directories are extremely robust. If any of the index nodes are corrupted, the kernel or the filesystem consistency checker can find all of the directory entries using the traditional linear directory data structures.

Daniel Phillips created an initial implementation for the Linux 2.4 kernel, and Theodore Ts'o significantly cleaned up the implementation and merged it into the mainline kernel during the Linux 2.5 development cycle, as well as implementing e2fsck support for the HTree data structures. This feature was extremely well received, since for very large directories, performance improvements were often better by a factor of 50–100 or more.

While the HTree algorithm significantly improved lookup times, it could cause some performance regressions for workloads that used `readdir()` to perform some operation of all of the files in a large directory. This is caused by `readdir()` returning filenames in a hash-sorted order, so that reads from the inode table would be done in a random order. This performance regression can be easily fixed by modifying applications to sort the directory entries returned by `readdir()` by inode number. Alternatively, an `LD_PRELOAD` library can be used, which intercepts calls to `readdir()` and returns the directory entries in sorted order.

One potential solution to mitigate this performance issue, which has been suggested by Daniel Phillips and Andreas Dilger, but not yet implemented, involves the kernel choosing free inodes whose inode numbers meet a property that groups the inodes by their filename hash. Daniel and Andreas suggest allocating the inode from a range of inodes based on the size of the directory, and then choosing a free inode from that range based on the filename hash. This should in theory reduce the amount of thrashing that results when accessing the inodes referenced in the directory in readdir order. In it is not clear that this strategy will result in a speedup, however; in fact it could increase the total number of inode blocks that might have to be referenced, and thus make the performance of `readdir()` + `stat()` workloads worse. Clearly, some experimentation and further analysis is still needed.

2.2 Improving ext3 scalability

The scalability improvements in the block layer and other portions of the kernel during 2.5 development uncovered a scaling problem for

ext3/JBD under parallel I/O load. To address this issue, Alex Tomas and Andrew Morton worked to remove a per-filesystem superblock lock (`lock_super()`) from ext3 block allocations [13].

This was done by deferring the filesystem's accounting of the number of free inodes and blocks, only updating these counts when they are needed by `statfs()` or `umount()` system call. This lazy update strategy was enabled by keeping authoritative counters of the free inodes and blocks at the per-block group level, and enabled the replacement of the filesystem-wide `lock_super()` with fine-grained locks. Since a spin lock for every block group would consume too much memory, a hashed spin lock array was used to protect accesses to the block group summary information. In addition, the need to use these spin locks was reduced further by using atomic bit operations to modify the bitmaps, thus allowing concurrent allocations within the same group.

After addressing the scalability problems in the ext3 code proper, the focus moved to the journal (JBD) routines, which made extensive use of the big kernel lock (BKL). Alex Tomas and Andrew Morton worked together to reorganize the locking of the journaling layer in order to allow as much concurrency as possible, by using a fine-grained locking scheme instead of using the BKL and the per-filesystem journal lock. This fine-grained locking scheme uses a new per-bufferhead lock (`BH_JournalHead`), a new per-transaction lock (`t_handle_lock`) and several new per-journal locks (`j_state_lock`, `j_list_lock`, and `j_revoke_lock`) to protect the list of revoked blocks. The locking hierarchy (to prevent deadlocks) for these new locks is documented in the `include/linux/jbd.h` header file.

The final scalability change that was needed was to remove the use of `sleep_on()` (which is only safe when called from within code running under the BKL) and replacing it with the new `wait_event()` facility.

These combined efforts served to improve multiple-writer performance on ext3 noticeably: ext3 throughput improved by a factor of 10 on SDET benchmark, and the context switches are dropped significantly [2, 13].

2.3 Reservation based block allocator

Since disk latency is the key factor that affects the filesystem performance, modern filesystems always attempt to layout files on a filesystem contiguously. This is to reduce disk head movement as much as possible. However, if the filesystem allocates blocks on demand, then when two files located in the same directory are being written simultaneously, the block allocations for the two files may end up getting interleaved. To address this problem, some filesystems use the technique of *preallocation*, by anticipating which files will likely need allocate blocks and allocating them in advance.

2.3.1 Preallocation background

In ext2 filesystem, preallocation is performed on the actual disk bitmap. When a new disk data block is allocated, the filesystem internally preallocates a few disk data blocks adjacent to the block just allocated. To avoid filling up filesystem space with preallocated blocks too quickly, each inode is allowed at most seven preallocated blocks at a time. Unfortunately, this scheme had to be disabled when journaling was added to ext3, since it is incompatible with journaling. If the system were to crash before the unused preallocated blocks could be reclaimed, then during system recovery, the ext3

journal would replay the block bitmap update change. At that point the inode's block mapping could end up being inconsistent with the disk block bitmap. Due to the lack of full forced fsck for ext3 to return the preallocated blocks to the free list, preallocation was disabled when the ext3 filesystem was integrated into the 2.4 Linux kernel.

Disabling preallocation means that if multiple processes attempted to allocate blocks to two files in the same directory, the blocks would be interleaved. This was a known disadvantage of ext3, but this short-coming becomes even more important with extents (see Section 3.1) since extents are far more efficient when the file on disk is contiguous. Andrew Morton, Mingming Cao, Theodore Ts'o, and Badari Pulavarty explored various possible ways to add preallocation to ext3, including the method that had been used for preallocation in ext2 filesystem. The method that was finally settled upon was a reservation-based design.

2.3.2 Reservation design overview

The core idea of the reservation based allocator is that for every inode that needs blocks, the allocator reserves a range of blocks for that inode, called a reservation window. Blocks for that inode are allocated from that range, instead of from the whole filesystem, and no other inode is allowed to allocate blocks in the reservation window. This reduces the amount of fragmentation when multiple files are written in the same directory simultaneously. The key difference between reservation and preallocation is that the blocks are only reserved in memory, rather than on disk. Thus, in the case the system crashes while there are reserved blocks, there is no inconsistency in the block group bitmaps.

The first time an inode needs a new block, a block allocation structure, which describes the reservation window information and other block allocation related information, is allocated and linked to the inode. The block allocator searches for a region of blocks that fulfills three criteria. First, the region must be near the ideal "goal" block, based on ext2/3's existing block placement algorithms. Secondly, the region must not overlap with any other inode's reservation windows. Finally, the region must have at least one free block. As an inode keeps growing, free blocks inside its reservation window will eventually be exhausted. At that point, a new window will be created for that inode, preferably right after the old with the guide of the "goal" block.

All of the reservation windows are indexed via a per-filesystem red-black tree so the block allocator can quickly determine whether a particular block or region is already reserved by a particular inode. All operations on that tree are protected by a per-filesystem global spin lock.

Initially, the default reservation window size for an inode is set to eight blocks. If the reservation allocator detects the inode's block allocation pattern to be sequential, it dynamically increases the window size for that inode. An application that knows the file size ahead of the file creation can employ an ioctl command to set the window size to be equal to the anticipated file size in order to attempt to reserve the blocks immediately.

Mingming Cao implemented this reservation based block allocator, with help from Stephen Tweedie in converting the per-filesystem reservation tree from a sorted link list to a red-black tree. In the Linux kernel versions 2.6.10 and later, the default block allocator for ext3 has been replaced by this reservation based block allocator. Some benchmarks, such as tiobench and dbench, have shown significant improvements on sequential writes and subsequent sequential reads with this reservation-based block

allocator, especially when a large number of processes are allocating blocks concurrently.

2.3.3 Future work

Currently, the reservation window only lasts until the last process writing to that file closes. At that time, the reservation window is released and those blocks are available for reservation or allocation by any other inode. This is necessary so that the blocks that were reserved can be released for use by other files, and to avoid fragmentation of the free space in the filesystem.

However, some files, such as log files and UNIX® mailbox files, have a *slow growth* pattern. That is, they grow slowly over time, by processes appending a small amount of data, and then closing the file, over and over again. For these files, in order to avoid fragmentation, it is necessary that the reservation window be preserved even after the file has been closed.

The question is how to determine which files should be allowed to retain their reservation window after the last close. One possible solution is to tag the files or directories with an attribute indicating that they contain files that have a slow growth pattern. Another possibility is to implement heuristics that can allow the filesystem to automatically determines which file seems to have a slow growth pattern, and automatically preserve the reservation window after the file is closed.

If reservation windows can be preserved in this fashion, it will be important to also implement a way for preserved reservation windows to be reclaimed when the filesystem is fully reserved. This prevents an inode that fails to find a new reservation from falling back to no-reservation mode too soon.

2.4 Online resizing

The online resizing feature was originally developed by Andreas Dilger in July of 1999 for the 2.0.36 kernel. The availability of a Logical Volume Manager (LVM), motivated the desire for on-line resizing, so that when a logical volume was dynamically resized, the filesystem could take advantage of the new space. This ability to dynamically resize volumes and filesystems is very useful in server environments, where taking downtime for unmounting a filesystem is not desirable. After missing the code freeze for the 2.4 kernel, the ext2online code was finally included into the 2.6.10 kernel and e2fsprogs 1.36 with the assistance of Stephen Tweedie and Theodore Ts'o.

2.4.1 The online resizing mechanism

The online resizing mechanism, despite its seemingly complex task, is actually rather simple in its implementation. In order to avoid a large amount of complexity it is only possible to increase the size of a filesystem while it is mounted. This addresses the primary requirement that a filesystem that is (nearly) full can have space added to it without interrupting the use of that system. The online resizing code depends on the underlying block device to handle all aspects of its own resizing prior to the start of filesystem resizing, and does nothing itself to manipulate the partition tables of LVM/MD block devices.

The ext2/3 filesystem is divided into one or more block allocation groups of a fixed size, with possibly a partial block group at the end of the filesystem [4]. The layout of each block group (where the inode and block allocation bitmaps and the inode table are stored) is kept in the group descriptor table. This table is stored at the start of at the first block group, and

consists of one or more filesystem blocks, depending on the size of the filesystem. Backup copies of the group descriptor table are kept in more groups if the filesystem is large enough.

There are three primary phases by which a filesystem is grown. The first, and simplest, is to expand the last partial block group (if any) to be a full block group. The second phase is to add a new block group to an existing block in the group descriptor table. The third phase is to add a new block to the group descriptor table and add a new group to that block. All filesystem resizes are done incrementally, going through one or more of the phases to add free space to the end of the filesystem until the desired size is reached.

2.4.2 Resizing within a group

For the first phase of growth, the online resizing code starts by briefly locking the superblock and increasing the total number of filesystem blocks to the end of the last group. All of the blocks beyond the end of the filesystem are already marked as "in use" by the block bitmap for that group, so they must be cleared. This is accomplished by the same mechanism that is used when deleting a file—`ext3_free_blocks()` and can be done without locking the whole filesystem. The online resizer simply pretends that it is deleting a file that had allocated all of the blocks at the end of the filesystem, and `ext3_free_blocks()` handles all of the bitmap and free block count updates properly.

2.4.3 Adding a new group

For the second phase of growth, the online resizer initializes the next group beyond the end of the filesystem. This is easily done because this area is currently unused and unknown to the filesystem itself. The block bitmap for that group is initialized as empty, the superblock and group descriptor backups (if any) are copied from the primary versions, and the inode bitmap and inode table are initialized. Once this has completed successfully the online resizing code briefly locks the superblock to increase the total and free blocks and inodes counts for the filesystem, add a new group to the end of the group descriptor table, and increase the total number of groups in the filesystem by one. Once this is completed the backup superblock and group descriptors are updated in case of corruption of the primary copies. If there is a problem at this stage, the next e2fsck will also update the backups.

The second phase of growth will be repeated until the filesystem has fully grown, or the last group descriptor block is full. If a partial group is being added at the end of the filesystem the blocks are marked as "in use" before the group is added. Both first and second phase of growth can be done on any ext3 filesystem with a supported kernel and suitable block device.

2.4.4 Adding a group descriptor block

The third phase of growth is needed periodically to grow a filesystem over group descriptor block boundaries (at multiples of 16 GB for filesystems with 4 KB blocksize). When the last group descriptor block is full, a new block must be added to the end of the table. However, because the table is contiguous at the start of the first group and is normally followed immediately by the block and inode bitmaps and the inode table, the online resize code needs a bit of assistance while the filesystem is unmounted (offline) in order to maintain compatibility with older kernels. Either at `mke2fs`

time, or for existing filesystems with the assistance of the `ext2prepare` command, a small number of blocks at the end of the group descriptor table are reserved for online growth. The total amount of reserved blocks is a tiny fraction of the total filesystem size, requiring only a few tens to hundreds of kilobytes to grow the filesystem 1024-fold.

For the third phase, it first gets the next reserved group descriptor block and initializes a new group and group descriptor beyond the end of the filesystem, as is done in second phase of growth. Once this is successful, the superblock is locked while reallocating the array that indexes all of the group descriptor blocks to add another entry for the new block. Finally, the superblock totals are updated, the number of groups is increased by one, and the backup superblock and group descriptors are updated.

The online resizing code takes advantage of the journaling features in ext3 to ensure that there is no risk of filesystem corruption if the resize is unexpectedly interrupted. The ext3 journal ensures strict ordering and atomicity of filesystem changes in the event of a crash—either the entire resize phase is committed or none of it is. Because the journal has no rollback mechanism (except by crashing) the resize code is careful to verify all possible failure conditions prior to modifying any part of the filesystem. This ensures that the filesystem remains valid, though slightly smaller, in the event of an error during growth.

2.4.5 Future work

Future development work in this area involves removing the need to do offline filesystem manipulation to reserve blocks before doing third phase growth. The use of Meta Block Groups [15] allows new groups to be added to the filesystem without the need to allocate contiguous blocks for the group descriptor table. Instead the group descriptor block is kept in the first group that it describes, and a backup is kept in the second and last group for that block. The Meta Block Group support was first introduced in the 2.4.25 kernel (Feb. 2004) so it is reasonable to think that a majority of existing systems could mount a filesystem that started using this when it is introduced.

A more complete description of the online growth is available in [6].

2.5 Extended attributes

2.5.1 Extended attributes overview

Many new operating system features (such as access control lists, mandatory access controls, Posix Capabilities, and hierarchical storage management) require filesystems to be able associate a small amount of custom metadata with files or directories. In order to implement support for access control lists, Andreas Gruenbacher added support for extended attributes to the ext2 filesystems. [7]

Extended attributes as implemented by Andreas Gruenbacher are stored in a single EA block. Since a large number of files will often use the same access control list, as inherited from the directory's default ACL as an optimization, the EA block may be shared by inodes that have identical extended attributes.

While the extended attribute implementation was originally optimized for use to store ACL's, the primary users of extended attributes to date have been the NSA's SELinux system, Samba 4 for storing extended attributes from Windows clients, and the Lustre filesystem.

In order to store larger EAs than a single filesystem block, work is underway to store

large EAs in another EA inode referenced from the original inode. This allows many arbitrary-sized EAs to be attached to a single file, within the limitations of the EA interface and what can be done inside a single journal transaction. These EAs could also be accessed as additional file forks/streams, if such an API were added to the Linux kernel.

2.5.2 Large inode support and EA-in-inode

Alex Tomas and Andreas Dilger implemented support for storing the extended attribute in an expanded ext2 inode, in preference to using a separate filesystem block. In order to do this, the filesystem must be created using an inode size larger than the default 128 bytes. Inode sizes must be a power of two and must be no larger than the filesystem block size, so for a filesystem with a 4 KB blocksize, inode sizes of 256, 512, 1024, 2048, or 4096 bytes are valid. The 2 byte field starting at offset 128 (`i_extra_size`) of each inode specifies the starting offset for the portion of the inode that can be used for storing EA's. Since the starting offset must be a multiple of 4, and we have not extended the fixed portion of the inode beyond `i_extra_size`, currently `i_extra_size` is 4 for all filesystems with expanded inodes. Currently, all of the inode past the initial 132 bytes can be used for storing EAs. If the user attempts to store more EAs than can fit in the expanded inode, the additional EAs will be stored in an external filesystem block.

Using the EA-in-inode, a very large (seven-fold improvement) difference was found in some Samba 4 benchmarks, taking ext3 from last place when compared to XFS, JFS, and Reiserfs3, to being clearly superior to all of the other filesystems for use in Samba 4. [5] The in-inode EA patch started by Alex Tomas and Andreas Dilger was re-worked by Andreas Gruenbacher. And the fact that this feature was such a major speedup for Samba 4, motivated it being integrated into the mainline 2.6.11 kernel very quickly.

3 Extents, delayed allocation and extent allocation

This section and the next (Section 4) will discuss features that are currently under development, and (as of this writing) have not been merged into the mainline kernel. In most cases patches exist, but they are still being polished, and discussion within the ext2/3 development community is still in progress.

Currently, the ext2/ext3 filesystem, like other traditional UNIX filesystems, uses a direct, indirect, double indirect, and triple indirect blocks to map file offsets to on-disk blocks. This scheme, sometimes simply called an indirect block mapping scheme, is not efficient for large files, especially large file deletion. In order to address this problem, many modern filesystems (including XFS and JFS on Linux) use some form of extent maps instead of the traditional indirect block mapping scheme.

Since most filesystems try to allocate blocks in a contiguous fashion, extent maps are a more efficient way to represent the mapping between logical and physical blocks for large files. An *extent* is a single descriptor for a range of contiguous blocks, instead of using, say, hundreds of entries to describe each block individually.

Over the years, there have been many discussions about moving ext3 from the traditional indirect block mapping scheme to an extent map based scheme. Unfortunately, due to the complications involved with making an incompatible format change, progress on an actual implemention of these ideas had been slow.

Alex Tomas, with help from Andreas Dilger, designed and implemented extents for ext3. He posted the initial version of his extents patch on August, 2003. The initial results on file creation and file deletion tests inspired a round of discussion in the Linux community to consider adding extents to ext3. However, given the concerns that the format changes were ones that all of the ext3 developers will have to support on a long-term basis, and the fact that it was very late in the 2.5 development cycle, it was not integrated into the mainline kernel sources at that time.

Later, in April of 2004, Alex Tomas posted an updated extents patch, as well as additional patches that implemented delayed allocation and multiple block allocation to the ext2-devel mailing list. These patches were reposted in February 2005, and this re-ignited interest in adding extents to ext3, especially when it was shown that the combination of these three features resulted in significant throughput improvements on some sequential write tests.

In the next three sections, we will discuss how these three features are designed, followed by a discussion of the performance evaluation of the combination of the three patches.

3.1 Extent maps

This implementation of extents was originally motivated by the problem of long truncate times observed for huge files.[1] As noted above, besides speeding up truncates, extents help improve the performance of sequential file writes since extents are a significantly smaller amount of metadata to be written to describe contiguous blocks, thus reducing the filesystem overhead.

Most files need only a few extents to describe their logical-to-physical block mapping, which can be accommodated within the inode or a single extent map block. However, some extreme cases, such as sparse files with random allocation patterns, or a very badly fragmented filesystem, are not efficiently represented using extent maps. In addition, allocating blocks in a random access pattern may require inserting an extent map entry in the middle of a potentially very large data representation.

One solution to this problem is to use a tree data structure to store the extent map, either a B-tree, B+ tree, or some simplified tree structure as was used for the HTree feature. Alex Tomas's implementation takes the latter approach, using a constant-depth tree structure. In this implementation, the extents are expressed using a 12 byte structure, which include a 32-bit logical block number, a 48-bit physical block number, and a 16-bit extent length. With 4 KB blocksize, a filesystem can address up to 1024 petabytes, and a maximum file size of 16 terabytes. A single extent can cover up to 2^{16} blocks or 256 MB.[2]

The extent tree information can be stored in the inode's `i_data` array, which is 60 bytes long. An attribute flag in the inode's `i_flags` word indicates whether the inode's `i_data` array should be interpreted using the traditional indirect block mapping scheme, or as an extent data structure. If the entire extent information can be stored in the `i_data` field, then it will be treated as a single leaf node of the extent tree; otherwise, it will be treated as the root node of inode's extent tree, and additional filesystem blocks serve as intermediate or leaf nodes in the extent tree.

At the beginning of each node, the `ext3_ext_header` data structure is 12 bytes long,

[1] One option to address the issue is performing asynchronous truncates, however, while this makes the CPU cycles to perform the truncate less visible, excess CPU time will still be consumed by the truncate operations.

[2] Currently, the maximum block group size given a 4 KB blocksize is 128 MB, and this will limit the maximum size for a single extent.

and contains a 16-bit magic number, 2 16-bit integers containing the number of valid entries in the node, and the maximum number of entries that can be stored in the node, a 16-bit integer containing the depth of the tree, and a 32-bit tree generation number. If the depth of the tree is 0, then root inode contains leaf node information, and the 12-byte entries contain the extent information described in the previous paragraph. Otherwise, the root node will contain 12-byte intermediate entries, which consist of 32-bit logical block and a 48-bit physical block (with 16 bits unused) of the next index or leaf block.

3.1.1 Code organization

The implementation is divided into two parts: Generic extents support that implements initialize/lookup/insert/remove functions for the extents tree, and VFS support that allows methods and callbacks like `ext3_get_block()`, `ext3_truncate()`, `ext3_new_block()` to use extents.

In order to use the generic extents layer, the user of the generic extents layer must declare its tree via an `ext3_extents_tree` structure. The structure describes where the root of the tree is stored, and specifies the helper routines used to operate on it. This way one can root a tree not only in `i_data` as described above, but also in a separate block or in EA (Extended Attributes) storage. The helper routines described by struct `ext3_extents_helpers` can be used to control the block allocation needed for tree growth, journaling metadata, using different criteria of extents mergability, removing extents etc.

3.1.2 Future work

Alex Tomas's extents implementation is still a work-in-progress. Some of the work that needs to be done is to make the implementation independent of byte-order, improving the error handling, and shrinking the depth of the tree when truncated the file. In addition, the extent scheme is less efficient than the traditional indirect block mapping scheme if the file is highly fragmented. It may be useful to develop some heuristics to determine whether or not a file should use extents automatically. It may also be desirable to allow block-mapped leaf blocks in an extent-mapped file for cases where there is not enough contiguous space in the filesystem to allocate the extents efficiently.

The last change would necessarily change the on-disk format of the extents, but it is not only the extent format that has been changed. For example, the extent format does not support logical block numbers that are greater than 32 bits, and a more efficient, variable-length format would allow more extents to be stored in the inode before spilling out to an external tree structure.

Since deployment of the extent data structure is disruptive because it involved an non-backwards-compatible change to the filesystem format, it is important that the ext3 developers are comfortable that the extent format is flexible and powerful enough for present and future needs, in order to avoid the need for additional incompatible format changes.

3.2 Delayed allocation

3.2.1 Why delayed allocation is needed

Procrastination has its virtues in the ways of an operating system. Deferring certain tasks un-

til an appropriate time often improves the overall efficiency of the system by enabling optimal deployment of resources. Filesystem I/O writes are no exception.

Typically, when a filesystem `write()` system call returns success, it has only copied the data to be written into the page cache, mapped required blocks in the filesystem and marked the pages as needing write out. The actual write out of data to disk happens at a later point of time, usually when writeback operations are clustered together by a background kernel thread in accordance with system policies, or when the user requests file data to be synced to disk. Such an approach ensures improved I/O ordering and clustering for the system, resulting in more effective utilization of I/O devices with applications spending less time in the `write()` system call, and using the cycles thus saved to perform other work.

Delayed allocation takes this a step further, by deferring the allocation of new blocks in the filesystem to disk blocks until writeback time [12]. This helps in three ways:

- Reduces fragmentation in the filesystem by improving chances of creating contiguous blocks on disk for a file. Although preallocation techniques can help avoid fragmentation, they do not address fragmentation caused by multiple threads writing to the file at different offsets simultaneously, or files which are written in a non-contiguous order. (For example, the `libfd` library, which is used by the GNU C compiler will create object files that are written out of order.)
- Reduces CPU cycles spent in repeated `get_block()` calls, by clustering allocation for multiple blocks together. Both of the above would be more effective when combined with a good multi-block allocator.
- For short lived files that can be buffered in memory, delayed allocation may avoid the need for disk updates for metadata creation altogether, which in turn reduces impact on fragmentation [12].

Delayed allocation is also useful for the Active Block I/O Scheduling System (ABISS) [1], which provides guaranteed read/write bit rates for applications that require guaranteed real-time I/O streams. Without delayed allocation, the synchronous code path for `write()` has to read, modify, update, and journal changes to the block allocation bitmap, which could disrupt the guaranteed read/write rates that ABISS is trying to deliver.

Since block allocation is deferred until background writeback when it is too late to return an error to the caller of `write()`, the `write()` operation requires a way to ensure that the allocation will indeed succeed. This can be accomplished by carving out, or reserving, a claim on the expected number of blocks on disk (for example, by subtracting this number from the total number of available blocks, an operation that can be performed without having to go through actual allocation of specific disk blocks).

Repeated invocations of `ext3_get_block()/ext3_new_block()` is not efficient for mapping consecutive blocks, especially for an extent based inode, where it is natural to process a chunk of contiguous blocks all together. For this reason, Alex Tomas implemented an extents based multiple block allocation and used it as a basis for extents based delayed allocation. We will discuss the extents based multiple block allocation in Section 3.3.

3.2.2 Extents based delayed allocation implementation

If the delayed allocation feature is enabled for an ext3 filesystem and a file uses extent maps, then the address space operations for its inode are initialized to a set of ext3 specific routines that implement the write operations a little differently. The implementation defers allocation of blocks from `prepare_write()` and employs extent walking, together with the multiple block allocation feature (described in the next section), for clustering block allocations maximally into contiguous blocks.

Instead of allocating the disk block in `prepare_write()`, the the page is marked as needing block reservation. The `commit_write()` function calculates the required number of blocks, and reserves them to make sure that there are enough free blocks in the filesystem to satisfy the write. When the pages get flushed to disk by `writepage()` or `writepages()`, these functions will walk all the dirty pages in the specified inode, cluster the logically contiguous ones, and submit the page or pages to the bio layer. After the block allocation is complete, the reservation is dropped. A single block I/O request (or `BIO`) is submitted for write out of pages processed whenever a new allocated extent (or the next mapped extent if already allocated) on the disk is not adjacent to the previous one, or when `writepages()` completes. In this manner the delayed allocation code is tightly integrated with other features to provide best performance.

3.3 Buddy based extent allocation

One of the shortcomings of the current ext3 block allocation algorithm, which allocates one block at a time, is that it is not efficient enough for high speed sequential writes. In one experiment utilizing direct I/O on a dual Opteron workstation with fast enough buses, fiber channel, and a large, fast RAID array, the CPU limited the I/O throughput to 315 MB/s. While this would not be an issue on most machines (since the maximum bandwidth of a PCI bus is 127 MB/s), but for newer or enterprise-class servers, the amount of data per second that can be written continuously to the filesystem is no longer limited by the I/O subsystem, but by the amount of CPU time consumed by ext3's block allocator.

To address this problem, Alex Tomas designed and implemented a multiple block allocation, called mballoc, which uses a classic buddy data structure on disk to store chunks of free or used blocks for each block group. This buddy data is an array of metadata, where each entry describes the status of a cluster of 2^n blocks, classified as free or in use.

Since block buddy data is not suitable for determining a specific block's status and locating a free block close to the allocation goal, the traditional block bitmap is still required in order to quickly test whether a specific block is available or not.

In order to find a contiguous extent of blocks to allocate, mballoc checks whether the goal block is available in the block bitmap. If it is available, mballoc looks up the buddy data to find the free extent length starting from the goal block. To find the real free extent length, mballoc continues by checking whether the physical block right next to the end block of the previously found free extent is available or not. If that block is available in the block bitmap, mballoc could quickly find the length of the next free extent from buddy data and add it up to the total length of the free extent from the goal block.

For example, if block M is the goal block and

is claimed to be available in the bitmap, and block M is marked as free in buddy data of order n, then initially the free chunk size from block M is known to be 2^n. Next, mballoc checks the bitmap to see if block $M + 2^n + 1$ is available or not. If so, mballoc checks the buddy data again, and finds that the free extent length from block $M + 2^n + 1$ is k. Now, the free chunk length from goal block M is known to be $2^n + 2^k$. This process continues until at some point the boundary block is not available. In this manner, instead of testing dozens, hundreds, or even thousands of blocks' availability status in the bitmap to determine the free blocks chunk size, it can be enough to just test a few bits in buddy data and the block bitmap to learn the real length of the free blocks extent.

If the found free chunk size is greater than the requested size, then the search is considered successful and mballoc allocates the found free blocks. Otherwise, depending on the allocation criteria, mballoc decides whether to accept the result of the last search in order to preserve the goal block locality, or continue searching for the next free chunk in case the length of contiguous blocks is a more important factor than where it is located. In the later case, mballoc scans the bitmap to find out the next available block, then, starts from there, and determines the related free extent size.

If mballoc fails to find a free extent that satisfies the requested size after rejecting a predefined number (currently 200) of free chunks, it stops the search and returns the best (largest) free chunk found so far. In order to speed up the scanning process, mballoc maintains the total number of available blocks and the first available block of each block group.

3.3.1 Future plans

Since in ext3 blocks are divided into block groups, the block allocator first selects a block group before it searches for free blocks. The policy employed in mballoc is quite simple: to try the block group where the goal block is located first. If allocation from that group fails, then scan the subsequent groups. However, this implies that on a large filesystem, especially when free blocks are not evenly distributed, CPU cycles could be wasted on scanning lots of almost full block groups before finding a block group with the desired free blocks criteria. Thus, a smarter mechanism to select the right block group to start the search should improve the multiple block allocator's efficiency. There are a few proposals:

1. Sort all the block groups by the total number of free blocks.
2. Sort all the groups by the group fragmentation factor.
3. Lazily sort all the block groups by the total number of free blocks, at significant change of free blocks in a group only.
4. Put extents into buckets based on extent size and/or extent location in order to quickly find extents of the correct size and goal location.

Currently the four options are under evaluation though probably the first one is a little more interesting.

3.4 Evaluating the extents patch set

The initial evaluation of the three patches (extents, delayed allocation and extent allocation) shows significant throughput improvements, especially under sequential tests. The

tests show that the extents patch significantly reduces the time for large file creation and removal, as well as file rewrite. With extents and extent allocation, the throughput of Direct I/O on the aforementioned Opteron-based workstation is significantly improved, from 315 MB/sec to 500MB/sec, and the CPU usage is significantly dropped from 100% to 50%. In addition, extensive testing on various benchmarks, including dbench, tiobench, FFSB [11] and sqlbench [16], has been done with and without this set of patches. Some initial analysis indicates that the multiple block allocation, when combined with delayed allocation, is a key factor resulting in this improvement. More testing results can be obtained from `http://www.bullopensource.org/ext4`.

4 Improving ext3 without changing disk format

Replacing the traditional indirect block mapping scheme with an extent mapping scheme, has many benefits, as we have discussed in the previous section. However, changes to the on-disk format that are not backwards compatible are often slow to be adopted by users, for two reasons. First of all, robust e2fsck support sometimes lags the kernel implementation. Secondly, it is generally not possible to mount the filesystem with an older kernel once the filesystem has been converted to use these new features, preventing rollback in case of problems.

Fortunately, there are a number of improvements that can be made to the ext2/3 filesystem without making these sorts of incompatible changes to the on-disk format.

In this section, we will discuss a few of features that are implemented based on the current ext3 filesystem. Section 4.1 describes the effort to reduce the usage of bufferheads structure in ext3; Section 4.2 describes the effort to add delayed allocation without requiring the use of extents; Section 4.3 discusses the work to add multiple block allocation; Section 4.4 describes asynchronous file unlink and truncate; Section 4.5 describes a feature to allow more than 32000 subdirectories; and Section 4.6 describes a feature to allow multiple threads to concurrently create/rename/link/unlink files in a single directory.

4.1 Reducing the use of bufferheads in ext3

Bufferheads continue to be heavily used in Linux I/O and filesystem subsystem, even though closer integration of the buffer cache with the page cache since 2.4 and the new block I/O subsystem introduced in Linux 2.6 have in some sense superseded part of the traditional Linux buffer cache functionality.

There are a number of reasons for this. First of all, the buffer cache is still used as a metadata cache. All filesystem metadata (superblock, inode data, indirect blocks, etc.) are typically read into buffer cache for quick reference. Bufferheads provide a way to read/write/access this data. Second, bufferheads link a page to disk block and cache the block mapping information. In addition, the design of bufferheads supports filesystem block sizes that do not match the system page size. Bufferheads provide a convenient way to map multiple blocks to a single page. Hence, even the generic multi-page read-write routines sometimes fall back to using bufferheads for fine-graining or handling of complicated corner cases.

Ext3 is no exception to the above. Besides the above reasons, ext3 also makes use of bufferheads to enable it to provide ordering guarantees in case of a transaction commit. Ext3's or-

dered mode guarantees that file data gets written to the disk before the corresponding metadata gets committed to the journal. In order to provide this guarantee, bufferheads are used as the mechanism to associate the data pages belonging to a transaction. When the transaction is committed to the journal, ext3 uses the bufferheads attached to the transaction to make sure that all the associated data pages have been written out to the disk.

However, bufferheads have the following disadvantages:

- All bufferheads are allocated from the "buffer_head" slab cache, thus they consume low memory[3] on 32-bit architectures. Since there is one bufferhead (or more, depending on the block size) for each filesystem page cache page, the bufferhead slab can grow really quickly and consumes a lot of low memory space.
- When bufferheads get attached to a page, they take a reference on the page. The reference is dropped only when VM tries to release the page. Typically, once a page gets flushed to disk it is safe to release its bufferheads. But dropping the bufferhead, right at the time of I/O completion is not easy, since being in interrupt handler context restricts the kind of operations feasible. Hence, bufferheads are left attached to the page, and released later as and when VM decides to re-use the page. So, it is typical to have a large number of bufferheads floating around in the system.
- The extra memory references to bufferheads can impact the performance of memory caches, the Translation Lookaside Buffer (TLB) and the Segment Lookaside Buffer[4] (SLB). We have observed that when running a large NFS workload, while the ext3 journaling thread kjournald() is referencing all the transactions, all the journal heads, and all the bufferheads looking for data to flush/clean it suffers a large number of SLB misses with the associated performance penalty. The best solution for these performance problems appears to be to eliminate the use of bufferheads as much as possible, which reduces the number of memory references required by kjournald().

To address the above concerns, Badari Pulavarty has been working on removing bufferheads usage from ext3 from major impact areas, while retaining bufferheads for uncommon usage scenarios. The focus was on elimination of bufferhead usage for user data pages, while retaining bufferheads primarily for metadata caching.

Under the writeback journaling mode, since there are no ordering requirements between when metadata and data gets flushed to disk, eliminating the need for bufferheads is relatively straightforward because ext3 can use most recent generic VFS helpers for writeback. This change is already available in the latest Linux 2.6 kernels.

For ext3 ordered journaling mode, however, since bufferheads are used as linkage between pages and transactions in order to provide flushing order guarantees, removal of the use of bufferheads gets complicated. To address this issue, Andrew Morton proposed a new ext3 journaling mode, which works without bufferheads and provides semantics that are somewhat close to that provided in ordered mode[9]. The idea is that whenever there is a transaction commit, we go through all the dirty inodes and

[3] Low memory is memory that can be directly mapped into kernel virtual address space, i.e. 896MB, in the case of IA32.

[4] The SLB is found on the 64-bit Power PC.

dirty pages in that filesystem and flush every one of them. This way metadata and user data are flushed at the same time. The complexity of this proposal is currently under evaluation.

4.2 Delayed allocation without extents

As we have discussed in Section 3.2, delayed allocation is a powerful technique that can result in significant performance gains, and Alex Tomas's implementation shows some very interesting and promising results. However, Alex's implementation only provide delayed allocation when the ext3 filesystem is using extents, which requires an incompatible change to the on-disk format. In addition, like past implementation of delayed allocation by other filesystems, such as XFS, Alex's changes implement the delayed allocation in filesystem-specific versions of `prepare_write()`, `commit_write()`, `writepage()`, and `writepages()`, instead of using the filesystem independent routines provided by the Linux kernel.

This motivated Suparna Bhattacharya, Badari Pulavarty and Mingming Cao to implement delayed allocation and multiple block allocation support to improve the performance of the ext3 to the extent possible without requiring any on-disk format changes.

Interestingly, the work to remove the use of bufferheads in ext3 implemented most of the necessary changes required for delayed allocation, when bufferheads are not required. The `nobh_commit_write()` function, delegates the task of writing data to the `writepage()` and `writepages()`, by simply marking the page as dirty. Since the `writepage()` function already has to handle the case of writing a page which is mapped to a sparse memory-mapped files, the `writepage()` function already handles block allocation by calling the filesystem specific `get_block()` function. Hence, if the `nobh_prepare_write` function were to omit call `get_block()`, the physical block would not be allocated until the page is actually written out via the `writepage()` or `writepages()` function.

Badari Pulavarty implemented a relatively small patch as a proof-of-concept, which demonstrates that this approach works well. The work is still in progress, with a few limitations to address. The first limitation is that in the current proof-of-concept patch, data could be dropped if the filesystem was full, without the `write()` system call returning -ENOSPC.[5] In order to address this problem, the `nobh_prepare_write` function must note that the page currently does not have a physical block assigned, and request the filesystem reserve a block for the page. So while the filesystem will not have assigned a specific physical block as a result of `nobh_prepare_write()`, it must guarantee that when `writepage()` calls the block allocator, the allocation must succeed.

The other major limitation is, at present, it only worked when bufferheads are not needed. However, the `nobh` code path as currently present into the 2.6.11 kernel tree only supports filesystems when the ext3 is journaling in writeback mode and not in ordered journaling mode, and when the blocksize is the same as the VM pagesize. Extending the `nobh` code paths to support sub-pagesize blocksizes is likely not very difficult, and is probably the appropriate way of addressing the first part of this shortcoming.

[5] The same shortcoming exists today if a sparse file is memory-mapped, and the filesystem is full when `writepage()` tries to write a newly allocated page to the filesystem. This can potentially happen after user process which wrote to the file via `mmap()` has exited, where there is no program left to receive an error report.

However, supporting delayed allocation for ext3 ordered journaling using this approach is going to be much more challenging. While metadata journaling alone is sufficient in writeback mode, ordered mode needs to track I/O submissions for purposes of waiting for completion of data writeback to disk as well, so that it can ensure that metadata updates hit the disk only after the corresponding data blocks are on disk. This avoids potential exposures and inconsistencies without requiring full data journaling[14].

However, in the current design of generic multi-page writeback routines, block I/O submissions are issued directly by the generic routines and are transparent to the filesystem specific code. In earlier situations where bufferheads were used for I/O, filesystem specific wrappers around generic code could track I/O through the bufferheads associated with a page and link them with the transaction. With the recent changes, where I/O requests are built directly as multi-page bio requests with no link from the page to the bio, this no longer applies.

A couple of solution approaches are under consideration, as of the writing of this paper:

- Introducing yet another filesystem specific callback to be invoked by the generic multi-page write routines to actually issue the I/O. ext3 could then track the number of in-flight I/O requests associated with the transaction, and wait for this to fall to zero at journal commit time. Implementing this option is complicated because the multi-page write logic occasionally falls back to the older bufferheads based logic in some scenarios. Perhaps ext3 ordered mode writeback would need to provide both the callback and the page bufferhead tracking logic if this approach is employed.
- Find a way to get ext3 journal commit to effectively reuse a part the fsync/O_SYNC implementation that waits for writeback to complete on the pages for relevant inodes, using a radix-tree walk. Since the journal layer is designed to be unaware of filesystems [14], this could perhaps be accomplished by associating a (filesystem specific) callback with journal commit, as recently suggested by Andrew Morton[9].

It remains to be seen which approach works out to be the best, as development progresses. It is clear that since ordered mode is the default journaling mode, any delayed allocation implementation must be able to support it.

4.3 Efficiently allocating multiple blocks

As with the Alex Tomas's delayed allocation patch, Alex's multiple block allocator patch relies on an incompatible on-disk format change of the ext3 filesystem to support extent maps. In addition, the extent-based mballoc patch also required a format change in order to store data for the buddy allocator which it utilized. Since oprofile measurements of Alex's patch indicated the multiple block allocator seemed to be responsible for reducing CPU usage, and since it seemed to improve throughput in some workloads, we decided to investigate whether it was possible to obtain most of the benefits of a multiple block allocator using the current ext3 filesystem format. This seemed to be a reasonable approach since many of the advantages of supporting Alex's mballoc patch seemed to derive from collapsing a large number of calls to `ext3_get_block()` into much fewer calls to `ext3_get_blocks()`, thus avoiding excess calls into the journaling layer to record changes to the block allocation bitmap.

In order to implement a multiple-block allocator based on the existing block allocation

bitmap, Mingming Cao first changed `ext3_new_block()` to accept a new argument specifying how many contiguous blocks the function should attempt to allocate, on a best efforts basis. The function now allocates the first block in the existing way, and then continues allocating up to the requested number of adjacent physical blocks at the same time if they are available.

The modified `ext3_new_block()` function was then used to implement ext3's `get_blocks()` method, the standardized filesystem interface to translate a file offset and a length to a set of on-disk blocks. It does this by starting at the first file offset and translating it into a logical block number, and then taking that logical block number and mapping it to a physical block number. If the logical block has already been mapped, then it will continue mapping the next logical block until the requisite number of physical blocks have been returned, or an unallocated block is found.

If some blocks need to be allocated, first `ext3_get_blocks()` will look ahead to see how many adjacent blocks are needed, and then passes this allocation request to `ext3_new_blocks()`, searches for the requested free blocks, marks them as used, and returns them to `ext3_get_blocks()`. Next, `ext3_get_blocks()` will update the inode's direct blocks, or a single indirect block to point at the allocated blocks.

Currently, this `ext3_get_blocks()` implementation does not allocate blocks across an indirect block boundary. There are two reasons for this. First, the `JBD` journaling requests the filesystem to reserve the maximum of blocks that will require journaling, when a new transaction handle is requested via `ext3_journal_start()`. If we were to allow a multiple block allocation request to span an indirect block boundary, it would be difficult to predict how many metadata blocks may get dirtied and thus require journaling. Secondly, it would be difficult to place any newly allocated indirect blocks so they are appropriately interleaved with the data blocks.

Currently, only the Direct I/O code path uses the `get_blocks()` interfaces; the `mpage_writepages()` function calls `mpage_writepage()` which in turn calls `get_block()`. Since only a few workloads (mainly databases) use Direct I/O, Suparna Bhattacharya has written a patch to change `mpage_writepages()` use `get_blocks()` instead. This change should be generically helpful for any filesystems which implement an efficient `get_blocks()` function.

Draft patches have already been posted to the ext2-devel mailing list. As of this writing, we are trying to integrate Mingming's `ext3_get_blocks()` patch, Suparna Bhattacharya's `mpage_writepage()` patch and Badari Pulavarty's generic delayed allocation patch (discussed in Section 4.2) in order to evaluate these three patches together using benchmarks.

4.4 Asynchronous file unlink/truncate

With block-mapped files and ext3, truncation of a large file can take a considerable amount of time (on the order of tens to hundreds of seconds if there is a lot of other filesystem activity concurrently). There are several reasons for this:

- There are limits to the size of a single journal transaction (1/4 of the journal size). When truncating a large fragmented file, it may require modifying so many block bitmaps and group descriptors that it forces a journal transaction to close out, stalling the unlink operation.

- Because of this per-transaction limit, truncate needs to zero the [dt]indirect blocks starting from the end of the file, in case it needs to start a new transaction in the middle of the truncate (ext3 guarantees that a partially-completed truncate will be consistent/completed after a crash).
- The read/write of the file's [dt]indirect blocks from the end of the file to the beginning can take a lot of time, as it does this in single-block chunks and the blocks are not contiguous.

In order to reduce the latency associated with large file truncates and unlinks on the Lustre® filesystem (which is commonly used by scientific computing applications handling very large files), the ability for ext3 to perform asynchronous unlink/truncate was implemented by Andreas Dilger in early 2003.

The delete thread is a kernel thread that services a queue of inode unlink or truncate-to-zero requests that are intercepted from normal `ext3_delete_inode()` and `ext3_truncate()` calls. If the inode to be unlinked/truncated is small enough, or if there is any error in trying to defer the operation, it is handled immediately; otherwise, it is put into the delete thread queue. In the unlink case, the inode is just put into the queue and the delete thread is woke up, before returning to the caller. For the truncate-to-zero case, a free inode is allocated and the blocks are moved over to the new inode before waking the thread and returning to the caller. When the delete thread is woke up, it does a normal truncate of all the blocks on each inode in the list, and then frees the inode.

In order to handle these deferred delete/truncate requests in a crash-safe manner, the inodes to be unlinked/truncated are added into the ext3 orphan list. This is an already existing mechanism by which ext3 handles file unlink/truncates that might be interrupted by a crash. A persistent singly-linked list of inode numbers is linked from the superblock and, if this list is not empty at filesystem mount time, the ext3 code will first walk the list and delete/truncate all of the files on it before the mount is completed.

The delete thread was written for 2.4 kernels, but is currently only in use for Lustre. The patch has not yet been ported to 2.6, but the amount of effort needed to do so is expected to be relatively small, as the ext3 code has changed relatively little in this area.

For extent-mapped files, the need to have asynchronous unlink/truncate is much less, because the number of metadata blocks is greatly reduced for a given file size (unless the file is very fragmented). An alternative to the delete thread (for both files using extent maps as well as indirect blocks) would be to walk the inode and pre-compute the number of bitmaps and group descriptors that would be modified by the operation, and try to start a single transaction of that size. If this transaction can be started, then all of the indirect, double indirect, and triple indirect blocks (also referenced as [d,t] indirect blocks) no longer have to be zeroed out, and we only have to update the block bitmaps and their group summaries, reducing the amount of I/O considerably for files using indirect blocks. Also, the walking of the file metadata blocks can be done in forward order and asynchronous readahead can be started for indirect blocks to make more efficient use of the disk. As an added benefit, we would regain the ability to undelete files in ext3 because we no longer have to zero out all of the metadata blocks.

4.5 Increased nlinks support

The use of a 16-bit value for an inode's link count (`i_nlink`) limits the number of hard links on an inode to 65535. For directories, it

starts with a link count of 2 (one for "." and one for "..") and each subdirectory has a hard link to its parent, so the number of subdirectories is similarly limited.

The ext3 implementation further reduced this limit to 32000 to avoid signed-int problems. Before indexed directories were implemented, the practical limit for files/subdirectories was about 10000 in a single directory.

A patch was implemented to overcome this subdirectory limit by not counting the subdirectory links after the counter overflowed (at 65000 links actually); instead, a link count of one is stored in the inode. The ext3 code already ignores the link count when determining if a directory is full or empty, and a link count of one is otherwise not possible for a directory.

Using a link count of one is also required because userspace tools like "find" optimize their directory walking by only checking a number of subdirectories equal to the link count minus two. Having a directory link count of one disables that heuristic.

4.6 Parallel directory operations

The Lustre filesystem (which is built on top of the ext3 filesystem) has to meet very high goals for concurrent file creation in a single directory (5000 creates/second for 10 million files) for some of its implementations. In order to meet this goal, and to allow this rate to scale with the number of CPUs in a server, the implementation of parallel directory operations (pdirops) was done by Alex Tomas in mid 2003. This patch allows multiple threads to concurrently create, unlink, and rename files within a single directory.

There are two components in the pdirops patches: one in the VFS to lock individual entries in a directory (based on filesystem preference), instead of using the directory inode semaphore to provide exclusive access to the directory; the second patch is in ext3 to implement proper locking based on the filename.

In the VFS, the directory inode semaphore actually protects two separate things. It protects the filesystem from concurrent modification of a single directory and it also protects the dcache from races in creating the same dentry multiple times for concurrent lookups. The pdirops VFS patch adds the ability to lock individual dentries (based on the dentry hash value) within a directory to prevent concurrent dcache creation. All of the places in the VFS that would take `i_sem` on a directory instead call `lock_dir()` and `unlock_dir()` to determine what type of locking is desired by the filesystem.

In ext3, the locking is done on a per-directory-leaf-block basis. This is well suited to the directory-indexing scheme, which has a tree with leaf blocks and index blocks that very rarely change. In the rare case that adding an entry to the leaf block requires that an index block needs locking the code restarts at the top of the tree and keeps the lock(s) on the index block(s) that need to be modified. At about 100,000 entries, there are 2-level index blocks that further reduce the chance of lock collisions on index blocks. By not locking index blocks initially, the common case where no change needs to be made to the index block is improved.

The use of the pdirops VFS patch was also shown to improve the performance of the tmpfs filesystem, which needs no other locking than the dentry locks.

5 Performance comparison

In this section, we will discuss some performance comparisons between the ext3 filesys-

tem found on the 2.4 kernel and the 2.6 kernel. The goal is to evaluate the progress ext3 has made over the last few years. Of course, many improvements other than the ext3 specific features, for example, VM changes, block I/O layer re-write, have been added to the Linux 2.6 kernel, which could affect the performance results overall. However, we believe it is still worthwhile to make the comparison, for the purpose of illustrating the improvements made to ext3 on some workload(s) now, compared with a few years ago.

We selected linux 2.4.29 kernel as the baseline, and compared it with the Linux 2.6.10 kernel. Linux 2.6.10 contains all the features discussed in Section 2, except the EA-in-inode feature, which is not relevant for the benchmarks we had chosen. We also performed the same benchmarks using a Linux 2.6.10 kernel patched with Alex Tomas' extents patch set, which implements extents, delayed allocation, and extents-based multiple block allocation. We plan to run the same benchmarks against a Linux 2.6.10 kernel with some of the patches described in Section 4 in the future.

In this study we chose two benchmarks. One is tiobench, a benchmark testing filesystem sequential and random I/O performance with multiple running threads. Another benchmark we used is filemark, a modified postmark[8] benchmark which simulates I/O activity on a mail server with multiple threads mode. Filemark was used by Ray Bryant when he conducted filesystem performance study on Linux 2.4.17 kernel three years ago [3].

All the tests were done on the same 8-CPU 700 MHZ Pentium III system with 1 GB RAM. All the tests were run with ext3's writeback journaling mode enabled. When running tests with the extents patch set, the filesystem was mouted with the appropriate mount options to enable the extents, multiple block allocation, and delayed allocation features. These test runs

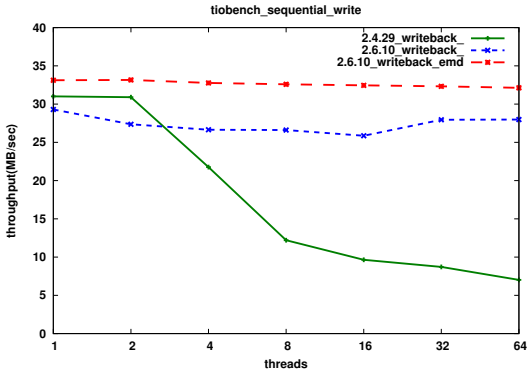


Figure 1: tiobench sequential write throughput results comparison

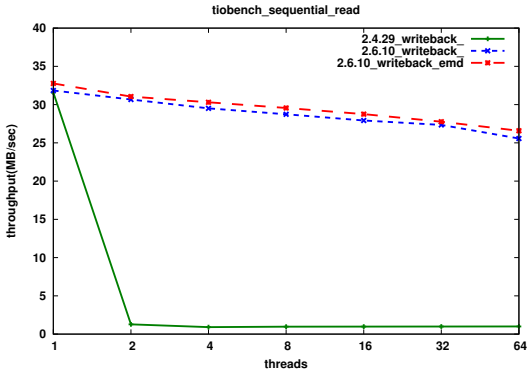


Figure 2: tiobench sequential read throughput results comparison

are shown as "`2.6.10_writeback_emd`" in the graphs.

5.1 Tiobench comparison

Although there have been a huge number of changes between the Linux 2.4.29 kernel to the Linux 2.6.10 kernel could affect overall performance (both in and outside of the ext3 filesystem), we expect that two ext3 features, removing BKL from ext3 (as described in Section 2.2) and reservation based block allocation (as described in Section 2.3) are likely to significantly impact the throughput of the tiobench

benchmark. In this sequential write test, multiple threads are sequentially writing/allocating blocks in the same directory. Allowing allocations concurrently in this case most likely will reduces the CPU usage and improves the throughput. Also, with reservation block allocation, files created by multiple threads in this test could be more contiguous on disk, and likely reduce the latency while writing and sequential reading after that.

Figure 1 and Figure 2 show the sequential write and sequential read test results of the tiobench benchmark, on the three selected kernels, with threads ranging from 1 to 64. The total files size used in this test is 4GB and the blocksize is 16348 byte. The test was done on a single 18G SCSI disk. The graphs indicate significant throughput improvement from the 2.4.29 kernel to the Linux 2.6.10 kernel on this particular workload. Figure 2 shows the sequential read throughput has been significantly improved from Linux 2.4.29 to Linux 2.6.10 on ext3 as well.

When we applied the extents patch set, we saw an additional 7-10% throughput improvement on tiobench sequential write test. We suspect the improvements comes from the combination of delayed allocation and multiple block allocation patches. As we noted earlier, having both features could help lay out files more contiguously on disk, as well as reduce the times to update the metadata, which is quite expensive and happens quite frequently with the current ext3 single block allocation mode. Future testing are needed to find out which feature among the three patches (extents, delayed allocation and extent allocation) is the key contributor of this improvement.

5.2 Filemark comparison

A Filemark execution includes three phases: creation, transaction, and delete phase. The

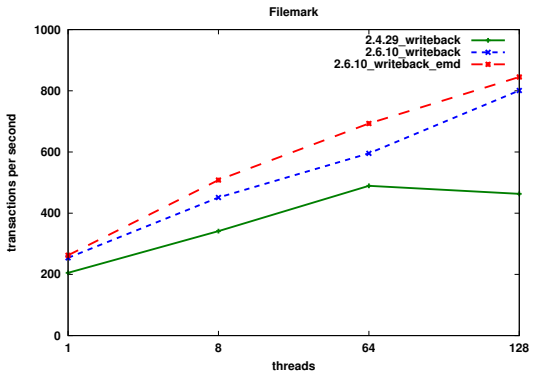


Figure 3: Filemark benchmark transaction rate comparison

transaction phase includes file read and append operations, and some file creation and removal operations. The configuration we used in this test is the so called “medium system” mentioned in Bryant’s Linux filesystem performance study [3]. Here we run filemark with 4 target directories, each on a different disk, 2000 subdirectories per target directory, and 100,000 total files. The file sizes ranged from 4KB to 16KB and the I/O size was 4KB. Figure 3 shows the average transactions per second during the transaction phase, when running Filemark with 1, 8, 64, and 128 threads on the three kernels.

This benchmark uses a varying number of threads. We therefore expected the scalability improvements to the ext3 filesystem in the 2.6 kernel should improve Linux 2.6’s performance for this benchmark. In addition, during the transaction phase, some files are deleted soon after the benchmark creates or appends data to those files. The delayed allocation could avoid the need for disk updates for metadata changes at all. So we expected Alex’s delayed allocation to improve the throughput on this benchmark as well.

The results are shown in Figure 3. At 128 threads, we see that the 2.4.29 kernel had sig-

nificant scalability problems, which were addressed in the 2.6.10 kernel. At up to 64 threads, there is approximately a 10% to 15% improvement in the transaction rate between Linux 2.4.29 and Linux 2.6.10. With the extents patch set applied to Linux 2.6.10, the transaction rate is increased another 10% at 64 threads. In the future, we plan to do futher work to determine how much of the additional 10% improvement can be ascribed to the different components of the extents patch set.

More performance results, both of the benchmark tests described above, and additional benchmark tests expected to be done before the 2005 OLS conference can be found at `http://ext2.sourceforge.net/ols05-testing`.

6 Future Work

This section will discuss some features that are still on the drawing board.

6.1 64 bit block devices

For a long time the Linux block layer limited the size of a single filesystem to 2 TB ($2^{32} *$ 512-byte sectors), and in some cases the SCSI drivers further limited this to 1TB because of signed/unsigned integer bugs. In the 2.6 kernels there is now the ability to have larger block devices and with the growing capacity and decreasing cost of disks the desire to have larger ext3 filesystems is increasing. Recent vendor kernel releases have supported ext3 filesystems up to 8 TB and which can theoretically be as large as 16 TB before it hits the 2^{32} filesystem block limit (for 4 KB blocks and the 4 KB `PAGE_SIZE` limit on i386 systems). There is also a page cache limit of 2^{32} pages in an address space, which are used for buffered block devices. This limit affects both ext3's internal metadata blocks, and the use of buffered block devices when running e2fsprogs on a device to create the filesystem in the first place. So this imposes yet another 16TB limit on the filesystem size, but only on 32-bit architectures.

However, the demand for larger filesystems is already here. Large NFS servers are in the tens of terabytes, and distributed filesystems are also this large. Lustre uses ext3 as the backend storage for filesystems in the hundreds of terabytes range by combining dozens to hundreds of individual block devices and smaller ext3 filesystems in the VFS layer, and having larger ext3 filesystems would avoid the need to artificially fragment the storage to fit within the block and filesystem size limits.

Extremely large filesystems introduce a number of scalability issues. One such concern is the overhead of allocating space in very large volumes, as described in Section 3.3. Another such concern is the time required to back up and perform filesystem consistency checks on very large filesystems. However, the primier issue with filesystems larger than 2^{32} filesystem blocks is that the traditional indirect block mapping scheme only supports 32-bit block numbers. The additional fact that filling such a large filesystem would take many millions of indirect blocks (over 1% of the whole filesystem, at least 160 GB of just indirect blocks) makes the use of the indirect block mapping scheme in such large filesystems undesirable.

Assuming a 4 KB blocksize, a 32-bit block number limits the maximum size of the filesystem to 16 TB. However, because the superblock format currently stores the number of block groups as a 16-bit integer, and because (again on a 4 KB blocksize filesystem) the maximum number of blocks in a block group is 32,768 (the number of bits in a single 4k block, for the block allocation bitmap), a combination of

these constraints limits the maximum size of the filesystem to 8 TB.

One of the plans for growing beyond the 8/16 TB boundary was to use larger filesystem blocks (8 KB up to 64 KB), which increases the filesystem limits such as group size, filesystem size, maximum file size, and makes block allocation more efficient for a given amount of space. Unfortunately, the kernel currently limits the size of a page/buffer to virtual memory's page size, which is 4 KB for i386 processors. A few years ago, it was thought that the advent of 64-bit processors like the Alpha, PPC64, and IA64 would break this limit and when they became commodity parts everyone would be able to take advantage of them. The unfortunate news is that the commodity 64-bit processor architecture, x86_64, also has a 4 KB page size in order to maintain compatibility with its i386 ancestors. Therefore, unless this particular limitation in the Linux VM can be lifted, most Linux users will not be able to take advantage of a larger filesystem block size for some time.

These factors point to a possible paradigm shift for block allocations beyond the 8 TB boundary. One possibility is to use only larger extent based allocations beyond the 8 TB boundary. The current extent layout described in Section 3.1 already has support for physical block numbers up to 2^{48} blocks, though with *only* 2^{32} blocks (16 TB) for a single file. If, at some time in the future larger VM page sizes become common, or the kernel is changed to allow buffers larger than the the VM page size, then this will allow filesystem growth up to 2^{64} bytes and files up to 2^{48} bytes (assuming 64 KB blocksize). The design of the extent structures also allows for additional extent formats like a full 64-bit physical and logical block numbers if that is necessary for 4 KB `PAGE_SIZE` systems, though they would have to be 64-bit in order for the VM to address files and storage devices this large.

It may also make sense to restrict inodes to the first 8 TB of disk, and in conjunction with the extensible inode table discussed in Section 6.2 use space within that region to allocate all inodes. This leaves the > 8 TB space free for efficient extent allocations.

6.2 Extensible Inode Table

Adding an dynamically extensible inode table is something that has been discussed extensively by ext2/3 developers, and the issues that make adding this feature difficult have been discussed before in [15]. Quickly summarized, the problem is a number of conflicting requirements:

- We must maintain enough backup metadata about the dynamic inodes to allow us to preserve ext3's robustness in the presence of lost disk blocks as far as possible.
- We must not renumber existing inodes, since this would require searching and updating all directory entries in the filesystem.
- Given the inode number the block allocation algorithms must be able to determine the block group where the inode is located.
- The number of block groups may change since ext3 filesystems may be resized.

Most obvious solutions will violate one or more of the above requirements. There is a clever solution that can solve the problem, however, by using the space counting backwards from $2^{31} - 1$, or "negative" inode. Since the number of block groups is limited by $2^{32}/(8 * \mathit{blocksize})$, and since the maximum number of inodes per block group is also the same as the maximum number of blocks per block group

is $(8 * blocksize)$, and if inode numbers and block numbers are both 32-bit integers, then the number of inodes per block group in the "negative" inode space is simply $(8 * blocksize)$ - *normal-inodes-per-blockgroup*. The location of the inode blocks in the negative inode space are stored in a reserved inode.

This particular scheme is not perfect, however, since it is not extensible to support 64 bit block numbers unless inode numbers are also extended to 64 bits. Unfortunately, this is not so easy, since on 32-bit platforms, the Linux kernel's internal inode number is 32 bits. Worse yet, the `ino_t` type in the `stat` structure is also 32 bits. Still, for filesystems that are utilizing the traditional 32 bit block numbers, this is still doable.

Is it worth it to make the inode table extensible? Well, there are a number of reasons why an extensible inode table is interesting. Historically, administrators and the `mke2fs` program have always over-allocated the number of inodes, since the number of inodes can not be increased after the filesystem has been formatted, and if all of the inodes have been exhausted, no additional files can be created even if there is plenty of free space in the filesystem. As inodes get larger in order to accommodate the `EA-in-inode` feature, the overhead of over-allocating inodes becomes significant. Therefore, being able to initially allocate a smaller number of inodes and adding more inodes later as needed is less wasteful of disk space. A smaller number of initial inodes also makes the the initial `mke2fs` takes less time, as well as speeding up the `e2fsck` time.

On the other hand, there are a number of disadvantages of an extensible inode table. First, the "negative" inode space introduces quite a bit of complexity to the inode allocation and read/write functions. Second, as mentioned earlier, it is not easily extensible to filesystems that implement the proposed 64-bit block number extension. Finally, the filesystem becomes more fragile, since if the reserved inode that describes the location of the "negative" inode space is corrupted, the location of all of the extended inodes could be lost.

So will extensible inode tables ultimately be implemented? Ultimately, this will depend on whether an ext2/3 developer believes that it is worth implementing—whether someone considers extensible inode an "itch that they wish to scratch." The authors believe that the benefits of this feature only slightly outweigh the costs, but perhaps not by enough to be worth implementing this feature. Still, this view is not unanimously held, and only time will tell.

7 Conclusion

As we have seen in this paper, there has been a tremendous amount of work that has gone into the ext2/3 filesystem, and this work is continuing. What was once essentially a simplified BSD FFS descendant has turned into an enterprise-ready filesystem that can keep up with the latest in storage technologies.

What has been the key to the ext2/3 filesystem's success? One reason is the forethought of the initial ext2 developers to add compatibility feature flags. These flags have made ext2 easily extensible in a variety of ways, without sacrificing compatibility in many cases.

Another reason can be found by looking at the company affiliations of various current and past ext2 developers: Cluster File Systems, Digeo, IBM, OSDL, Red Hat, SuSE, VMWare, and others. Different companies have different priorities, and have supported the growth of ext2/3 capabilities in different ways. Thus, this diverse and varied set of developers has allowed the ext2/3 filesystem to flourish.

The authors have no doubt that the ext2/3 filesystem will continue to mature and come to be suitable for a greater and greater number of workloads. As the old Frank Sinatra song stated, "The best is yet to come."

Patch Availability

The patches discussed in this paper can be found at `http://ext2.sourceforge.net/ols05-patches`.

Acknowledgments

The authors would like to thank all ext2/3 developers who make ext2/3 better, especially grateful to Andrew Morton, Stephen Tweedie, Daniel Phillips, and Andreas Gruenbacher for many enlightening discussions and inputs.

We also owe thanks to Ram Pai, Sonny Rao, Laurent Vivier and Avantika Mathur for their help on performance testing and analysis, and to Paul Mckenney, Werner Almesberger, and David L Stevens for paper reviewing and refining. And lastly, thanks to Gerrit Huizenga who encouraged us to finally get around to submitting and writing this paper in the first place.`:-)`

References

[1] ALMESBERGER, W., AND VAN DEN BRINK, B. Active block i/o scheduling systems (abiss). In *Linux Kongress* (2004).

[2] BLIGH, M. Re: 2.5.70-mm1, May, 2003. `http://marc.theaimsgroup.com/?l=linux-mm&m=105418949116972&w=2`.

[3] BRYANT, R., FORESTER, R., AND HAWKES, J. Filesystem performance and scalability in linux 2.4.17. In *USENIX Annual Technical Conference* (2002).

[4] CARD, R., TWEEDIE, S., AND TS'O, T. Design and implementation of the second extended filesystem. In *First Dutch International Symposium on Linux* (1994).

[5] CORBET, J. Which filesystem for samba4? `http://lwn.net/Articles/112566/`.

[6] DILGER, A. E. Online resizing with ext2 and ext3. In *Ottawa Linux Symposium* (2002), pp. 117–129.

[7] GRUENBACHER, A. Posix access control lists on linux. In *USENIX Annual Technical Conference* (2003), pp. 259–272.

[8] KATCHER, J. Postmark a new filesystem benchmark. Tech. rep., Network Appliances, 2002.

[9] MORTON, A. Re: [ext2-devel] [rfc] adding 'delayed allocation' support to ext3 writeback, April, 2005. `http://marc.theaimsgroup.com/?l=ext2-devel&m=111286563813799&w=2`.

[10] PHILLIPS, D. A directory index for ext2. In *5th Annual Linux Showcase and Conference* (2001), pp. 173–182.

[11] RAO, S. Re: [ext2-devel] re: Latest ext3 patches (extents, mballoc, delayed allocation), February, 2005. `http://marc.theaimsgroup.com/?l=ext2-devel&m=110865997805872&w=2`.

[12] SWEENY, A. Scalability in the xfs file system. In *USENIX Annual Technical Conference* (1996).

[13] TOMAS, A. Speeding up ext2, March, 2003. `http://lwn.net/Articles/25363/`.

[14] TWEEDIE, S. Ext3 journalling filesystem. In *Ottawa Linux Symposium* (2000).

[15] TWEEDIE, S., AND TS'O, T. Y. Planned extensions to the linux ext2/3 filesystem. In *USENIX Annual Technical Conference* (2002), pp. 235–244.

[16] VIVIER, L. Filesystems comparison using sysbench and mysql, April, 2005. `http://marc.theaimsgroup.com/?l=ext2-devel&m=111505444514651&w=2`.

Legal Statement

Using a the Xen Hypervisor to Supercharge OS Deployment

Mike D. Day
International Business Machines
ncmike@us.ibm.com

Ryan Harper
International Business Machines
ryanh@us.ibm.com

Michael Hohnbaum
International Business Machines
hohnbaum@us.ibm.com

Anthony Liguori
International Business Machines
aliguori@us.ibm.com

Andrew Theurer
International Business Machines
habanero@us.ibm.com

Abstract

Hypervisor technology presents some promising opportunities for optimizing Linux deployment. By isolating a server's unique properties into a set of patches to initialization scripts and other selected files, deployment of a new server will be demonstrated to occur in a few seconds by creating a new Xen domain, re-using an existing file system image and applying patches to it during domain initialization. To capture changes to a server's configuration that occur while it is running, the paper discusses the potential of copy-on-write file systems to hold changes to selected files. By separating the initialization and file data that make a linux server instance unique, that data can be stored and retrieved in a number of ways. The paper demonstrates how to store and retrieve different initialization patches over the network and integrate these capabilities into the Xen tools. Potential uses for the techniques demonstrated in the paper include capacity on demand, and new methods of provisioning servers and workstations.

1 Introduction

Virtual machine technology is rapidly becoming ubiquitous for commodity processors. Commercial product has established a foothold in this space. Open Source products are emerging and maturing at a rapid pace. This paper demonstrates how the use of virtualization technology can improve deployment and maintenance of Linux servers.

The virtualization technology used for this paper is Xen, an open source hypervisor developed at the University of Cambridge[1]. Xen supports para-virtualized guests, that is operating systems are modified to run in domains on top of Xen.

[1] http://www.cl.cam.ac.uk/Research/SRG/netos/xen/

All I/O devices are owned by one or more privileged domains. Typically the first domain to be created (called domain 0), but other domains may have control over one or more I/O device. The privileged domain runs a kernel that is configured with regular device drivers. The privileged domain initializes and services I/O hardware.

Block, network, and USB devices are virtualized by the privileged domain. Backend device drivers run in the privilege domain to provide a bridge between the physical device and the user domains. Front end virtual device drivers execute in user domains and appear to Linux as a regular device driver.

While Xen includes management and control tools (`xend` and others), an alternate toolset, vmtools[2], is used for the work discussed in this paper. vmtools is a re-implementation in "C" of the Xen toolset, which is implemented in python. vmtools provides the capabilities needed to configure domains.

`vmtools` consists of a daemon, `xenctld`; a set of command line tools, `vm-*`; and `vmm`—a script that provides a more user-friendly frontend to the vmtools. vmtools provides commands for creating a domain, assigning resources to the domain, starting and stopping a domain, querying information about domains. The tools are modular, provide ease of use within scripts, and are easy to modify and extend.

vmtools are used to demonstrate the flexibility of the Xen architecture by showing it can be controlled by multiple toolsets, and also as a vehicle for extending the Xen configuration syntax[3].

[2] `http://www.cs.utexas.edu/users/aliguori/vm-tools-0.0.9a.tar.gz`

[3] `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/user/user.html`

2 DEPLOYMENT OVERVIEW

Deployment is the provisioning of a new operating system and associated configuration for a unique instance of a computer system. Throughout this paper a unique instance of an operating system is referred to as a system image. Traditionally, each computer system has one system image deployed on it. With virtualization technology, each computer system may have one to many system images deployed, each executing within its own virtual machine environment.

Related to deployment is maintenance. After a system image is established, it must be maintained. Software components must be updated (for example, replaced with new versions) to address security problems, provide new funtionality, or correct problems with existing software. Sometimes this involves replacing one component, a subset of the overall software components, or a complete replacement of all operating system software. Similarly, application software and middleware needs to be maintained.

Data centers have numerous computer systems, and numerous system images. To keep things manageable, most datacenters strive to keep system images as common as possible. Thus, it is common practice to choose one specific version of an operating system and deploy that on all (or a large percentage of) the system images.

2.1 Deployment Tasks

Deploying a new system image involves:

- Configuring the physical (or virtual) machine, such as processor count, physical memory, I/O devices

- Installing the operating system software, such as kernel configuration (smp vs up, highmem, and so on), device drivers, shells, tools, documentation, and so on.
- Configuring the operating system (such as hostname, network parameters, security, and so on).
- Creating user accounts
- Installing application software
- Configuring application environment

2.2 Current Deployment Methods

There are different ways to deploy multiple copies of the same system. These include manual deployment, use of a higher-level installation tool for example kickstart, and installation customization then cloning.

2.2.1 Manual

The most basic mechanism is to do a manual install from the same installation media to each system image. This method is time consuming and can be error prone (as the system administrator must execute a series of steps and with repetition is inclined to miss a step or make a subtle variation in the process that can have unforseen consequences).

2.2.2 Kickstart

Kickstart[4] is a tool provided by Red Hat that enables repeating a system install with identical parameters. In effect, all questions that are normally asked by the system installer are answered in advanced and saved in a configuration file. Thus, identical system images may be installed on multiple machines with reasonable automation.

2.2.3 YaST Auto Installer

AutoYaST[5] functions according to the same principal as Kickstart. Configuration and deployment of the platform is driven by a configuration file, and the process can be repeated (with configuration changes) for multiple deployments.

2.2.4 Clone/Customize

Another install method is to clone an installed system and customize the resulting system image. In many cases a clone operation, which consists of copying the contents of the original installed root file system, is quicker than going through the complete install process. After the clone operation, system image specific customization is then performed. For example, setting hostname.

3 IMPROVEMENTS AVAILABLE THROUGH VIRTUALIZATION

Virtualization technology provides opportunities to improve deployment mechanisms. Improved aspects of deployment include:

- normalization of hardware configuration
- dynamic control over hardware configuration

[4] `http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/custom-guide/part-install-info.html`.

[5] `http://yast.suse.com/autoinstall/ref.html`.

- omission of hardware discovery and probing
- use of virtual block devices (VBD)
- file system reuse
- virtual networking (VLAN)

3.1 Dynamic Control Of Hardware Configuration

Without virtualization, changing the number of CPUs available, the amount of physical memory, or the types and quantity of devices requires modifying the physical platform. Typically this requires shutting down the system, modifying the hardware resources, then restarting the system. (It may also involve rebuilding the kernel.)

Using virtualization, resources can be modified through software control. This makes it possible to take disparate hardware, and still create normalized virtual machine configurations, without having to physically reconfigure the machine. Further, it provides the capability of redefining virtual machines with more resources available to address capacity issues.

For example, Xen allows you to add and remove processors, network, and block devices from and to user domains by editing a configuration file and running a command-line utility. No kernel configuration is necessary, and you don't need to shut down the physical computer. This operation can be repeated as often as necessary.

In addition to the advantages in deploying and maintaining Linux systems, dynamic hardware configuration makes more advanced workload management applications easier to implement.

3.2 Virtual Block Devices

Xen privileged domains virtualize block devices by exporting virtual block devices (VBD) to domU's. Any block device accessible by Linux can be exported as a VBD. As part of the process of setting up a VBD, the system administrator specifies the device path the VBD should appear to the domU as. For example `/dev/sda1` or `/dev/hda5`. Disk partitions may be exported this way, or a VBD may be backed by a file in dom0's file system.

Virtual block devices provide two benefits to deployment and maintenance of Linux servers. First, they provide hardware normalization as described above. (Every domain can have an identical fstab, for example). Secondly, VBDs make the reuse of file systems with Xen domains exceedingly simple, even for read/write file systems.

3.3 Virtual Networking

Xen privileged domains virtualize network devices in a manner similar to VDBs. The privileged domain kernel initializes network interfaces and starts networking services just as a normal kernel does. In addition, Xen privileged domains implement a virtual LAN and use the Xen network back end (netback) driver to export virtual network interfaces to user domains.

User domains import virtualized network interfaces as "devices," usually `eth0...ethN`. The virtualized eth0, for example, is really a stub that uses Xen inter-domain communication channels to communicate with the netback driver running in a privileged domain. Finally, the Xen privileged domain bridges virtualized network interfaces to the physical network using standard Linux bridge tools.

The most common practice is to use private IP addresses for all the virtual network interfaces

and then bridge them to a physical network interface that is forwarded using Network Address Translation (NAT) to the "real world."

A significant benefit of this method for deployment and maintenance of servers is that every server can have identical network configurations. For example, every user domain can have the same number of network interfaces and can use the same IP configuration for each interface. Each server can use the bridging and NAT forwarding services of the privileged domain to hide their private addresses. Note that bridging without NAT is also a common practice, and allows user domains to host externally visible network interfaces.

3.4 File System Reuse

Xen's Virtual Machine technology can export file systems and file images to virtual machines as devices. Sharing file systems among Linux platforms is a time-honored technique for deploying Linux servers, and virtual machine technology simplifies the sharing of file systems.

File System reuse is an especially helpful technique for deploying and maintaining Linux systems. The vast majority of the time spent deploying a new Linux system is spent creating and populating the file systems.

Re-using read-only file systems is exceedingly simple in Xen. All you have to do is export the file system as a device to Xen. For example, the line `disk = ['file:/var/images/xen_usr,sda1,r']` causes the file system image `/var/images/xen_usr` to be exported to the user domain as `/dev/sda1`. (All configuration commands are relative to the privilged domain's view of the world.) Because this is a read-only file system you don't need to do anything special to synchronize access among domians.

In addition to file system images, the Xen domain configuration syntax allows you to export both physical devices and network file systems as devices into the new domain. A future version of Xen will the exporting of a VFS directory tree to a Xen domain as a device.

Read/write file systems are not as easy to share among domains because write access must be synchronized among domains. There are at least three ways to do this:

- Use a storage server that provides external, sychronized shared storage. There is a range of systems that have this capability.
- Use a copy-on-write file system. One such file system is unionfs.[6]
- "Fork" an existing file system by duplicating it for each new domain. This is a simple and expedient (if not efficient) way to re-use read-write file systems.

The Logical Volume Manager (LVM)[7] has an interesting snapshot capability that was designed primarily to support hot backups of file systems, but which could evolve into a copy-on-write file system appropriate for use with Xen.

One problem with re-use of read-write file systems is that they usually contain configuration files that are specific to an individual Linux system. For example, `/etc` on a Linux system contains most of the uniqueness of a system. If you are going to re-use an `/etc` file system, you need an automated way to "fork" and modify it. Fortunately the typical system does not

[6] `http://www.fsl.cs.sunysb.edu/project-unionfs.html`

[7] `http://www.tldp.org/HOWTO/LVM-HOWTO/index.html`.

need a vast number of changes in `/etc` and as a result it is possible to automate the "forking" process. Later this paper discusses some tools we have developed to automate the creation, modification, and exporting of file systems under Xen.

4 EXPLOITING XEN TO DEPLOY SERVERS

A variation of the clone and modify approach is proposed to deploy Linux on Xen virtual machines. In addition, an extended configuration syntax and Xen deployment tool is proposed to integrate the deployment process with Xen domain creation. This approach uses Xen to improve on the existing clone methods in the following ways:

- Xen allows exporting of VBDs to domains, where they appear as virtual devices, such as SCSI or IDE drives. This is an improvement over cloning a file system image to a bare-metal server.
- Xen allows the exporting of NFS volumes as virtual devices. This privides a file system with some of the same advantages as VBDs.
- Xen allows provides control over the "hardware" environment of each new server. By exporting specific devices to the new domain, it is not necessary to accomodate all the possible hardware configurations when deploying a new server. For example, all domains within an organization may appear to have only SCSI block devices, despite variation in the underlying physical hardware.

4.1 Deploying Application Stacks

The flexibility to export specific file systems to the new partitions means that it is much easier to deploy new servers for specific applications. For example, a file system image can be prepared with a complete DBMS stack. When a new data base server is needed, a Xen domain can be created using the DBMS file system images. In this case, Xen can export the DBMS image to the new domain. The new domain can and mount the image read-only as `/opt/dbms/`. Exporting of pre-built file systems as virtual devices to Xen domains simplifies the deployment of application-specific servers.

4.2 Xen Deployment Methodology

The general methodology used is to create a handful of "canned" file systems that can be mixed-and-matched to create new Xen domains by exporting them as VDBs or NFS mounts. For example, `/usr` and `/bin` as standard read-only file systems; `/etc` as a read/write file system that needs to be preprocessed; `/var/` and `/home` as read-write file systems that need COW or snapshot capability; Variations of `/opt` for specific application stacks, and so on.

Extending vmtools to support integrated deployment and domain creation requires some new configuration properties for domains, as well as some shell scripts to perform preprocessing on the images to customize them (when necessary) for each domain.

The "Xen Domain Container" is comprised of the following:

- An overall configuration file for the new domain. This is an extended version of

the existing domain configuration file used by the `vmm` command. The extensions include information about the domain's VDB or NFS file systems and how they should be processed by vmtools prior to domain creation. The extended-syntax configuration file is called a "container file."

- File System Images. Each image consists of a file system stored in a compressed cpio archive (just as initrd). In addition, each file system image has metadata in the container file for the file system and processing instructions for vmtools. The metadata and processing instructions describe characteristics of the file system including where it should be mounted by the new domain, whether it should be read-only or read-write, and how it needs to be customized for each new domain.

  For example, a file system that is to be mounted by the new domain as `/etc` needs to be customized for each new domain. The `/etc` file system includes the sysinit data and configuration files, plus user and group accounts, file system mounting, hostnames, terminal configuration, etc.

- Init hooks. Each file system can include shell scripts that will be driven by a configuration file, also in that file system. The idea is to have vmtools preprocess the file system, then mount it on a device (or export it using NFS). During domain startup, the initrd/init process looks for a "post processing" shell script and executes the script on the mounted file system. Depending upon the context of the init process, it may remount file systems and execute a pivot-root and restart the init process.

4.3 Composing a Xen Container

An important goal is to make composing and maintaining Xen domain containers as simple as possible. The container file may contain standard Xen domain configuration statements in addition to "container" syntax. Both types of statements (standard Xen configuration and container) may be intermixed throughout the file.

The container syntax refers to file system images using URIs. Each URI may point to a file system image stored locally, as in `file:///var/images/etc.cpio.gz`; or remotely, as in `http://foo.org/images/etc.cpio.gz`. This reference syntax has two important advantages:

- Simplification of file system deployment. Using a URI reference for each file system image allows the administrator to keep a canonical image on a network server. When starting the domain, vmtools will follow the URI and download the file system and perform pre-processing on a local copy. The tools follow this process for each URI reference configured for use by the domain.

- Simplification of file system maintenenace. For read-only file systems that contain applications, such as `/bin`, `/sbin`, and `/usr`, applying updates and patches comprise a large percentage of the administrator's time. The URI reference allows the administrator to patch or update the cononical, network-resident file system image. Domains can be configured to retrieve their file system images every time they start. A more advanced design would provide a way for the domain initialization to check for recent updates to its file system images.

4.3.1 Domain Customization

Domain customization involves applying modifications to Linux configuration files residing within a file system image. After retrieving a file system image, vmtools can mount and modify the image before starting the Xen domain.

The container syntax provides three different methods for modifying files within a file system image:

- File replacement. This mechanism causes vmtools to replace the content of a file with text embedded in the configuration file itself. The conainter syntax for file replacement is shown in Figure 1.

  This simple expression in Figure 1 will cause vmtools to retrieve the file system archive at `http://images.xen.foo.org/etc.cpio.gz`, expand the archive, and replace the file /etc/HOSTNAME with a new file. The new file will contain a single line, "FCDOBBS." If `/etc/HOSTNAME` does not exist, it will be created.

  There are additional options in the file replacement syntax to create a patch file by comparing the original and modified file systems, and to "fork" the archive by creating a new copy (with the modifications)

  vmtools makes some simple attempts to be efficient. It will only retrieve and expand file system image once per invocation. Thereafter, it will use a locally expanded copy. The creator of the container file can order expressions so that the file system is forked only after it has been completely processed.

  The remaining methods for modifying files follow the same patterns as the replacement method.

- File copy. This mechanism causes vmtools to retrieve a file and copy the retrieved file over an existing file.

- File system patching. This mechanism retrieves a patch and then applies the patch to the file system.

4.3.2 Steps to Compose a Xen "Container"

Composing a Xen container, then, involves:

- Preparing file system images. This step only needs to be performed initially, after which you can use the same file system images repeatedly to deploy further Linux domains. The tools discussed in this paper provide commands that automate file system preparation. (Remember, a file system image is simply a compressed cpio archive).

- Creating the container file. The container file defines the new domain, including the location of the kernel, the amount of memory, the number of virtual processors, virtual block devices, virtual ethernets, and so on. The proposed container expressions prepare, retrieve, and process file system images for use by the new domain.

All information describing the domain is present in the container file: resources, devices, kernel, and references to file systems. Further, the container file includes processing instructions for each file system, with the ability to retrieve updated file systems whenever the domain is started. This collection of information is referred to as a "domain container" because it is self-contained and portable from one xen platform to another.

At the present time one container file must be created for each domain. However, because

CONTAINER SYNTAX FOR FILE REPLACEMENT

```
[replace /etc/HOSTNAME
 archive http://foo.org/images/etc.cpio.gz

 FCDOBBS

] [end]
```

Figure 1: Container Syntax for File Replacement. This simple example shows the `/etc/HOSTNAME` file being replaced with one text line containing "FCDOBBS."

most of the configuration syntax (including the extensions we propose) is boilerplate, there are improvements which will allow reuse of a container template to control the deployment and maintenance of multiple domains.

To complete the deployment, you must process the domain container using `vm-container`, as shown in Figure 2. This example is assumed to be running as a user process in the Xen Domain0 virtual machine. Domain0 is always the first domain to run on a Xen platform, and it is created implicitly by Xen at boot time.

The command in Figure 2 parses the container file `my-domain` and processes all the extended-syntax expressions within that file. It also produces the standard Xen configuration file `my-config`. Output is logged to `/var/log/domain`, and `/var/images` is used as the working directory for processing file system images.

At this point all that's left is to start the domain using `vmm create my-config`.

4.4 Xen Container Syntax

The Xen container syntax is a superset of "standard" Xen configuration syntax. Using the standard Xen syntax you can define the domain boot kernel and boot parameters, the amount of memory to allocate for the domain, which network and disk devices to virtualize, and more. The expressions discussed below are *in addition* to the standard Xen syntax and both types of expressions may be mingled in the same container file.

The Xen container syntax will expand as further experience using it to deploy Linux systems is gained. The syntax is presently complete enough to manage the creation, deployment, and maintenance of Xen domains, including the composition and reuse of file system images.

The Xen container syntax is explained below using examples. In actual use, the container file will have a number of container expressions. The `vm-container` parser only makes one pass through the container file and it processes each expression in the order it is declared within the file. Dependent expressions, such as a `populate` expression which refers to an archive instantiated by a `create` expression, must be in the correct order.

4.5 Creating a File System Image

A file system image for a Xen container can be created from any existing file system. For example, the expression

PROCESSING THE DOMAIN CONTAINER

```
vm-container --container my-domain \
--stripped my-config --log /var/log/domain \
--dir /var/images
```

Figure 2: Processing the Domain Container

```
[create
 /etc/
ftp://foo.org/images/etc.cpio.gz
][end]
```

will create a compressed `cpio` archive out of the contents of the local `/etc/` directory tree. It will then store that archive using `ftp` to the URI `ftp://foo.org/images/etc.cpio.gz`

4.6 Creating a Sparse File System

Loopback devices are especially convenient to use with Xen. The `image` expression will cause `vm-container` to create a sparse file system image, formatted as an `ext3` volume.

```
[image /var/images/fido-etc
 50MB
fido_etc] [end]
```

This example will cause `vm-container` to create a sparse 50 MB file system image at `/var/images/fido-etc`. The file system will be formatted and labelled as `fido-etc`.

4.7 Populating a File System Image

Any type of volume (LVM, NFS, loopback, or physical device) exported to a Xen domain needs to have a formatted file system and be populated with files. The `populate` expression will make it happen.

```
[populate image
/var/images/fido-etc
/mnt/
ftp://foo.org/images/etc.cpio.gz
][end]
```

The example above will cause `vm-container` to mount the file system `/var/images/fido-etc` to `/mnt` using a loopback device. It will then retrieve the archive `ftp://foo.org/images/etc.cpio.gz`, expand the archive into `/mnt`, `sync`, `umount`, and delete the loop device.

4.8 Replacing and Copying

Figure 1 shows an example of replacing a specific file within a file system. The `replace` expression can also be used to generate a `diff` file that isolates the modifications made to the file system. It can also create an new file system archive based on the modifications.

The `copy` expression is almost identical to `replace`, except that it retrieves whole files using URI references and copies those file into the file system being modified. It also supports patch generation and forking.

4.9 Patching a File System

The `replace` and `copy` expressions can both generate a patch file that isolates modifications

to a file system. Once that patch file is created, you can use it repeatedly to modify file systems during domain initialization.

```
[patch
file://var/ images/fido-etc
ftp://foo.org/ images/fido-etc.patch 1
file:///var/ images/fido-etc-patched
][end]
```

This example will retrieve a patch file from `ftp://foo.org/images/fido-etc.patch1`. It will then expand and patch the file system image at `/var/images/fido-etc`. It will then "fork" the file system by saving a patched archive at `file:///var/images/fido-etc-patched`.

4.10 Forking File Systems

While each of the `replace`, `copy`, and `patch` expressions will "fork" the file system, doing so should only occur after that file system had undergone all the modifications indicated by the container file. The statement that causes the file system to be copied and stored is always optional.

5 Further Work

The notion of using a hypervisor to supercharge OS deployment is valuable and warrants further development effort. In particular, the integration of file system image customization with Xen management and control tools proved very successful. The concept of capturing the unique personality of a domain as a set of changes to file system images was straightforward and familiar, and it worked as expected. A number of files were successfully patched during domain initialization, including the `/etc/passwd`, `/etc/shadow`, and `/etc/groups`. These last three examples show how user accounts and group member can be modified during domain initialization.

Patching user accounts and authorization data during domain initialization is dangerous, especially since our tools retrieved patchfiles over the network. High on the list of further work is generation and verification of cryptographic signatures for all file system images and difference files. It would also be prudent to generate and verify signatures for the extended configuration file.

While modifying file systems during domain initialization from Domain 0's userspace was very reliable, mixed success was achieved when modifying file systems during the kernel init process. Sometimes patches were successful but usually the patches failed or the init process died and was respawned. Continued experimentation with the init process as a vehicle for domain customization is warrented.

5.0.1 LVM

LVM has great potential to augment the approach to domain deployment. In fact, it is already a great tool for use with virtual machines. The LVM snapshot capability, while design for hot backups, works as a COW file system but needs to be evaluated further with this particular use model in mind.

6 Legal Statement

This work represents the views of the authors and does not necessarily represent the view of IBM.

Active Block I/O Scheduling System (ABISS)

Giel de Nijs & Benno van den Brink
Philips Research
{giel.de.nijs,benno.van.den.brink}@philips.com

Werner Almesberger
werner@almesberger.net

Abstract

The Active Block I/O Scheduling System (ABISS) is an extension of the storage subsystem of Linux. It is designed to provide guaranteed reading and writing bit rates to applications, with minimal overhead and low latency.

In this paper, the various components of ABISS as well as their actual implementation are described. This includes work on the Linux elevator and support for delayed allocation.

In a set of experimental runs with real-life data we have measured great improvements of the real-time response of read and write operations under heavy system load.

1 Introduction

As storage space is getting cheaper, the use of hard disk drives in home or mobile consumer devices is becoming more and more mainstream. As this class of devices like HDD video recorders, media centers and personal audio and video players were originally intended to be used by one person at a time (or by multiple persons, but watching the same content), performance of the hard disk drives was not a real issue. Adding more video sources to such a device (more tuners, for instance), however, will strain the storage subsystem by demanding the recording of multiple streams simultaneously. As these devices are being enabled with connectivity options and become interconnected through home networks or personal area networks, a device should also be able to serve a number of audio or video streams to multiple clients. For example, a media center should be able to provide a number of so-called media extenders or renderers throughout the house with recorded content. Putting aside high bit rate tasks, even simple low-end devices could benefit from a very low latency storage system.

Consumer electronics (CE) equipment has to consist of fairly low-cost hardware and often has to meet a number of other constraints like low power consumption and low-noise operation. Devices serving media content should therefore do this in an efficient way, instead of using performance overkill to provide their soft-real-time services. To be able to accomplish this sharing of resources in an effective way, either the applications have to be aware of each other or the system has to be aware of the applications.

In this paper we will present the results of work done on the storage subsystem of Linux, re-

sulting in the *Active Block I/O Scheduling System* (ABISS). The main purpose of ABISS is to make the system application-aware by either providing a guaranteed reading and writing bit rate to any application that asks for it or denying access when the system is fully committed. Apart from these guaranteed real-time (RT) streams, our solution also introduces priority-based best-effort (BE) disk traffic.

The system consists of a framework included in the kernel, with a policy and coordination unit implemented in user space as daemon. This approach ensures separation between the kernel infrastructure (the framework) and the policies (e.g. admission control) in user space.

The kernel part consists mainly of our own *elevator* and the ABISS *scheduler*. The elevator implements I/O priorities to correctly distinguish between real-time guaranteed streams and background best-effort requests. The scheduler is responsible for timely preloading and buffering of data. Furthermore, we have introduced an alternative allocation mechanism to be more effectively able to provide real-time writing guarantees. Apart from these new features, some minor modifications were made to file system drivers to incorporate our framework. ABISS supports the FAT, ext2, and ext3 filesystems.

ABISS works from similar premises as RTFS [1], but puts less emphasis on tight control of low-level operations, and more on convergence with current Linux kernel development.

In Section 2 a general overview of the ABISS architecture is given. Section 3 describes the steps involved in reading and explains the solutions incorporated in ABISS to control the involved latencies. The same is done for the writing procedure in Section 4. Performance measurements are presented in Section 5, followed by future work in Section 6 and the conclusions in Section 7.

The ABISS project is hosted at `http://abiss.sourceforge.net`.

2 Architecture

An application reading or writing data from a hard drive in a streaming way needs timely availability of data to avoid skipping of the playback or recording. Disk reads or writes can introduce long and hard-to-predict delays both from the drive itself as well as from the various operating system layers providing the data to the application. Therefore, conventionally a streaming application introduces a relatively large buffer to bridge these delays. The problem however is that as the delays are theoretically unbounded and can be quite long in practice (especially on a system under heavy load), the application cannot predict how much buffer space will be needed. Worst-case buffering while reading means loading the whole file into memory, while a worst-case write buffer should be large enough to hold all the data which is being written to disk.

2.1 Adaptive buffering

If I/O priorities are introduced and thus the involved delays become more predictable, an adaptive buffering scheme may be a useful approach. The adaptive algorithm can compensate for disk latency, system speed and various other variables. Still, an application will need to know how much competition it will face and what the initial parameters should be. Also, the algorithm would need some way to correctly dimension the buffer to be able to sustain some background activity.

Furthermore, some fairness against lower-priority I/O should be maintained. If any application can raise its priority uncontrolled, best-effort traffic can be completely starved. Too

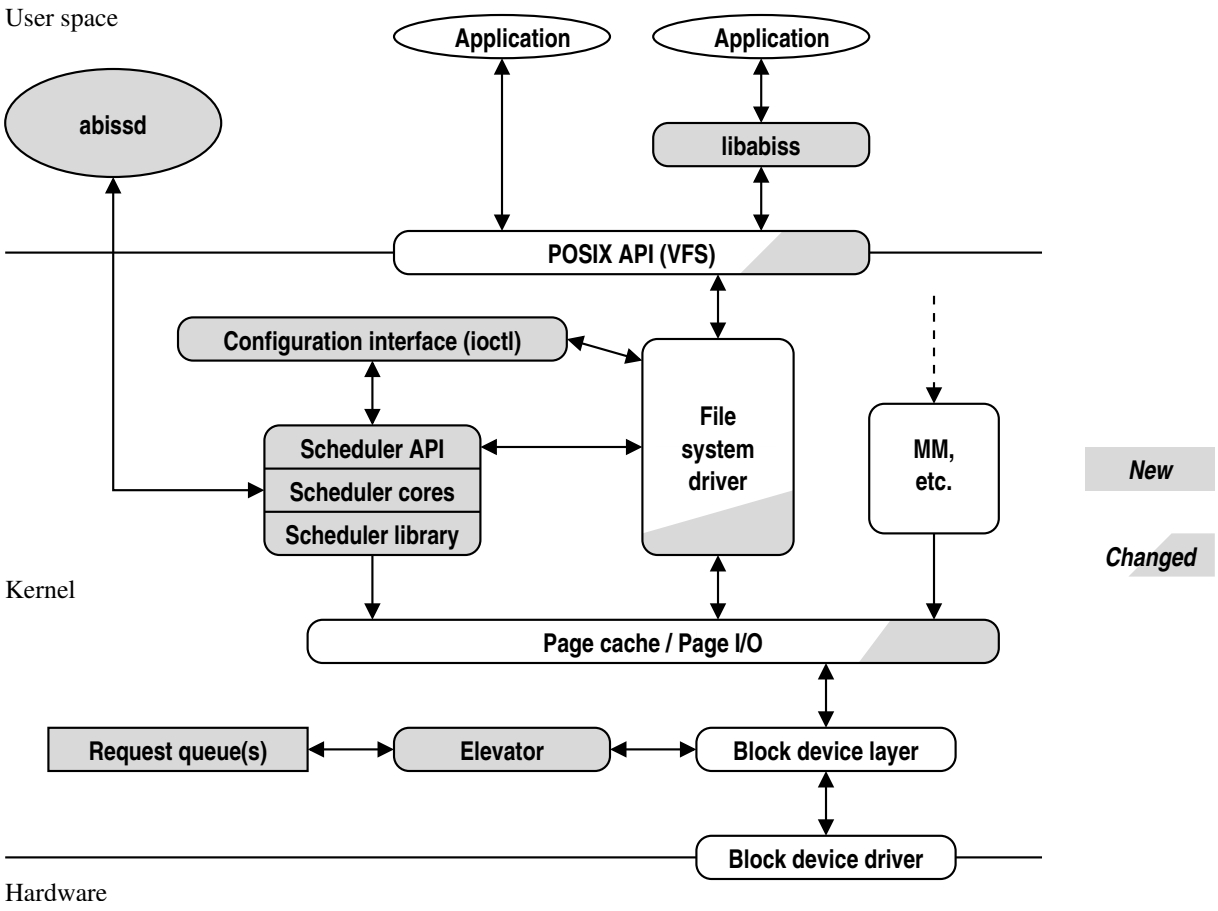


Figure 1: Global ABISS architecture layout.

many applications doing too much I/O at a high priority can also result in unbounded delays for those applications, simply because there are not enough system resources available. Clearly, admission control is needed.

ABISS implements such an adaptive buffering algorithm as a service for streaming applications on a relatively coarse time scale; buffer sizes are determined when the file is opened and may be adapted when the real-time load changes (i.e., when other high-priority files are opened). It makes use of elevated I/O priorities to be able to guarantee bounded access times and a real-time CPU priority to be able to more effectively predict the various operating system related delays. Furthermore, the file system meta-data is cached. All delays are thus predictable in non-degenerate cases and can be caught by a relatively small buffer on system level, outside of the application.

Furthermore, an admission control system is implemented in a user-space daemon to make sure no more commitments are made than the available resources allow. It should be noted that although our daemon offers a framework for extensive admission control, only a very basic system is available at the moment. The architecture of our framework as incorporated in the Linux kernel is shown in Figure 1.

Prior versions of ABISS used very fine-grained administration and measurement instrumentation to have very narrowly defined performance

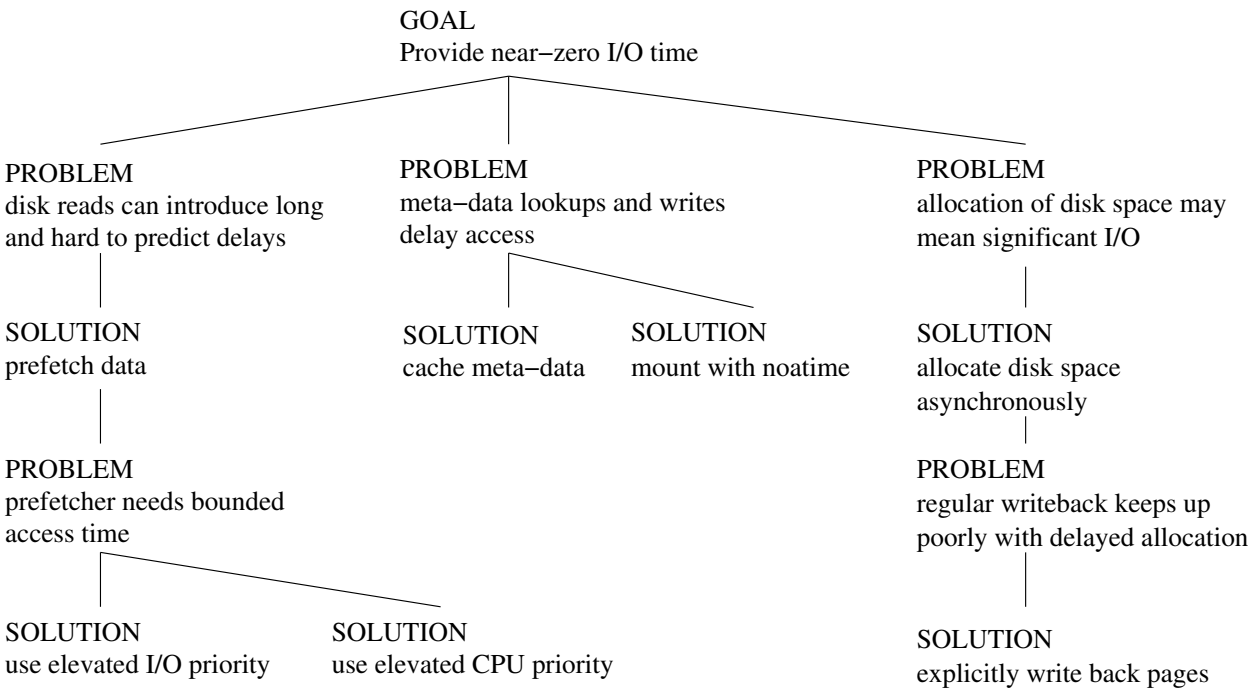


Figure 2: Overview of the solutions incorporated in ABISS.

characteristics. With time, these demands on the underlying layers have gotten "softer." Since we are covering larger parts of the system, leading to influences beyond our full control like the allocation of disk space, we cannot predict the involved delays with such precision as before.

2.2 Service model

When an application requests the services of ABISS (we call such an application an *ABISS user*, or, more specifically, an *ABISS reader* or *writer*), it informs the system about both the bit rate as well as the maximum read or write burst size it is planning to use. A function which opens a file and sets these parameters is available in the ABISS *middleware library*. Given knowledge of the general system responsiveness (I/O latencies, system speed and background load), the buffer can be correctly dimensioned using these variables. This information is also used in the admission control scheme in the daemon which oversees the available system resources.

As the behavior of a streaming application is highly predictable, a fairly simple prefetcher can be used to determine which data should be available in the buffer. The prefetching policy is concentrated in the ABISS scheduler. A separate worker thread performs the actual reading of the data asynchronously, to keep the response time to the application to a minimum.

We use the prefetcher mechanism also when writing, in which case it is not only responsible for the allocating and possibly loading of new pages, but also for coordinating writeback.

To minimize the response time during writing the operations which introduce delays are removed from the calling path of the write operation of the application. This is done by postponing the allocation, to make sure this I/O intensive task is done asynchronously at a moment

the system has time to spare. In our "*delayed allocation*" solution, space for new data in the buffer does not get allocated until the moment of writeback.

An overview of the above solutions is shown graphically in Figure 2. The technical implementations will be elaborated below.

2.3 Formal service definition

The real-time service offered to an application is characterized by a data rate r and a maximum burst read size b. The application sets the *playout point* to mark the location in the file after which it will perform accesses. As long as the playout point moves at rate r or less, accesses to up to b bytes after the playout point will be guaranteed to be served from memory.

If we consider reading a file as a sequence of n single-byte accesses with the i-th access at location a_i at time t_i and with the playout point set to p_i, the operating system then guarantees that all accesses are served from memory as long as the following conditions are met for all i, j in $1, \ldots, n$ with $t_i < t_j$:

$$p_i \leq p_j < p_i + b + r(t_j - t_i)$$
$$p_j \leq a_j < b + \min(p_j, p_i + r(t_j - t_i))$$

The infrastructure can also be used to implement a prioritized best-effort service without guarantees. Such a service would ensure that, on average and when measured over a sufficiently long interval, a reader that has always at least one request pending, will experience better latency and throughput, than any reader using a lower priority.

3 Reading

When reading a page of file data, the kernel first allocates a free page. Then it determines the location of the corresponding disk blocks, and may create so-called *buffer heads*[1] for them. Next, it submits disk I/O requests for the buffer heads, and waits for these requests to complete. Finally, the data is copied to the application's buffer, the *access time* is updated, and the `read` system call returns. This procedure is illustrated in Figure 3.

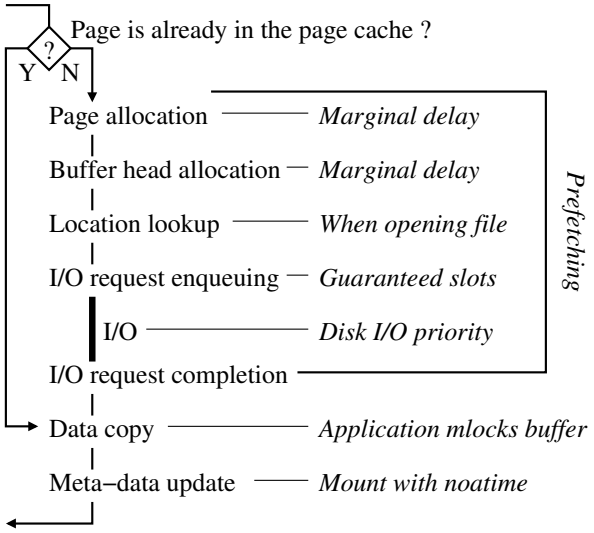


Figure 3: The steps in reading a page, and how ABISS controls their latency.

If trying to read a page that is already present in memory (in the so-called *page cache*), the data becomes available immediately, without any prior I/O. Thus, to avoid waiting for data to be read from disk, we make sure that it is already in the page cache when the application needs it.

3.1 Prefetching

We can accurately predict which data will be read, and can therefore initiate the read process ahead of time. We call this *prefetching*. Pages

[1] A buffer head describes the status and location of a block of the corresponding file system, and is used to communicate I/O requests to the block device layer.

read in advance are placed in a *playout buffer*, illustrated in Figure 4, in which they are kept until the application has read them. After that, pages with old data are evicted from the playout buffer, and new pages with data further into the file are loaded. This can also be thought of as a buffer sliding over the file data.

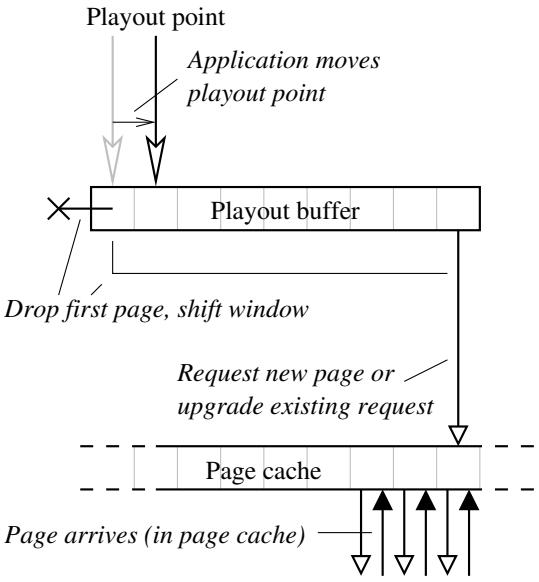


Figure 4: Playout buffer movement is initiated by the application moving its playout point. More than one page may be "in flight" at once.

The playout buffer maintained by ABISS is not a buffer with the actual file data, but an array of pointers to the page structures, which in turn describe the data pages.

Since the maximum rate at which the application will read is known, we can, given knowledge of how long the data retrieval will take, size the playout buffer accordingly, as shown in Figure 5. For this, we consider the space determined by the application, and the buffering needed by the operating system to load data in time. The application requests the total buffer size it needs, which comprises the maximum amount of data it will read at once, and the space needed to compensate for imperfections in its scheduling. To this, buffering is added to cover the maximum time that may pass between initiating retrieval of a page and its arrival, and the batching described in Section 3.4.

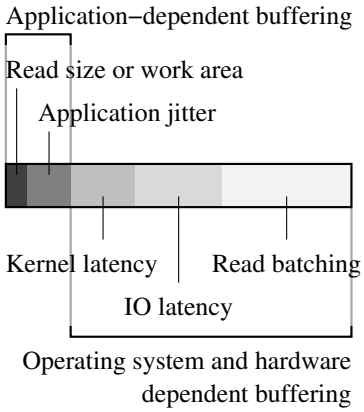


Figure 5: The playout buffer of the scheduler provides for buffering needs resulting from application properties and from latencies caused by the operating system and the hardware.

Prefetching is similar to the *read-ahead* process the kernel performs regularly when sequentially reading files. The main differences are that read-ahead uses heuristics to predict the application behaviour, while applications explicitly tell ABISS how they will read files, and that ABISS keeps a reference to the pages in the playout buffer, so that they cannot be reclaimed before they have actually been used.

Prefetching is done in a separate kernel thread, so the application does not get delayed.

For prefetching to work reliably, and without consuming excessive amounts of memory, data retrieval must be relatively quick, and the worst-case retrieval time should not be much larger than the typical retrieval time. In the following sections, we describe how ABISS accomplishes this.

3.2 Memory allocation

When reading a page from disk, memory allocation happens mainly at three places: (1) when allocating the page itself, (2) when allocating the buffer heads, and (3) when allocating disk I/O request structures.

The first two are regular memory allocation processes, and we assume that they are not sources of delays significantly larger than disk I/O latency.[2]

The number of disk I/O request structures is limited by the maximum size of the request queue of the corresponding device. If the request queue is full, processes wanting to enqueue new requests have to wait until there is room in the queue. Worse yet, once there is room, all processes waiting for it will be handled in FIFO order, irrespective of their CPU priority.

In order to admit high priority I/O requests (see below) instantly to the request queue, the ABISS elevator can be configured to guarantee a certain number of requests for any given priority. Note that this does not affect the actual allocation of the request data structure, but only whether a process has to wait before attempting an allocation.

3.3 Prioritized disk I/O

The key purpose of ABISS is to hide I/O latency from applications. This is accomplished mainly through the use of prefetching. Now, in order to make prefetching work properly, we also have to limit the worst-case duration[3] of

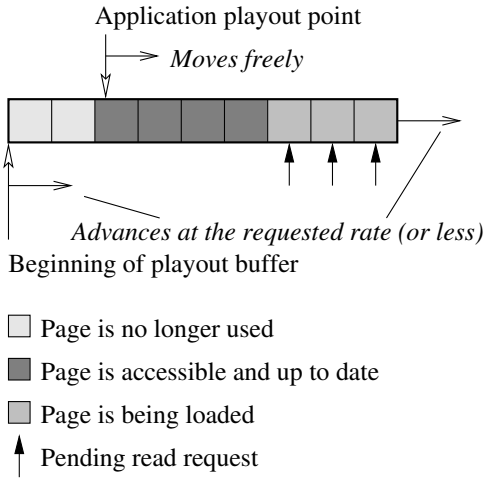


Figure 6: Playout buffer movement is controlled by the kernel, and tracks the position of the playout point, controlled by the application.

I/O requests, independent from what competing applications may do.

ABISS achieves isolation against applications not using ABISS by giving I/O requests issued by the prefetcher thread a higher priority than requests issued by regular applications. The priorities are implemented in the *elevator*:[4] requests with a high priority are served before any requests with a lower priority. We currently use an elevator specifically designed for ABISS. In the future, we plan to migrate to Jens Axboe's more versatile time-sliced CFQ elevator [2].

An interesting problem occurs if a page enters an ABISS playout buffer while being read at a low priority. In order to avoid having to wait until the low priority requests get processed, the prefetcher *upgrades* the priority of the requests associated with the page.

We have described the ABISS elevator in more detail in [3].

[2] In fact, they are much shorter most of the time, except when synchronous memory reclaim is needed.

[3] We ignore degenerate cases, such as hardware errors.

[4] Also called "I/O scheduler." In this paper, we use "elevator" to avoid confusion with the CPU scheduler and the ABISS scheduler.

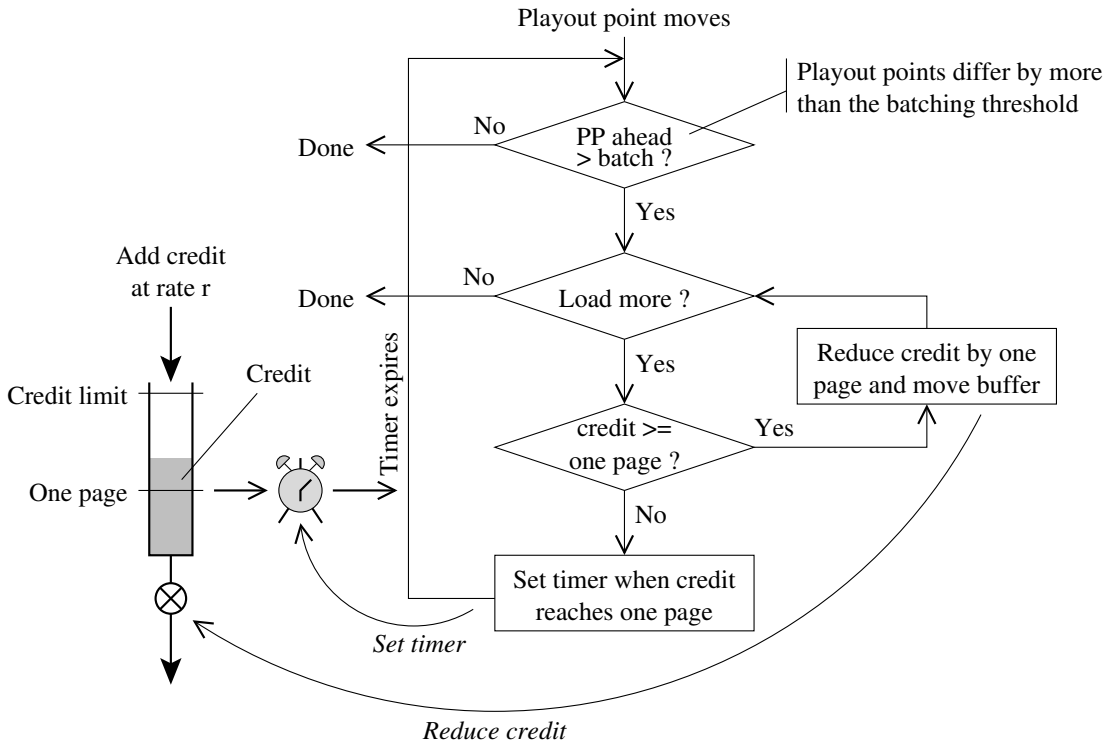


Figure 7: Playout buffer movement is limited by a credit that accumulates at the rate requested by the application, and which is spent when the playout buffer advances through the file.

ABISS users may also compete among each other for I/O. To ensure that there is enough time for requests to complete, the playout buffer must be larger if more ABISS users are admitted. Dynamically resizing of playout buffers is currently not implemented. Instead, the initial playout buffer size can be chosen such that it is sufficiently large for the expected maximum competing load.

3.4 Rate control

Movement of the playout buffer is limited to the rate the application has requested. Application and kernel synchronize through the so-called *playout point*: when the application is done accessing some data, it moves the playout point after this data. This tells the kernel that the playout buffer can be shifted such that its beginning lines up with the playout point again, as shown in Figure 6.

We require explicit updating of the playout point, because, when using `read` and `write`, the file position alone may not give an accurate indication of what parts of the file the application has finished reading. Furthermore, in the case of memory-mapped files, or when using `pread` and `pwrite`, there is no equivalent of the file position anyway.

The ABISS scheduler maintains a *credit* for playout buffer movements. If enough credit is available to align the playout buffer with the playout point, this is done immediately. Otherwise, the playout buffer catches up as far as it can until all credit is consumed, and then advances whenever enough new credit becomes available. This is illustrated in Figure 7.

The credit allows the playout buffer to "catch up" after small distortions. Its accumulation is capped to the batch size described below, plus the maximum latency for timer-driven playout buffer movement, as shown in Figure 8.

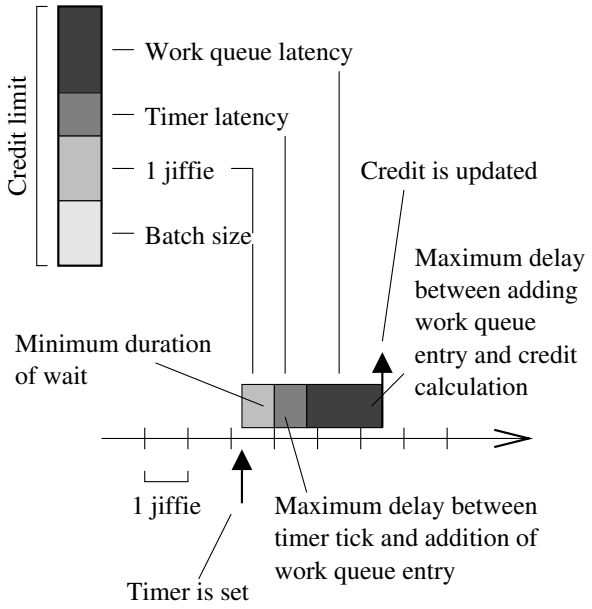


Figure 8: The limit keeps the scheduler from accumulating excessive credit, while allowing it to compensate for the delays occurring when scheduling operations.

If the file was read into the playout buffer one page at a time, and there is also concurrent activity, the disk would spend an inordinate amount of time seeking. Therefore, prefetching only starts when a configurable *batching threshold* is exceeded, as shown in Figure 9. This threshold defaults to ten pages (40 kB).

Furthermore, to avoid interrupting best-effort activity for every single ABISS reader, prefetching is done for all files that are at or near (i.e., half) the batching threshold, as soon as one file reaches that threshold. This is illustrated in Figure 10.

3.5 Wrapping up

Copying the data to user space could consume a significant amount of time if memory for the buffer needs to be allocated or swapped in at that time. ABISS makes no special provisions for this case, because an application can easily avoid it by `mlock`ing this address region into memory.

Finally, the file system may maintain an access time, which is updated after each read operation. Typically, the access time is written back to disk once per second, or less frequently. Updating the access time can introduce particularly large delays if combined with journaling. Since ABISS currently provides no mechanism to hide these delays, file systems used with it should be mounted with the `noatime` option.

4 Writing

When writing a page, the overall procedure is similar to reading, but a little more complicated, as shown in Figure 11: if the page is not already present in the page cache, a new page is allocated. If there is already data for this page in the file, i.e., if the page does not begin beyond the end of file, and does not in its entirety coincide with a hole in the file, the old data is read from disk.

If we are about to write new data, the file system driver looks for free space (which may involve locking and reading file system meta-data), allocates it, and updates the corresponding file system meta-data.

Next, the data is copied from the user space buffer to the page. Finally, the status of the buffer heads and the page is set to "dirty" to indicate that data needs to be written back to disk,

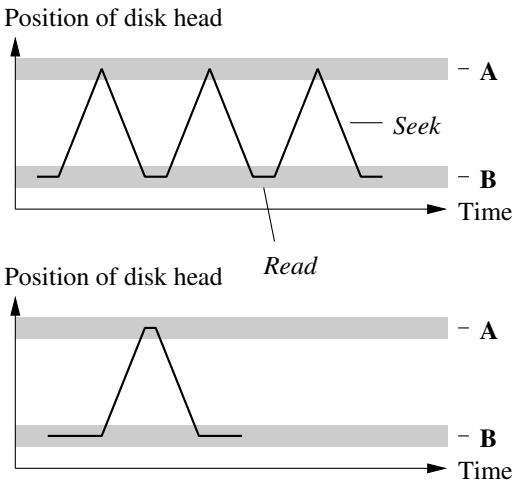


Figure 9: Reading a file (**A**) with ABISS one page at a time (above) would cause many seeks, greatly slowing down any concurrent best-effort reader (**B**). Therefore, we batch reads (below).

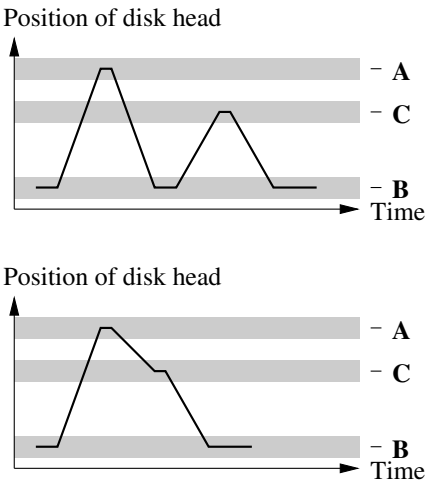


Figure 10: If there are multiple ABISS readers (**A** and **C**), further seeks can be avoided if prefetching is synchronized (below).

and to "up to date" to indicate that the buffers, or even the entire page, are now filled with valid data. Also file meta-data, such as the file size, is updated.

At this point, the data has normally not been written to disk yet. This *writeback* is done asynchronously, when the kernel scans for *dirty* pages to flush.

If using journaling, some of the steps above involve accesses to the journal, which have to complete before the write process can continue.

If overwriting already allocated regions of the file, the steps until after the data has been copied are the same as when reading data, and ABISS applies the same mechanisms for controlling delays.

4.1 Delayed allocation

When writing new data, disk space for it would have to be allocated in the `write` system call. It is not possible to do the allocation at prefetch time, because this would lead to inconsistent file state, e.g., the nominal end-of-file could differ from the one effectively stored on disk.

A solution for this problem is to defer the allocation until after the application has made the `write` system call, and the data has been copied to the page cache. This mechanism is called *delayed allocation*.

For ABISS, we have implemented experimental delayed allocation at the VFS level: when a page is prefetched, the new `PG_delalloc` page flag is set. This flag indicates to other VFS functions that the corresponding on-disk location of the data is not known yet.

Furthermore, `PG_delalloc` indicates to memory management that no attempt should be made to write the page to disk, e.g., during normal writeback or when doing a `sync`. If such a writeback were to happen, the kernel would automatically perform the allocation, and the page would also get locked during this. Since allocation may involve disk I/O, the page may stay locked for a comparably long time, which

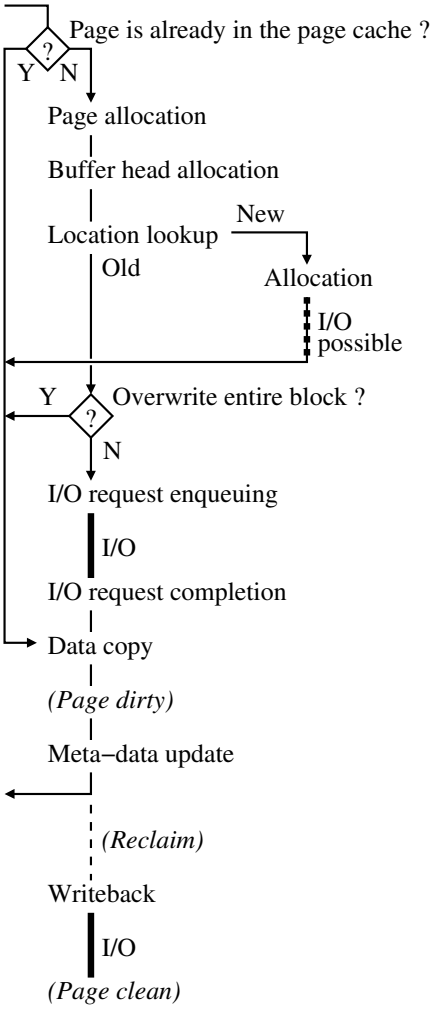


Figure 11: The steps in writing a page (without ABISS).

could block an application using ABISS that is trying to access this page. Therefore, we ensure that the page does not get locked while it is still in any playout buffer.

The code to avoid allocation is mainly in `fs/buffer.c`, in the functions `__block_commit_write` (we set the entire page dirty), `cont_prepare_write` and `block_prepare_write` (do nothing if using delayed allocation), and also in `mpage_writepages` in `fs/mpage.c` (skip pages marked for delayed allocation).

Furthermore, `cont_prepare_write` and `block_prepare_write` may now see pages that have been prefetched, and thus are already up to date, but are not marked for delayed allocation, so these functions must not zero them.

The prefetching is done in `abiss_read_page` in `fs/abiss/sched_lib.c`, and writeback in `abiss_put_page`, using `write_one_page`.

Support for delayed allocation in ABISS currently works with FAT, ext2, and ext3 in `data=writeback` mode.

4.2 Writeback

ABISS keeps track of how many playout buffers share each page, and only clears `PG_delalloc` when the last reference is gone. At that time, the page is explicitly written back by the prefetcher. This also implies allocating disk space for the page.

In order to obtain a predictable upper bound for the duration of this operation, the prefetcher uses high disk I/O priority.

We have tried to leave final writeback to the regular memory scan and writeback process of the kernel, but could not obtain satisfactory performance, resulting in the system running out of memory. Therefore, writeback is now done explicitly when the page is no longer in any ABISS playout buffer. It would be desirable to avoid this special case, and more work is needed to identify why exactly regular writeback performed poorly.

4.3 Reserving disk space

A severe limitation of our experimental implementation of delayed allocation is that errors,

in particular allocation failures due to lack of disk space or quota, are only detected when a page is written back to disk, which is long after the `write` system call has returned, indicating apparent success.

This could be solved by asking the file system driver to reserve disk space when considering a page for delayed allocation, and using this reservation when making the actual allocation. Such a mechanism would require file system drivers to supply the corresponding functionality, e.g., through a new VFS operation.

There is a set of extensions for the ext3 file system by Alex Tomas [4], which also adds, among other things, delayed allocation, along with reservation. Unfortunately, this implementation is limited to the ext3 file system, and extending it to support the prefetching done by ABISS would require invasive changes.

More recent work on delayed allocation with fewer dependencies on ext3 [4] may be considerably easier to adapt to our needs. However, actively preventing allocation while a page is in any playout buffer, which is a requirement unique to ABISS, may be a controversial addition.

4.4 Meta-data updates

When writing, file meta-data such as the file size and the modification time is also changed, and needs to be written back to disk. When reading, we could just suppress meta-data updates, but this is not an option when writing. Instead, we count on these updates to be performed asynchronously, and therefore not to delay the ABISS user.

This is clearly not an optimal solution, particularly when considering journaling, which implies synchronous updates of on-disk data, and we plan to look into whether meta-data updates can be made fully asynchronous, while still honoring assurances made by journaling.

Figure 12 shows the modified write process when using ABISS, with all read and write operations moved into the prefetcher.

4.5 FAT's contiguous files

Files in a FAT file system are always logically contiguous, i.e., they may not have holes. If adding data beyond the end of file, the in-between space must be filled first. This causes a conflict, if we encounter a page marked for delayed allocation while filling such a gap. If we write this page immediately, we may inflict an unexpected delay upon the ABISS user(s) whose playout buffer contains this page. On the other hand, if we defer writing this page until it has left all playout buffers, we must also block the process that is trying to extend the file, or turn also this write into a delayed allocation.

Since our infrastructure for delayed allocations does not yet work for files accessed without ABISS, and because a page can be held in a playout buffer indefinitely, we chose to simply ignore the delayed allocation flag in this case, and to write the page immediately.

A more subtle consequence of all files being contiguous is that new space can only be allocated in a `write` call, never when writing back memory-mapped data. With delayed allocation this changes, and allocations may now happen during writeback, triggered by activity of the allocation code. As a consequence, the locking in the allocation code of the FAT file system driver has to be changed to become reentrant.[5]

[5] This reorganization is partly completed at the time of writing.

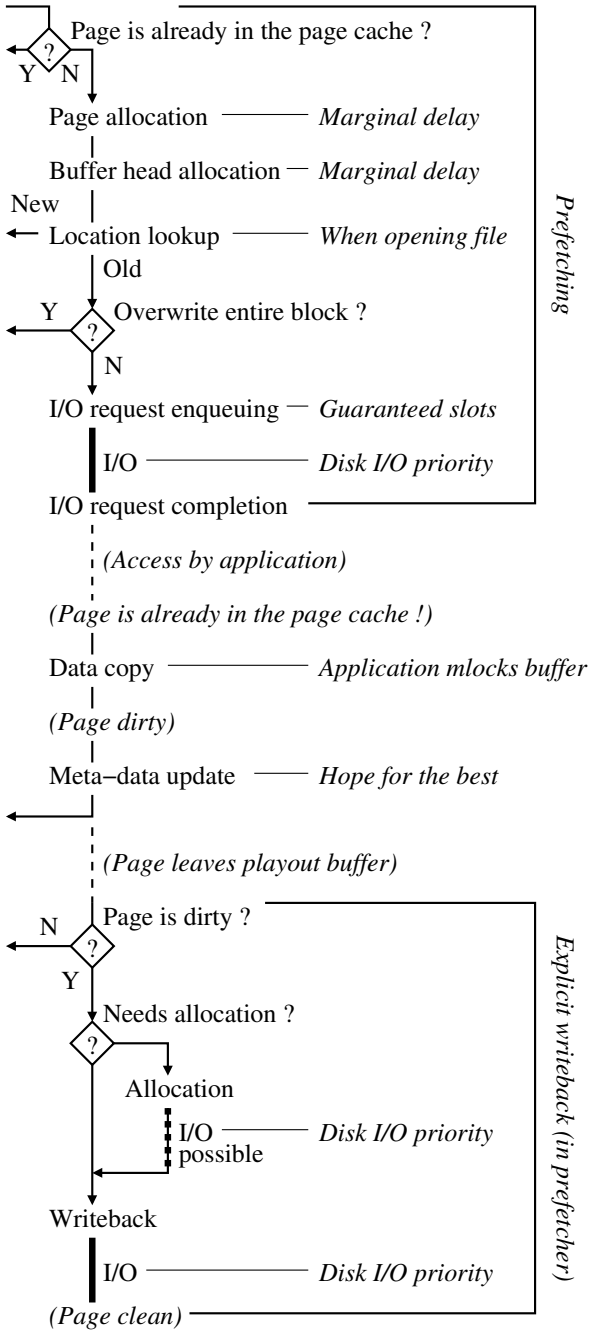


Figure 12: The modified sequence of steps in writing a page using ABISS.

5 Measurements

To be able to assure we have reached our main goal as stated before, near-zero I/O delays, a testing setup was created. The machine running ABISS was deliberately a fairly low-end machine, to assess the results in the light of embedded consumer devices. The data was gathered by `rwrt`, a tool in the ABISS distribution which performs isochronous read or write operations on a file with a certain specified data rate. We have compared the results obtained using ABISS with those obtained using the standard Linux disk I/O. For fair comparison, we used the ABISS elevator on all occasions.

The measurements are performed on a system built around a Transmeta Crusoe TM5800 CPU [5], running at 800 MHz, equipped with 128 MB of main memory of which about 92 MB is available for applications, according to `free`. Two hard drives were connected to the system: the primary drive containing the operating system and applications, and a secondary drive purely for measurement purposes. The drive on which our tests were performed was a 2.5” 4200 RPM Hitachi Travelstar drive.

We have measured the jitter and the latency of reads and writes, the latency of advancing the playout point, the duration of the sleeps of our measurement tool between the I/O calls and the effective distance of the playout point movements. Of these values the jitter is the most interesting one, as it includes both the system call time as well as any effects on time-keeping. Therefore it is a realistic view of what an application can really expect to get. This is further explained in Figure 13. Furthermore, the behaviour of background best-effort readers was analyzed.

Last but not least, we made sure that the streams we read or write are not corrupted in

the process. This was done by adding sequence numbers in the streams, either in prepared streams for reading or on-the-fly while writing.

```
due_time = now;
while (work_to_do) {
    // A (should ideally be due_time)
    read();
    // B
    move_playout();
    // C
    due_time = when next read is due;
    if (due_time < now)
        due_time = now;
    sleep_until(due_time);
}
```

Figure 13: Main loop in rwrt used for reading. Latency is the time from A to B, jitter is B – due_time.[6] Playout point advancement latency is C – B. A similar loop is used for writing. Missed deadlines are forgiven by making sure the next due_time will never be in the past.

5.1 Reading and writing performance

The delays of both the read and write system call with ABISS were measured under heavy system load, to show we are effectively able to guarantee our promised real-time behaviour. Using the `rwrt` tool, we have read or written a stream of 200 MB with a data rate of 1 MB/s, in blocks of 10 kB. The playout buffer size was set to 564 kB for reading and a generous 1 MB for writing, as the latter stressed the system noticeably more. The number of guaranteed real-time requests in the elevator queue was set to 200.

For the tests involving writing, data was written to a new file. The system load was generated by simultaneously running eight greedy best-effort readers or writers[7] during the tests, using separate files with an as high as possible data rate. The background writers were overwriting old data to avoid too many allocations.

5.2 Timeshifting scenario test

To show a realistic scenario for applications mentioned in the introduction of this paper, we have measured the performance of three foreground, real-time writers writing new data, while one foreground real-time reader was reading the data of one of the writers. This is comparable with recording two streams while watching a third one using timeshifting[8]. We have used the same setup as with the previous measurements, i.e., the same bit rate and file sizes.

5.3 Results

The top two plots in Figure 14 show the measured jitter for reading operations. The plots are cumulative proportional, i.e., each point expresses the percentage of requests (on the y-axis) that got executed after a certain amount of time (on the x-axis). For example, a point at (5 ms, 0.1%) on the graph would indicate that 0.1% of all operations took longer than 5 ms. This nicely shows the clustering of the delays; a steep part of the graphs indicates a cluster.

It can be seen that only a small percentage of the requests experience delays significantly longer than average. However, those measurements are the most interesting ones, as we try

[6] We considered using the interval C – due_time instead, but found no visible difference in preparatory tests.

[7] Greedy readers or writers try to read or write as fast as possible, in this case in a best-effort way, using a lower CPU and I/O priority than the ABISS processes.

[8] Timeshifting is essentially recording a stream and playing the same stream a few minutes later. For example, this can be used for pausing while watching a broadcast.

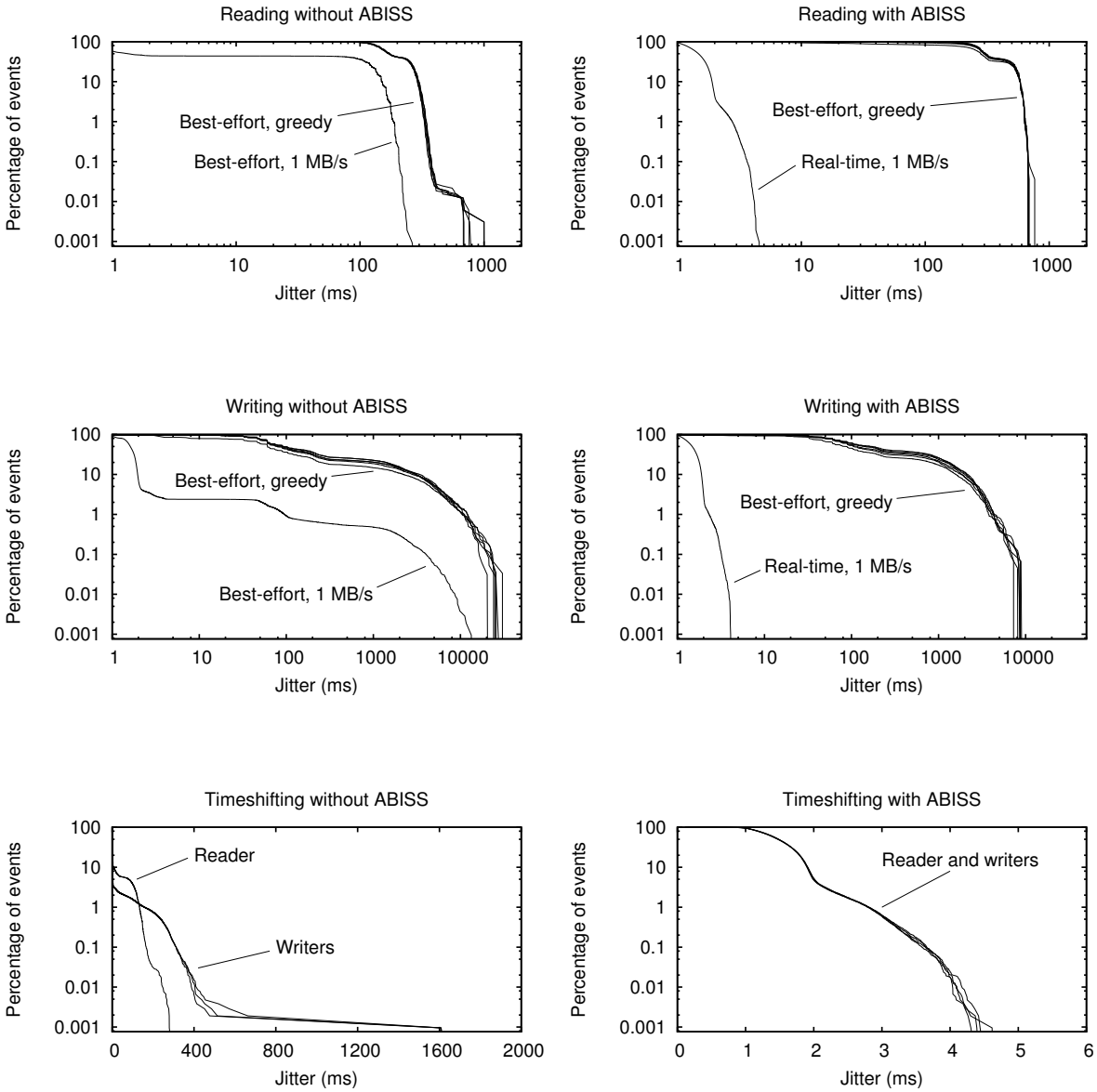


Figure 14: Cumulative proportional plots of the jitter measurements. In all cases the ABISS elevator was used and the measurements were performed on a FAT filesystem.

to bound the experienced delays heuristically. To be able to focus on these delays, the y-axis is logarithmic. As the greedy best-effort readers experience delays of orders of magnitude longer than the real-time delays, the x-axis is logarithmic as well.

Without using the ABISS prefetching mechanism or I/O priorities, all traffic is basically unbounded best-effort. Under the load of the greedy readers, the requested 1 MB/s can definitely not be provided by the system. Although the majority of the requests are served within a few milliseconds, occasional delays of up to a 300 ms were measured. The performance of the greedy readers is even worse: maximum service times of more than a second occurred.

When ABISS is used, we see an enormous decrease of the maximum delay: the reading requests of the 1 MB/s foreground reader now get serviced within less than 5 ms, while the background readers are hardly influenced.

Similar results were observed when using ABISS for writing, as can be concluded from the middle two plots in Figure 14. Using no buffering, prefetching or real-time efforts, but with the ABISS elevator, both the 1 MB/s writer of new data as the greedy background writers experience delays of up to ten seconds. ABISS is able to decrease the service times of the foreground writer to the same level as when it is used for reading: a maximum delay of less than 5 ms, while again the background writers experience little discomfort.

As for the timeshifting scenario with multiple high-priority real-time writers and a ditto reader, the results conform with the above. The results are shown in the last two plots in Figure 14. Without the help of ABISS, especially the writers cannot keep up at all and some request only get served after seconds. Again, using ABISS shortens the delays to less than 5 ms, for both the reader and the writers.

6 Future work

We have briefly experimented with a mechanism based on the NUMA emulator [6], to provide a guaranteed amount of memory to ABISS users. With our changes, we generally observed worse results with than without this mechanism, which suggests that Linux memory management is usually capable to fend for itself, and can maintain sufficient free memory reserves. In periods of extreme memory pressure, this is not true, and additional help may be needed.

When additional ABISS users are admitted or applications close their files, I/O latency changes. In response to this, playout buffers should be adjusted. We currently only provide the basic infrastructure for this, i.e., the ABISS daemon that oversees system-wide resource use, and a set of communication mechanisms to affect schedulers, but we do not implement dynamic playout buffer resizing so far.

Since improvements are constantly being made to the memory management subsystem, it would be good to avoid the explicit writeback described in Section 4.2, and use the regular writeback mechanism instead. We need to identify why attempts to do so have only caused out of memory conditions.

As discussed in Section 4.3, error handling when using delayed allocation is inadequate for most applications. This is due to the lack of a reservation mechanism that can presently be used by ABISS. Possible solutions include either the introduction of reservations at the VFS level, or to try to use file system specific reservation mechanisms, such as the one available for ext3, also with ABISS.

Since delayed allocation seems to be useful in many scenarios, it would be worthwhile to try to implement a general mechanism, that is neither tied to a specific usage pattern (such as the

ABISS prefetcher), nor confined to a single file system. Also, delayed allocation is currently very experimental in ABISS, and some corner cases may be handled improperly.

Last but not least, it would be interesting to explore to what extent the functionality of ABISS could be moved into user space, e.g., by giving regular applications limited access to disk I/O priorities.

7 Conclusion

The ABISS framework is able to provide a number of different services for controlling the way reads and writes are executed. It furthermore allows for a highly controlled latency due to the use of elevated CPU and I/O priorities by using a custom elevator. These properties have enabled us to implement a service providing guaranteed I/O throughput and service times, without making use of an over-dimensioned system. Other strategies might also be implemented using ABISS, e.g., a HDD power management algorithm to extend the battery life of a portable device.

Reading is a more clearly defined operation than writing and the solutions for controlling the latencies involved have matured, yielding good results with FAT, ext2, and ext3. We have identified the problem spots of the writing operation and have implemented partial solutions, including delayed allocation. Although these implementations are currently in a proof-of-concept state, the results are good for both FAT and ext2. The interface complexity of our framework is hidden from the application requesting the service, by introducing a middleware library.

To determine the actual effectiveness and performance of both the framework as well as the implemented scheduler, we have carried out several measurements. The results of the standard Linux I/O system have been compared with the results of using ABISS. Summarizing, using ABISS for reading and writing streams with a maximum bit rate which is known *a priori* leads to heuristically bounded service times in the order of a few milliseconds. Therefore, buffering requirements for the application are greatly reduced or even eliminated, as all data will be readily available.

References

[1] Li, Hong; Cumpson, Stephen R.; Korst, Jan; Jochemsen, Robert; Lambert, Niek. *A Scalable HDD Video Recording Solution Using A Real-time File System*. IEEE Transactions on Consumer Electronics, Vol. 49, No. 3, 663–669, 2003.

[2] Axboe, Jens. [*PATCH*][*CFT*] *time sliced cfq ver18*. Posted on the linux-kernel mailing list, December 21, 2004. `http://article.gmane.org/gmane.linux.kernel/264676`

[3] Van den Brink, Benno; Almesberger, Werner. *Active Block I/O Scheduling System (ABISS)*. Proceedings of the 11th International Linux System Technology Conference (Linux-Kongress 2004), pp. 193–207, September 2004. `http://abiss.sourceforge.net/doc/LK2004_abiss.pdf`

[4] Cao, Mingming; Ts'o, Theodore Y.; Pulavarty, Badari; Bhattacharya, Suparna; Dilger, Andreas; Tomas, Alex; Tweedie, Stephen C. *State of the Art: Where we are with the Ext3 filesystem*. To appear in the Proceedings of the Linux Symposium, Ottawa, July 2005.

[5] The Transmeta Corporation `http://www.transmeta.com/crusoe/crusoe_tm5800_tm5500.html`

[6] Kleen, Andi. [*PATCH*] *x86_64: emulate NUMA on non-NUMA hardware*. Posted on the linux-kernel mailing list, August 31, 2004. `http://article.gmane.org/gmane.linux.kernel.commits.head/38563`

UML and the Intel VT extensions

Jeff Dike
Intel Corp.
jeffrey.g.dike@intel.com

Abstract

Intel has added virtualization extensions (VT) to the x86 architecture. It adds a new set of rings, guest rings 0 through 3, to the traditional rings, which are now called the host rings.

User-mode Linux (UML) is in the process of being enhanced to make use of these extensions for greater performance. It will run in guest ring 0, gaining the ability to directly receive software interrupts. This will allow it to handle process system calls without needing assistance from the host kernel, which will let UML handle system calls at hardware speed.

In spite of running in a ring 0, UML will appear to remain in userspace, making system calls to the host kernel and receiving signals from it. So, it will retain its current manageability, while getting a performance boost from its use of the hardware.

1 Introduction

Intel's new Vanderpool Technology[1] (VT) adds virtualization extensions to the IA architecture which enable hardware support for virtual machines. A full set of "guest" rings are added to the current rings, which are now called "host" rings. The guest OS will run in the guest ring 0 without perceiving any difference from running in the host rings 0 (or on a non-VT system). The guest is controlled by the host regaining control whenever one of a set of events happens within the guest.

The architecture is fully virtualized within the guest rings, so the guest can be an unmodified OS. However, there is also support for paravirtualization in the form of a *VMCALL* instruction which may be executed by the guest, which transers control to the host OS or hypervisor.

The hypervisor has fine-grained control over when the guest traps out to it (a *VMEXIT* event to the host) and over the state of the guest when it is restarted. The hypervisor can cause the guest to be re-entered at an arbitrary point, with arbitrary state.

The paravirtualization support is key to supporting environments other than unmodified kernels. User-mode Linux (UML) is one such environment. It is a userspace port of the Linux kernel, and, as such, would be considered a "modified" guest. It is heavily paravirtualized, as it contains a complete reimplementation, in terms of Linux system calls, of the architecture-specific layer of the kernel.

The reason to consider making UML use this support, when it is not obvious that it is useful, is that there are performance benefits to be

[1] AMD subsequently introduced a compatible technology code-named Pacifica.

realized by doing so. A sore spot in UML performance is its system call speed. Currently, UML must rely on ptrace in order to intercept and handle its process system calls. The context switching between the UML process and the UML kernel and the host kernel entries and exits when the process executes a system call imposes an order of magnitude greater overhead than a system call executing directly on the host. As will be described later, the VT architecture allows a guest to receive software interrupts directly, without involving the host kernel or hypervisor. This will allow UML/VT to handle process system calls at hardware speed.

2 Overview of UML/VT support

The VT paravirtualization support can be used to allow UML to run in a guest ring. For various reasons that will be discussed later, UML will be made to run as a real kernel, in guest ring 0. This would seem to contradict the "user-mode" part of UML's name, but as we shall see, the basic character of UML will remain the same, and the fact that it's running in a ring 0 can be considered an implementation detail.

The essential characteristics of UML are

- It makes system calls to the host kernel.
- It receives signals from the host kernel.
- It resides in a normal, swappable, process address space.

We are going to preserve the first by using the VT paravirtualization support to make system calls to the host kernel from the guest ring 0. Signals from the host will be injected into the guest by the host manipulating the guest state appropriately, and VMENTERing the guest. The third will be preserved as a side-effect of the rest of the design. UML/VT will start in a process address space, and the host will see page faults in the form of VMEXITs whenever the guest causes an access violation. Thus, the normal page fault mechanism will be used to populate the UML/VT kernel address space, and the normal swapping mechanism can be used to swap it out if necessary.

The fact that UML will be running in kernel mode means that it can't make system calls in the normal way, by calling the glibc system call wrappers, which execute *int 0x80* or *sysenter* instructions. Since we can't use glibc for system calls any more, we must implement our own system call layer in terms of *VMCALL*. glibc is UMLs interface to the host Linux kernel, so replacing that with a different interface to the underlying OS can be considered a port of UML to a different OS. Another way of looking at it is to observe that UML will now be a true kernel, in the sense of running in ring 0, and must be ported to that environment, making this a kernel-mode port of UML.

There must be something in the host kernel to receive those VMCALLs, interpret them as system calls, and invoke the normal system call mechanism. A *VMCALL* instruction invokes the VMEXIT handler in the host kernel, as does any event which causes a trap out of the guest to the host. The VMEXIT handler will see all such events, be they hardware interrupts, processor exceptions caused by the guest, or an explicit VMCALL.

3 Porting UML to VT

The first step in porting UML to VT is to make UML itself portable between host operating systems. To date, UML has run only on Linux, so it is strongly tied to the Linux system

call interface. To fix this, we must first abstract out the Linux-specific code and put it under an interface which is somewhat OS-independent. Total OS-independence is not possible with only two examples which are very similar to each other, and is a more of a process than a goal in any case. What we are aiming for is an interface which supports both Linux and VT, and can be made to support other operating systems with modest changes.

To this end, we are moving all of the Linux-specific code to its own directory within the UML architecture (arch/um/os-Linux) and exposing a somewhat OS-independent interface to it. This task is simplified to some extent by the fact that glibc-dependent code had to be separated from kernel-dependent code anyway. The reason is that the former needs to include glibc headers and the latter needs to include kernel headers. The two sets of headers are very incompatible with each other—including both glibc and kernel headers into the same file will produce something that has no chance of compiling. So, from the beginning, UML has been structured such that glibc code and kernel code have been in separate files.

So, to some extent, this part of the port has involved simply moving those files from the main UML source, where they are intermingled with kernel source files, to the os-Linux directory. There are functions which are neither glibc- or kernel-dependent, so these need to be recognized and moved to a kernel file.

Once this code movement has happened, and the resulting interface has been cleaned up and minimized, the next step is to actually implement the interface in terms of VT, using *VMCALL*. So, we will create a new directory, possibly arch/um/os-vt, and implement this interface there. To actually build a VT-enabled UML, we will need to tell the kernel build process (kbuild) to use the os-vt directory rather than the os-Linux one. This is currently determined at runtime by setting a make variable to the output of `uname -s`, and forming the OS directory from that. We can override this variable on the command line by adding `OS=vt` to it, forcing kbuild to use the OS interface implementation in os-vt rather than os-Linux.

4 Host kernel support

As previously mentioned, there will need to be support added to the host kernel in order for it to run UML as a VT guest. Linux currently has no real support for being a hypervisor, and this is what is needed for this project.

The host kernel will need to do the following new things:

- Handle VMEXITs caused by the guest explicitly executing *VMCALL* instructions in order to make system calls.
- Handle hardware interrupts that happen while the guest is running, but which the guest doesn't need to deal with.
- Handle processor faults caused by the guest.
- Force the guest to handle whatever signals it receives from elsewhere on the host.
- Launch the guest and handle its exit.

The design for this calls for a kernel thread in the host to be created when a UML/VT instance is launched. This thread will do the VT-specific work in order to create the guest context and to start UML within it.

Once the UML instance is launched and running, this thread will become the VMEXIT

handler for the instance. It will be invoked whenever the CPU transfers control from the guest to the host for any of a number of reasons.

VMCALL The guest will invoke the *VMCALL* whenever it wants to make a system call to the host. The handler will need to interpret the guest state in order to determine what system call is requested and what its arguments are. Then it will invoke the normal system call mechanism. When the system call returns, it will write the return value into the guest state and resume it. The VT-specific system call layer within the guest will retrieve the return value and pass it back to its caller within UML.

Hardware interrupts Whenever a hardware interrupt, such as a timer tick or a device interrupt, happens while the UML guest is running, the host kernel will need to handle it. So, the VMEXIT handler will need to recognize that this was the cause of the transfer back to the host and invoke the IRQ system in the host.

Processor faults The guest will cause CPU faults in the normal course of operation. Most commonly, these will be page faults on its own text and data due to the guest either not having been fully faulted in or having been swapped out. These interrupts will be handled in the same way as hardware interrupts—they will be passed to the normal host interrupt mechanism for processing.

This thread will be the guest's representative within the host kernel. As such, it will be the target of any signals intended for the guest, and it must ensure that these signals are passed to the UML, or not, as appropriate.

In order to see that there is a signal that needs handling, the thread must explicitly check for pending signals queued against it. When a signal is queued to a process, that process is make runnable, and scheduled. So, if the signal arrives while the guest is not sleeping, then the thread will see the signal as soon as it has been scheduled, and deliver it at that point. If the signal is queued while the guest is running, then delivery will wait until the next time the thread regains control, which will be a hardware timer interrupt, at the latest. This is exactly the same as a signal being delivered to a normal process, except that the wakeup and delivery mechanisms are somewhat different.

If the signal is to be handled by the UML instance, as with a timer or I/O interrupt, then the thread must cause the signal to be delivered to the guest. This is very similar to normal process signal delivery. The existing guest CPU state must be saved, and that state must be modified (by changing the IP and SP, among others) so that when the guest resumes, it is executing the registered handler for that signal. When the handler returns, there will be another exit to the host kernel, analogous to sigreturn, at which point the thread will restore the state it had previously saved and resume the guest at the point at which the signal arrived.

If the signal is fatal, as when a `SIGKILL` is sent to the guest, the thread will shut the guest down. It will destroy the VT context associated with the guest and then call `exit()` on its own behalf. The first step will release any VT-specific resources held by the guest, and the second will release any host kernel resources held by the thread.

This is the same process that will happen on a normal UML shutdown, when the UML instance is halted, and it calls `exit()` after performing its own cleanup.

The final thing that the thread must do is check

for rescheduling. Since it's in the kernel, it must do this explicitly. If the guest's quantum has expired, or a higher priority task can run, then a flag will be set in the thread's task structure indicating that it must call `schedule()`. The thread must check this periodically and schedule whenever the flag is set.

5 Guest setup

When it is launched, a UML/VT guest must do some setup which is hardware-dependent since it is running in ring 0. There are two principal things which must be initialized, system call handling and kernel memory protection.

System call handling As mentioned earlier, this is the area where we expect the greatest performance benefit from using VT. Before launching the guest, the host has specified to the hardware that it does not want a VMEXIT whenever a process within the guest causes a soft interrupt, as happens whenever it makes a system call. The guest will handle these directly, and the guest IDT must be initialized so that the guest's system call handler is invoked.

This will cause UML process system calls to be handled by the guest kernel without any involvement by the host. The host involvement (through `ptrace`) is what currently makes UML system calls so much slower than host system calls. This VT support will make UML process system calls run at hardware speed.

Kernel memory protection Another benefit of running in ring 0 is that UML gets to use the same hardware mechanisms as the host to protect itself from it processes. This is not available to processes—they cannot have two protection domains with the higher one being inaccessible by something running in the lower one. However, by initializing the guest GDT appropriately, UML/VT can install itself as the kernel within the guest domain.

6 Current status

The port of UML to VT is ongoing, as a project within Intel. All of the actual work is being done by two Intel engineers in Moscow, Gennady Sharapov and Mikhail Kharitonov. At this writing, they have finished the OS abstraction work, and I have that as patches in my development tree. These patches have started to be included in the mainline kernel.

The VT-specific work is now in progress. They are making VT system calls to the host and making the guest handle signals sent from the host. The next steps are the hardware initialization to handle system calls and to enable the protection of the kernel.

Following that will be the actual port. The OS abstraction work will be hooked up to the VT system calls in the os-vt layer. The host kernel thread will need to be fleshed out to handle all of the events it will see. Once this is done, it will be possible to start booting UML on VT and to start debugging it.

7 Conclusion

This paper has described the changes needed to make UML work in guest ring 0 with the VT extensions. However, a great deal won't change, and will continue to work exactly as it does today.

The UML address space will still be a completely normal process address space, under the

full control of the host kernel. In the host, the address space will be associated with the kernel thread that is standing in for the VT guest. It will be swappable and demand paged just like any other process address space.

Because of this, and because UML will create its own processes as it does today, UML's copy-user mechanisms will work just as they do currently.

Resource accounting will similarly work exactly as it does today. UML/VT will use the address space occupied by its host kernel thread, and its memory consumption will show up in `/proc` as usual. Similarly, when the guest is running, its kernel thread will be shown as running, and it will accrue time. Thus, CPU accounting, scheduling priority, and other things which depend on process CPU time will continue to work normally.

In spite of being run as a kernel, in a ring 0, UML/VT will continue to maintain the characteristics of a process running within the host kernel. So, it will gain the performance advantages of using the hardware support provided by VT, while retaining all of the benefits of being a process.

SNAP Computing and the X Window System

James Gettys
Hewlett-Packard Company
jim.gettys@hp.com

Abstract

Today's computing mantra is "One keyboard, one mouse, one display, one computer, one user, one role, one administration"; in short, one of everything. However, if several people try to use the same computer today, or cross adminstrative boundaries, or change roles from work to home life, chaos generally ensues.

Several hardware technologies will soon push this limited model of computing beyond the breaking point. Projectors and physically large flat panel displays have become affordable and are about to take a leap in resolution[12]. Cellphone-size devices can now store many gigabytes of information, take high resolution photographs, have significant computation capability, and are small enough to *always* be with you.

Ask yourself "Why can't we sit with friends, family, or coworkers in front of a large display with audio system, and all use it at once?"

You should be able change roles or move locations, and reassociate with the local computing environment. The new mantra must become 'many' and 'mobile' everywhere 'one' has sufficed in the past.

Change will be required in many areas from base system, through the window system and toolkits, and in applications to fully capitalize on this vision.

1 Introduction

As much as three quarters of the cost of computing in enterprise environments now goes to system management and support; the hardware and software purchase cost is well under half of the total expense. In the some parts of the developing world, expertise may be in shorter supply than computers. Personally, I now manage three systems at home, in addition to three for work. Clearly something needs to be done.

Project Athena[13], a joint project of Digital, MIT, and IBM in the mid 1980's, had the vision of centrally administrated, personal computing, in which mobile students and faculty could use whichever computer was most convenient or appropriate for their work. Out of this project was born a number of technologies that we take for granted today, including Kerberos[24], the X Window System[31], central administration of configuration information using Hesiod[18] (now mostly supplanted by LDAP), and Zephyr[17], the first instant message system.

Due to the lack of a critical mass of applications, UNIX divisions, and UNIX workstations costing more than PC's, the Athena environment did not reach critical mass in the marketplace, despite demonstrating much lower cost of ownership, due to much easier system management. The Wintel environment has caused almost everyone to become poor system man-

agers of an ever-increasing number of computers, and it is now clear that Athena had more right than wrong about it. The "solution" of having to carry laptops everywhere is poor, at best. Some of Athena's technologies escaped and became significant parts of our computing environment as individual components, but the overall vision was lost.

Athena's vision was right on many points:

- People are mobile, the computing infrastructure is not.
- People should be able to use any computing system in the environment so long as they are authorized.
- There is a mixture of personally owned and organizationally owned equipment and facilities.
- Authentication enables an organization to control its resources.
- Collaborative tools, either realtime or non-realtime, are central to everyone's lives.
- Your information should be available to you wherever you go.

The Fedora Stateless Project[11] is resurrecting most aspects of the Athena environment and extending it to the often connected laptop; and the LTSP project[6] uses X terminal technology for low system management overhead, thin client computing. These technologies reduce cost of ownership due to system management to something much closer to proportional to the number of people served rather than the number of computers. Deployment of systems based on these technologies, the continuing declining cost of hardware, and open source systems' zero software cost, will enable computers to be located wherever may be convenient. We need to go beyond the Athena vision, however, good as it is for centralized system management.

History also shows Athena's presumptions incorrect or insufficient:

- We presumed display technology limited to one individual at a time, possibly with someone looking over the shoulder.
- That users play a single role, where in the adult world we play many roles: job, home life, church, schools, clubs, and often more. Computer systems must enable people to play multiple roles simultaneously.
- That universal authentication was possible. This is probably a chimera despite efforts of Microsoft and Sun Microsystems—it implies universal trust, unlikely between organizations. At best, you may have a single USB fob or wireless device with many keys that authenticate you for your many roles in life; at worst, many such devices, attached to your physical keyring.
- That there would be very small wearable devices, with significant storage and computing power (soon sufficient for most user's entire computing environment).
- That wireless networking would become very cheap and commonplace.
- That the computing environment is a PC, file, compute and print services: today's environments include projectors and large format displays, (spatial) audio systems, display walls, and so on.

So long as large displays are few and far between, and limited in resolution, the pseudo-solution of the laptop VGA connector attached to a projector has been a poor but adequate

solution. Projectors are now cheap and commonplace, but with the imminent advent of 1080i and 1080p large screen HDTV displays and projectors (1080p is 1920x1080 resolution in computer-speak), we face a near future in which we will finally have displays with enough pixels that sharing of the display makes sense. We will soon be asking: "Why can't I use the environment easily? Why can't I combine my 100 gig cell phone with the surrounding environment to always be able to have my computing environment with me? Why can't I easily shift from work, to home, to school, to church, to hobby?"

Computing systems should enable the reassociation of people, any computing devices they have with them, and the computing infrastructure available wherever they meet, work, and play. While many devices can be used by only one person at a time (e.g. keyboards, mice, etc.), others, such as large screens and audio systems can and should be usable by multiple people simultaneously. It is time we make this possible.

2 User Scenarios

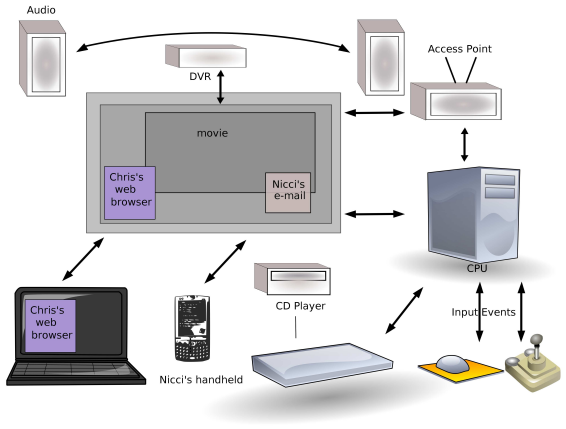
My great thanks to my colleagues Andrew Christian et al. for exploring wider insights into SNAP Computing[15]. Some of the scenarios below are excerpted from that paper. This paper will provide a concrete proposal for work on the X Window System, but without providing background material explaining the SNAP vision, it would be impossible to understand the rationale of the design changes proposed.

2.1 Office

You are sitting in your office. Your incoming frantic call is from your spouse, who is having problems with a complicated formatting problem in the word processor of a document that must be sent before you get home that evening. You ask that the window be cloned to your display, so you can help solve the problem together. When finished, you close the cloned window and the document is finished by the deadline.

2.2 Home

In this example Nikki and her friend Chris are sitting in Nikki's living room watching television on a big, high-resolution video screen, but also doing a little work and web browsing (see below). The living room's personal video recorder (PVR) is playing a movie on the video screen and sending audio to the living room audio system. Nikki has pulled out a portable keyboard, connected to the home office CPU, and pulled up her e-mail on a corner of the living room video screen. As she browses her remote mail store, audio attachments are routed and mixed in the local audio system and played through the living room speakers so that they appear on her side of the room (spatially located so as to not distract Chris).



Meanwhile, Chris has pulled out a wireless handheld computer. Nikki has previously granted Chris some access rights for using the home's broadband connection and living room equipment, so Chris grabs a section of the big video screen and displays output from a web browser running on the handheld computer. Audio output from Chris's web browser is spatially located to help disambiguate it from Nikki's e-mail. Without a keyboard Chris must use the handheld computer for handwriting recognition and cursor control. To speed things up, Chris borrows a wireless keyboard from Nikki's home office. The keyboard detects it is in the living room and bonds automatically to the big screen. Through the handheld computer, Chris assigns the keyboard to work with the web browser and goes back to surfing.

Most of the time Chris and Nikki are working within the confines of the big video screen. For example, both may be driving their own private pointing cursor on the screen. Security policies prevent them from controlling each others' applications; Nikki typing at her e-mail is kept separate from Chris's web browsing. However, the big screen also provides high level services that both can request and access. For example, a screen window manager service positions the individual windows and a screen cut-and-paste service allows data to be shared across users. Should Chris or Nikki wish to change channels or control audio volume in the room, either can ask for video screen control and use the shared, built-in video browser to access the audio volume control or bind it to their local device (Chris' handheld or Nikki's keyboard).

2.3 Conference Room

Functionally, a conference room is not greatly dissimilar from Nikki's living room. The conference room provides shared video screens that multiple users can access from their laptop/handheld computers, or via broadband connections back to their desktop machines.

The conference room provides several business-specific services. First, the room itself can provide scheduling and journaling functions. Because the conference room display screens are intelligent—rather than simple projectors—it is easy to allow them to record and store information about what was done in the room. Each user provides authentication before accessing services, so a clean record of users and activities can be journalled and made available to participants later.

Adding video conferencing introduces a second interesting feature: virtual proximity. A video conference establishes a virtual location relationship between people and devices. For example, the remote user may wish to print a file in the conference room, display and control a presentation on the video screen, and play audio through the local speakers.

To make this more concrete, imagine you are at a meeting of a industry working group with representatives from competitors, to work on a standards document. Several of you put up drafts on the conference room display screens to work on from the laptops you brought with you. The computer of one of your working group members has failed entirely, but he has the information cached in his cellphone, so using a spare keyboard in the conference room, he is able to find the needed information using a corner of the screen for the group.

Such conference rooms were described by Isaac Asimov in his *Foundation* series, in which his First Foundation mathematicians work together in rooms whose walls *are* displays. Such conference rooms are no longer science fiction; display wall systems are already being built[9][2], and their cost will continue to fall.

3 Design Requirements of the SNAP Vision

If we are to disassemble, or unsnap, the components of the classic computer and allow the flexible reassociation (or snapping together) of components, while enabling people to reassociate with the computing environment as they move, we require some networking connectors to snap the components back together. I argue that the networking connectors now exist, and if we disassemble our systems and combine the pieces using these connectors, we can then easily snap them back together dynamically at will. I will touch on some of the resulting topics in this section, before diving into X Window System-specific issues.

These software components include:

- distributed caching file systems (e.g. Coda[23])
- encryption of all network communication
- roaming between networks
- software which can easily roam among multiple differing authentication systems
- discovery of network services
- network connectors replacing hard wires to snap the computing components back together
- network audio, so that you can easily use audio facilities in the environment
- the window system that supports multiple people collaborating, and helps protects you from other malicious people

3.1 Service Discovery

People need to be able to discover that facilities are available and take advantage of them. Open source implementations of the IETF Zeroconf[10] protocols are now available such as Howl[4]); zeroconf is also used to good effect as Apple's Bonjour[1] in OSX. We can leverage such protocols to discover file systems, X Window System servers for large displays, scanners, printers, and other services that may be interesting to mobile users in the environment; and zeroconf support is beginning to appear in open source desktop projects.

3.2 Localization

For ease of use, you need to know what devices are in a given physical location. Presenting a user with a long list of devices present in many work environments, even just a local subnet, would result in confusion. Research shows that it may be feasible to localize 802.11[abg] to roughly the granularity of an office or a conference room, but such systems are not generally available at this date. Given these results it is clear that location tracking systems will become available in the near-term future, and there are startup companies working actively to bring them to market.

Bluetooth was intended as a cable replacement, but experience with it has not been very favorable in a SNAP system application. Introducing a new bluetooth device to its controller is involved and time-consuming, not something that is done casually, at least as cumbersome as dragging a cable across a room and plugging it in.

The 801.15.4 ZigBee local wireless technology, just becoming available, does not suffer from these limitations that make Bluetooth so cumbersome. Additionally, IR is ubiquitous can

be used for local line of sight localization, and handheld devices often have consumer IR (intended for remote control use), which has much longer range than that found in laptops.

There are multiple efforts in the research community to provide location based lookup of resources, and this work and expertise should be leveraged.

3.3 Audio

There is a long history of inadequate audio servers on UNIX and Linux.

ESD and NAS are inadequate even for local multimedia use (lacking any provision for tight time synchronization), much less low-latency applications like teleconferencing.

There are a number of possible paths:

- The Media Application Server (MAS)[7] may be adequate.
- We can build a network layer for the JACK[5] audio system.

These possibilities are not mutually exclusive (Jack and MAS could be used in concert), and we can start from scratch, if they will not serve.

Detailed discussion of the need/requirements for network audio, needed to complement our network transparent window system are beyond the overall scope of this paper. The AF audio server[25] on UNIX of the early 1990's showed that both very low latency and tight synchronization is in fact possible in a network-transparent audio server.

3.4 Network Roaming

There is much work to be done to take what is possible and reduce it to practice. Real-time roaming between networks can be as short as fractions of a second; we should not accept the current delays or manual nature we find today as we DHCP and manually suspend/resume as we transition between networks. Handoff between networks can and should be similar in duration to the cellphone network, so short as to be effectively unnoticed.

4 X Window System

The 'One' mantra is most clearly ingrained in all of today's window systems, where one keyboard, one mouse, one user is the norm. Our base operating system, however, was designed as a multi-user system, with the operating system providing protection between users. The X Window System has at least been, since its inception, network transparent, allowing applications to run on multiple displays, potentially including displays in our environment.

4.1 Multiple People Systems

X's current design presumes a single person using the window system server, and therefore only provided access control. To allow multiple people, particularly in open environments where people cannot trust each other, to use a common screen means that privacy and security problems must be solved.

The core X protocol allows applications to spy on input. Furthermore, cut-and-paste can quickly transfer megabytes of data between applications. Multiple simultaneous users therefore pose a serious security challenge. X needs

better access control to input events, pixmap data, X properties, and other X resources.

During the mid 1990's, there was work to extend X for the requirements posed by military multi-level 'orange book' security. The resulting extension provided no policy flexibility, and still presumed a single user. The resulting X Security extension[35] has remained entirely unused, as far as can be determined.

Recently, Eamon Walsh, an intern at the NSA, implemented an SELinux-style X extension [34] with the explicit goal of enabling multiple possible security policies, that might provide the kinds of policy flexibility. Differing environments, in which different levels of trust between users exist and different sensitivities of information displayed on the screen simultaneously, will clearly need different policies. One policy can clearly not fit all needs. Eamon's work was updated this spring by Bryan Ericson, Chad Hanson, and others at Trusted Computing Solutions, Inc., and provides a general framework that may be sufficient to explore the policies required for this use of the window system.

X has internally no *explicit* concept of a 'user,' without which it is impossible to devise any security policy for systems being used by multiple people. Given good security policies and enforcement, in many environments even unknown people should have unprivileged access to a display. An explicit concept of a user, and the window resources they are using, is clearly needed in X, and once present, policy development using this framework should become feasible. X also lack explicit knowledge of a people's sessions, and since several sessions may be going on simultaneously, I also expect X will require this concept as well.

On Linux and some UNIX systems you can determine the person's identity on the other end of a local socket. We also need the identity of the person's application on the far end of a network connection. In a corporate environment, this might best be served by the person's Kerberos credentials. In other environments, ssh keys or certificate-based authentication systems may be more appropriate. Fortunately, it appears that Cyrus SASL[19] may fill the authentication bill, as it supports multiple authentication families.

Even with this work, there is work remaining to do to define usable security profiles, and work that should take place in toolkits rather than relying solely in the window system. For example, a person cutting from their application and pasting into another person's application does not have the same security consequence as the opposite situation, of others being able to cut from your application into their application: in this case, the person is giving away information explicitly that they already control. It is easier to trust the implementation of the toolkit you are using, than the implementation of a remote X server that you may have much less reason to trust.

More subtle questions arise for which there are not yet obvious answers: How do you know what security profile is currently in force in the X server you are using? Why should you trust that that profile is actually being enforced? These class of problems are not unique to X, of course.

Distinguishing different pointing devices according to the people using them will require an extension to X to support multiple mouse cursors that can be visually distinguished from each other. Since hardware supports a single cursor at most, X already commonly uses software cursors, and by compositing another image with the cursor shape, we can easily indicate whose cursor it is.

4.2 Securing the wire and the SSH trap

At the time of X11's design (1987), and until just a few years ago, the U.S. government actively discouraged the use of encryption; the best we could do was to leave minimal hooks in the wire protocol to enable a later retrofit. Even pluggable architectures allowing the easy addition of encryption were actively opposed and might cause the U.S. Government to forbid export of software. Export of encryption without export control only became feasible in open source projects in the last several years.

In the era of little or no security problems of the 1980's and early 1990's, X was for a time routinely used unencrypted over the network. With network sniffers on insecure systems everywhere today, this usage today is clearly insane.

The Swiss army knife of encryption and authentication, "ssh"[14], appeared as a solution, which provides authentication, encryption, and compression by allowing tunneling of arbitrary streams (including X traffic). While it has been a wonderful crutch for which we are very grateful, a crutch it is, for the following reasons:

- SSH requires you to have an account on a machine before tunneling is possible. This prevents the casual use of remote displays, even those we might intend for such use.
- SSH requires extra context switches between the ssh daemon, costing performance, memory, latency, and latency variation, likely an issue on LTSP servers.
- A remote person's identity cannot be determined; only the identity of their local account.

IPSEC might seem to be the easiest solution, and may be necessary to implement as a 'check off' marketing item: however, it does not ensure end-to-end encryption of traffic, and even worse, does not provide user authentication. In IPSEC's use in VPN software, the data is often unencrypted at corporate firewalls and delivered unencrypted, unacceptable for use that involves user keyboard input. It is therefore at a minimum insufficient for SNAP computing, and in some uses, in fact completely insecure.

Therefore authentication, encryption, and compression must be integrated into the X Window System transport to allow for a wider range of authentication and encryption options, to be proxyable to enable secure traversal of administrative boundaries, and to enable use of display resources on displays where you cannot be authenticated. Compression can provide a huge performance benefit over low bandwidth links[27].

4.3 Remote Devices

It has always been trivial for an X application to use a remote display, but when the application is running to a remote X server, there has been a presumption that the input devices are also attached to the remote machine. Having to drape input device cables across the room to plug into the computer driving the display, is clearly ludicrous. We therefore need network transparent input devices.

People may want to use either spare keyboards, a laptop they brought with them, their PDA, or other input devices available in the room to interact with that application. In any case, input events must be routed from the input device to the appropriate X server, whether connected via wires or wireless.

Input devices present security challenges, along with a further issue: we need some way to associate an input device with a particular user. Association setup needs to be both secure and easy

to use, which may present the largest single research challenge; most of the other tasks described in this paper are simple engineering efforts, applying existing technology in obvious ways. One might have hoped that USB's HID serial numbers on devices would help; however, due to the very low cost of many input devices, most manufacturers do not provide actual serial numbers in their hardware.

4.4 Who is in Control?

The X server implementation has presumed that it is in control of all of its input devices, and worse yet, that these do not change during an X session. It uses a static configuration file, only read during server reset (which only occurs when a user logs out). This static model of configuration is clearly wrong, and hotplug is a necessary. The X server needs (as in all good server processes) to react to changes in the environment.

Over the last two years, open source systems have developed extensive infrastructure to support hotplug, with kernel facilities, and the D-BUS[29] and HAL[36] facilities. These should greatly simplify the problem, and allow the policy decisions of whether an input device (local or remote) is connected to a particular X server.

D-BUS can inform the X server of the changes in configuration of input devices. This itself poses a further challenge, as the X server must be able to become a client of the D-BUS daemon. To avoid possible dead-lock situations between X and the D-BUS daemon, some of the internal X infrastructure needs updating.

With only slight care, an interface can be designed that will allow input devices to either use local or remote input devices. Input device association policy should be kept outside of the X server.

4.5 X Input Extension

The X Input Extension[28] provides support for additional input devices beyond the 'core' pointing device (typically mouse) and keyboard. It has a competent design, though it shows its age. XInput lacks:

- Hotplug notification of devices being connected or disconnected.
- The valuator axes should have abstract names (e.g. you would like to know that valuator 0 is the X coordinate, valuator 1 is the Y coordinate, valuator 2 is pressure, and so on).
- Support for multiple users and devices that all users might share.
- A modern reimplementation exploiting the standardization of USB HID (and the `/dev/input` abstraction on Linux); most of the current implementation is supporting old serial devices with many strange proprietary protocols.
- A limit on 255 input devices in the wire encoding (which might become an issue in an auditorium setting); however, if input events are augmented by a identity information, this should be sufficient.

Whether a upward compatible wire protocol version is possible or a new major version of the X Input extension is not yet completely clear, though an upward compatible API looks very likely.

4.6 Toolkits

Replication and migration of running applications has in fact been possible from X's inception: GNU emacs has had the capability to both

share buffers on different X servers, allowing for shared editing of text, and therefore migration of emacs from X server to X server for more than 15 years.

In practice, due to the level of abstraction of the most commonly used toolkit of X's first era (Xt/Motif[22]), migration and/or replication of windows has been very difficult, as such applications initially adjust themselves to the visual types available on the X server and then draw for the rest of their execution with the same pixel values.

Modern toolkits (e.g. GTK[21] and Qt[16]) operate at a higher level of abstraction, where pixel values are typically hidden from applications, and migration of most applications is feasible[20]: a prototype of migration capability first appeared in GTK+ 2.2.

One to one replication of information is the wrong level of abstraction, since not only is the resolution of different screens extremely wide, but different users on different displays should be able to control the allocation of the screen real-estate. A multi-view approach is clearly correct and to be preferred over the existing server based pixel sharing solutions such as xmove[32], useful though such tools are, particularly for migration of old X applications that are unlikely to be updated to modern toolkits. Work to ease replication of windows for application developers awaits suitably motivated contributors.

Since the resolution between a handheld device and a display wall is over an order of magnitude, applications often need to be able to reload their UI layout on the fly for migration to work really well; again, using the Glade user interface builder[3], libglade and GTK+, this capability is already demonstrable for a few applications.

In the face of unreliable wireless connections, the X library needs minor additions to allow toolkits to recover from connection failures. This work is on hold pending completion of a new implementation of Xlib called Xcb[26], which is well underway and now able to run almost all applications. Testing connection loss recovery may be more of a challenge than its implementation.

Lest you think these facilities are interesting only to SNAP computing, it also aids migration of X sessions from one display (say work) to another (e.g. home). As always, security must be kept in mind: it would not be good for someone to be able to steal one or all of your running applications.

4.7 Window Management and Applications

Besides the infrastructure modifications outlined above, window managers need some modification to support a collaborative environment.

Certain applications may want to be fully aware of multiple users: a good example is an editor that keeps changes that each person applies to a document.

Existing applications can run in such a collaborative environment unchanged. Wallace et al.[33] recently reported experience in a deployed system using somewhat jury-rigged support for multiple cursors and using a modified X window manager on a large shared display at Princeton's Plasma Physics Lab's control room. They report easier simultaneous use of existing applications such as GNU Image Manipulation Program (gimp). They also confirm, as hypothesized above, multiple people working independently side-by-side require sufficient display real-estate to be effective; here they may be looking at different views of

the same dataset using separate application instances. And finally, they report that even sequential use of the display was improved due to less dragging of the mouse back and forth.

5 Summary

Most of the problems SNAP computing pose have obvious solutions; in a few areas, further research is required, but none of the research topics appear intractable.

Network display software systems such as Microsoft's RDP[8] and Citrix and VNC[30] are popular, though by operating at a very low level of abstraction, badly compromise full application integration (e.g. cut and paste, selections, window management meta information) between applications sharing a display from many remote systems. They do, however, do a fine job of simple access to remote applications, but are *fatally flawed* if full collaboration among multiple users is desired.

Open source systems SNAP systems should be able to exist quickly, not only since our technology starts off close to the desired end-state, is more malleable, but also that it does not threaten our business model in the same way that such a shift might to commercial systems.

While this paper has primarily explored X Window System design issues, there is obviously plenty of work elsewhere to fully exploit the vision of SNAP Computing.

References

[1] Bonjour. `http://developer.apple.com/darwin/projects/bonjour/index.html/`.

[2] Distributed Multihead X Project. `http://dmx.sourceforge.net/`.

[3] Glade - a User Interface Builder for GTK+ and GNOME. `http://glade.gnome.org/`.

[4] Howl: Man's new best friend. `http://www.porchdogsoft.com/products/howl/`.

[5] Jack audio connection kit. `http://jackit.sourceforge.net/`.

[6] Linux Terminal Server Project. `http://www.ltsp.org/`.

[7] Media Applications Server. `http://www.mediaapplicationserver.net/`.

[8] RDP Protocol Documentation. `http://www.rdesktop.org/#docs`.

[9] Scalable Display Wall. `http://www.cs.princeton.edu/omnimedia/index.html`.

[10] Zero Configuration Networking (Zeroconf). `http://www.zeroconf.org/`.

[11] Stateless Linux, 2004. `http://fedora.redhat.com/projects/stateless/`.

[12] Will Allen and Robert Ulichney. Wobulation: Doubling the addressed resolution of projection displays. In *SID 2005*, volume 47.4. The Society for Information Display, 2005. `http://sid.aip.org/digest`.

[13] Edward Balkovich, Steven Lerman, and Richard P. Parmelee. Computing in higher education: the athena experience. *Commun. ACM*, 28(11):1214–1224, 1985.

[14] Daniel J. Barrett and Richard Silverman. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., 2001.

[15] Andrew Christian, Brian Avery, Steven Ayer, Frank Bomba, and Jamey Hicks. Snap computing: Shared wireless plug and play. 2005. `http://www.hpl.hp.com/techreports/2005/`.

[16] Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly & Associates, Inc., second edition, May 2001.

[17] C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Sommerfeld. The zephyr notification service. In *USENIX Winter*, pages 213–219, 1988.

[18] S. P. Dyer. The hesiod name server. In *Proceedings of the USENIX Winter 1988 Technical Conference*, pages 183–190, Berkeley, CA, 1988. USENIX Association.

[19] Rob Earhart, Tim Martin, Larry Greenfield, and Rob Siemborski. Simple Autentication and Security Layer. `http://asg.web.cmu.edu/sasl/`.

[20] James Gettys. The Future is Coming, Where the X Window System Should Go. In *FREENIX Track, 2002 Usenix Annual Technical Conference*, Monterey, CA, June 2002. USENIX.

[21] Eric Harlow. *Developing Linux Applications with GTK+ and GDK*. MacMillan Publishing Company, 1999.

[22] Dan Heller. *Motif Programming Manual for OSF/Motif Version 1.1*, volume 6. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991.

[23] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[24] J. Kohl and B. Neuman. The kerberos network authentication service. Technical report, 1991.

[25] T. Levergood, A. Payne, J. Gettys, G. Treese, and L. Stewart. Audiofile: A network-transparent system for distributed audio applications, 1993.

[26] Bart Massey and Jamey Sharp. XCB: An X protocol c binding. In *XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[27] Keith Packard and James Gettys. X Window System Network Performance. In *FREENIX Track, 2003 Usenix Annual Technical Conference*, San Antonio, TX, June 2003. USENIX.

[28] Mark Patrick and George Sachs. X11 Input Extension Protocol Specification, Version 1.0. X consortium standard, X Consortium, Inc., 1991.

[29] Havoc Pennington, Anders Carlsson, and Alexander Larsson. D-BUS Specification. `http://dbus.freedesktop.org/doc/dbus-specification.html`.

[30] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[31] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, fourth edition, 1996.

[32] Ethan Solomita, James Kempf, and Dan Duchamp. XMOVE: A pseudoserver for X window movement. *The X Resource*, 11(1):143–170, 1994.

[33] Grant Wallace, Peng Bi, Kai Li, and Otto Anshus. A MultiCursor X Window Manager Supporting Control Room Collaboration. Technical report tr-707-04, Princeton University Computer Science, July 2004.

[34] Eamon Walsh. Integrating XFree86 With Security-Enhanced Linux. In *X Developers Conference*, Cambridge, MA, April 2004. `http://freedesktop.org/Software/XDevConf/x-security-walsh.pdf`.

[35] David P. Wiggins. Security Extension Specification, Version 7.0. X consortium standard, X Consortium, Inc., 1996.

[36] David Zeuthen. HAL Specification 0.2. `http://freedesktop.org/~david/hal-0.2/spec/hal-spec.html`.

Linux Multipathing

Edward Goggin
EMC Corporation
egoggin@emc.com

Alasdair Kergon
Red Hat
agk@redhat.com

Christophe Varoqui
christophe.varoqui@free.fr

David Olien
OSDL
dmo@osdl.org

Abstract

Linux multipathing provides io failover and path load sharing for multipathed block devices. In this paper, we provide an overview of the current device mapper based multipathing capability and describe Enterprise level requirements for future multipathing enhancements. We describe the interaction amongst kernel multipathing modules, user mode multipathing tools, hotplug, udev, and kpartx components when considering use cases. Use cases include path and logical unit re-configuration, partition management, and path failover for both active-active and active-passive generic storage systems. We also describe lessons learned during testing the MD scheme on high-end storage.

1 Introduction

Multipathing provides the host-side logic to transparently utilize the multiple paths of a redundant network to provide highly available and higher bandwidth connectivity between hosts and block level devices. Similar to how TCP/IP re-routes network traffic between 2 hosts, multipathing re-routes block io to an alternate path in the event of one or more path connectivity failures. Multipathing also transparently shares io load across the multiple paths to the same block device.

The history of Linux multipathing goes back at least 3 years and offers a variety of different architectures. The multipathing personality of the multidisk driver first provided block device multipathing in the 2.4.17 kernel. The Qlogic FC HBA driver has provided multipathing across Qlogic FC initiators since 2002. Storage system OEM vendors like IBM, Hitachi, HP, Fujitsu, and EMC have provided multipathing solutions for Linux for several years. Strictly software vendors like Veritas also provide Linux multipathing solutions.

In this paper, we describe the current 2.6 Linux kernel multipathing solution built around the kernel's device mapper block io framework and consider possible enhancements. We first describe the high level architecture, focusing on both control and data flows. We then describe the key components of the new architecture residing in both user and kernel space. This is followed by a description of the interaction amongst these components and other user/kernel components when considering sev-

eral key use cases. We then describe several outstanding architectural issues related to the multipathing architecture.

2 Architecture Overview

This chapter describes the overall architecture of Linux multipathing, focusing on the control and data paths spanning both user and kernel space multipathing components. Figure 1 is a block diagram of the kernel and user components that support volume management and multipath management.

Linux multipathing provides path failover and path load sharing amongst the set of redundant physical paths between a Linux host and a block device. Linux multipathing services are applicable to all block type devices, (e.g., SCSI, IDE, USB, LOOP, NBD). While the notion of what constitutes a path may vary significantly across block device types, for the purpose of this paper, we consider only the SCSI upper level protocol or session layer definition of a path—that is, one defined solely by its endpoints and thereby indifferent to the actual transport and network routing utilized between endpoints. A SCSI physical path is defined solely by the unique combination of a SCSI initiator and a SCSI target, whether using iSCSI, Fiber Channel transport, RDMA, or serial/parallel SCSI. Furthermore, since SCSI targets typically support multiple devices, a logical path is defined as the physical path to a particular logical device managed by a particular SCSI target. For SCSI, multiple logical paths, one for each different SCSI logical unit, may share the same physical path.

For the remainder of this paper, "physical path" refers to the unique combination of a SCSI initiator and a SCSI target, "device" refers to a SCSI logical unit, and a "path" or logical path refers to the logical connection along a physical path to a particular device.

The multipath architecture provides a clean separation of policy and mechanism, a highly modular design, a framework to accommodate extending to new storage systems, and well defined interfaces abstracting implementation details.

An overall philosophy of isolating mechanism in kernel resident code has led to the creation of several kernel resident frameworks utilized by many products including multipathing. A direct result of this approach has led to the placement of a significant portion of the multipathing code in user space and to the avoidance of a monolithic kernel resident multipathing implementation. For example, while actual path failover and path load sharing take place within kernel resident components, path discovery, path configuration, and path testing are done in user space.

Key multipathing components utilize frameworks in order to benefit from code sharing and to facilitate extendibility to new hardware. Both kernel and user space multipathing frameworks facilitate the extension of multipathing services to new storage system types, storage systems of currently supported types for new vendors, and new storage system models for currently supported vendor storage systems.

The device mapper is the foundation of the multipathing architecture. The device mapper provides a highly modular framework for stacking block device filter drivers in the kernel and communicating with these drivers from user space through a well defined libdevmapper API. Automated user space resident device/path discovery and monitoring components continually push configuration and policy information into the device mapper's multipathing filter driver and pull configuration and path state information from this driver.

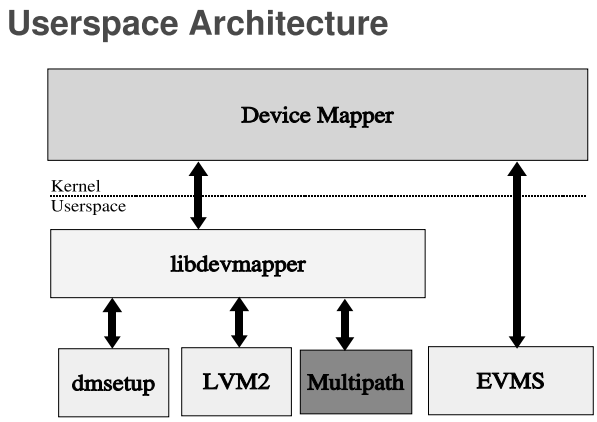


Figure 1: Overall architecture

The primary goals of the multipathing driver are to retry a failed block read/write io on an alternate path in the event of an io failure and to distribute io load across a set of paths. How each goal is achieved is controllable from user space by associating path failover and load sharing policy information on a per device basis.

It should also be understood that the multipath device mapper target driver and several multipathing sub-components are the only multipath cognizant kernel resident components in the linux kernel.

3 Component Modules

The following sections describe the kernel and user mode components of the Linux multipathing implementation, and how those components interact.

3.1 Kernel Modules

Figure 2 is a block diagram of the kernel device mapper. Included in the diagram are components used to support volume management as well as the multipath system. The primary kernel components of the multipathing subsystem are

- the device mapper pseudo driver
- the multipathing device mapper target driver
- multipathing storage system Device Specific Modules (DSMs),
- a multipathing subsystem responsible for run time path selection.

3.1.1 Device Mapper

The device mapper provides a highly modular kernel framework for stacking block device filter drivers. These filter drivers are referred to as

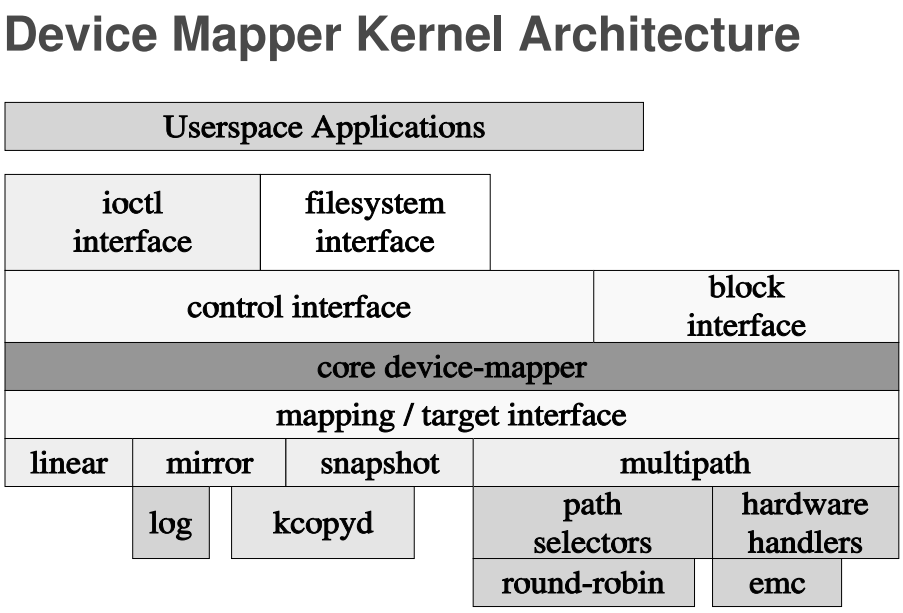


Figure 2: device mapper kernel architecture

target drivers and are comparable to multidisk personality drivers. Target drivers interact with the device mapper framework through a well defined kernel interface. Target drivers add value by filtering and/or redirecting read and write block io requests directed to a mapped device to one or more target devices. Numerous target drivers already exist, among them ones for logical volume striping, linear concatenation, and mirroring; software encryption; software raid; and various other debug and test oriented drivers.

The device mapper framework promotes a clean separation of policy and mechanism between user and kernel space respectively. Taking this concept even further, this framework supports the creation of a variety of services based on adding value to the dispatching and/or completion handling of block io requests where the bulk of the policy and control logic can reside in user space and only the code actually required to effectively filter or redirect a block io request must be kernel resident.

The interaction between user and kernel device mapper components takes place through device mapper library interfaces. While the device mapper library currently utilizes a variety of synchronous ioctl(2) interfaces for this purpose, fully backward compatible migration to using Sysfs or Configfs instead is certainly possible.

The device mapper provides the kernel resident mechanisms which support the creation of different combinations of stacked target drivers for different block devices. Each io stack is represented at the top by a single mapped device. Mapped device configuration is initiated from user space via device mapper library interfaces. Configuration information for each mapped device is passed into the kernel within a map or table containing one or more targets or segments. Each map segment consists of a start sector and length and a target driver specific number of

target driver parameters. Each map segment identifies one or more target devices. Since all sectors of a mapped device must be mapped, there are no mapping holes in a mapped device.

Device mapper io stacks are configured in bottom-up fashion. Target driver devices are stacked by referencing a lower level mapped device as a target device of a higher level mapped device. Since a single mapped device may map to one or more target devices, each of which may themselves be a mapped device, a device mapper io stack may be more accurately viewed as an inverted device tree with a single mapped device as the top or root node of the inverted tree. The leaf nodes of the tree are the only target devices which are not device mapper managed devices. The root node is only a mapped device. Every non-root, non-leaf node is both a mapped and target device. The minimum device tree consists of a single mapped device and a single target device. A device tree need not be balanced as there may be device branches which are deeper than others. The depth of the tree may be viewed as the tree branch which has the maximum number of transitions from the root mapped device to leaf node target device. There are no design limits on either the depth or breadth of a device tree.

Although each target device at each level of a device mapper tree is visible and accessible outside the scope of the device mapper framework, concurrent open of a target device for other purposes requiring its exclusive use such as is required for partition management and file system mounting is prohibited. Target devices are exclusively recognized or claimed by a mapped device by being referenced as a target of a mapped device. That is, a target device may only be used as a target of a single mapped device. This restriction prohibits both the inclusion of the same target device within multiple device trees and multiple references to the same target device within the same device tree, that is, loops within a device tree are not allowed.

It is strictly the responsibility of user space components associated with each target driver to

- discover the set of associated target devices associated with each mapped device managed by that driver
- create the mapping tables containing this configuration information
- pass the mapping table information into the kernel
- possibly save this mapping information in persistent storage for later retrieval.

The multipath path configurator fulfills this role for the multipathing target driver. The `lvm(8)`, `dmraid(8)`, and `dmsetup(8)` commands perform these tasks for the logical volume management, software raid, and the device encryption target drivers respectively.

While the device mapper registers with the kernel as a block device driver, target drivers in turn register callbacks with the device mapper for initializing and terminating target device metadata; suspending and resuming io on a mapped device; filtering io dispatch and io completion; and retrieving mapped device configuration and status information. The device mapper also provides key services, (e.g., io suspension/resumption, bio cloning, and the propagation of io resource restrictions), for use by all target drivers to facilitate the flow of io dispatch and io completion events through the device mapper framework.

The device mapper framework is itself a component driver within the outermost `generic_make_request` framework for block devices. The `generic_make_request`

framework also provides for stacking block device filter drivers. Therefore, given this architecture, it should be at least architecturally possible to stack device mapper drivers both above and below multidisk drivers for the same target device.

The device mapper processes all read and write block io requests which pass through the block io subsystem's `generic_make_request` and/or `submit_bio` interfaces and is directed to a mapped device. Architectural symmetry is achieved for io dispatch and io completion handling since io completion handling within the device mapper framework is done in the inverse order of io dispatch. All read/write bios are treated as asynchronous io within all portions of the block io subsystem. This design results in separate, asynchronous and inversely ordered code paths through both the `generic_make_request` and the device mapper frameworks for both io dispatch and completion processing. A major impact of this design is that it is not necessary to process either an io dispatch or completion either immediately or in the same context in which they are first seen.

Bio movement through a device mapper device tree may involve fan-out on bio dispatch and fan-in on bio completion. As a bio is dispatched down the device tree at each mapped device, one or more cloned copies of the bio are created and sent to target devices. The same process is repeated at each level of the device tree where a target device is also a mapped device. Therefore, assuming a very wide and deep device tree, a single bio dispatched to a mapped device can branch out to spawn a practically unbounded number of bios to be sent to a practically unbounded number of target devices. Since bios are potentially coalesced at the device at the bottom of the `generic_make_request` framework, the io request(s) actually queued to one or more target devices at the bottom may bear little relationship to the single bio initially sent to a mapped device at the top. For bio completion, at each level of the device tree, the target driver managing the set of target devices at that level consumes the completion for each bio dispatched to one of its devices, and passes up a single bio completion for the single bio dispatched to the mapped device. This process repeats until the original bio submitted to the root mapped device is completed.

The device mapper dispatches bios recursively from top (root node) to bottom (leaf node) through the tree of device mapper mapped and target devices in process context. Each level of recursion moves down one level of the device tree from the root mapped device to one or more leaf target nodes. At each level, the device mapper clones a single bio to one or more bios depending on target mapping information previously pushed into the kernel for each mapped device in the io stack since a bio is not permitted to span multiple map targets/segments. Also at each level, each cloned bio is passed to the map callout of the target driver managing a mapped device. The target driver has the option of

1. queuing the io internal to that driver to be serviced at a later time by that driver,
2. redirecting the io to one or more different target devices and possibly a different sector on each of those target devices, or
3. returning an error status for the bio to the device mapper.

Both the first or third options stop the recursion through the device tree and the `generic_make_request` framework for that matter. Otherwise, a bio being directed to the first target device which is not managed by the device mapper causes the bio to exit the device mapper

framework, although the bio continues recursing through the `generic_make_request` framework until the bottom device is reached.

The device mapper processes bio completions recursively from a leaf device to the root mapped device in soft interrupt context. At each level in a device tree, bio completions are filtered by the device mapper as a result of redirecting the bio completion callback at that level during bio dispatch. The device mapper callout to the target driver responsible for servicing a mapped device is enabled by associating a target_io structure with the bi_private field of a bio, also during bio dispatch. In this fashion, each bio completion is serviced by the target driver which dispatched the bio.

The device mapper supports a variety of push/pull interfaces to enhance communication across the system call boundary. Each of these interfaces is accessed from user space via the device mapper library which currently issues ioctls to the device mapper character interface. The occurrence of target driver derived io related events can be passed to user space via the device mapper event mechanism. Target driver specific map contents and mapped device status can be pulled from the kernel using device mapper messages. Typed messages and status information are encoded as ASCII strings and decoded back to their original form according dictated by their type.

3.1.2 Multipath Target Driver

A multipath target driver is a component driver of the device mapper framework. Currently, the multipath driver is position dependent within a stack of device mapper target drivers: it must be at the bottom of the stack. Furthermore, there may not be other filter drivers, (e.g., multidisk), stacked underneath it. It must be stacked immediately atop driver which services a block request queue, for example, `/dev/sda`.

The multipath target receives configuration information for multipath mapped devices in the form of messages sent from user space through device mapper library interfaces. Each message is typed and may contain parameters in a position dependent format according to message type. The information is transferred as a single ASCII string which must be encoded by the sender and decoded by the receiver.

The multipath target driver provides path failover and path load sharing. Io failure on one path to a device is captured and retried down an alternate path to the same device. Only after all paths to the same device have been tried and failed is an io error actually returned to the io initiator. Path load sharing enables the distribution of bios amongst the paths to the same device according to a path load sharing policy.

Abstractions are utilized to represent key entities. A multipath corresponds to a device. A logical path to a device is represented by a path. A path group provides a way to associate paths to the same device which have similar attributes. There may be multiple path groups associated with the same device. A path selector represents a path load sharing algorithm and can be viewed as an attribute of a path group. Round robin based path selection amongst the set of paths in the same path group is currently the only available path selector. Storage system specific behavior can be localized within a multipath hardware handler.

The multipath target driver utilizes two subcomponent frameworks to enable both storage system specific behavior and path selection algorithms to be localized in separate modules which may be loaded and managed separately from the multipath target driver itself.

3.1.3 Device Specific Module

A storage system specific component can be associated with each target device type and is referred to as a hardware handler or Device Specific Module (DSM). A DSM allows for the specification of kernel resident storage system specific path group initialization, io completion filtering, and message handling. Path group initialization is used to utilize storage system specific actions to activate the passive interface of an active-passive storage system. Storage system specific io completion filtering enables storage system specific error handling. Storage system specific message handling enables storage system specific configuration.

DSM type is specified by name in the multipath target driver map configuration string and must refer to a DSM pre-loaded into the kernel. A DSM may be passed paramters in the configuration string. A hardware context structure passed to each DSM enables a DSM to track state associated with a particular device.

Associating a DSM with a block device type is optional. The EMC CLARiion DSM is currently the only DSM.

3.1.4 Path Selection Subsystem

A path selector enables the distribution of io amongst the set of paths in a single path group.

Path selector type is specified by name in the multipath target driver map configuration string and must refer to a path selector pre-loaded into the kernel. A path selector may be passed paramters in the configuration string. The path selector context structure enables a path selector type to track state across multiple ios to the paths of a path group.

Each path group must be associated with a path selector. A single round robin path selector exists today.

3.2 User Modules

Figure 3 outlines the architecture of the user-mode multipath tools. Multipath user space components perform path discovery, path policy management and configuration, and path health testing. The multipath configurator is responsible for discovering the network topology for multipathed block devices and for updating kernel resident multipath target driver configuration and state information. The multipath daemon monitors the usability of paths both in response to actual errors occurring in the kernel and proactively via periodic path health testing. Both components share path discovery and path health testing services. Furthermore, these services are implemented using an extensible framework to facilitate multipath support for new block device types, block devices from new vendors, and new models. The kpartx tool creates mapped devices for partitions of multipath managed block devices.

3.2.1 Multipath Configurator

Path discovery involves determining the set of routes from a host to a particular block device which is configured for multipathing. Path discovery is implemented by scanning Sysfs looking for block device names from a multipath configuration file which designate block device types eligible for multipathing. Each entry in /sys/block corresponds to the gendisk for a different block device. As such, path discovery is independent of whatever path transport is used between host and device. Since devices are assumed to have an identifier attribute which is unique in both time and space, the

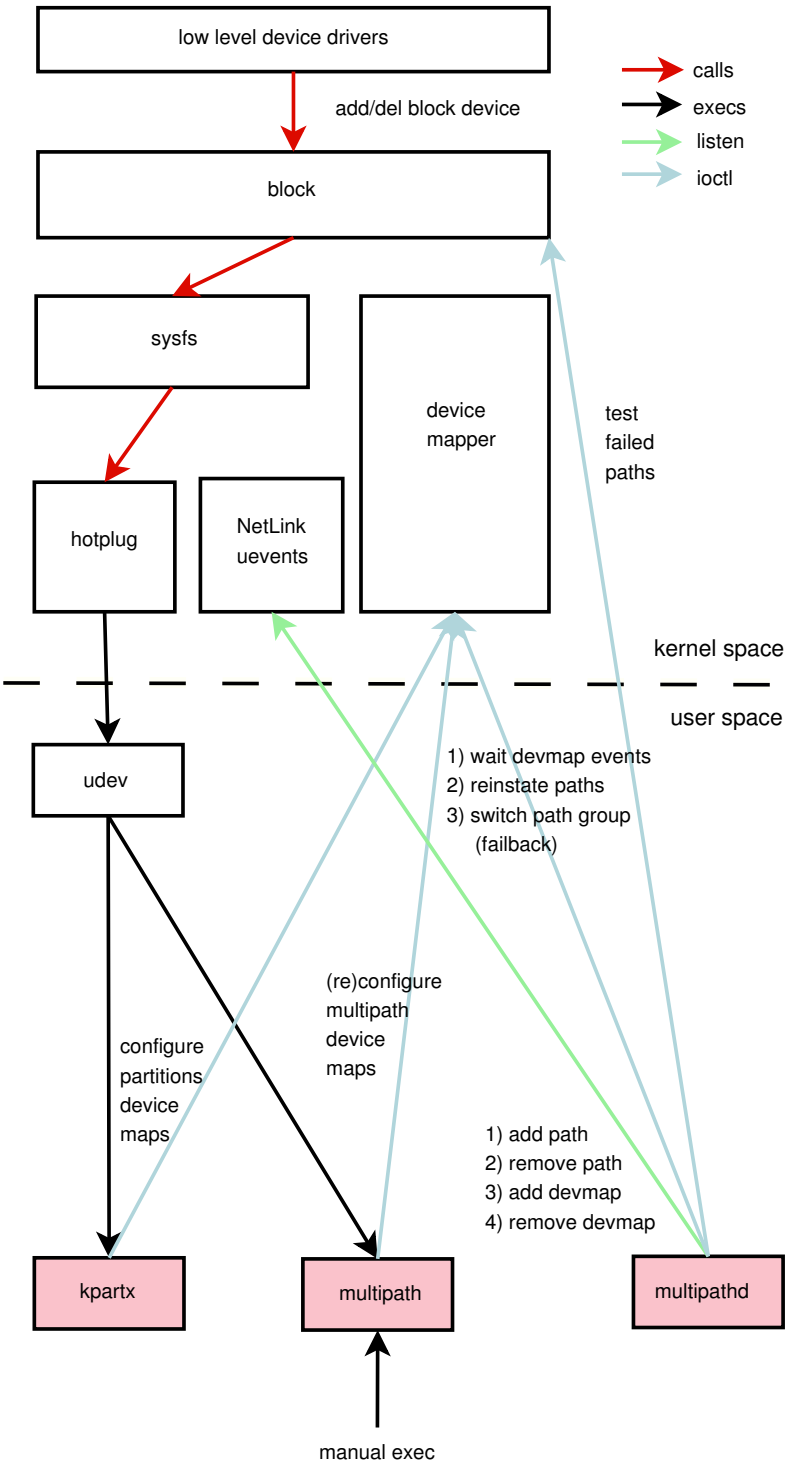


Figure 3: multipath tools architecture

cumulative set of paths found from Sysfs are coalesced based on device UID. Configuration driven multipath attributes are setup for each of these paths.

The multipath configurator synchronizes path configuration and path state information across both user and kernel multipath components. The current configuration path state is compared with the path state pulled from the multipath target driver. Most discrepancies are dealt with by pushing the current configuration and state information into the multipath target driver. This includes creating a new multipath map for a newly discovered device; changing the contents of an existing multipath map for a newly discovered path to a known device, for a path to a known device which is no longer visible, and for configuration driven multipath attributes which may have changed; and for updating the state of a path.

Configuration and state information are passed between user and kernel space multipath components as position dependent information as a single string. The entire map for a mapped device is transferred as a single string and must be encoded before and decoded after the transfer.

The multipath configurator can be invoked manually at any time or automatically in reaction to a hotplug event generated for a configuration change for a block device type managed by the multipathing subsystem. Configuration changes involve either the creation of a new path or removal of an existing path.

3.2.2 Multipath Daemon

The multipath daemon actively tests paths and reacts to changes in the multipath configuration.

Periodic path testing performed by the multipath daemon is responsible for both restoring failed paths to an active state and proactively failing active paths which fail a path test. Currently, while the default is to test all active and failed paths for all devices every 5 seconds, this interval can be changed via configuration directive in the multipath configuration file. The current non-optimized design could be enhanced to reduce path testing overhead by

- testing the physical transport components instead of the logical ones
- varying the periodic testing interval.

An example of the former for SCSI block devices is to

- associate across all devices those paths which utilize common SCSI initiators and targets and
- for each test interval test only one path for every unique combination of initiator and target.

An example of the latter is to vary the periodic test interval in relationship to the recent past history of the path or physical components, that is, paths which fail often get tested more frequently.

The multipath daemon learns of and reacts to changes in both the current block device configuration and the kernel resident multipathing configuration. The addition of a new path or the removal of an already existing path to an already managed block device is detected over a netlink socket as a uevent triggered callback which adds or removes the path to or from the set of paths which will be actively tested. Changes to the kernel resident multipathing state are detected as device-mapper generated event callbacks. Events of this kind invole block io errors, path state change, and changes in the highest priority path group for a mapped device.

3.2.3 Multipath Framework

The multipath framework enables the use of block device vendor specific algorithms for

1. deriving a UID for identifying the physical device associated with a logical path
2. testing the health of a logical path
3. determining how to organize the logical paths to the same device into separate sets,
4. assigning a priority to each path
5. determining how to select the next path within the same path set
6. specifying any kernel resident device specific multipathing capabilities.

While the last two capabilities must be kernel resident, the remaining user resident capabilities are invoked either as functions or executables. All but item four and item six are mandatory. A built-in table specifying each of these capabilities for each supported block device vendor and type may, but need not be, overridden by configuration directives in a multipath configuration file. Block device vendor and type are derived from attributes associated with the Sysfs device file probed during device discovery. Configuration file directives may also be used to configure these capabilities for a specific storage system instance.

A new storage system is plugged into this user space multipath framework by specifying a configuration table or configuration file entry for the storage system and providing any of the necessary, but currently missing mechanism needed to satisfy the six services mentioned above for the storage system. The service selections are specified as string or integer constants. In some cases, the selection is made from a restricted domain of options. In other cases a new mechanism can be utilized to provide the required service. Services which are invoked as functions must be integrated into the multipath component libraries while those invoked as executables are not so restricted. Default options provided for each service may also be overridden in the multipath configuration file.

Since the service which derives a UID for a multipath device is currently invoked from the multipath framework as an executable, the service may be, and in fact is now external to the multipath software. All supported storage systems (keep in mind they are all SCSI at the moment) utilize `scsi_id(8)` to derive a UID for a SCSI logical unit. Almost all of these cases obtain the UID directly from the Vendor Specified Identifier field of an extended SCSI inquiry command using vital product page 0x83. This is indeed the default option. Although `scsi_id` is invoked as an executable today, a `scsi_id` service library appears to be inplan, thereby allowing in-context UID generation from this framework in the near future.

Path health testing is invoked as a service function built into the libcheckers multipath library. While SCSI specific path testing functions already exist in this library based on reading sector 0 (this is the default) and issuing a TUR, path health testing can be specified to be storage system specific but must be included within libcheckers.

The selection of how to divide up the paths to the same device into groups is restricted to a set of five options:

- failover
- multibus
- group-by-priority
- group-by-serial

- group-by-node-name.

Failover, the default policy, implies one path per path group and can be used to disallow path load sharing while still providing path failover. Multibus, by far the most commonly selected option, implies one path group for all paths and is used in most cases when access is symmetric across all paths, e.g., active-active storage systems. Group-by-priority implies a grouping of paths with the same priority. This option is currently used only by the active-passive EMC CLARiion storage array and provides the capability to assign a higher priority to paths connecting to the portion of the storage system which has previously been assigned to be the default owner of the SCSI logical unit. Group-by-serial implies a grouping based on the Vendor Specified Identifier returned by a VPD page 0x80 extended SCSI inquiry command. This is a good way to group paths for an active-passive storage system based on which paths are currently connected to the active portion of the storage system for the SCSI logical unit. Group-by-node-name currently implies a grouping by by SCSI target.

Paths to the same device can be assigned priorities in order to both enable the group-by-priority path grouping policy and to affect path load sharing. Path group priority is a summation of the priority for each active path in the group. An io is always directed to a path in the highest priority path group. The `get_priority` service is currently invoked as an executable. The default option is to not assign a priority to any path, which leads to all path groups being treated equally. The `pp_balance_paths` executable assigns path priority in order to attempt to balance path usage for all multipath devices across the SCSI targets to the same storage system. Several storage system specific path priority services are also provided.

Path selectors and hardware contexts are specified by name and must refer to specific kernel resident services. A path selector is mandatory and currently the only option is round-robin. A hardware context is by definition storage system specific. Selection of hardware context is optional and only the EMC CLARiion storage system currently utilizes a hardware context. Each may be passed parameters, specified as a count followed by each parameter.

3.2.4 Kpartx

The kpartx utility creates device-mapper mapped devices for the partitions of multipath managed block devices. Doing so allows a block device partition to be managed within the device mapper framework as would be any whole device. This is accomplished by reading and parsing a target device's partition table and setting up the device-mapper table for the mapped device from the start address and length fields of the paritition table entry for the partition in question. Kpartx uses the same devmapper library interfaces as does the multipath configurator in order to create and initialize the mapped device.

4 Interaction Amongst Key Kernel and User Components

The interaction between key user and kernel multipath components will be examined while considering several use cases. Device and path configuration will be considered first. Io scheduling and io failover will then be examined in detail.

4.1 Block Device and Path Discovery

Device discovery consists of obtaining information about both the current and previous multipath device configurations, resolving any differences, and pushing the resultant updates into the multipath target driver. While these tasks are primarily the responsibility of the multipath configurator, many of the device discovery services are in fact shared with the multipath daemon.

The device discovery process utilizes the common services of the user space multipath framework. Framework components to identify, test, and prioritize paths are selected from preestablished table or config driven policy options based on device attributes obtained from probing the device's Sysfs device file.

The discovery of the current configuration is done by probing block device nodes created in Sysfs. A block device node is created by udev in reaction to a hotplug event generated when a block device's request queue is registered with the kernel's block subsystem. Each device node corresponds to a logical path to a block device since no kernel resident component other than the multipath target driver is multipath cognizant.

The set of paths for the current configuration are coalesced amongst the set of multipath managed block devices. Current path and device configuration attributes are retrieved configuration file and/or table entries.

The previous device configuration stored in the collective set of multipath mapped device maps is pulled from the multipath target driver using target driver specific message ioctls issued by the device-mapper library.

Discrepancies between the old and new device configuration are settled and the updated device configuration and state information is pushed into the multipath target driver one device at a time. Several use cases are enumerated below.

- A new mapped device is created for a multipath managed device from the new configuration which does not exist in the old configuration.
- The contents of an existing multipath map are updated for a newly discovered path to a known device, for a path to a known device which is no longer visible and for multipath attributes which may have changed. Examples of multipath attributes which can initiate an update of the kernel multipath device configuration are enumerated below.
  - device size
  - hardware handler
  - path selector
  - multipath feature parameters
  - number of path groups
  - assignment of paths to path groups
  - highest priority path group
- Path state is updated based on path testing done during device discovery.

Configuration updates to an existing multipath mapped device involve the suspension and subsequent resumption of io around the complete replacement of the mapped device's device-mapper map. Io suspension both blocks all new io to the mapped device and flushes all io from the mapped device's device tree. Path state updates are done without requiring map replacement.

Hotplug initiated invocation of the multipath configurator leads to semi-automated multipath response to post-boot time changes in the block

device configuration. For SCSI target devices, a hotplug event is generated for a SCSI target device when the device's gendisk is registered after the host attach of a SCSI logical unit and unregistered after the host detach of a SCSI logical unit.

4.2 Io Scheduling

The scheduling of bios amongst the multiple multipath target devices for the same multipath mapped device is controlled by both a path grouping policy and a path selection policy. While both path group membership and path selection policy assignment tasks are performed in user space, actual io scheduling is implemented via kernel resident mechanism.

Paths to the same device can be separated into path groups, where all paths in the same group have similar path attributes. Both the number of path groups and path membership within a group are controlled by the multipath configurator based on one of five possible path grouping policies. Each path grouping policy uses different means to assign a path to a path group in order to model the different behavior in the physical configuration. Each path is assigned a priority via a designated path priority callout. Path group priority is the summation of the path priorities for each path in the group. Each path group is assigned a path selection policy governing the selection of the next path to use when scheduling io to a path within that group.

Path group membership and path selection information are pushed into the kernel where it is then utilized by multipath kernel resident components to schedule each bio on one of multipath paths. This information consists of the number of path groups, the highest priority path group, the path membership for each group (target devices specified by `dev_t`), the name of the path selection policy for each group, a count of optional path selection policy parameters, and the actually path selection policy parameters if the count value is not zero. As is the case for all device mapper map contents passed between user and kernel space, the collective contents is encoded and passed as a single string, and decoded on the other side according to its position dependent context.

Path group membership and path selection information is pushed into the kernel both when a multipath mapped device is first discovered and configured and later when the multipath configurator detects that any of this information has changed. Both cases involve pushing the information into the multipath target driver within a device mapper map or table. The latter case also involves suspending and resuming io to the mapped device during the time the map is updated.

Path group and path state are also pushed into the kernel by the multipath configurator independently of a multipath mapped device's map. A path's state can be either active or failed. Io is only directed by the multipath target driver to a path with an active path state. Currently a path's state is set to failed either by the multipath target driver after a single io failure on the path or by the multipath configurator after a path test failure. A path's state is restored to active only in user space after a multipath configurator initiated path test succeeds for that path. A path group can be placed into bypass mode, removed from bypass mode, or made the highest priority path group for a mapped device. When searching for the next path group to use when there are no active paths in the highest priority path group, unless a new path group has been designated as the highest priority group, all path groups are searched. Otherwise, path groups in bypass mode are first skipped over and selected only if there are no path groups for the mapped device which are not in bypass mode.

The path selection policy name must refer to an already kernel resident path selection policy module. Path selection policy modules register half dozen callbacks with the multipath target driver's path selection framework, the most important of which is invoked in the dispatch path of a bio by the multipath target driver to select the next path to use for the bio.

Io scheduling triggered during the multipath target driver's bio dispatch callout from the device mapper framework consists of first selecting a path group for the mapped device in question, then selecting the active path to use within that group, followed by redirecting the bio to the selected path. A cached value of the path group to use is saved with each multipath mapped device in order to avoid its recalculation for each bio redirection to that device. This cached value is initially set from to the highest priority path group and is recalculated if either

- the highest priority path group for a mapped device is changed from user space or
- the highest priority path group is put into bypassed mode either from kernel or user space multipathing components.

A cached value of the path to use within the highest priority group is recalculated by invoking the path selection callout of a path selection policy whenever

- a configurable number of bios have already been redirected on the current path,
- a failure occurs on the current path,
- any other path gets restored to a usable state, or
- the highest priority path group is changed via either of the two methods discussed earlier.

Due to architectural restrictions and the relatively (compared with physical drivers) high positioning of the multipath target driver in the block io stack, it is difficult to implement path selection policies which take into account the state of shared physical path resources without implementing significant new kernel resident mechanism. Path selection policies are limited in scope to the path members of a particular path group for a particular multipath mapped device. This multipath architectural restriction together with the difficulty in tracking resource utilization for physical path resources from a block level filter driver makes it difficult to implement path selection policies which could attempt to minimize the depth of target device request queues or the utilization of SCSI initiators. Path selectors tracking physical resources possibly shared amongst multiple hosts, (e.g., SCSI targets), face even more difficulties.

The path selection algorithms are also impacted architecturally by being positioned above the point at the bottom of the block io layer where bios are coalesced into io requests. To help deal with this impact, path reselection within a priority group is done only for every n bios, where n is a configurable repeat count value associated with each use of a path selection policy for a priority group. Currently the repeat count value is set to 1000 for all cases in order to limit the adverse throughput effects of dispersing bios amongst multiple paths to the same device, thereby negating the ability of the block io layer to coalesce these bios into larger io requests submitted to the request queue of bottom level target devices.

A single round-robin path selection policy exists today. This policy selects the least recently used active path in the current path group for the particular mapped device.

4.3 Io Failover

While actual failover of io to alternate paths is performed in the kernel, path failover is controlled via configuration and policy information pushed into the kernel multipath components from user space multipath components.

While the multipath target driver filters both io dispatch and completion for all bios sent to a multipath mapped device, io failover is triggered when an error is detected while filtering io completion. An understanding of the error handling taking place underneath the multipath target driver is useful at this point. Assuming SCSI target devices as leaf nodes of the device mapper device tree, the SCSI mid-layer followed by the SCSI disk class driver each parse the result field of the `scsi_cmd` structure set by the SCSI LLDD. While parsing by the SCSI mid-layer and class driver filter code filter out some error states as being benign, all other cases lead to failing all bios associated with the io request corresponding to the SCSI command with -EIO. For those SCSI errors which provide sense information, SCSI sense key, Additional Sense Code (ASC), and Additional Sense Code Qualifier (ASCQ) byte values are set in the `bi_error` field of each bio. The -EIO, SCSI sense key, ASC, and ASCQ are propagated to all parent cloned bios and are available for access by the any target driver managing target devices as the bio completions recurse back up to the top of the device tree.

Io failures are first seen as a non-zero error status, (i.e., -EIO), in the error parameter passed to the multipath target driver's io completion filter. This filter is called as a callout from the device mapper's bio completion callback associated with the leaf node bios. Assuming one exists, all io failures are first parsed by the storage system's hardware context's error handler. Error parsing drives what happens next for the path, path group, and bio associated with the io failure. The path can be put into a failed state or left unaffected. The path group can be placed into a bypassed state or left unaffected. The bio can be queued for retry internally within the multipath target driver or failed. The actions on the path, the path group, and the bio are independent of each other. A failed path is unusable until restored to a usable state from the user space multipath configurator. A bypassed path group is skipped over when searching for a usable path, unless there are no usable paths found in other non-bypassed path groups. A failed bio leads to the failure of all parent cloned bios at higher levels in the device tree.

Io retry is done exclusively in a dedicated multipath worker thread context. Using a worker thread context allows for blocking in the code path of an io retry which requires a path group initialization or which gets dispatched back to `generic_make_request`—either of which may block. This is necessary since the bio completion code path through the device mapper is usually done within a soft interrupt context. Using a dedicated multipath worker thread avoids delaying the servicing of non-multipath related work queue requests as would occur by using the kernel's default work queue.

Io scheduling for path failover follows basically the same path selection algorithm as that for an initial io dispatch which has exhausted its path repeat count and must select an alternate path. The path selector for the current path group selects the best alternative path within that path group. If none are available, the next highest priority path group is made current and its path selector selects the best available path. This algorithm iterates until all paths of all path groups have been tried.

The device mapper's kernel resident event mechanism enables user space applications to determine when io related events occur in the

kernel for a mapped device. Events are generated by the target driver managing a particular mapped device. The event mechanism is accessed via a synchronous device mapper library interface which blocks a thread in the kernel in order to wait for an event associated with a particular mapped device. Only the event occurrence is passed to user space. No other attribute information of the event is communicated.

The occurrence of a path failure event (along with path reinstatement and a change in the highest priority path group) is communicated from the multipath target driver to the multipath daemon via this event mechanism. A separate multipath daemon thread is allocated to wait for all multipath events associated with each multipath mapped device. The detection of any multipath event causes the multipath daemon to rediscover its path configuration and synchronize its path configuration, path state, and path group state information with the multipath target driver's view of the same.

A previously failed path is restored to an active state only as a result of passing a periodically issued path health test issued by the multipath daemon for all paths, failed or active. This path state transition is currently enacted by the multipath daemon invoking the multipath configurator as an executable.

A io failure is visible above the multipathing mapped device only when all paths to the same device have been tried once. Even then, it is possible to configure a mapped device to queue for an indefinite amount of time such bios on a queue specific to the multipath mapped device. This feature is useful for those storage systems which can possibly enter a transient all-paths-down state which must be ridden through by the multipath software. These bios will remain where they are until the mapped device is suspended, possibly done when the mapped device's map is updated, or when a previously failed path is reinstated. There are no practical limits on either the amount of bios which may be queued in this manner nor on the amount of time which these bios remain queued. Furthermore, there is no congestion control mechanism which will limit the number of bios actually sent to any device. These facts can lead to a significant amount of dirty pages being stranded in the page cache thereby setting the stage for potential system deadlock if memory resources must be dynamically allocated from the kernel heap anywhere in the code path of reinstating either the map or a usable path for the mapped device.

5 Future Enhancements

This section enumerates some possible enhancements to the multipath implementation.

5.1 Persistent Device Naming

The cryptic name used for the device file associated with a device mapper mapped device is often renamed by a user space component associated with the device mapper target driver managing the mapped device. The multipathing subsystem sets up udev configuration directives to automatically rename this name when a device mapper device file is first created. The dm-<minor #> name is changed to the ASCII representation of the hexi-decimal values for each 4-bit nibble of the device's UID utilized by multipath. Yet, the resultant multipath device names are still cryptic, unwieldly, and their use is prone to error. Although an alias name may be linked to each multipath device, the setup requires manipulcation of the multipath configuration file for each device. The automated management of multipath alias names by both udev and multipath components seems a reasonable next step.

It should be noted that the Persistent Storage Device Naming specification from the Storage Networking SIG of OSDL is attempting to achieve consistent naming across all block devices.

5.2 Event Mechanism

The device mapper's event mechanism enables user space applications to determine when io related events occur in the kernel for a mapped device. Events are generated by the target driver managing a particular mapped device. The event mechanism is currently accessed via a synchronous device mapper library interface which blocks a thread in the kernel in order to wait for an event associated with a particular mapped device. Only the event occurrence is passed back to user space. No other attribute information of the event is communicated.

Potential enhancements to the device mapper event mechanism are enumerated below.

1. Associating attributes with an event and providing an interface for communicating these attributes to user space will improve the effectiveness of the event mechanism. Possible attributes for multipath events include (1) the cause of the event, (e.g., path failure or other), (2) error or status information associated with the event, (e.g., SCSI sense key/ASC/ASCQ for a SCSI error), and (3) an indication of the target device on which the error occurred.

2. Providing a multi-event wait synchronous interface similar to select(2) or poll(2) will significantly reduce the thread and memory resources required to use the event mechanism. This enhancement will allow a single user thread to wait on events for multiple mapped devices.

3. A more radical change would be to integreate the device-mappers event mechanism with the kernel's kobject subsystem. Events could be send as uevents to be received over an `AF_NETLINK` socket.

5.3 Monitoring of io via Iostat(1)

Block io to device mapper mapped devices cannot currently be monitored via `iostat(1)` or `/proc/diskstats`. Although an io to a mapped device is tracked on the actual target device(s) at the bottom of the `generic_make_request` device tree, io statistics are not tracked for any device mapper mapped devices positioned within the device tree.

Io statistics should be tracked for each device mapper mapped device positioned on an io stack. Multipathing must account for possibly multiple io failures and subsequent io retry.

5.4 IO Load Sharing

Additional path selectors will be implemented. These will likely include state based ones which select a path based on the minimum number of outstanding bios or minimum round trip latency. While the domain for this criteria is likely a path group for one mapped device, it may be worth looking sharing io load across actual physical components, (e.g., SCSI initiator or target), instead.

5.5 Protocol Agnostic Multipathing

Achieving protocol agnostic multipathing will require the removal of some SCSI specific affinity in the kernel, (e.g., SCSI-specific error information in the bio), and user, (e.g., path discovery), multipath components.

5.6 Scalable Path Testing

Proactive path testing could be enhanced to support multiple path testing policies and new policies created which provide improved resource scalability and improve the predictability of path failures. Path testing could emphasize the testing of the physical components utilized by paths instead of simply exhaustively testing every logical path. For example, the availability through Sysfs of path transport specific attributes for SCSI paths could will make it easier to group paths which utilize common physical components. Additionally, the frequency of path testing can be based on the recent reliability of a path, that is, frequently and recently failed paths are more often.

6 Architectural Issues

This section describes several critical architectural issues.

6.1 Elevator Function Location

The linux block layer performs the sorting and merging of IO requests (elevator modules) in a layer just above the device driver. The dm device mapper supports the modular stacking of multipath and RAID functionality above this layer.

At least for the device mapper multipath module, it is desirable to either relocate the elevator functionality to a layer above the device mapper in the IO stack, or at least to add an elevator at that level.

An example of this need can be seen with a multipath configuration where there are four equivalent paths between the host and each target. Assume also there is no penalty for switching paths. In this case, the multipath module wants to spread IO evenly across the four paths. For each IO, it may choose a path based on which path is most lightly loaded.

With the current placement of the elevator then, IO requests for a given target tend to be spread evenly across each of the four paths to that target. This reduces the chances for request sorting and merging.

If an elevator were placed in the IO stack above the multipath layer, the IO requests coming into the multipath would already be sorted and merged. IO requests on each path would at least have been merged. When IO requests on different paths reach their common target, the IO's will may nolonger be in perfect sorted order. But they will tend to be near each other. This should reduce seeking at the target.

At this point, there doesn't seem to be any advantage to retaining the elevator above the device driver, on each path in the multipath. Aside from the additional overhead (more memory occupied by the queue, more plug/unplug delay, additional cpu cycles) there doesn't seem to be any harm from invoking the elevator at this level either. So it may be satisfactory to just allow multiple elevators in the IO stack.

Regarding other device mapper targets, it is not yet clear whether software RAID would benefit from having elevators higher in the IO stack, interspersed between RAID levels. So, it maybe be sufficient to just adapt the multipath layer to incorporate an elevator interface.

Further investigation is needed to determine what elevator algorithms are best for multipath. At first glance, the Anticipatory scheduler seems inappropriate. It's less clear how the deadline scheduler of CFQ scheduler would

perform in conjunction with multipath. Consideration should be given to whether a new IO scheduler type could produce benefits to multipath IO performance.

6.2 Memory Pressure

There are scenarios where all paths to a logical unit on a SCSI storage system will appear to be failed for a transient period of time. One such expected and transient all paths down use case involves an application transparent upgrade of the micro-code of a SCSI storage system. During this operation, it is expected that for a reasonably short period of time likely bounded by a few minutes, all paths to a logical unit on the storage system in question will appear to a host to be failed. It is expected that a multipathing product will be capable of riding through this scenario without failing ios back to applications. It is expected that the multipathing software will both detect when one or more of the paths to such a device become physically usable again, do what it takes to make the paths usable, and retry ios which failed during the all paths down time period.

If this period coincides with a period of extreme physical memory congestion it must still be possible for multipath components to enable the use of these paths as they become physically usable. While a kernel resident congestion control mechanism based on block request allocation exists to ward off the over committal of page cache memory to any one target device, there are no congestion control mechanisms that take into account either the use of multiple target devices for the same mapped device or the internal queuing of bios within device mapper target drivers.

The multipath configuration for several storage systems must include the multipath feature `queue_if_no_path` in order to not immediately return to an application an io request whose transfer has failed on every path to its device. Yet, the use of this configuration directive can result in the queuing of an indefinite number of bios each for an indefinite period of time when there are no usable paths to a device. When coincident with a period of heavy asynchronous write-behind in the page cache, this can lead to lots of dirty page cache pages for the duration of the transient all paths down period.

Since memory congestion states like this cannot be detected accurately, the kernel and user code paths involved with restoring a path to a device must not ever execute code which could result in blocking while an io is issued to this device. A blockable (i.e., `__GFP_WAIT`) memory allocation request in this code path could block for write-out of dirty pages to this device from the synchronous page reclaim algorithm of `__alloc_pages`. Any modification to file system metadata or data could block flushing modified pages to this device. Any of these actions have the potential of deadlocking the multipathing software.

These requirements are difficult to satisfy for multipathing software since user space intervention is required to restore a path to a usable state. These requirements apply to all user and kernel space multipathing code (and code invoked by this code) which is involved in testing a path and restoring it to a usable state. This precludes the use of fork, clone, or exec in the user portion of this code path. Path testing initiated from user space and performed via ioctl entry to the block scsi ioctl code must also conform to these requirements.

The pre-allocation of memory resources in order to make progress for a single device at a time is a common solution to this problem. This approach may require special case code for tasks such as the kernel resident path testing. Furthermore, in addition to being “locked to core,” the user space components must only

invoke system calls and library functions which also abide by these requirements. Possibly combining these approaches with a bit of congestion control applied against bios (to account for the ones internally queued in device-mapper target drivers) instead of or in addition to block io requests and/or a mechanism for timing out bios queued within the multipath target driver as a result of the `queue_if_no_path` multipath feature is a reasonable starting point.

7 Conclusion

This paper has analyzed both architecture and design of the block device multipathing indigenous to linux. Several architectural issues and potential enhancements have been discussed.

The multipathing architecture described in this paper is actually implemented in several linux distributions to be released around the time this paper is being written. For example, SuSE SLES 9 service pack 2 and Red Hat AS 4 update 1 each support Linux multipathing. Furthermore, several enhancements described in this paper are actively being pursued.

Please reference `http://christophe.varoqui.free.fr` and `http://sources.redhat.com/dm` for the most up-to-date development versions of the user- and kernel-space resident multipathing software respectively. The first web site listed also provides a detailed description of the syntax for a multipathing device-mapper map.

Kdump, A Kexec-based Kernel Crash Dumping Mechanism

Vivek Goyal
IBM
vgoyal@in.ibm.com

Eric W. Biederman
Linux NetworkX
ebiederman@lnxi.com

Hariprasad Nellitheertha
IBM
hari@in.ibm.com

Abstract

Kdump is a kexec based kernel crash dumping mechanism, which is being perceived as a reliable crash dumping solution for Linux®. This paper begins with brief description of what kexec is and what it can do in general case, and then details how kexec has been modified to boot a new kernel even in a system crash event.

Kexec enables booting into a new kernel while preserving the memory contents in a crash scenario, and kdump uses this feature to capture the kernel crash dump. Physical memory layout and processor state are encoded in ELF core format, and these headers are stored in a reserved section of memory. Upon a crash, new kernel boots up from reserved memory and provides a platform to retrieve stored ELF headers and capture the crash dump. Also detailed are ELF core header creation, dump capture mechanism, and how to configure and use the kdump feature.

1 Introduction

Various crash dumping solutions have been evolving over a period of time for Linux and other UNIX® like operating systems. All solutions have their pros and cons, but the most important consideration for the success of a solution has been the reliability and ease of use. Kdump is a crash dumping solution that provides a very reliable dump generation and capturing mechanism [01]. It is simple, easy to configure and provides a great deal of flexibility in terms of dump device selection, dump saving mechanism, and plugging-in filtering mechanism.

The idea of kdump has been around for quite some time now, and initial patches for kdump implementation were posted to the Linux kernel mailing list last year [03]. Since then, kdump has undergone significant design changes to ensure improved reliability, enhanced ease of use and cleaner interfaces. This paper starts with an overview of the kdump design and development history. Then the limitations of existing designs are highlighted and this paper goes on to detail the new design and enhancements.

Section 2 provides background of kexec and kdump development. Details regarding how kexec has been enhanced to boot-into a new kernel in panic events are covered in section 3. Section 4 details the new kdump design. Details about how to configure and use this mechanism are captured in Section 5. Briefly discussed are advantages and limitations of this approach in section 6. A concise description of current status of project and TODOs are in-

cluded in Section 7.

2 Background

This section provides an overview of the kexec and original kdump design philosophy and implementation approach. It also brings forward the design deficiencies of kdump approach so far, and highlights the requirements that justified kexec and kdump design enhancements.

2.1 Kexec

Kexec is a kernel-to-kernel boot-loader [07], which provides the functionality to boot into a new kernel, over a reboot, without going through the BIOS. Essentially, kexec pre-loads the new kernel and stores the kernel image in RAM. Memory required to store the new kernel image need not be contiguous and kexec keeps a track of pages where new kernel image has been stored. When a reboot is initiated, kexec copies the new kernel image to destination location from where the new kernel is supposed to run, and after executing some setup code, kexec transfers the control to the new kernel.

Kexec functionality is constituted of mainly two components; kernel space [08] and user space [02]. Kernel space component implements a new system call `kexec_load()` which facilitates pre-loading of new kernel. User space component, here onwards called kexec tools, parses the new kernel image, prepares the appropriate parameter segment, and setup code segment and passes the this data to the running kernel through newly implemented system call for further processing.

2.2 A Brief History of Kdump Development

The core design principle behind this approach is that dump is captured with the help of a custom built kernel that runs with a small amount of memory. This custom built kernel is called capture kernel and is booted into upon a system crash event without clearing crashed kernel's memory. Here onwards, for discussion purposes, crashing kernel is referred to as first kernel and the kernel which captures the dump after a system crash is called capture kernel.

While capture kernel boots, first kernel's memory is not overwritten except for the small amount of memory used by new kernel for its execution. Kdump used this feature of kexec and added hooks in kexec code to boot into a capture kernel in a panic event without stomping crashed kernel's memory.

Capture kernel used the first 16 MB of memory for booting and this region of memory needed to be preserved before booting into capture kernel. Kdump added the functionality to copy the contents of the first 16 MB to a reserved memory area called backup region. Memory for the backup region was reserved during the first kernel's boot time, and location and size of the backup region was specified using kernel config options. Kdump also copied over the CPU register states to an area immediately after the backup region during a crash event [03].

After the crash event, the system is unstable and usual device shutdown methods can not be relied upon, hence, devices are not shutdown after a crash. This essentially means that any ongoing DMAs at the time of crash are not stopped. In the above approach, the capture kernel was booting from the same memory location as the first kernel (1 MB) and used first 16 MB to boot, hence, it was prone to corruption due to any on-going DMA in that re-

gion. An idea was proposed and a prototype patch was provided for booting the capture kernel from a reserved region of memory instead of a default location. This reduced the chances of corruption of the capture kernel due to ongoing DMA [04] [05]. Kdump's design was updated to accommodate this change and now the capture kernel booted from reserved location. This reserved region was still being determined by kernel config options [06].

Despite the fact that the capture kernel was booting from a reserved region of memory, it needed first 640 KB of memory to boot for SMP configurations. This memory was required to retrieve configuration data like the MP configuration table saved by BIOS while booting the first kernel. It was also required to place the trampoline code needed to kick-start application processors in the system. Kdump reserved 640 KB of memory (backup region) immediately after the reserved region, and preserved the first 640 KB of memory contents by copying it to a backup region just before transferring control to capture kernel. CPU register states were being stored immediately after the backup region [06].

After booting, capture kernel retrieved the saved register states and backup region contents, and made available the old kernel's dump image through two kernel interfaces. The first one was through the `/proc/vmcore` interface, which exported the dump image in ELF core format, and other one being the `/dev/oldmem`, which provided a linear raw view of memory.

2.3 Need for Design Enhancement

Following are some of the key limitations of the above approach that triggered the design enhancement of kdump.

1. In the design above, kexec pre-loads the capture kernel wherever it can manage to grab a page frame. At the time of crash, the capture kernel image is copied to the destination location and control is transferred to the new kernel. Given the fact that the capture kernel runs from a reserved area of memory, it can be loaded there directly and extra copying of kernel can be avoided. In general terms, kexec can be enhanced to provide a fast reboot path to handle booting into a new kernel in crash events also.

2. Capture kernel and the associated data is pre-loaded and stored in the kernel memory, but there is no way to detect any data corruption due to faulty kernel programming.

3. During the first kernel boot, kdump reserves a chunk of memory for booting the capture kernel. The location of this region is determined during kernel compilation time with the help of config options. Determining the location of reserved region through config options is a little cumbersome. It brings in hard-coding in many places, at the same time it is static in nature and a user has to compile the kernel again if he decides to change the location of reserved region.

4. Capture kernel has to boot into a limited amount of memory, and to achieve this, the capture kernel is booted with user defined memory map with the help of `memmap=exactmap` command line options. User has to provide this user defined memory map while pre-loading the capture kernel and need to be explicitly aware of memory region reserved for capture kernel. This process can be automated by kexec tools and these details can be made opaque to the user.

5. When the capture kernel boots up, it needs to determine the location of the backup region to access the crashed kernel's backed-up memory contents. Capture kernel receives this information through hard coded config options. It also retrieves the saved register states assuming these to be stored immediately after the backup region and this introduces another level of hard-coding.

   In this approach, the capture kernel is explicitly aware of the presence of the backup region, which can be done away with. In general, there is no standard format for the exchange of information between two kernels which essentially makes two kernel dependent on each other, and it might now allow kernel skew between the first kernel and the capture kernel as kernel development progresses.

6. The `/proc/vmcore` implementation does not support discontiguous memory systems and assumes memory is contiguous, hence exports only one ELF program header for the whole of the memory.

3 Kexec On Panic

Initially, kexec was designed to allow booting a new kernel from a sane kernel over a reboot. Emergence of kdump called for kexec to allow booting a new kernel even in a crash scenario. Kexec has now been modified to handle system crash events, and it provides a separate reboot path to a new kernel in panic situations.

Kexec as a boot-loader supports loading of various kinds of images for a particular platform. For i386, vmlinux, bzImage, and multiboot images can be loaded. Capture kernel is compiled to load and run from a reserved memory location which does not overlap with the first kernel's memory location (1 MB). However, currently only a vmlinux image can be used as a capture kernel. A bzImage can not be used as capture kernel because even if it is compiled to run from a reserved location, it always first loads at 1 MB and later it relocates itself to the memory location it was compiled to run from. This essentially means that loading bzImage shall overwrite the first kernel's memory contents at 1 MB location and that is not the desired behavior.

From here on out the discussion is limited to the loading of a vmlinux image for i386 platform. Details regarding loading of other kind of images is outside the scope of this paper.

3.1 Capture Kernel Space Reservation

On i386, the default location a kernel runs from is 1 MB. The capture kernel is compiled and linked to run from a non default location like 16 MB. The first kernel needs to reserve a chunk of memory where the capture kernel and associated data can be pre-loaded. Capture kernel will directly run from this reserved memory location. This space reservation is done with the help of `crashkernel=X@Y` boot time parameter to first kernel, where `X` is the the amount of memory to be reserved and `Y` indicates the location where reserved memory section starts.

3.2 Pre-loading the Capture Kernel

Capture kernel and associated data are pre-loaded in the reserved region of memory. Kexec tools parses the capture kernel image and loads it in reserved region of memory using `kexec_load()` system call. Kexec tools manage a contiguous chunk of data belonging to the same group in the form of segment. For example, bzImage code is considered as one

segment, parameter block is treated as another segment and so on. Kexec tools parses the capture kernel image and prepares a list of segments and passes the list to kernel. This list basically conveys the information like location of various data blocks in user space and where these blocks have to be loaded in reserved region of memory. `kexec_load()` system call does the verification on destination location of a segments and copies the segment data from user space to kernel space. Capture kernel is directly loaded into the memory where it is supposed to run from and no extra copying of capture kernel is required.

`purgatory` is an ELF relocatable object that runs between the kernels. Apart from setup code, purgatory also implements a sha256 hash to verify that loaded kernel is not corrupt. In addition, `purgatory` also saves the contents to backup region after the crash (section 4.3).

Figure 1 depicts one of the possible arrangements of various segments after being loaded into a reserved region of memory. In this example, memory from 16 MB to 48 MB has been reserved for loading capture kernel.

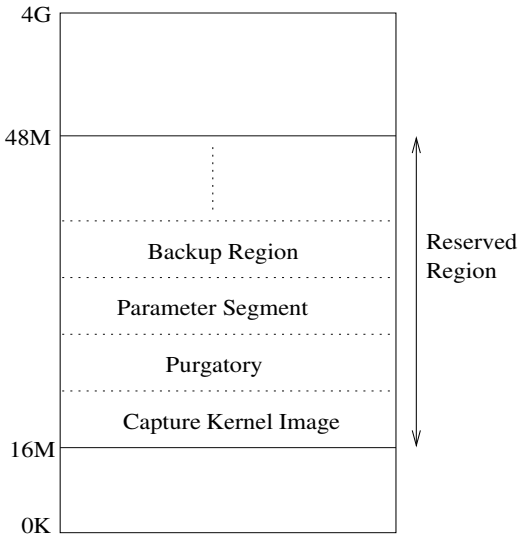


Figure 1: Various Data Segments in Reserved Region

3.3 Post Crash Processing

Upon a crash, kexec performs a minimum machine shutdown procedure and then jumps to the `purgatory` code. During machine shutdown, crashing CPU sends the NMI IPIs to other processors to halt them. Upon receiving NMI, the processor saves the register state, disables the local APIC and goes into halt state. After stopping the other CPUs, crashing CPU disables its local APIC, disables IOAPIC, and saves its register states.

CPU register states are saved in ELF note section format [09]. Currently the processor status is stored in note type NT_PRSTATUS at the time of crash. Framework provides enough flexibility to store more information down the line, if needed. One kilobyte of memory is reserved for every CPU for storing information in the form of notes. A final null note is appended at the end to mark the end of notes. Memory for the note section is allocated statically in the kernel and the memory address is exported to user space through `sysfs`. This address is in turn used by kexec tools while generating the ELF headers (Section 4.2).

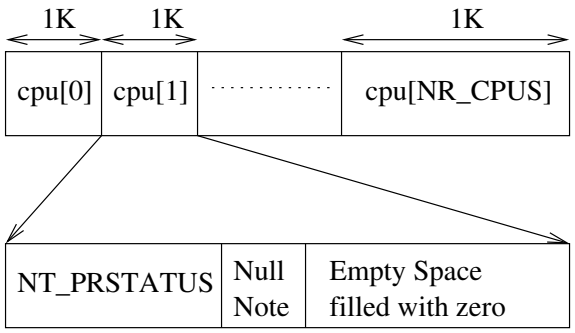


Figure 2: Saving CPU Register States

After saving register states, control is transferred to `purgatory`. `purgatory` runs sha256 hash to verify the integrity of the capture kernel and associated data. If no corruption is detected, purgatory goes on to copy the first

640 KB of memory to the backup region (Section 4.3). Once the backup is completed control flow jumps to start of the new kernel image and the new kernel starts execution.

4 Kdump

Previous kdump design had certain drawbacks which have been overcome in the new design. Following section captures the details of the new kdump design.

4.1 Design Overview

Most of the older crash dumping solutions have had the drawback of capturing/writing out the dump in the context of crashing kernel, which is inherently unreliable. This led to the idea of first booting into a sane kernel after the crash and then capturing the dump. Kexec enables kdump to boot into the already loaded capture kernel without clearing the memory contents and this sets the stage for a reliable dump capture.

The dump image can be represented in many ways. It can be a raw snapshot of memory read from a device interface similar to `/dev/mem`, or it can be exported in ELF core format. Exporting a dump image in ELF core format carries the advantage of being a standard approach for representing core dumps and provides the compatibility with existing analysis tools like `gdb`, `crash`, and so on. Kdump provides ELF core view of a dump through `/proc/vmcore` interface and at the same time it also provides `/dev/oldmem` interface presenting linear raw view of memory.

ELF core headers encapsulate the information like processor registers, valid RAM locations, and backup region, if any. ELF headers are prepared by kexec tools and stored in a reserved memory location along with other segments as shown in Figure 3.

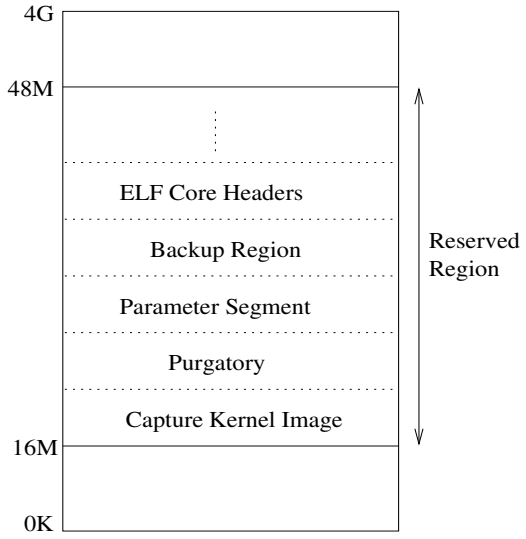


Figure 3: ELF Core Headers in Reserved Region

Memory for ELF core headers is reserved by bootmem allocator during first kernel boot using the `reserve_bootmem()` function call. Upon crash, system boots into new kernel and stored ELF headers are retrieved and exported through `/proc/vmcore` interface.

This provides a platform for capturing the dump image and storing it for later analysis. Implementation details are discussed in following sections of this paper.

4.2 ELF Core Header Generation

Kdump uses the ELF core format to exchange the information about dump image, between two kernels. ELF core format provides a generic and flexible framework for exchange of dump information. The address of the start of these headers is passed to the new kernel through a command line option. This provides

a cleaner interface between the two kernels, and at the same time ensures that the two kernels are independent of each other. It also allows kernel skew between the crashing kernel and the capture kernel, which essentially means that version of the crashing kernel and the capture kernel do not need to be the same. Also, an older capture kernel should be able to capture the dump for a relatively newer first kernel.

Kexec tools are responsible for ELF core header generation. ELF64 headers are sufficient to encode all the required information, but `gdb` can not open a ELF64 core file for 32 bit systems. Hence, kexec also provides a command line option to force preparation of ELF32 headers. This is useful for the users with non-PAE systems.

One PT_LOAD type program header is created for every contiguous memory chunk present in the system. Information regarding valid RAM locations is obtained from `/proc/iomem`. Considering system RAM as a file, physical address represents the offset in the file. Hence the `p_offset` field of program header is set to actual physical address of the memory chunk. `p_paddr` is the same as `p_offset` except in case of a backup region (Section 4.3). Virtual address (`p_vaddr`) is set to zero except for the linearly mapped region as virtual addresses for this region can be determined easily at the time of header creation. This allows a restricted debugging with `gdb` directly, without assistance from any other utility used to fill in virtual addresses during post crash processing.

One PT_NOTE type program header is created per CPU for representing note information associated with that CPU. Actual notes information is saved at the time of crash (Section 3.3), but PT_NOTE type program header is created in advance at the time of loading the capture kernel. The only information required at this point is the address of location where actual notes section reside. This address is exported to user space through `sysfs` by kexec. Kexec user space tools read in the `/sys/kernel/crash_notes` file and prepare the PT_NOTE headers accordingly.

In the event of memory hotplug, the capture kernel needs to be reloaded so that the ELF headers are generated again reflecting the changes.

4.3 Backup Region

Capture kernel boots from the reserved area of memory after a crash event. Depending on the architecture, it may still need to use some fixed memory locations that were used by the first kernel. For example, on i386, it needs to use the first 640 KB of memory for trampoline code for booting SMP kernel. Some architectures like ppc64 need fixed memory locations for storing exception vectors and other data structures. Contents of these memory locations are copied to a reserved memory area (backup region) just after crash to avoid any stomping by the capture kernel. `purgatory` takes care of copying the contents to backup region (Section 3.2).

Capture kernel/capture tool need to be aware of the presence of a backup region because effectively some portion of the physical memory has been relocated. ELF format comes in handy here as it allows to envelop this information without creating any dependencies. A separate PT_LOAD type program header is generated for the backup region. The `p_paddr` field is filled with the original physical address and the `p_offset` field is populated with the relocated physical address as shown in Figure 4.

Currently, kexec user space tools provide the backup region handling for i386, and the first 640 KB of memory is backed-up. This code is more or less architecture dependent. Other architectures can define their own backup regions

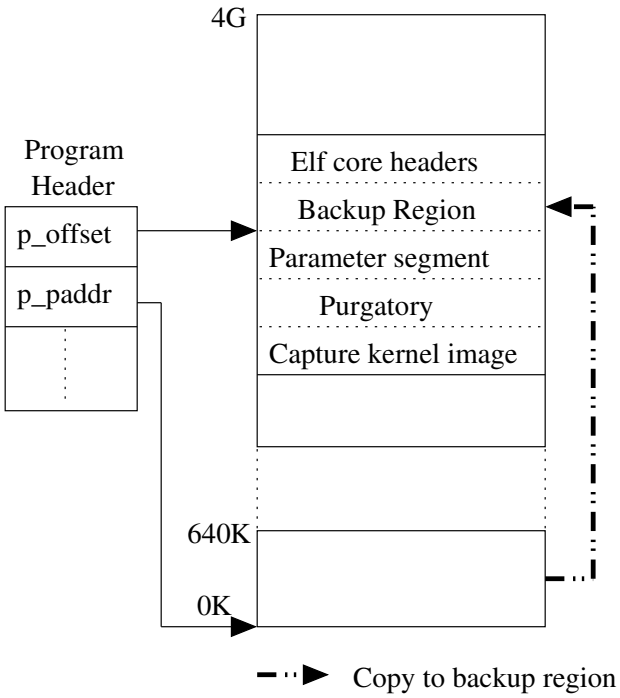


Figure 4: Saving Contents To Backup Region

and plug-in the implementations into existing kexec user space code.

4.4 Booting into Capture Kernel

The capture kernel is compiled to boot from a non-default memory location. It should not stomp over crashed kernel's memory contents to be able to retrieve a sane dump. Hence, capture kernel is booted with an user defined memory map instead of the one provided by BIOS or one passed in parameter segment by kexec. The command line option `memmap=exactmap` along with `memmap=X@Y` is used to override BIOS provided memory map and define user memory map.

These boot time parameters are automatically added to command line by kexec tools while loading the capture kernel and it's details are opaque to user. Internally, kexec prepares a list of memory regions that the capture kernel can safely use to boot into, and appropriate `memmap` options are appended to the command line accordingly. The backup region and ELF header segments are excluded from this list to avoid stomping of these memory areas by new kernel.

Address of start of ELF header segment is passed to the capture kernel through the `elfcorehdr=` command line option. This option is also added automatically to command line by kexec tools.

4.5 Dump Capture Mechanism

Once the capture kernel has booted there are multiple design options for dump capturing mechanism. Few of them are as following.

- **Kernel Space**

  Export ELF core image through the `/proc/vmcore` interface which can be directly used by ELF core format aware analysis tools such as `gdb`. Also export raw linear view of memory through device interface `/dev/oldmem`. Other crash analysis tools can undergo required modifications to adapt to these formats.

  This is an easy to use solution which offers a wide variety of choices. Standard tools like `cp`, `scp`, and `ftp` can be used to copy the image to the disk either locally or over the network. `gdb` can be used directly for limited debugging. The flip side is that the `/proc/vmcore` code is in the kernel and debugging the kernel code is relatively harder.

- **User Space**
  User space utilities which read the raw

physical memory through suitable interfaces like `/dev/oldmem` and write out the dump image.

- **Early User Space**
  Utilities that run from initial ramdisk and perform a raw dump to pre-configured disk. This approach is especially useful in a scenario when root file system happens to be corrupted after the crash.

For now, we stick to kernel space implementation and other solutions (user space or early user space) can evolve slowly to cater to wide variety of requirements. The following sections cover implementation details.

4.5.1 Accessing Dump Image in ELF Core Format

ELF core headers, as stored by crashed kernel, are parsed and the dump image is exported to user space through `/proc/vmcore`. Backup region details are abstracted in ELF headers, and `/proc/vmcore` implementation is not even aware of the presence of the backup region. The physical address of the start of the ELF header is passed to the capture kernel through the `elfcorehdr=` command line option. Stored ELF headers undergo a sanity check during the `/proc/vmcore` initialization and if valid headers are found then initialization process continues otherwise `/proc/vmcore` initializaiton is aborted and the `vmcore` file size is set to zero.

CPU register states are saved in note sections by crashing kernel and one PT_NOTE type program header is created for every CPU. To be fully compatible with ELF core format, all the PT_NOTE program headers are merged into one during the `/proc/vmcore` initialization. Figure 5 depicts what a `/proc/vmcore` exported ELF core images looks like.

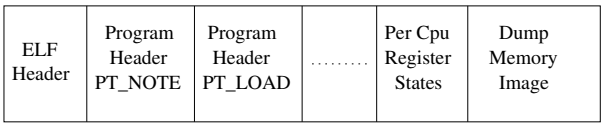


Figure 5: ELF Core Format Dump Image

Physical memory can be discontiguous and this means that offset in the core file can not directly map to a physical address unless memory holes are filled with zeros in the core file. On some architectures like IA64, holes can be big enough to deter one from taking this approach.

This new approach does not fill memory holes with zeros, instead it prepares one program header for every contiguous memory chunk. It maintains a linked list in which each element represents one contiguous memory region. This list is prepared during init time and also contains the data to map a given offset to respective physical address. This enables `/proc/vmcore` to determine where to get the contents from associated with a given offset in ELF core file when a read is performed.

`gdb` can be directly used with `/proc/vmcore` for limited debugging. This includes processor status at the time of crash as well as analyzing linearly mapped region memory contents. Non-linearly mapped areas like vmalloced memory regions can not be directly analyzed because kernel virtual addresses for these regions have not been filled in ELF headers. Probably a user space utility can be written to read in the dump image, determine the virtual to physical address mapping for vmalloced regions and export the modified ELF headers accordingly.

Alternatively, the `/proc/vmcore` interface can be enhanced to fill in the virtual addresses in exported ELF headers. Extra care needs to be taken while handling it in kernel space because determining the virtual to physical mapping shall involve accessing VM data structures

of the crashed kernel, which are inherently unreliable.

4.5.2 Accessing Dump Image in linear raw Format

The dump image can also be accessed in linear raw format through the `/dev/oldmem` interface. This can be especially useful for the users who want to selectively read out portions of the dump image without having to write out the entire dump. This implementation of `/dev/oldmem` does not possess any knowledge of the backup region. It's a raw dummy interface that treats the old kernel's memory as high memory and accesses its contents by stitching up a temporary page table entry for the requested page frame. User space application needs to be intelligent enough to read in the stored ELF headers first, and based on these headers retrieve rest of the contents.

5 How to Configure and Use

Following is the detailed procedure to configure and use the kdump feature.

1. Obtain a kernel source tree containing kexec and kdump patches.
2. Obtain appropriate version of kexec-tools.
3. Two kernels need to be built in order to get this feature working. The first kernel is the production kernel and the second kernel is the capture kernel. Build the first kernel as follows.
   - Enable `kexec system call` feature.
   - Enable `sysfs file system support` feature.
4. Build the capture kernel as follows.
   - Enable `kernel crash dumps` feature.
   - The capture kernel needs to boot from the memory area reserved by the first kernel. Specify a suitable value for `Physical address where kernel is loaded`.
   - Enable `/proc/vmcore` support. (Optional)
5. Boot into the first kernel with the commandline `crashkernel=Y@X`. Pass appropriate values for X and Y. Y denotes how much memory to reserve for the second kernel and X denotes at what physical address the reserved memory section starts. For example, `crashkernel=32M@16M`
6. Preload the capture kernel using following commandline.

   ```
   kexec -p <capture kernel>
   --crash-dump --args-linux
   --append="root=<root-dev>
   maxcpus=1 init 1"
   ```

7. Either force a panic or press `Alt SysRq c` to force execution of kexec on panic. System reboots into the capture kernel.
8. Access and save the dump file either through the `/proc/vmcore` interface or the `/dev/oldmem` interface.
9. Use appropriate analysis tool for debugging. Currently `gdb` can be used with the `/proc/vmcore` for limited debugging.

6 Advantages and Limitations

Every solution has its advantages and limitations and kdump is no exception. Section 6.1

highlights the advantages of this approach and limitations have been captured in Section 6.2.

6.1 Advantages

- More reliable as it allows capturing the dump from a freshly booted kernel as opposed to some of other methods like `LKCD`, where dump is saved from the context of crashing kernel, which is inherently unreliable.
- Offers much more flexibility in terms of choosing the dump device. As dump is captured from a newly booted kernel, virtually it can be saved to any storage media supported by kernel.
- Framework is flexible enough to accommodate filtering mechanism. User space or kernel space based filtering solutions can be plugged in, unlike firmware based solutions. For example, a kernel pages only filter can be implemented on top of the existing infrastructure.

6.2 Limitations

- Devices are not shutdown/reset after a crash, which might result in driver initialization failure in capture kernel.
- Non-disruptive dumping is not possible.

7 Status and TODOS

Kdump has been implemented for i386 and initial set of patches are in -mm tree. Following are some of the TODO items.

- Harden the device drivers to initialize properly in the capture kernel after a crash event.
- Modify `crash` to be able to analyze kdump generated crash dumps.
- Port kdump to other platforms like x86_64 and ppc64.
- Implement a kernel pages only filtering mechanism.

8 Conclusions

Kdump has made significant progress in terms of overcoming some of the past limitations, and is on its way to become a mature crash dumping solution. Reliability of the approach is further bolstered with the capture kernel now booting from a reserved area of memory, making it safe from any DMA going on at the time of crash. Dump information between the two kernels is being exchanged via ELF headers, providing more flexibility and allowing kernel skew. Usability of the solution has been further enhanced by enabling the kdump to support PAE systems and discontiguous memory.

Capture kernel provides `/proc/vmcore` and `/dev/oldmem` interfaces for retrieving the dump image, and more dump capturing mechanisms can evolve based on wide variety of requirements.

There are still issues with driver initialization in the capture kernel, which need to be looked into.

References

[01] Hariprasad Nellitheertha, *The kexec way to lightweight reliable system crash dumping*, Linux Kongress, 2004.

[02] The latest kexec tools patches, `http://www.xmission.com/~ebiederm/files/kexec/`

[03] Initial Kdump patches, `http://marc.theaimsgroup.com/?l=linux-kernel&m=109274443023485&w=2`

[04] Booting kernel from non default location patch (bzImage), `http://www.xmission.com/~ebiederm/files/kexec/2.6.8.1-kexec3/broken-out/highbzImage.i386.patch`

[05] Booting kernel from non default location patch (vmlinux), `http://www.xmission.com/~ebiederm/files/kexec/2.6.8.1-kexec3/broken-out/vmlinux-lds.i386.patch`

[06] Improved Kdump patches `http://marc.theaimsgroup.com/?l=linux-kernel&m=109525293618694&w=2`

[07] Andy Pfiffer, Reducing System Reboot Time with kexec, `http://developer.osdl.org/rddunlap/kexec/whitepaper/kexec.pdf`

[08] Hariprasad Nellitheertha, Reboot Linux Faster using kexec, `http://www-106.ibm.com/developerworks/linux/library/l-kexec.html`

[09] Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification (version 1.2)

Acknowledgments

The authors wish to express their sincere thanks to Suparna Bhattacharya who had been continuously providing ideas and support. Thanks to Maneesh Soni for numerous suggestions, reviews and feedback. Thanks to all the others who have helped us in our efforts.

Legal Statement

The Novell Linux Kernel Debugger, NLKD

Clyde Griffin
Novell, Inc.
cgriffin@novell.com

Jan Beulich
Novell, Inc.
jbeulich@novell.com

Abstract

In this paper we introduce the Novell Linux Kernel Debugger. After a brief introduction we will go into an in-depth discussion of NLKD's architecture. Following the architecture discussion we will cover some of the features supported by NLKD and its supported debug agents. We wrap up the discussion with some of the work items that still need to be done and follow with a brief conclusion.

1 Introduction

NLKD began its life as an R&D project in 1998 by Novell engineers Jan Beulich and Clyde Griffin. The effort to build a new debugger was driven by a need for a robust kernel debugger for future operating systems running on Intel's Itanium Processor Family.

The project was a success and soon there was demand for similar functionality on other hardware architectures. The debugger has since been ported for use on x86, x86-64, and EM64T.

Novell has never formally shipped this debugger as a stand-alone product or with any other Novell products or operating systems. To dispel any myths about its origin, it was never targeted for or used with Novell NetWare. It remained a research effort until the summer of 2004 when Novell engineering determined that the capabilities of this tool would be a boost to Linux development and support teams.

At the time of the publication of this paper, NLKD is functional on x86, x86-64, and EM64T SUSE Linux platforms. A port to IA64 Linux is pending.

1.1 Non-Goals

While we believe NLKD is one of the most stable and capable kernel debuggers available on Linux, we in no way want to force other developers to use this tool. We, like most developers on Linux, have our personal preferences and enjoy the freedom to use the right tool for the job at hand. To this end, NLKD is separated into layers, any of which could benefit existing debugging practices. At the lowest level, our exception handling framework could add stability and flexibility to existing Linux kernel debuggers. The Core Debug Engine can be controlled by add-on debug agents. As a final example, NLKD ships with a module that understands GDB's wire protocol, so that remote kernel debugging can be done with GDB or one of the many GDB interfaces.

1.2 Goals

Novell's primary interest in promoting NLKD is to provide a robust debugging experience

for kernel development engineers and enable support organizations to provide quick response times on critical customer support issues. While Novell development may favor NLKD as its primary kernel debugger, Novell will continue to support other kernel debugger offerings as long as sufficient demand exists.

NLKD has been released under the GPL with Novell retaining the copyright for the original work. Novell plans to ship NLKD as part of the SUSE distribution and at the same time enable it for inclusion into the mainline Linux kernel.

2 NLKD Architecture

Like any kernel debugger, at the core of NLKD is a special purpose exception handler. However, unlike many exception handlers, kernel debuggers must be able to control the state of other processors in the system in order to ensure a stable debugging experience. The fact that all processors in the system can generate simultaneous exceptions complicates the issue and makes the solution even more interesting.

Getting all processors into a quiescent state for examination has been a common challenge for multiprocessor kernel debuggers. Sending these processors back to the run state with a variety of debug conditions attached can be even more challenging, especially when processors are in critical sections of kernel code or operating on the same set of instructions.

The architecture that we describe here deals with this complex set of issues in a unique way, providing the user with a stable debugging experience.

In the following discussion we introduce the major components comprising NLKD. These include the exception handling framework supporting NLKD, the Core Debug Engine (CDE), and the debug agents that plug into CDE. CDE is a complex piece, so we spend extra time discussing its state machine and breakpoint logic. Figure 1 depicts these components and their interactions.

So let's start with the exception handling framework.

2.1 Exception Handling Framework

The first task in providing a robust debugging experience is to get an exception handling framework in place that properly serializes exception handlers according to function and priority.

While NLKD does not define the exception handling framework, our research at Novell has led us to a solution that solves the problem in a simple and elegant way.

The first thing to recognize is that not all exception handlers are created equal. For some exceptions, all registered handlers must be called no matter what. The best example of this is the x86 NMI. Other handlers are best called serially and others round robin. We should also note that interrupt handlers sharing a single interrupt vector should be called round robin to avoid priority inversion or starvation. Some exception handlers are passive and do nothing but monitor events and these, too, must be called in the right order.

To enable this flexibility, we defined a variety of exception handler types. They are: Exception Entry Notifiers, Registered Exception Handlers, Debug Exception Handler, Default Exception Handler, and Exception Exit Notifiers. Each of these handler types have strict semantics, such as how many handlers of each type may be registered, and whether all or just one

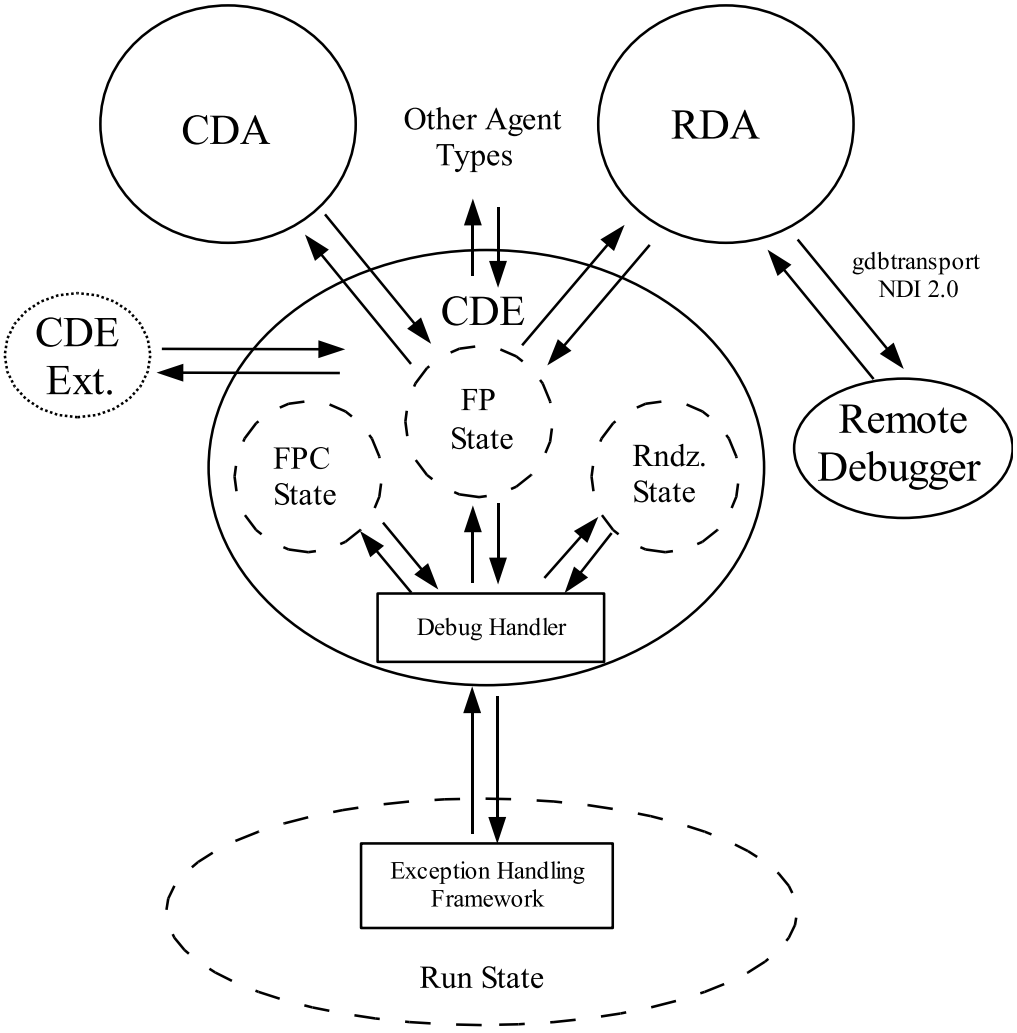


Figure 1: Architecture and state transitions

is called when an exception occurs. The various exception handler types are called in a well-defined order. Taken together, these rules ensure the system remains stable, and event counters remain correct, in all debugging situations.

The following sections describe the registration and calling conventions for each of these handler types. The handler types are listed in the order they are called.

Exception Entry Notifiers An exception entry notifier is a passive handler that does not change the state of the stack frame. It is typically used for monitors that want to know when an exception has occurred and what type it is. Zero or more such handlers may be registered, and all will be called.

Registered Exception Handlers These exception handlers are dynamically registered at runtime. If any of these handlers claim the exception, then no other registered exception handlers, nor the debug handler, are called. Zero or more such handlers may be registered.

Debug Exception Handler The debug exception handler invokes the debugger. (This may be NLKD or any other kernel debugger.) At most, one such handler may exist. If no debugger was registered, the exception is passed on to the default exception handler.

Default Exception Handler The kernel's default exception handler is included at compile time. Depending upon the exception type, it may cause the kernel to panic if no other handlers have claimed the exception.

Exception Exit Notifiers This is a passive handler that does not change the state of the stack frame. It is typically used for monitors wanting to know that an exception has been completed and what type it was. Zero or more such handlers may be registered, and all will be called.

The overhead of such an exception handler framework is extremely lightweight. For example:

```
// Multiple handlers test/call loop
while (ptr1) {
    if (ptr1->handler() == HANDLED)
        break;
    ptr1 = ptr1->next;
}
// Single handler test/call
if (ptr2)
    ptr2->handler();
```

There is very little overhead in this scheme, yet great flexibility is achieved.

With the framework in place allowing for exception handlers to be prioritized according to purpose, and by allowing those handlers to be registered at run time, we have enabled the kernel to load NLKD at run time. Note that the usefulness of such an exception system extends beyond just NLKD, as it enables a whole class of debuggers and monitors to be loaded dynamically.

Our current implementation does not actually load the Core Debug Engine (CDE) at runtime. CDE is currently a compile time option. However, with CDE in place, the debug agents (which we will discuss later) are loadable/unloadable at run time. This allows a user to switch from no debugger to an on-box kernel debugger or a remote source level debugger by simply loading the appropriate modules.

There have been many customer support scenarios that require debugging, monitoring, or profiling on production boxes in live environments. This must happen without taking the box down or changing the environment by rebuilding the kernel to enable a debugger.

There is some argument that a loadable kernel debugger is a security risk. To some degree this is true, but only insomuch as the root user is a security risk. Since root is the only user that can load kernel modules, such security concerns are negated.

It could easily be argued that the benefit of being able to load and then unload a debugger on demand provides even greater security in situations where a debugger is needed, since we can easily restrict the actual time that the debugger is available.

Let us reiterate that in our current implementation, adding support for CDE is a compile time option, like KDB, not a runtime option. But with CDE in place, kernel modules to support local or remote debugging can easily be loaded. Without a corresponding debug agent attached, CDE is inactive.

Let's now turn our attention to the next layer in the stack, the Core Debug Engine (CDE).

2.2 Core Debug Engine (CDE)

Sitting on top of the exception handling framework is a debugger infrastructure piece we have named the Core Debug Engine. This layer of the debugger provides three main functions. First, all NLKD state machine logic is located within CDE. Second, CDE provides a framework against which debug agents load and assume responsibility for driving the state machine and for providing interaction with the user. Finally, CDE provides a means of extending the functionality of the debugger.

The state machine also provides the infrastructure supporting the breakpoint logic, which is a key component and distinguishing capability of NLKD.

We will now examine each of these in turn.

2.2.1 CDE State Machine

NLKD divides the state of each processor in the system into four simple yet well-defined states. These states are: RUN state, FOCUS PROCESSOR state, RENDEZVOUS state, and FOCUS PROCESSOR COMMIT state.

Run State The RUN state is defined as the state in which the operating system is normally running. This is the time when the processor is in user and kernel modes, including the time spent in interruptions such as IO interrupts and processor exceptions. It does not include the debug exception handler where CDE will change the state from the RUN state to one of the other three defined states.

Focus Processor State When an exception occurs that results in a processor entering the debug exception handler and subsequently CDE, CDE determines whether this is the first processor to come into CDE. If it is the first processor, it becomes the focus processor and its state is changed from the RUN state to the FOCUS PROCESSOR state.

The focus processor controls the machine from this point on, until it yields to another processor or returns to the RUN state.

Once the focus processor has entered CDE, its first responsibility is to rendezvous all other processors in the system before debug operations are allowed by the registered debug agent. Rendezvous operations are typically accomplished by hardware specific methods and may be unique to each processor architecture. On x86, this is typically a cross-processor NMI.

After sending the rendezvous command to all other processors, the focus processor waits for all processors to respond to the request to rendezvous. As these processors come into CDE they are immediately sent to the RENDEZVOUS state where they remain until the focus processor yields control.

Once all processors are safely placed in the RENDEZVOUS state, the focus processor transfers control to the debug agent that was registered with CDE for subsequent control of the system.

Rendezvous State The RENDEZVOUS state is sort of a corral or holding pen for processors while a debug agent examines the processor currently in the FOCUS PROCESSOR state. Processors in the RENDEZVOUS state do nothing but await a command to change state or to deliver information about their state to the focus processor.

It should be noted at this point that processors could have entered the debugger for reasons

other than being asked to rendezvous. This happens when there are exceptions occurring simultaneously on more than one processor. This is to be expected. A processor could, in fact, receive a rendezvous request just before entering CDE on is own accord. This can result in spurious rendezvous requests that are detected and handled by the state machine. Again, this is normal. These sorts of race conditions are gracefully handled by CDE, such that those processors end up in the RENDEZVOUS state just as any other processor does.

As stated above, a processor may end up in the RENDEZVOUS state when it has a valid exception condition that needs evaluation by the active debug agent. Before ever sending any processor back to the RUN state, CDE examines the reason for which all other processors have entered the debugger. This may result in the processor in the FOCUS PROCESSOR state moving to the RENDEZVOUS state and a processor in the RENDEZVOUS state becoming the focus processor for further examination.

This careful examination of each processor's exception status forces all pending exceptions to be evaluated by the debug agent before allowing any processor to continue execution. This further contributes to the stability of the debugger.

Once all processors have been examined, any processors that have been in the FOCUS PROCESSOR state are moved to the FOCUS PROCESSOR COMMIT state, which we will now discuss.

Focus Processor Commit State The logic in this state is potentially the most complex part of CDE. Processors that have been moved to this state may need to adjust the breakpoint state in order to resume execution without re-triggering the breakpoint that caused the debugger to be entered.

The FOCUS PROCESSOR COMMIT state is the state that ensures that no processor is run or is further examined by the debug agents until the conditions specified by CDE are met. This contributes greatly to the stability of the debugger.

2.2.2 Breakpoint Logic

A distinguishing feature of NLKD is its rich breakpoint capability. NLKD supports the notion of qualifying breakpoints. Breakpoints can be set to qualify when a number of conditions are met. These conditions are:

- execute/read/write
- address/symbol, optionally with a length
- agent-evaluated condition (e.g. expression)
- global/engine/process/thread
- rings 0, 1, 2, 3
- count

This allows for a number of restrictions to be placed on the breakpoint before it would actually be considered as "qualifying," resulting in the debug agent being invoked.

The number of supported read/write break points is restricted by hardware, while the number of supported execute breakpoints is limited by software and is currently a `#define` in the code. NLKD uses the debug exception, INT3 on x86, for execute breakpoints and the processor's watch/debug registers for read/write breakpoints.

The debug agents work in cooperation with CDE to provide breakpoint capabilities. The

debug agents define the conditions that will trigger a debug event and CDE modifies the code with debug patterns (INT3 on x86) as necessary. When the breakpoint occurs, CDE determines if it actually qualifies before calling the debug agent.

CDE's breakpoint logic is one of the most powerful features of the tool and a distinguishing feature of NLKD. CDE's breakpoint logic combined with CDE's state machine sets the stage for a stable on-box or remote debugging experience.

2.2.3 CDE APIs

CDE exports a number of useful APIs. These interfaces allow the rest of the system to interact with the debugger and allow debug agents to be extended with new functionality.

CDE supports an API to perform DWARF2 frame-pointer-less reliable stack unwinding, using the `-fasynchronous-unwind-tables` functionality available with gcc.

The programmatic interfaces to the debugger also include support for various debug events (such as assertions and explicit requests to enter the debugger) and the ability to register and unregister debugger extensions. Extensions can be either loadable binaries or statically linked modules.

APIs also exist to support pluggable debug agents, which we will discuss in the next section.

2.2.4 Debug Agents

In earlier discussions, we briefly introduced the notion of debug agents. Debug agents plug into CDE and provide some sort of interface to the user. Debug agents can be loadable kernel modules or statically linked into the kernel.

NLKD provides two debug agents: the Console Debug Agent (CDA) for on-box kernel debugging, and the Remote Debug Agent (RDA) for remote debugging including remote source level debugging.

Other debug agents can be written and plugged into CDE's framework, thus benefiting from the state logic provided by CDE.

It should be noted that CDE only allows one agent to be active at a time. However, a new agent can be loaded on the fly and replace the currently active one. This scenario commonly happens when a server is being debugged on-site (using CDA), but is then enabled for debugging by a remote support team (using RDA). This is possible by simply unloading CDA and loading RDA.

Console Debug Agent (CDA) CDA is NLKD's on-box kernel debugger component. It accepts keyboard input and interacts with the screen to allow users to do on-box kernel debugging.

Remote Debug Agent (RDA) RDA is an agent that sits on-box and communicates with a remote debug client. RDA would typically be used by users who want to do remote source level debugging.

Other Agent Types It should be noted that NLKD's architecture does not limit itself to the use of these two agents. CDE allows for other agent types to plug in and take advantage of the environment provided by CDE.

NLKD's agents support the ability to apply certain settings to the debug environment before the debugger is initialized. Some examples are a request to break at the earliest possible moment during system boot, or setting screen color preferences for CDA. These configuration settings are held in a file made available early in the boot process, but only if the agent is built into the kernel.

2.3 Architecture Summary

At this point we have introduced the exception handling framework and NLKD's architecture, including CDE with its state machine, debug agents, breakpoint logic, and finally NLKD's ability to be extended.

Further discussion of the NLKD will follow but will not be presented as an architectural discussion. The remainder of this discussion will focus on features provided by NLKD and the debug agents CDA and RDA.

3 Console Debug Agent (CDA) Features

This section discusses the features of the Console Debug Agent. We should note that this section lists features, but it is not meant to be a user's guide. To see the full user's guide for NLKD, go to `http://forge.novell.com` and then search for "NLKD".

3.1 User Interface Overview

CDA supports on-box debugging. Interaction with the debugger is via the keyboard and screen.

3.1.1 Keyboard

Input from PS2 keyboards and the 8042 keyboard controller is currently supported. The debugger can be invoked by a special keystroke when CDA is loaded.

3.1.2 Screen IO

CDA can operate in text or graphics mode. The mode CDA uses is determined by the mode that the kernel switched to during boot.

Since CDA has the ability to display data in graphics mode, we also have the ability to enter the debugger directly from graphics mode at run time. This is extremely useful but requires that the screen resolution and color depth of the user graphics environment match the screen resolution and color depth of the kernel console environment.

3.1.3 Screen Layout

CDA supports both command line and window-pane based debugging. The debugging screen is divided into six window panes as shown in Figure 2. One of these panes hosts a command line. All panes are resizable.

Each pane's features can be accessed via a number of keystroke combinations. These keystrokes are documented in the user's guide, and are also available by pressing F1 while in the debugger. Help can also be obtained by typing `h` in the command line pane.

Code Pane The code pane shows instruction disassembly. There are a variety of format specifier commands that can alter the way the information is displayed. Currently CDA supports the Intel assembly format.

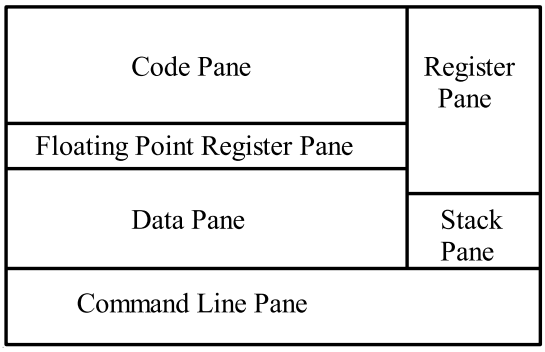


Figure 2: Screen layout of CDA

Data Pane The data pane supports the display and modification of logical and physical memory addresses, including IO ports and PCI config space. Data can be displayed in a variety of formats.

Register Pane The register pane supports the display and modification of processor registers. Registers with special bitmap definitions are decoded.

On some architectures, IA64 for example, there are too many registers to view all at once in the register pane. Hence, support exists to scroll up or down through the registers in the register pane.

Stack Pane / Predicate Pane (IA64) A special pane exists for displaying the stack pointed to by the processor's stack pointer. Since on IA64 the register stack engine is used instead of a normal stack, CDA uses this pane to display the IA64 predicate registers instead of stack information.

Code or data browsing can be initiated directly from the stack.

Floating Point Register Pane The floating point register pane supports the display and modification of floating point registers. The data can be displayed in a variety of formats. Since kernel debugging rarely requires access to floating point registers, this pane is normally hidden.

Command Line Pane The command line pane supports a command line parser that allows access to most of the capabilities found in other CDA panes. This pane can assume the size of the entire screen, and it can also be entirely hidden.

The command line pane exports APIs so that other modules can further extend the debugger.

3.2 CDA User Interface Features

3.2.1 Viewing Program Screens

The screen that was active when the debugger was invoked can be viewed from the debugger. Viewing both text and graphics (such as an X session) is supported.

3.2.2 Processor Selection

Support to switch processors and view information specific to that processor is supported. Some information is processor specific such as the registers, per processor data, etc. CDA also supports viewing the results of such commands as the CPUID instruction on x86.

3.2.3 Command Invocation

There are a number of pane-sensitive hot keys and pane-sensitive context menus available for command execution. Additionally, there is a global context menu for commands common to all panes.

3.2.4 Expression Evaluation

Support for expression evaluation exists. The expressions use mostly C-style operators, operating on symbolic or numeric addresses.

3.2.5 Browsing

The code, data, and stack panes support the ability to browse. For example, in the code pane we can browse (follow) branch instructions, including call and return statements.

In the data pane we can follow data as either code pointers or data pointers. The same is true for the register and stack panes.

Functionality exists to follow a pointer, to go to the previously visited location, or to go to the place of origin where we started.

Of course, we can always browse to a specific code or data address.

3.2.6 Stepping

CDA supports typical processor stepping capabilities.

- Single Step
- Branch Step
- Step Over
- Step Out
- Continue Execution (Go)

3.2.7 Symbols

Provided that symbols are available from the kernel, support for symbolic debugging is supported throughout the debugger.

3.2.8 Smart Register Mode

A mode exists to make it easier to watch only the registers that change as the code is stepped through. This is particularly useful on architectures like IA64 that have many more registers that we can display at once.

3.2.9 Aliases

Aliases are supported in the code and register panes. For example, on IA64 a register named `cr12` may also be a the stack pointer. With aliasing off, the name `cr12` will be displayed everywhere. With aliasing on, `sp` will be displayed.

3.2.10 Special Registers

Support exists for viewing (and in some cases, setting) special processor and platform registers. On x86, these are:

- CPUID
- Debug Registers
- Data Translation Registers
- Last Branch
- MSR
- MTTR

3.2.11 Debugger Events

NLKD has a level of interaction with the host operating system, enabling certain operating system events or code to invoke the debugger.

APIs exist to generate debug messages from source code. When a message event is hit, CDA displays the message on the screen. The user then enters a keystroke to resume execution.

CDA also can notify the user when a kernel module is loaded or unloaded, as well as when a thread is created or destroyed.

The user may enable and disable each event notification type at run-time.

On a per processor basis, CDA can display the most recent event that caused the debugger to be invoked.

3.2.12 OS Structures

CDA understands certain Linux kernel data structures. In particular, CDA can list all threads in the system. It also can list all kernel modules that are currently loaded, and information about them.

3.2.13 Linked List Browsing

A useful feature in the data pane is the ability for the user to inform the debugger which offsets from the current address are forward and backward pointers. This features enables users to easily browse singly and doubly linked lists.

3.2.14 Set Display Format

CDA is very flexible with how data is displayed.

Code Pane The code pane allows a variety of formating options.

- Show Address
- Show Aliases
- Show NOPs (Mainly used for IA64 templates.)
- Set Opcode Display Width
- Display Symbolic Information
- Show Templates/Bundles (IA64)

Data Pane Data sizing, format, and radix can be set. CDA provides support for displaying or operating on logical and physical memory.

Floating Point Pane Data sizing and format can be set.

3.2.15 Search

CDA can search for a set of bytes in memory. The search can start at any location in memory, and can be restricted to a range. Both forward and backward searches are supported.

3.2.16 List PCI Devices

On architectures that support PCI buses, CDA has knowledge of the PCI config space and allows browsing of the config space of all the devices.

3.2.17 System Reboot

NLKD can perform a warm and/or cold boot. Cold boot is dependent upon hardware support.

4 Remote Debug Agent (RDA) Features

RDA provides a means whereby a remote debugger can communicate with NLKD breakpoint logic and drive NLKD's abilities to support debugging.

As expected, there are a number of verbs and events supported by RDA that enable remote source level debuggers to drive the system. Some examples include setting and clearing breakpoints, controlling processors' execution, and reading and writing registers and memory.

4.1 Protocol Support

RDA currently supports the gdbtransport protocol.

Novell has an additional protocol, Novell Debug Interface (NDI) 2.0, which we hope to introduce into a new remote debug agent. NDI provides advantages in three areas:

- NDI has support for describing multiprocessor configurations.
- When debugging architectures with a large register set, NDI allows the debugger to transfer only a portion of the full register set. This is especially important when debugging over slow serial lines.
- NDI fully supports control registers, model specific registers, and indirect or architecture-specific registers.

4.2 Wire Protocols

RDA currently supports RS232 serial port connections to the debugger.

5 Work To Do

Currently, a number of things still need to be worked on. We welcome help from interested and capable persons in these areas.

5.1 USB Keyboards

Support for other keyboard types, mainly USB, will be added to CDA.

5.2 AT&T Assembler Format

CDA currently supports the Intel assembler format. We would like to add support for the AT&T assembler format.

5.3 Additional Command Line Parsers

There is a native command line interface provided by CDA. We would also like to see a KDB command line parser and a GDB command line parser for those familiar with these debuggers and who prefer command line debugging to pane-based debugging.

5.4 Novell Debug Interface 2.0

We plan to create a new debug agent supporting the NDI 2.0 protocol. We also need support for NDI in a remote source level debugger.

5.5 Additional Wire Protocols

We would like to support additional wire protocols for remote debugging, such as LAN, IPMI/BMC LAN, and USB.

5.6 Additional Architectures

We would like to finish the NLKD port to IA64 Linux, and port it to Power PC Linux.

6 Conclusion

We recognize that no one tool fits all, and that user familiarity with a tool often dictates what is used in spite of the existence of tools that may offer more features and capabilities. In introducing this tool, we make no claim to superiority over existing tools. Each and every user will make that decision themselves.

As we stated earlier, our goal is to provide developers and support engineers a robust kernel development tool enabling them to be successful in their respective roles. We believe we have introduced a debug architecture that needs no apology. At the same time, we welcome input as to how to improve upon this foundation.

Our hope is that kernel debugging with this architecture will become standard and that the capabilities of NLKD will find broad acceptance with the goal of creating a better Linux for everyone.

7 Acknowledgments

We would like to express thanks to Charles Coffing and Jana Griffin for proofreading this paper, and to Charles for typesetting.

8 References

Data Center Linux: DCL Goals and Capabilities Version 1.1
`http://www.osdl.org/lab_activities/data_center_linux/DCL_Goals_Capabilities_1.1.pdf`

Novell Linux Kernel Debugger (NLKD)
`http://forge.novell.com`

Built-in Kernel Debugger (KDB)
`http://oss.sgi.com/projects/kdb/`

GNU Project Debugger (GDB)
`http://www.gnu.org/software/gdb/gdb.html`

Large Receive Offload implementation in Neterion 10GbE Ethernet driver

Leonid Grossman
Neterion, Inc.
leonid@neterion.com

Abstract

The benefits of TSO (Transmit Side Offload) implementation in Ethernet ASICs and device drivers are well known. TSO is a *de facto* standard in version 2.6 Linux kernel and provides a significant reduction in CPU utilization, especially with 1500 MTU frames. When a system is CPU-bound, these cycles translate into a dramatic increase in throughput. Unlike TOE (TCP Offload Engine) implementations, stateless offloads do not break the Linux stack and do not introduce either security or support issues. The benefits of stateless offloads are especially apparent at 10 Gigabit rates. TSO hardware support on a 10GbE sender uses a fraction of a single CPU to achieve full line rate, still leaving plenty of cycles for applications. On the receiver side, however, the Linux stack presently does not support an equivalent stateless offload. Receiver CPU utilization, as a consequience, becomes the bottleneck that prevents 10GbE adapters from reaching line rate with 1500 MTU. Neterion Xframe adapter, implementing a LRO (Large Receive Offload) approach, was designed to address this bottleneck and reduce TCP processing overhead on the receiver. Both design and performance results will be presented.

1 Introduction

With the introduction of 10 Gigabit Ethernet, server I/O re-entered the "fast network, slow host" scenario that occurred with both the transitions to 100Base-T and 1G Ethernet.

Specifically, 10GbE has exposed three major system bottlenecks that limit the efficiently of high-performance I/O Adapters:

- PCI-X bus bandwidth
- CPU utilization
- Memory bandwidth

Despite Moore's law and other advances in server technology, completely overcoming these bottlenecks will take time. In the interim, network developers and designers need to find reliable ways to work around these limitations.

One approach to improve system I/O performance has come through the introduction of Jumbo frames. Increasing the maximum frame size to 9600 byte reduces the number of packets a system has to process and transfer across the bus.

While Jumbo frames have became universally supported in all operating systems, they have

not been universally deployed outside of the datacenter.

As a consequence, for the foreseeable future, networks will still need some kind of offloading relief in order to process existing 1500 MTU traffic.

As occurred in previous "fast network, slow host" scenarios, the need to improve performance has triggered renewed industry interest in developing NIC (Network Interface Card) hardware assists, including stateless and stateful TCP assists, as well as the all-critical operating system support required for widespread deployment of these NIC assists.

To date, the acceptance of stateless and stateful TCP assist has varied.

Stateful TCP Offload Engines (TOE) implementations never achieved any significant market traction or OS support. Primary reasons for lack of adoption include cost, implementation complexity, lack of native OS support, security/TCO concerns, and Moores law. On the other hand, stateless assists, including checksum offload and TSO (Transmit Side Offload) have achieved universal support in all major operating systems and became a de-facto standard for high-end server NICs. TSO is especially effective for 10GbE applications since it provides a dramatic reduction in CPU utilization and supports 10Gbps line rate for normal frames on current server systems.

Unfortunately, TSO offloads the transmit-side only, and there is no similar stateless offload OS support today on the receive side. To a large degree, this negates the overall effect of implementing LSO, especially in 10GbE applications like single TCP session and back-to-back setups.

This is not surprising, since receive-side offloads are less straightforward to implement due to potential out-of-order receive and other reasons. However, there are several NIC hardware assists that have existed for some time and could be quite effective, once Linux support is in place.

For example, some of the current receive-side assists that are shipped in Neterion 10GbE NICs and can be used for receive-side stateless offload include:

- MAC, IP, and TCP IPv4 and IPv6 header separation; used for header pre-fetching and LRO (Large Receive Offload). Also improves PCI bus utilization by providing better data alignment.
- RTH (Receive Traffic Hashing), based on Jenkins Hash, and SPDM (Socket Pair Direct Match); used for LRO and RTD (Receive Traffic Distribution).
- Multiple transmit and receive queues with advanced steering criteria; used for RTD, as well as for NIC virtualization, NIC sharing, and operations on multi-core CPU architectures.
- MSI and MSI-X interrupts; used in RTD, as well as for reducing interrupt overhead
- Dynamic utilization-based and timer-based interrupt moderation schemes; used to reduce CPU utilization.

2 PCI-X bus bandwidth bottleneck

Theoretically, a PCI-X 1.0 slot is limited in throughput to 8+Gbps, with a practical TCP limit (unidirectional or bidirectional) around 7.6Gbps. PCI-X 2.0 and PCI-Express slots support unidirectional 10Gbps traffic at line rate Neterion has measured 9.96Gbps (unidirectional) with PCI-X 2.0 Xframe-II adapters.

In order to saturate the PCI bus, a high-end 10GbE NIC needs to implement an efficient DMA engine, as well as support Jumbo frames, TSO, and data alignment.

3 Memory bandwidth bottleneck

Typically, memory bandwidth is not a limitation in Opteron and Itanium systems, at least not for TCP traffic. Xeon systems, however, encounter memory bandwidth limitations before either PCI bus or CPU saturation occurs. This can be demonstrated on Xeon systems with 533Mhz FSB vs. 800Mhz FSB. In any case, memory bandwidth will increase as a bottleneck concern since advances in silicon memory architectures proceed at a much slower pace than CPU advances. Neither stateful nor stateless TCP offload addresses this problem. Upcoming RDMA over Ethernet RNIC adapters will ease memory bandwidth issues, and if RNIC technology is successful in the market, this will be one application where TOE can be deployed (most likely, without exposing the TOE as a separate interface)

4 CPU utilization bottleneck

On the transmit side, LSO and interrupt moderation provide the desired result Neterion has achieved utilization in the range of 10-15% of a single Opteron CPU in order to saturate a PCI-X 1.0 bus with TCP traffic. On the receive side, however, CPU utilization emerged as the biggest bottleneck to achieving 10GbE line rate with 1500 bytes frames. With current NICs and operating systems, using multiple processors doesnt help much because in order to support cache locality and optimize CPU utilization, a TCP session needs to be kept on the same CPU. Without achieving the cache locality, additional CPU cycles are being used in a very inefficient fashion. Moores law is often cited as a main argument against deploying TCP assists and offloads. However, the industry wants to deploy full-rate 10GbE and cannot wait for CPUs that don't required offloading. Also, from an application prospective CPU utilization expended on stack processing must drop to single digits, and on current systems, the only way to achieve such a low utilization rate for 10GbE processing is to bring in some sort of hardware assist. The resolution to the CPU bottleneck is to add Linux support for header separation and prefetching, as well as for Receive Traffic Distribution and Receive Side Offload.

5 Header separation and prefetching

Neterion's Xframe-II supports several flavors of true hardware separation of Ethernet, IP and TCP (both IPv4 and IPv6) headers. This has been proven to be effective in achieving optimal data alignment, but since cache misses on headers represent one of the most significant sources of TCP processing overhead, the real benefit is expected to come from the ability to support OS header pre-fetching and LRO.

6 Receive Traffic Distribution

The key to efficient distribution of TCP processing across multiple CPUs is maintaining an even load between processors while at the same time keeping each TCP session on the same CPU. In order to accomplish this, the host must be able to identify each TCP flow and dynamically associate the flow to its particular hardware receive queue, particular MSI, DPC,

and CPU. In this way, load-balancing multiple TCP sessions across CPUs while preserving cache locality is possible. Neterion's Xframe-II 10GbE ASIC achieves this through receive descriptors that carry SPDM or RTH information on a per packet basis, giving the host enough visibility into packets to identify and associate flows.

7 Large Receive Offload

In short, LRO assists the host in processing incoming network packets by aggregating them on-the-fly into fewer but larger packets. This is done with some hardware assist from the NIC. It's important that an LRO implementation avoid a very expensive state-aware TOE implementation that would break compatibility with current operating systems and therefore have only limited application.

To illustrate the effectiveness of LRO, consider a network passing 1500 MTU packets at a data rate of 10 Gigabit per second. In this best possible case network traffic consists of universally full-sized packets the host-resident network stack will have to process more than 800,000 packets per second. If it takes on average 2000 instructions to process each packet and one CPU clock cycle to execute each instruction, processing in the best case will take consume more than 80% of a 2Ghz CPU, leaving little for doing anything other than receiving data. This simplified calculation demonstrates the critical characteristic of networks that the performance of transport protocols is dependent upon the granularity of data packets. The fewer packets presented to the protocol stacks, the less CPU utilization required leaving more cycle for the host to run applications.

The idea of Large Receive Offload, as the name implies, is to give the host the same amount of data but in bigger "chunks." Reducing the number of packets the stacks have to process lowers the load on the CPU. LRO implementation relies on the bursty nature of TCP traffic, as well as the low packet loss rates that are typical for 10GbE datacenter applications.

To implement Large Receive Offload, Neterion's Xframe-II 10GbE ASIC separates TCP headers from the payload and calculates SPDM or RTH information on a per packet basis. In this way, it is possible to identify a burst of consecutive packets that belong to the same TCP session and can be aggregated into a single oversized packet. Additionally, the LRO engine must perform a number of checks on the packet to ensure that it can be added to an LRO frame.

The initial implementation of Neterion's LRO is a combination of hardware (NIC) and software (Linux driver). The NIC provides the following:

- multiple hardware-supported receive queues
- link-layer, IP, and UDP/TCP checksum offloading
- header and data split, with link, IP, TCP, and UDP headers placed in the host-provided buffers separately from their corresponding packet datas
- SPDM or RTH "flow identifier."

The Linux driver controls the NIC and coordinates operation with the host-resident protocol stack. It is the driver that links payloads together and builds a single header for the LRO packet. If the flow is "interrupted," such as a sequence gap, the driver signals the host-resident network stack and sends all the accumulated receive data.

The simple algorithm below capitilizes on the fact that the receive handling code at any point in time potentially "sees" multiple new frames. This is because of the interrupt coalescing, which may or may not be used in combination with polling (NAPI).

Depending on the interrupt moderation scheme configured "into" the adapter, at high throughput we are seeing batches of 10s or 100s received frames within a context of a single interrupt.

The same is true for NAPI, except that the received "batch" tends to be even bigger, and the processing is done in the `net_device->poll()` softirq context.

Within this received batch the LRO logic looks for multiple back-to-back frames that belong to the same stream.

The 12-step algorithm below is essentially a set of simple hardware-friendly checks (see check A, check B, etc. below) and a simple hardware-friendly header manipulation.

Note that by virtue of being a pseudo-code certain low-level details were simplified out.

8 Large Receive Offload algorithm

1) for each (Rx descriptor, Rx frame) pair from the received "batch":

2) get LRO object that corresponds to the descriptor->ring.

3) check A:
- should the frame be dropped? (check FCS and a number of other conditions, including ECC) if the frame is bad then drop it, increment the stats, and continue to 1).

4) is the LRO object (located at step 2) empty? if it contains previously accumulated data, goto step 6); otherwise proceed with a series of checks on the first to-be-LROed frame (next).

5) check B:
- is it IP frame?
- IP fragmented?
- passes a check for IP options?
- either TCP or UDP?
- both L3 and L4 offloaded checksums are good?
- for TCP: passes a check for flags?
- for TCP: passes a check for TCP options? if any check fails - goto step 11). otherwise goto to step 10).

6) use hardware-assisted Receive Traffic Hashing (RTH) to check whether the frame belongs to the same stream; if not (i.e. cannot be LROed), goto 11).

7) check C:
- IP fragmented?
- passes a check for IP options?
- offloaded checksums are good?
- for TCP: passes a check for flags?
- for TCP: passes a check for TCP options? if any of the checks fail, goto step 11).

8) check D:
- in-order TCP segment? if not, goto step 11).

9) append the new (the current) frame; update the header of the first frame in the already LROed sequence; update LRO state (for the given ring->LRO) accordingly.

10) check E:
- too much LRO data accumulated? (in terms of both total size and number of "fragments")
- is it the last frame in this received "batch"? if 'no' on both checks, continue to 1).

11) call netif_rx() or netif_receive_skb() (the latter for NAPI) for the LRO-ed frame, if exists; call netif_rx() or netif_receive_skb() for

the current frame, if not “appended” within this iteration (at step 9).

12) reset the LRO object and continue to 1).

9 Conclusion

Stateless hardware assists and TCP offloads have become a de-facto standard feature in both high-end Server NICs and operating systems. Support for additional stateless offloads on the receive-side, with native driver support in Linux, is required in order to provide 10Gbps Ethernet data rates in efficient manner.

10 References

- Xframe 10GbE Programming manual
- The latest Neterion Linux driver code (available in 2.6 kernel)

eCryptfs: An Enterprise-class Encrypted Filesystem for Linux

Michael Austin Halcrow
International Business Machines
mhalcrow@us.ibm.com

Abstract

eCryptfs is a cryptographic filesystem for Linux that stacks on top of existing filesystems. It provides functionality similar to that of GnuPG, except the process of encrypting and decrypting the data is done transparently from the perspective of the application. eCryptfs leverages the recently introduced Linux kernel keyring service, the kernel cryptographic API, the Linux Pluggable Authentication Modules (PAM) framework, OpenSSL/GPGME, the Trusted Platform Module (TPM), and the GnuPG keyring in order to make the process of key and authentication token management seamless to the end user.

1 Enterprise Requirements

Any cryptographic application is hard to implement correctly and hard to effectively deploy. When key management and interaction with the cryptographic processes are cumbersome and unwieldy, people will tend to ignore, disable, or circumvent the security measures. They will select insecure passphrases, mishandle their secret keys, or fail to encrypt their sensitive data altogether. This places the confidentiality and the integrity of the data in jeopardy of compromise in the event of unauthorized access to the media on which the data is stored.

While users and administrators take great pains to configure access control mechanisms, including measures such as user account and privilege separation, Mandatory Access Control[13], and biometric identification, they often fail to fully consider the circumstances where none of these technologies can have any effect – for example, when the media itself is separated from the control of its host environment. In these cases, access control must be enforced via cryptography.

When a business process incorporates a cryptographic solution, it must take several issues into account. How will this affect incremental backups? What sort of mitigation is in place to address key loss? What sort of education is required on the part of the employees? What should the policies be? Who should decide them, and how are they expressed? How disruptive or costly will this technology be? What class of cryptography is appropriate, given the risks? Just what are the risks, anyway? Whenever sensitive data is involved, it is incumbent upon those responsible for the information to reflect on these sorts of questions and to take action accordingly.

We see today that far too many businesses ne-

glect to effectively utilize on-disk encryption. We often see news reports of computer equipment that is stolen in trivial cases of burglary[5] or of backup tapes with sensitive customer data that people lose track of.[10] While the physical security measures in place in these business establishments are usually sufficient given the dollar value of the actual equipment, businesses often underrate the value of the data contained on the media in that equipment. Encryption can effectively protect the data, but there there exist a variety of practical barriers to using it effectively. eCryptfs directly addresses these issues.

1.1 Integration of File Encryption into the Filesystem

Cryptography extends access control beyond the trusted domain. Within the trusted domain, physical control, authentication mechanisms, DAC/MAC[14][13], and other technologies regulate what sort of behaviors users can take with respect to data. Through various mathematical operations, cryptographic applications can enforce the confidentiality and the integrity of the data when it is not under these forms of protection. The mathematics, however, is not enough. The cryptographic solution must take human behavior into account and compensate for tendencies to take actions that compromise the security afforded by the cryptographic application.

Several solutions exist that solve separate pieces of the data encryption problem. In one example highlighting transparency, employees within an organization that uses IBM™ Lotus Notes™ [11] for its email will not even notice the complex PKI or the encryption process that is integrated into the product. Encryption and decryption of sensitive email messages is seamless to the end user; it involves checking an "Encrypt" box, specifying a recipient, and sending the message. This effectively addresses a significant file in-transit confidentiality problem. If the local replicated mailbox database is also encrypted, then this addresses confidentiality (to some extent) on the local storage device, but the protection is lost once the data leaves the domain of Notes (for example, if an attached file is saved to disk). The process must be seamlessly integrated into *all* relevant aspects of the user's operating environment.

We learn from this particular application that environments that embody strong hierarchical structures can more easily provide the infrastructure necessary to facilitate an easy-to-use and effective organization-wide cryptographic solution. Wherever possible, systems should leverage this infrastructure to protect sensitive information. Furthermore, when organizations with differing key management infrastructures exchange data, the cryptographic application should be flexible enough to support alternate forms of key management.

Current cryptographic solutions that ship with Linux distributions do not fully leverage existing Linux security technologies to make the process seamless and transparent. Surprisingly few filesystem-level solutions utilize public key cryptography. eCryptfs brings together the kernel cryptographic API, the kernel keyring, PAM, the TPM, and GnuPG in such a way so as to fill many of the gaps[3] that exist with current popular cryptographic technologies.

1.2 Universal Applicability

Although eCryptfs is geared toward securing data in enterprise environments, we explored how eCryptfs can be flexible for use in a wide variety of circumstances. The basic passphrase mode of operation provides equivalent functionality to that of EncFS[23] or CFS[20], with the added advantage of the ability to copy an

encrypted file, as an autonomic unit, between hosts while preserving the associated cryptographic contexts. eCryptfs includes a pluggable Public Key Infrastructure API through which it can utilize arbitrary sources for public key management. One such plugin interfaces with GnuPG (see Section 5.7) in order to leverage the web-of-trust mechanism already in wide use among participants on the Internet.

1.3 Enterprise-class

We designed and implemented eCryptfs with the enterprise environment in mind. These environments entail a host of unique opportunities and requirements.

1.3.1 Ease of Deployment

eCryptfs does not require any modifications to the Linux kernel itself.[1] It is deployable as a stand-alone kernel module that utilizes a set of userspace tools to perform key management functions.

Many other cryptographic filesystem solutions, such as dm-crypt, require that a fixed partition (or image) be established upon which to write the encrypted data. This provides the flexibility of block-layer encryption; any application, such as swap, a database application, or a filesystem, can use it without any modification to the application itself. However, it is limited in that the amount of space allocated for the encrypted data is fixed. It is an inconvenient task to increase or decrease the amount of space available on the encrypted partition.

Cryptographic filesystems like EncFS[23] and CFS[20] are more easily deployable, as they operate at the VFS layer and can mount on top of any previously existing directory. These filesystems store cryptographic metadata in special files stored in the location mounted. Thus, the files themselves cannot be decrypted unless the user copies that metadata along with the encrypted files.

eCryptfs goes one step beyond other filesystems by storing cryptographic metadata directly in the files. This information is associated on a per-file basis, in a manner dictated by policies that are contained in special files on the target. These policies specify the behavior of eCryptfs as it works with individual files at the target. These policies are not required in order for the user to work with the files, but the policies can provide enhanced transparency of operation. Planned enhancements include utilities to aid in policy generation (see Section 7).

1.3.2 PKI Integration

Through its pluggable PKI interface (see Section 5.7), eCryptfs aims to be integrable with existing Public Key Infrastructures.

1.3.3 TPM Utilization

The Trusted Computing Group has published an architecture standard for hardware support for various secure operations.[7] Several vendors, including IBM, implement this standard in their products today. As an example, more recent IBM Thinkpad and workstation products ship with an integrated Trusted Computing Platform (TPM) chip.

The TPM can be configured to generate a public/private keypair in which the private exponent cannot be obtained from the chip. The session key to be encrypted or decrypted with this

[1] Note that the *key_type_user* symbol must be exported by the kernel keyring module, which may require a one-line patch for older versions of the module.

key must be passed to the chip itself, which will then use the protected private key to perform the operation. This hardware support provides a strong level of protection for the key that is beyond that which can be provided by a software implementation alone.

Using a TPM, eCryptfs can essentially "bind" a set of files to a particular host. Should the media ever be separated from the host which contains the TPM chip, the session keys (see Section 5.1) of the file will be irretrievable. The user can even configure the TPM in such a manner so that the TPM will refuse to decrypt data unless the machine is booted in a certain configuration; this helps to address attacks that involve booting the machine from untrusted media.

1.3.4 Key Escrow

Employees often forget or otherwise lose their credentials, and it is subsequently necessary for the administrator to reset or restore those credentials. Organizations expect this to happen and have processes in place to rectify the situations with a minimal amount of overhead. When strong cryptographic processes are in place to enforce data integrity and confidentiality, however, the administrator is no more capable of retrieving the keys than anyone else is, unless some steps are taken to store the key in a trustworthy escrow.

1.3.5 Incremental Backups

Cryptographic filesystem solutions that operate at the block layer do not provide adequate security when interoperating with incremental backup utilities. Solutions that store cryptographic contexts separately from the files to which they apply, as EncFS or CFS do, allow for incremental backup utilities to operate while maintaining the security of the data, but the administrator must take caution to assure that the backup tools are also recording the cryptographic metadata. Since eCryptfs stores this data in the body of the files themselves, the backup utilities do not need to take any additional measures to make a functional backup of the encrypted files.

2 Related Work

eCryptfs extends cryptfs, which is one of the filesystems instantiated by the stackable filesystem framework FiST.[9] Erez Zadok heads a research lab at Stony Brook University, where FiST development takes place. Cryptfs is an in-kernel implementation; another option would be to extend EncFS, a userspace cryptographic filesystem that utilizes FUSE to interact with the kernel VFS, to behave in a similar manner. Much of the functionality of eCryptfs revolves around key management, which can be integrated, without significant modification, into a filesystem like EncFS.

Other cryptographic filesystem solutions available under Linux include dm-crypt[18] (preceded by Cryptoloop and Loop-AES), CFS[20], BestCrypt[21], PPDD[19], TCFS[22], and CryptoFS[24]. Reiser4[25] provides a plugin framework whereby cryptographic operations can be implemented.

3 Design Structure

eCryptfs is unique from most other cryptographic filesystem solutions in that it stores a complete set of cryptographic metadata together with each individual file, much like

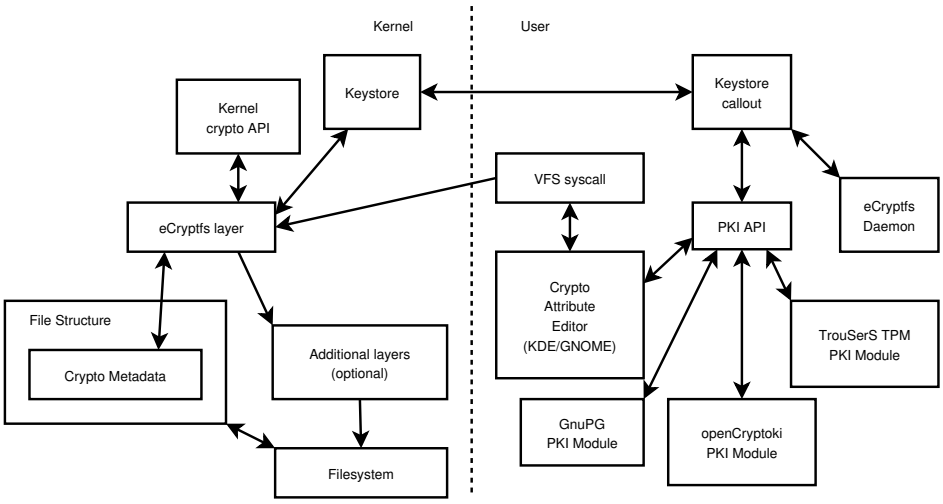


Figure 1: Overview of eCryptfs architecture

PGP-encrypted files are formatted. This allows for encrypted files to be transferred across trusted domains while maintaining the ability for those with the proper credentials to gain access to those files. Because the encryption and decryption takes place at the VFS layer, the process is made transparent from the application's perspective.

eCryptfs is implemented as a kernel module augmented with various userspace utilities for performing key management functions. The kernel module performs the bulk encryption of the file contents via the kernel cryptographic API. A keystore component extracts the header information from individual files[2] and forwards this data to a callout application. The callout application evaluates the header information against the target policy and performs various operations, such as prompting the user for a passphrase or decrypting a session key with a private key.

eCryptfs performs key management operations at the time that an application either opens or closes a file (see Figure 2). Since these events occur relatively infrequently in comparison to page reads and writes, the overhead involved in transferring data and control flow between the kernel and userspace is relatively insignificant. Furthermore, pushing key management functions out into userspace reduces the amount and the complexity of code that must run in kernel space.

4 Cryptographic Operations

eCryptfs performs the bulk symmetric encryption of the file contents in the kernel module portion itself. It utilizes the kernel cryptographic API.

4.1 File Format

The underlying file format for eCryptfs is based on the OpenPGP format described in RFC

[2]Note that the initial prototype of eCryptfs, demonstrated at OLS 2004, utilized Extended Attributes (EA) to store the cryptographic context. Due to the fact that EA's are not ubiquitously and consistently supported, this information was moved directly into the file contents. eCryptfs now uses EA's to cache cryptographic contexts, but EA support is not required for correct operation.

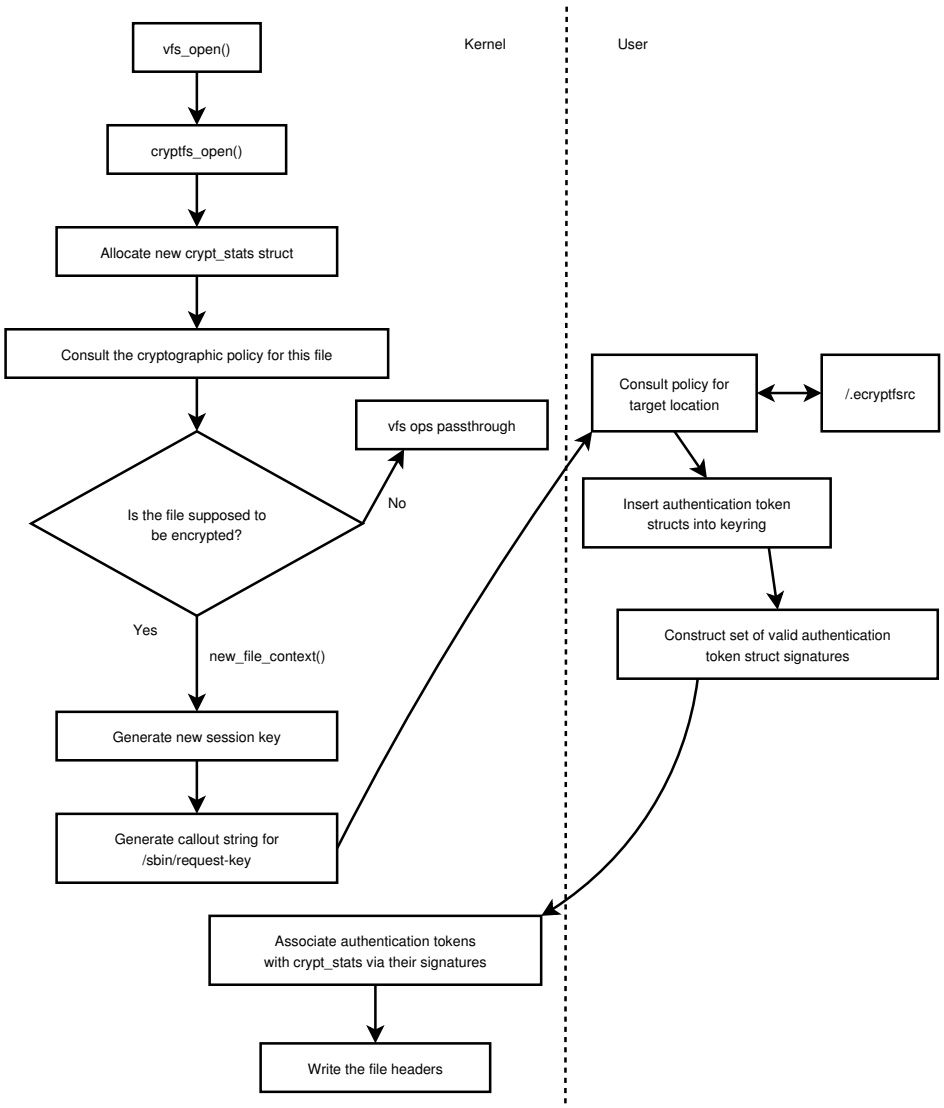


Figure 2: New file process

2440[2] (see Figure 3). In order to accommodate random access, eCryptfs necessarily deviates from that standard to some extent. The OpenPGP standard assumes that the encryption and decryption is done as an atomic operation over the entire data contents of the file; there is no concept of a partially encrypted or decrypted file. Since the data is encrypted using a chained block cipher, it would be impossible to read the very last byte of a file without first decrypting the entire contents of the file up to that point. Likewise, writing the very first byte of the file would require re-encrypting the entire contents of the file from that point.

To compensate for this particular issue while maintaining the security afforded by a cipher operating in block chaining mode[6], eCryptfs breaks the data into extents. These extents, by default, span the page size (as specified for each kernel build). Data is dealt with on a per-extent basis; any data read from the middle of an extent causes that entire extent to be decrypted, and any data written to that extent causes that

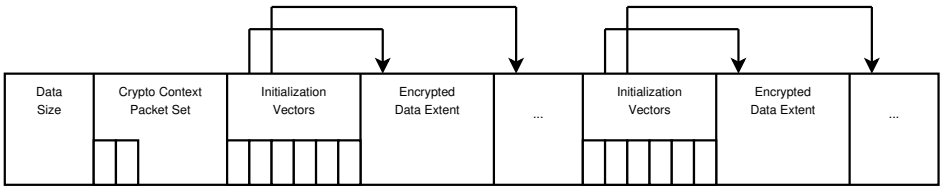


Figure 3: Underlying file format

entire extent to be encrypted.

Each extent has a unique initialization vector (IV) associated with it. One extent containing IV's precedes a group of extents to which those IV's apply. Whenever data is written to an extent, its associated IV is rotated and rewritten to the IV extent before the associated data extent is encrypted. The extents are encrypted with the block cipher selected by policy for that file and employ CBC mode to chain the blocks.

4.1.1 Sparse Files

Sparse files present a challenge for eCryptfs. Under UNIX semantics, a file becomes sparse when an application seeks past the end of a file. The regions of the file where no data is written represent *holes*. No data is actually written to the disk for these regions; the filesystem "fakes it" by specially marking the regions and setting the reported filesize accordingly. The space occupied on the disk winds up being less than the size of the file as reported by the file's inodes. When sparse regions are read, the filesystem simply pretends to be reading the data from the disk by filling in zero's for the data.

The underlying file structure for eCryptfs is amenable to accommodating this behavior; IV's consisting of all zero's can indicate that the underlying region that corresponds is sparse. The obvious problem with this approach is that it is readily apparent to an attacker which regions of the file consist of holes, and this may constitute an unacceptable breach of confidentiality. It makes sense to relegate eCryptfs's behavior with respect to sparse files as something that policy decides.

4.2 Kernel Crypto API

eCryptfs performs the bulk data encryption in the kernel module, and hence it takes advantage of the kernel cryptographic API to perform the encryption and the decryption. One of the primary motivators in implementing eCryptfs in the kernel is to avoid the overhead of context switches between userspace and kernel space, which is frequent when dealing with pages in file I/O. Any symmetric ciphers supported by the Linux kernel are candidates for usage as the bulk data ciphers for the eCryptfs files.

4.3 Header Information

eCryptfs stores the cryptographic context for each file as header information contained directly in the underlying file (see Figure 4). Thus, all of the information necessary for users with the appropriate credentials to access the file is readily available. This makes files amenable to transfer across untrusted domains while preserving the information necessary to decrypt and/or verify the contents of the file. In this respect, eCryptfs operates much like an OpenPGP application.

Most encrypted filesystem solutions either operate on the entire block device or operate on

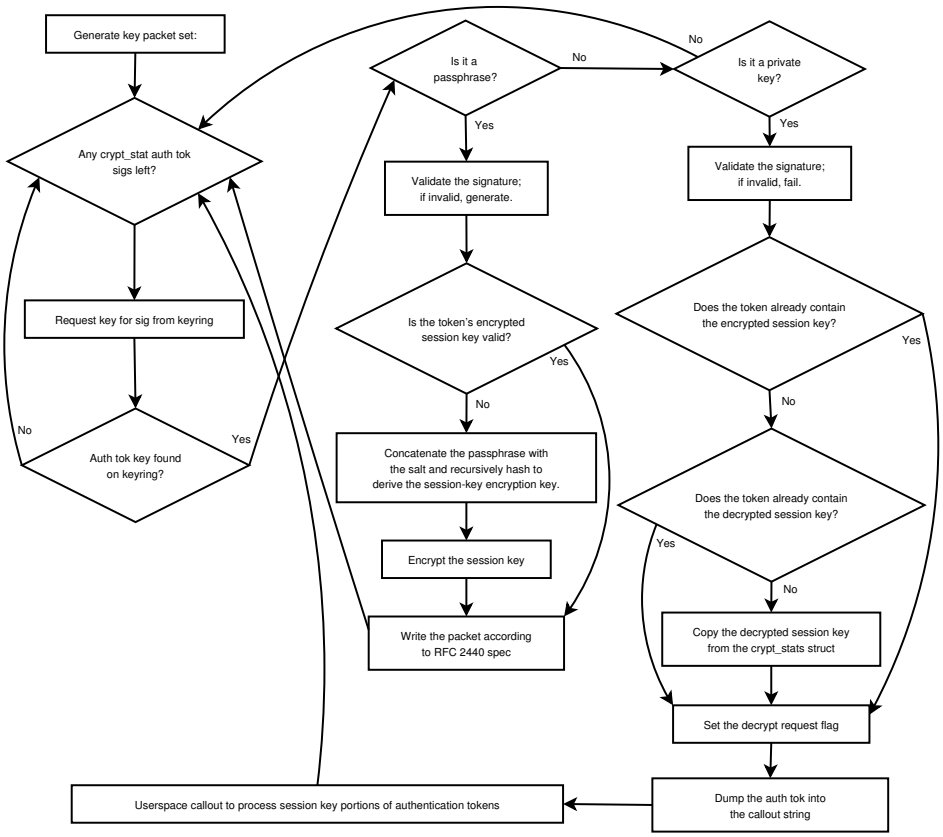


Figure 4: Writing file headers

entire directories. There are several advantages to implementing filesystem encryption at the filesystem level and storing encryption metadata in the headers of each file:

- Granularity: Keys can be mapped to individual files, rather than entire block devices or entire directories.
- Backup Utilities: Incremental backup tools can correctly operate without having to have access to the decrypted content of the files it is backing up.
- Performance: In most cases, only certain files need to be encrypted. System libraries and executables, in general, do not need to be encrypted. By limiting the actual encryption and decryption to only those files that really need it, system resources will not be taxed as much.
- Transparent Operation: Individual encrypted files can be easily transferred off of the block device without any extra transformation, and others with authorization will be able to decrypt those files. The userspace applications and libraries do not need to be modified and recompiled to support this transparency.

4.4 Rotating Initialization Vectors

eCryptfs extents span page lengths. For most architectures, this is 4096 bytes. Subsequent writes within extents may provide information to an attacker who aims to perform linear cryptanalysis against the file. In order to mitigate

this risk, eCryptfs associates a unique Initialization Vector with each extent. These IV's are interspersed throughout each file. In order to simplify and streamline the mapping of the underlying file data with the overlying file, IV's are currently grouped on a per-page basis.

4.5 HMAC's Over Extents

Integrity verification can be accomplished via sets of keyed hashes over extents within the file. Keyed hashes are used to prove that whoever modified the data had access to the shared secret, which is, in this case, the session key. Since hashes apply on a per-extent basis, eCryptfs need not generate the hash over the entire file before it can begin reading the file. If, at any time in the process of reading the file, eCryptfs detects a hash mismatch for an extent, it can flag the read operation as failing in the return code for the VFS syscall.

This technique can be applied to generate a built-in digital signature structure for files downloaded over the Internet. Given that an eCryptfs key management module is able to ascertain the trustworthiness of a particular key, then that key can be used to encode a verification packet into the file via HMAC's. This is accomplished by generating hashes over the extents of the files, as eCryptfs normally does when operating in integrity verification mode. When the file is closed, an HMAC is generated by hashing the concatenation of all of the hashes in the file, along with a secret key. This HMAC is then encrypted with the distributor's private key and written to an HMAC-type packet. The recipients of the file can proceed then to retrieve the secret key by decrypting it with the distributor's trusted public key and performing the hash operations to generate the final HMAC, which can be compared then against the HMAC that is stored in the file header in order to verify the file's integrity.

4.6 File Context

Each eCryptfs inode correlates with an inode from the underlying filesystem and has a cryptographic context associated with it. This context contains, but is not limited to, the following information:

- The session key for the file
- Whether the file is encrypted
- A pointer to the kernel crypto API context for that file
- The signatures of the authentication tokens associated with that file
- The size of the extents

eCryptfs can cache each file's cryptographic context in the user's session keyring in order to facilitate faster repeat access by bypassing the process of reading and interpreting of authentication token header information from the file.

4.7 Revocation

Since anyone with the proper credentials can extract a file's session key, revocation of access for any given credential to future versions of the file will necessitate regeneration of a session key and re-encryption of the file data with that key.

5 Key Management

eCryptfs aims to operate in a manner that is as transparent as possible to the applications and the end users of the system. Under most circumstances, when access control over the data

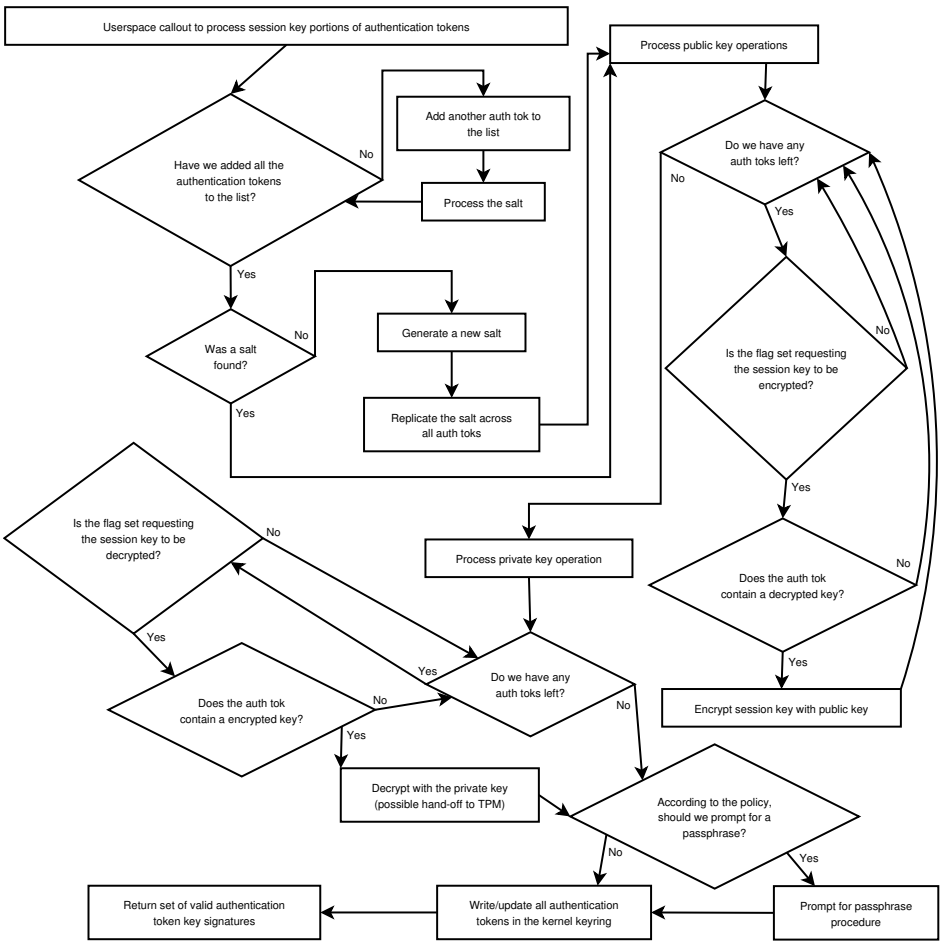


Figure 5: Key management

cannot be provided at all times by the host, the fact that the files are being encrypted should not be a concern for the user. Encryption must protect the confidentiality and the integrity of the files in these cases, and the system is configured to do just that, using the user's authentication credentials to generate or access the keys.

5.1 Session Keys

Every file receives a randomly generated session key, which eCryptfs uses in the bulk data encryption of the file contents. eCryptfs stores this session key in the cryptographic metadata for the file, which is in turn cached in the user's session keyring. When an application closes a newly created file, the eCryptfs encrypts the session key once for each authentication token associated with that file, as dictated by policy, then writes these encrypted session keys into packets in the header of the underlying file.

When an application later opens the file, eCryptfs reads in the encrypted session keys and chains them off of the cryptographic metadata for the file. eCryptfs looks through the user's authentication tokens to attempt to find a match with the encrypted session keys; it uses the first one found to decrypt the session key. In the event that no authentication tokens in the user's session keyring can decrypt any

of the encrypted session key packets, eCryptfs falls back on policy. This policy can dictate actions such as querying PKI modules for the existence of private keys or prompting the user for a passphrase.

5.2 Passphrase

> *Passwords just don't work anymore.*
> – Bruce Schneier

Many cryptographic applications in Linux rely too heavily on passphrases to protect data. Technology that employs public key cryptography provides stronger protection against brute force attacks, given that the passphrase-protected private keys are not as easily accessible as the encrypted data files themselves.

Passphrase authentication tokens in eCryptfs exist in three forms: non-passphrased, saltless, and salted. In order to address the threat of passphrase dictionary attacks, eCryptfs utilizes the method whereby a salt value is concatenated with a passphrase to generate a passphrase identifier. The concatenated value is iteratively hashed (65,537 times by default) to generate the identifying signature for the salted authentication token.

On the other hand, saltless authentication tokens exist only in the kernel keyring and are not at any time written out to disk. The userspace callout application combines these saltless authentication tokens with non-passphrased authentication tokens to generate candidate salted authentication tokens, whose signatures are compared against those in file headers.

While eCryptfs supports passphrase-based protection of files, we do not recommend using passphrases for relatively high-value data that requires more than casual protection. Most passphrases that people are capable of remembering are becoming increasingly vulnerable to brute force attacks. eCryptfs takes measures to make such attacks more difficult, but these measures can only be so effective against a determined and properly equipped adversary.

Every effort should be made to employ the use of a TPM and public key cryptography to provide strong protection of data. Keep in mind that using a passphrase authentication token in addition to a public key authentication token does not in any way combine the security of both; rather, it combines the *insecurity* of both. This is due to the fact that, given two authentication tokens, eCryptfs will encrypt and store two copies of the session key (see Section 5.1) that can individually be attacked.

5.3 Kernel Keyring

David Howells recently authored the keyring service, which kernel versions 2.6.10 and later now include. This keyring provides a host of features to manage and protect keys and authentication tokens. eCryptfs takes advantage of the kernel keyring, utilizing it to store authentication tokens, inode cryptographic contexts, and keys.

5.4 Callout and Daemon

The primary contact between the eCryptfs kernel module and the userspace key management code is the request-key callout application, which the kernel keyring invokes. This callout application parses policy information from the target, which it interprets in relation to the header information in each file. It may then make calls through the PKI API in order to satisfy pending public key requests, or it may go searching for a salted passphrase with a particular signature.

In order to be able to prompt the user for a passphrase via a dialog box, eCryptfs must have an avenue whereby it can get to the user's X session. The user can provide this means by simply running a daemon. The eCryptfs daemon listens to a socket (for which the location is written to the user's session keyring). Whenever policy calls for the user to be prompted for a passphrase, the callout application can retrieve the socket's location and use it to request the daemon to prompt the user; the daemon then returns the user's passphrase to the callout application.

5.5 Userspace Utilities

To accommodate those who are not running the eCryptfs layer on their systems, userspace utilities to handle the encrypted content comprise part of the eCryptfs package. These utilities act much like scaled-down versions of GnuPG.

5.6 Pluggable Authentication Module

Pluggable Authentication Modules (PAM) provide a Discretionary Access Control (DAC)[14] mechanism whereby administrators can parameterize how a user is authenticated and what happens at the time of authentication. eCryptfs includes a module that captures the user's login passphrase and stores it in the user's session keyring. This passphrase is stored in the user's session keyring as a saltless passphrase authentication token.

Future actions by eCryptfs, based on policy, can then use this passphrase to perform cryptographic operations. For example, the login passphrase can be used to extract the user's private key from his GnuPG keyring. It could be used to derive a key (via a string-to-key operation) that is directly used to protect a session key for a set of files. Furthermore, this derived key could be combined with a key stored in a TPM in order to offer two-factor authentication (i.e., in order to access a file, the user must have (1) logged into a particular host (2) using a particular passphrase).

Due to PAM's flexibility, these operations do not need to be restricted to a passphrase. There is no reason, for example, that a key contained on a SmartCard or USB device could not be used to help authenticate the user, after which point that key is used in the above named cryptographic operations.

5.7 PKI

eCryptfs offers a pluggable Public Key Infrastructure (PKI) interface. PKI modules accept key identifiers and data, and they return encrypted or decrypted data. Whether any particular key associated with an identifier is available, trustworthy, etc., is up to the PKI module to determine.

eCryptfs PKI modules need to implement a set of functions that accept as input the key identifier and a blob of data. The modules have the responsibility to take whatever course of action is necessary to retrieve the requisite key, evaluate the trustworthiness of that key, and perform the public key operation.

eCryptfs includes a PKI module that utilizes the GnuPG Made Easy (GPGME) interface to access and utilize the user's GnuPG keyring. This module can utilize the user's login passphrase credential, which is stored in the user's session keyring by the eCryptfs PAM (see Section 5.6), to decrypt and utilize the user's private key stored on the user's keyring.

The eCryptfs TPM PKI module utilizes the TrouSerS[26] interface to communicate with the Trusted Platform Module. This allows for

the use of a private key that is locked in the hardware, binding a file to a particular host.

The eCryptfs openCryptoki PKCS#11[15] framework PKI provides a mechanism for performing public key operations via various hardware devices supported by openCryptoki, including the IBM Cryptographic Accelerator (ICA) Model 2058, the IBM 4758 PCI Cryptographic Coprocessor, the Broadcom Crypto Accelerator, the AEP Crypto Accelerator, and the TPM.

It is easy to write additional PKI modules for eCryptfs. Such modules can interface with existing PKI's that utilize x.509 certificates, with certificate authorities, revocation lists, and other elements that help manage keys within an organization.

5.8 Key Escrow/Secret Sharing

In enterprise environments, it often makes sense for data confidentiality and integrity to be a shared responsibility. Just as prudent business organizations entail backup plans in the event of the sudden loss of any one employee, the data associated with business operations must survive any one individual in the company. In the vast majority of the cases, it is acceptable for all members of the business to have access to a set of data, while it is not acceptable for someone outside the company who steals a machine or a USB pen drive to have access to that data. In such cases, some forms of key escrow within the company are appropriate.

In enterprise environments where corporate and customer data are being protected cryptographically, key management and key recovery is an especially critical issue. Techniques such as secret splitting or (m,n)-threshold schemes[4] can be used within an organization to balance the need for key secrecy with the need for key recovery.

5.9 Target-centric Policies

When an application creates a new file, eCryptfs must make a number of decisions with regard to that file. Should the file be encrypted or unencrypted? If encrypted, which symmetric block cipher should be used to encrypt the data? Should the file contain HMAC's in addition to IV's? What should the session key length be? How should the session key be protected?

Protecting the session key on disk requires even more policy decisions. Should a passphrase be used? Which one, and how should it be retrieved? What should be the string-to-key parameters (i.e., which hash algorithm and the number of hash iterations)? Should any public keys be used? If so, which ones, and how should they be retrieved?

eCryptfs currently supports Apache-like policy definition files[3] that contain the policies that apply to the target in which they exist. For example, if the root directory on a USB pen drive device contains a .ecryptfsrc file, then eCryptfs will parse the policy from that file and apply it to all files under the mount point associated with that USB pen drive device.

Key definitions associate labels with (PKI, Id) tuples (see Figure 6).

Application directive definitions override default policies for the target, dependent upon the application performing the action and the type of action the application is performing (see Figure 7).

The action definitions associate labels with $(Action, Cipher, SessionKeySize)$ tuples. eCryptfs uses these directives to set cryptographic context parameters for files (see Figure 8).

[3] XML formats are currently being worked on.

```
<ApplicationDirectiveDef mutt_prompt_on_read_encrypted>
  Application /usr/bin/mutt
  Scenario OPEN_FOR_READ
  FileState ENCRYPTED
  Action PROMPT
</ApplicationDirectiveDef>

<ApplicationDirectiveDef mutt_decrypt>
  Application /usr/bin/mutt
  Scenario ALL
  FileState ENCRYPTED
  Action DECRYPT
</ApplicationDirectiveDef>

<ApplicationDirectiveDef openoffice_strong_encrypt>
  Application /usr/bin/ooffice
  Scenario OPEN_FOR_CREATE
  Action encrypt_strong
</ApplicationDirective>
```

Figure 7: Example policy: application directives

The directory policy definitions give default actions for files created under the specified directory location, along with application directives that apply to the directory location (see Figure 9).

6 Additional Measures

eCryptfs concerns itself mainly with protecting data that leaves trusted domains. Additional measures are necessary to address various threats outside the scope of eCryptfs's influence. For example, swap space should be encrypted; this can be easily accomplished with dm-crypt.[18]

Strong access control is outside the scope of the eCryptfs project, yet it is absolutely necessary to provide a comprehensive security solution for sensitive data. SE Linux[16] provides a robust Mandatory Access Control framework that can be leveraged with policies to protect the user's data and keys.

Furthermore, the system should judiciously employ timeouts or periods of accessibility/applicability of credentials. The kernel keyring provides a convenient and powerful mechanism for handling key permissions and expirations. These features must be used appropriately in order to address human oversight, such as failing to lock down a terminal or otherwise exit or invalidate a security context when the user is finished with a task.

7 Future Work

eCryptfs is currently in an experimental stage of development. While the majority of the VFS

```
<Directory />
  DefaultAction blowfish_encrypt
  DefaultState PROMPT
  DefaultPublicKeys mhalcrow legal
  DefaultPassphrase LOGIN
  # This directives for files under this location
  # that meet this criteria
  <FilePattern \.mutt_.*>
    ApplicationDirective mutt_decrypt
  </FilePattern>
  ApplicationDirective mutt_prompt_on_read_encrypted
</Directory>

# Overrides the prior set of policies
<Directory /gnucash>
  DefaultAction encrypt_strong
  DefaultPublicKeys host_tpm
</Directory>
```

Figure 9: Example policy: directory policies

functionality is implemented and functioning, eCryptfs requires testing and debugging across a wide range of platforms under a variety of workload conditions.

eCryptfs has the potential to provide weak file size secrecy in that the size of the file would only be determinable to the granularity of one extent size, given that the file size field in the header is encrypted with the session key. Strong file size secrecy is much more easily obtained through block device layer encryption, where everything about the filesystem is encrypted. eCryptfs only encrypts the data contents of the files; additional secrecy measures must address dentry's, filenames, and Extended Attributes, which are all within the realm of what eCryptfs can influence.

At this stage, eCryptfs requires extensive profiling and streamlining in order to optimize its performance. We need to investigate opportunities for caching cryptographic metadata, and variations on such attributes as the size of the extents could have a significant impact on speed.

eCryptfs policy files are equivalent to the Apache configuration files in form and complexity. eCryptfs policy files are amenable to guided generation via user utilities. Another significant area of future development includes the development of such utilities to aid in the generation of these policy files.

Desktop environments such as GNOME or KDE can provide users with a convenient interface through which to work with the cryptographic properties of the files. In one scenario, by right-clicking on an icon representing the file and selecting "Security", the user will be presented with a window that can be used to control the encryption status of the file. Such options will include whether or not the

```
<Key mhalcrow>
  PKI GPG
  Id 3F5C22A9
</Key>

<Key legal>
  PKI GPG
  Id 7AB1FF25
</Key>

<Key host_tpm>
  PKI TPM
  Id DEFAULT
</Key>
```

Figure 6: Example policy: key defs

```
<ActionDef blowfish_encrypt>
  Action ENCRYPT
  Cipher blowfish
  SessionKeySize 128
</ActionDef>

<ActionDef encrypt_strong>
  Action ENCRYPT
  Cipher aes
  SessionKeySize 256
</ActionDef>
```

Figure 8: Example policy: action defs

file is encrypted, which users should be able to encrypt and decrypt the file (identified by their public keys as reported by the PKI plugin module), what cipher is used, what keylength is used, an optional passphrase that is used to encrypt the symmetric key, whether or not to use keyed hashing over extents of the file for integrity, the hash algorithms to use, whether accesses to the file when no key is available should result in an error or in the encrypted blocks being returned (as dictated by target-centric policies; see Section 5.9), and other properties that are interpreted and used by the eCryptfs layer.

8 Recognitions

We would like to express our appreciation for the contributions and input on the part of all those who have laid the groundwork for an effort toward transparent filesystem encryption. This includes contributors to FiST and Cryptfs, GnuPG, PAM, and many others from which we are basing our development efforts, as well as several members of the kernel development community.

9 Conclusion

eCryptfs is an effort to reduce the barriers that stand in the way of the effective and ubiquitous utilization of file encryption. This is especially relevant as physical media remains exposed to theft and unauthorized access. Whenever sensitive data is being handled, it should be the *modus operandi* that the data be encrypted at all times when it is not directly being accessed in an authorized manner by the applications. Through strong and transparent key management that includes public key support, key->file association, and target-centric policies, eCryptfs provides the means whereby a cryptographic filesystem solution can be more easily and effectively deployed.

10 Availability

eCryptfs is licensed under the GNU General Public License (GPL). SourceForge is hosting the eCryptfs code base at `http://sourceforge.net/projects/ecryptfs`. We welcome any interested parties to become involved in the testing and development of eCryptfs.

11 Legal Statement

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM and Lotus Notes are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

References

[1] M. Blaze. "Key Management in an Encrypting File System", Proc. *Summer '94 USENIX Tech. Conference*, Boston, MA, June 1994.

[2] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. RFC 2440. November 1998. See `ftp://ftp.rfc-editor.org/in-notes/rfc2440.txt`.

[3] M. Halcrow. "Demands, Solutions, and Improvements for Linux Filesystem Security." *Proceedings of the 2004 Ottawa Linux Symposium*, Ottawa, Canada, July 2004.

[4] S.C. Kothari. "Generalized Linear Threshold Scheme." *Advances in Cryptology: Proceedings of CRYPTO 84*, Springer-Verlag, 1985, pp. 231–241.

[5] M. Liedtke. "Stolen UC Berkeley laptop exposes personal data of nearly 100,000." Associated Press. March 29, 2005. See `http://www.sfgate.com/cgi-bin/article.cgi?f=/n/a/2005/03/28/financial/f151143S80.DTL`

[6] B. Schneier. *Applied Cryptography*. New York: John Wiley & Sons, Inc., 1996. Pp. 193–197.

[7] The Trusted Computing Group. "TCG Specification Architecture Overview version 1.0." `https://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Overview.pdf`

[8] E. Zadok, L. Badulescu, and A. Shender. "Cryptfs: A stackable vnode level encryption file system." *Technical Report CUCS-021-98, Computer Science Department*, Columbia University, 1998.

[9] E. Zadok and J. Nieh. "FiST: A Language for Stackable File Systems." *Proceedings of the Annual USENIX Technical Conference*, pp. 55–70, San Diego, June 2000.

[10] "Ameritrade addresses missing tape." United Press International. April 19, 2005. See `http://washingtontimes.com/upi-breaking/20050419-110638-7335r.htm`

[11] For more information on IBM Lotus Notes, see `http://www-306.ibm.com/software/lotus/`. Information on Notes security can be obtained from `http://www-10.lotus.com/ldd/`

today.nsf/f01245ebfc115aaf8525661a006b86b9/232e604b847d2cad88256ab90074e298?OpenDocument

[12] For more information on Pluggable Authentication Modules (PAM), see http://www.kernel.org/pub/linux/libs/pam/

[13] For more information on Mandatory Access Control (MAC), see http://csrc.nist.gov/publications/nistpubs/800-7/node35.html

[14] For more information on Discretionary Access Control (DAC), see http://csrc.nist.gov/publications/nistpubs/800-7/node25.html

[15] For more information on openCryptoki, see http://sourceforge.net/projects/opencryptoki

[16] For more information on Security-Enhanced Linux (SE Linux), see http://www.nsa.gov/selinux/index.cfm

[17] For more information on Samhain, see http://la-samhna.de/samhain/

[18] For more information on DM-crypt, see http://www.saout.de/misc/dm-crypt/

[19] For more information on PPDD, see http://linux01.gwdg.de/~alatham/ppdd.html

[20] For more information on CFS, see http://sourceforge.net/projects/cfsnfs/

[21] For more information on BestCrypt, see http://www.jetico.com/index.htm#/products.htm

[22] For more information on TCFS, see http://www.tcfs.it/

[23] For more information on EncFS, see http://arg0.net/users/vgough/encfs.html

[24] For more information on CryptoFS, see http://reboot.animeirc.de/cryptofs/

[25] For more information on Reiser4, see http://www.namesys.com/v4/v4.html

[26] For more information on TrouSerS, see http://sourceforge.net/projects/trousers/

We Are Not Getting Any Younger: A New Approach to Time and Timers

John Stultz, Nishanth Aravamudan, Darren Hart
IBM Linux Technology Center
{johnstul, dvhltc, nacc}@us.ibm.com

Abstract

The Linux® time subsystem, which once provided only tick granularity via a simple periodic addition to xtime, now must provide nanosecond resolution. As more and more unique timekeeping hardware becomes available, and as virtualization and low-latency demands grow, the complexity of maintenance and bug resolution increases.

We are proposing a significant re-work of the time keeping subsystem of the kernel. Additionally, we demonstrate some possible enhancements this re-work facilitates: a more flexible soft-timer subsystem and dynamic interrupt source management.

1 The New `timeofday` Subsystem

The functionality required of the `timeofday` subsystem is deceptively simple. We must provide a monotonically increasing system clock, a fast and accurate method for generating the time of day, and a method for making small adjustments to compensate for clock drift. In the existing code, these are provided by the `wall_to_monotonic` offset to `do_gettimeofday()`, the `do_gettimeofday()` function and the Network Time Protocal(NTP) `adjtimex()` interface. This basic functionality, however, is required to meet increasingly stringent demands. Performance must improve and time resolution must increase, while keeping correctness. Meeting these demands with the current code has become difficult, thus new methods for time keeping must be considered.

1.1 Terminology

Before we get into the paper, let us just cover some quick terminology used, so there is no confusion.

System Time A monotonically increasing value that represents the amount of time the system has been running.

Wall Time (Time of Day) A value representing the the human time of day, as seen on a wrist-watch.

Timesource A representation of a free running counter running at a known frequency, usually in hardware.

Hardware-Timer (Interrupt Source) A bit of hardware that can be programmed to generate an interrupt at a specific point in time. Frequently the hardware-timer can be used as a timesource as well.

Soft-Timer A software kernel construct that runs a callback at a specified time.

Tick A periodic interrupt generated by a hardware-timer, typically with a fixed interval defined by `HZ`. Normally used for timekeeping and soft-timer management.

1.2 Reasons for Change

1.2.1 Correctness

So, why are we proposing these changes? There are many reasons, but the most important one is correctness. It is critical that time flow smoothly, accurately and does not go backwards at any point. The existing interpolation-based timekeeping used by most arches[1] is error prone.

Quickly, let us review how the existing timekeeping code works. The code is tick-based, so the `timeofday` is incremented a constant amount (plus a minor adjustment for NTP) every tick. A simplified pseudo-code example of this tick-based scheme would look like Figure 1.

Since, in this code, `abs(ntp_adjustment)` is always smaller then `NSECS_PER_TICK`, time cannot go backwards. Thus, the only issue with this example is that the resolution of time is not very granular, e.g. only 1 millisecond when $\texttt{HZ} = 1000$. To solve this, the existing `timeofday` code uses a high-resolution timesource to interpolate with the tick-based time keeping. Again, using a simplified pseudo-code example we get something like Figure 2.

The idea is that the interpolation function (`cycles2ns(now - hi_res_base)`) will smoothly give the inter-tick position in time. Now, a sharp eye will notice that this has some potential problems. If the interpolation function does not cover the entire tick interval (`NSECS_PER_TICK + ntp_adjustment`), time will jump forward. More worrisome though, if at any time the value of the interpolation function grows to be larger than the tick interval, there is the potential for time to go backwards. These two conditions can be caused by a number of factors: calibration error, changes to `ntp_adjustment`'s value, interrupt handler delay, timesource frequency changes, and lost ticks.

The first three cases typically lead to a small single-digit microsecond error, and thus, a small window for inconsistency. As processor speeds increase, `gettimeofday` takes less time and small interpolation errors become more apparent.

Timesource frequency changes are more difficult to deal with. While they are less common, some laptops do not notify the `cpufreq` subsystem when they change processor frequency. Should the user boot off of battery power, and the system calibrate the timesource at the slow speed then later plug into the wall, the TSC frequency can triple causing much more obvious (two tick length) errors.

Lost timer interrupts also cause large and easily visible time inconsistencies. If a bad driver blocks interrupts for too long, or a BIOS SMI interrupt blocks the OS from running, it is possible for the value of the interpolation function to grow to multiple tick intervals in length. When a timer interrupt is finally taken though, only one interval is accumulated. Since changing `HZ` to 1000 on most systems, missed timer interrupts have become more common, causing large time inconsistencies and clock drift.

Attempts have been made (in many cases by one of the authors of this paper) to reduce the

[1] Throughout this paper, we will refer to the software architecture(s), i.e. those found in `linux-source/arch`, as arch(es) and to hardware architecture(s) as architecture(s).

```
timer_interrupt():
    xtime += NSECS_PER_TICK + ntp_adjustment

gettimeofday():
    return xtime
```

Figure 1: A tick-based `timeofday` implementation

```
timer_interrupt():
    hi_res_base = read_timesource()
    xtime += NSECS_PER_TICK + ntp_adjustment

gettimeofday():
    now = read_timesource()
    return xtime + cycles2ns(now - hi_res_base)
```

Figure 2: An interpolated `gettimeofday()`

impact of lost ticks by trying to detect and compensate for them. At interrupt time, the interpolation function is used to see how much time has passed, and any lost ticks are then emulated. One problem with this method of detecting lost ticks is that it results in false positives when timesource frequency changes occur. Thus, instead of time inconsistencies, time races three times faster on those systems, necessitating additional heuristics.

In summary, the kernel uses a buggy method for calculating time using both ticks and timesources, neither of which can be trusted in all situations. This is not good.

1.2.2 A Much Needed Cleanup

Another problem with the current time keeping code is how fractured the code base has become. Every arch calculates time in basically the same manner, however, each one has its own implementation with minor differences. In many cases bugs have been fixed in one or two arches but not in the rest. The arch independent code is fragmented over a number of files (time.c, timer.c, time.h, timer.h, times.h, timex.h), where the divisions have lost any meaning or reason. Without clear and explicit purpose to these files, new code has been added to these files haphazardly, making it even more difficult to make sense of what is changing.

The lack of opacity in the current timekeeping code is an issue as well. Since almost all of the timekeeping variables are global and have no clear interface, they are accessed in a number of different ways in a number of different places in the code. This has caused much confusion in the way kernel code accesses time. For example, consider the many possible ways to calculate uptime shown in Figure 3.

Clearly some of these examples are more correct than others, but there is not one clear way of doing it. Since different ways to calculate time are used in the kernel, bugs caused by mixing methods are common.

1.2.3 Needed Flexibility

The current system also lacks flexibility. Current workarounds for things like lost ticks cause

```
uptime = jiffies * HZ
uptime = (jiffies - INITIAL_JIFFIES) * HZ
uptime = ((jiffies - INITIAL_JIFFIES) * ACTHZ) >> 8
uptime = xtime.tv_sec + wall_to_monotonic.tv_sec
uptime = monotonic_clock() / NSEC_PER_SEC;
```

Figure 3: Possible ways to calculate uptime

bugs elsewhere. One such place is virtualization, where the guest OS may be halted for many ticks while another OS runs. This requires the hypervisor to emulate a number of successive ticks when the guest OS runs. The lost tick compensation code notes the hang and tries to compensate on the first tick, but then the next ticks arrive causing time to run too fast. Needless to say, this dependency on ticks increases the difficulty of correctly implementing deeper kernel changes like those proposed in other patches such as: Dynamic Ticks, `NO_IDLE_HZ`, Variable System Tick, and High Res Timers.

1.3 Related Work

1.3.1 `timer_opts`

Now that we are done complaining about all the problems of the current timekeeping subsystem, one of the authors should stand up and claim his portion of the responsibility. John Stultz has been working in this area off and on for the last three years. His largest set of changes was the i386-specific `timer_opts` reorganization. The main purpose of that code was to modularize the high-resolution interpolator to allow for easy addition of alternative timesources. At that time the i386 arch supported the PIT and the TSC, and it was necessary to add support for a third timesource, the Cyclone counter. Thanks, in part, to to the `timer_opts` changes the kernel now supports the ACPI PM and HPET timesources, so in a sense the code did what was needed, but it is not without its problems.

The first and most irritating issue is the name. While it is called `timer_opts`, nothing in the structure actually uses any of the underlying hardware as a hardware-timer. Part of this confusion comes from the fact that hardware-timers can frequently be used as counters or timesources, but not the other way around, as timesources do not necessarilly generate interrupts. The lack of precision in naming and speaking about the code has caused continuing difficulty in discussing the issues surrounding it.

The other problem with the `timer_opts` structure is that its interface is too flexible and openly defined. It is unclear for those writing new `timer_opt` modules, how to use the interface as intended. Additionally, fixing a bug or adding a feature requires implementing the same code across each `timer_opt` module, leading to excessive code duplication. That along with additional interfaces being added (each needing their own implementation) has caused the `timer_opts` modules to become ugly and confusing. A cleanup is in order.

Finally, while the `timer_opts` structure is somewhat modular, the list of available timesources becomes fixed at build-time. Having the “clock=” boot-time parameter is useful, allowing users to override the default timesource; however, more flexibility in providing new timesources at run-time would be helpful.

1.3.2 `time_interpolator`

Right about the time the `timer_opts` structure got into the kernel, a similar bit of code called "time interpolation hooks" showed up implementing a somewhat similar interface. An additional benefit of this code was that it was arch independent, promising the ability to share interpolator "drivers" between different arches that had the same hardware. John followed a bit of its discussion and intended to move the i386 code over to it, but was distracted by other work requirements. That and the never-ending 2.6 freeze kept him from actually attempting the change.

John finally got a chance to really look at the code when he implemented the Cyclone interpolator driver. The code was nice and more modular than the `timer_opts` interface, but still had some of the same faults: it left too much up to the driver to implement and the `getoffset()`, `update()`, and `reset()` interfaces were not intuitive. Impressively, much of the time-interpolator code has been recently re-written, resolving many of issues and influencing this proposal. However, the time-interpolator design is still less than ideal. NTP adjustments are done by intentionally undershooting in converting from cycles to nanoseconds, causing time to run just a touch slow, and thus, forcing NTP to only make adjustments forward in time. This trick avoids time inconsistencies from NTP adjustments, but causes time drift on systems that do not run NTP. Additionally, while interpolation errors are no longer an issue, the code is still tick based, which makes it more difficult to understand and extend.

1.4 Our Proposal

Before we get into the implementation details of our proposal, let us review the goals:

1. Clean up and simplify time related code.
2. Provide clean and clear `timeofday` interfaces.
3. Use nanoseconds as the fundamental time unit.
4. Stop using tick-based time and avoid interpolation.
5. Make much of the implementation arch independent.
6. Use a modular design, allowing timesource drivers to be installed at runtime.

The core implementation has three main components: the `timeofday` code, timesource management, and the NTP state machine.

1.4.1 `timeofday` Core

The core of the timekeeping code provides methods for getting a monotonically increasing system time and the wall time. To avoid unnecessary complication, we layer these two values in a simple way. The monotonically increasing system time, accessed via `monotonic_clock()` is the base layer. On top of that we add a constant offset `wall_time_offset` to calculate the wall time value returned by `do_gettimeofday()`. The code looks like Figure 4.

The first thing to note in Figure 4, is that the `timeofday` code is not using interpolation. The amount of time accumulated in the `periodic_hook()` function is the exact same as would be calculated in `monotonic_clock()`. This means the `timeofday` code is no longer dependent on timer interrupts being received at a regular interval. If a tick arrives late, that is okay, we will just accumulate the actual amount of time that has past and reset the `offset_base`. In fact, `periodic_hook()` does not need to be called

```
nsec_t system_time
nsec_t wall_time_offset
cycle_t offset_base
int ntp_adj
struct timesource_t ts

monotonic_clock():
    now = read_timesource(ts)
    return system_time + cycles2ns(ts, now - offset_base, ntp_adj)

gettimeofday():
    return monotonic_clock() + wall_time_offset

periodic_hook():
    now = read_timesource(ts)
    interval = cycles2ns(ts, now - offset_base, ntp_adj)
    system_time += interval
    offset_base = now
    ntp_adj = ntp_advance(interval)
```

Figure 4: The new timekeeping psuedo-code

from `timer_interrupt()`, instead it can be called from a soft-timer scheduled to run every number of ticks. Additionally, notice that NTP adjustments are done smoothly and consistenetly throughout the time interval between `periodic_hook()` calls. This avoids the interpolation error that occurs with the current code when the NTP adjustment is only applied at tick time. Another benefit is that the core algorithm is shared between all arches. This consolidates a large amount of redundant arch specific code, which simplifies maintenance and reduces the number of arch specific time bugs.

1.4.2 Timesource Management

The timesource management code defines a timesource, and provides accessor functions for reading and converting timesource cycle values to nanoseconds. Additionally, it provides the interface for timesources to be registered, and then selected by the kernel. The timesoure structure is defined in Figure 5.

In this structure, the `priority` field allows for the best available timesource to be chosen. The `type` defines if the timesource can be directly accessed from the on-CPU cycle counter, via MMIO, or via a function call, which are defined by the `read_fnct` and `mmio_ptr` pointers respectively. The `mask` value ensures that subtraction between counter values from counters that are less then 64 bits do not need special overflow logic. The `mult` and `shift` approximate the frequency value of cycles over nanoseconds, where $\texttt{frequency} \approx \frac{\texttt{mult}}{2^{\texttt{shift}}}$ Finally, the `update_callback()` is used as a notifier for a safe point where the timesource can change its `mult` or `shift` values if needed, e.g. in the case of `cpufreq` scaling.

A simple example `timesource` structure can be seen in Figure 6. This small HPET driver can even be shared between i386, x86-64 and ia64 arches (or any arch that supports HPET). All that is necessary is an initialization function that sets the `mmio_ptr` and `mult` then calls `register_timesource()`. This can be done

```
struct timesource_t {
    char* name;
    int priority;
    enum {
        TIMESOURCE_FUNCTION,
        TIMESOURCE_CYCLES,
        TIMESOURCE_MMIO_32,
        TIMESOURCE_MMIO_64
    } type;
    cycle_t (*read_fnct)(void);
    void __iomem *mmio_ptr;
    cycle_t mask;
    u32 mult;
    u32 shift;
    void (*update_callback)(void);
};
```

Figure 5: The `timesource` structure

```
struct timesource_t timesource_hpet = {
    .name = ''hpet'',
    .priority = 300,
    .type = TIMESOURCE_MMIO_32,
    .mmio_ptr = NULL,
    .mask = (cycle_t)HPET_MASK,
    .mult = 0,
    .shift = HPET_SHIFT,
};
```

Figure 6: The HPET `timesource` structure

at any time while the system is running, even from a module. At that point, the timesource management code will choose the best available timesource using the priority field. Alternatively, a `sysfs` interface allows users to override the priority list and, while the system is running, manually select a timesource to use.

1.4.3 NTP

The NTP subsystem provides a way for the kernel to keep track of clock drift and calculate how to adjust for it. Briefly, the core interface from the timekeeping perspective is `ntp_advance()`, which takes a time interval and increments the NTP state machine by that amount, and then returns the signed parts per million adjustment value to be used to adjust time consistently over the next interval.

1.4.4 Related Changes

Other areas affected by this proposal are VDSO or vsyscall `gettimeofday()` implementations. These are chunks of kernel code mapped into user-space that implement `gettimeofday()`. These implementations are somewhat hackish, as they require heavy linker magic to map kernel variables into the address space twice. In the current code, this

dual mapping further entangles the global variables used for timekeeping. Luckily, our proposed changes can adapt to handle these cases.

The `timesource` structure has been designed to be a fairly translucent interface. Thus, any `timesource` of type `MMIO` or `CYCLES` can be easily used in a VDSO. By making a call to an arch specific function whenever the base time values change, the arch independent and specific code are able to be cleanly split. This avoids tangling the `timeofday` code with ugly linker magic, while still letting these significant optimizations occur in whatever method is appropriate for each arch.

1.5 What Have We Gained?

With these new changes, we have simplified the way time keeping is done in the kernel while providing a clear interface to the required functionality. We have provided higher resolution, nanoseconds based, time keeping. We have streamlined the code and allowed for methods which further increase gettimeofday performance. And finally, we have organized and cleaned up the code to fix a number of problematic bugs in the current implementation.

With a foundation of clean code and clear interfaces has been laid, we can look for deeper cleanups. A clear target for improvement is the soft-timer subsystem. The `timeofday` rework clearly redefines the split between system time and `jiffies`. Changing the soft-timer subsystem to human-time frees the kernel from inaccurate, tick-based time (see §3.2).

2 Human-Time

2.1 Why Human-Time Units?

Throughout the kernel, time is expressed in units of `jiffies`, which is only a timer interrupt counter. Since the interrupt interval differs between archs, the amount of time one jiffy represents is not absolute. In contrast, the amount of time one nanosecond represents is an independent concept.

When discussing human-time units, e.g. seconds, nanoseconds, etc., and the kernel, there are two main questions to be asked: "Why should the kernel interfaces change to use human-time units?" and "Why should the internal structures and algorithms change to use human-time units?" If a good answer to the latter can be found, then the former simply follows from it; a good answer can and will be provided, but we feel there are several reasons to make the interface change regardless of any changes to the infrastructure.

2.1.1 Interfaces in Human-Time

First of all, human-time units are the units of our thought; simultaneously, the units of computer design are in human-time (or their inverse for frequency measurements). The relation between human-time units and jiffies is vague, while it is clear how one human-time unit relates to another. Additionally, human-time units are effectively unbounded in terms of expressivity. That is to say, as systems achieve higher and higher granularity—currently expressed by moving to higher values of `HZ`—we simply multiply all of the existing constants by an appropriate power of 10 and change the internal resolution.

2.1.2 Infrastructure in Human-Time

The most straight-forward argument in favor of human-time interfaces stems from our proposed changes to the soft-timer subsystem. If the underlying algorithm adds and expires soft-timers in human-time units, then it follows that the interfaces to the subsystem should use the same units. But why change the infrastructure in the first place? All of the arguments mentioned in §2.1.1 apply equally well here. But our fundamental position—as alluded to in §1.2—is that the tick-based `jiffies` value is a poor representation of time.

The current soft-timer subsystem relies on the periodic timer tick, and its resolution is linked at compile time to the timer interrupt frequency value `HZ`. This approach to timer management works well for timers with expiration values at least an order of magnitude longer than that period. Higher resolution timers present several problems for a tick-based soft-timer system. The most obvious problem is that a timer set for a period shorter than a single tick cannot be handled efficiently. Even calls to `nanosleep()` with delays equal to the period of `HZ` will often experience latencies of three ticks!

On i386, for example, `HZ` is 1000, which indicates a timer interrupt should occur every millisecond. Because of limitations in hardware, the closest we can come to that is about 999,876 nanoseconds, just a little too fast. This actual frequency is represented by the `ACTHZ` constant. Since `jiffies` is kept in `HZ` units instead of `ACTHZ` units, when requests are made for one millisecond, two ticks are required to ensure one full millisecond elapses (instead of 999,876 nanoseconds). Then, since we do not know where in the current tick we are, an extra jiffy must be added. So a one millisecond `nanosleep()` turns into three jiffies.

2.1.3 A Concrete Example

To clarify our arguments, consider the example code in Figure 7.

It is clear that the old code is attempting to sleep for the shortest time possible (a single jiffy). Internally, a soft-timer will be added with an `expires` value of `jiffies` + 1. How long does this `expires` value actually represent? It is hard to say, as it depends on the value of `HZ`, which changed from 2.4 to 2.6. Perhaps it is 10 milliseconds (2.4), or perhaps it is one millisecond (2.6). What about the new kernel hacker who copies the code and uses it themselves—what assumption will they make regarding the timeout indicated? What happens when `HZ` has a dynamic value? Clearly, problems abound with this small chunk of code.

Consider, in contrast, the code after an update by the Kernel-Janitors[2] project. Now, it is unclear how the soft-timer subsystem will translate the milliseconds parameter to `msleep()` into an internal `expires` value, but in all honesty, that does not matter to the author. It is clear, however, that the author intends for the task to sleep for at least 10 milliseconds. `HZ` can change to any value it likes and the request is the same. In this case, it is up to the soft-timer subsystem to handle converting from human-time to `jiffies` and not other kernel developers (rejoice!).

These changes are in the best interest of the kernel; they will help with the long-term maintainability of much code, particularly in drivers.

3 The New soft-timer Subsystem

We have already argued that a human-time soft-timer subsystem is in the best interest of the

[2] http://www.kerneljanitors.org/

Old:

```
set_current_state(TASK_UNINTERRUPTIBLE);
schedule_timeout(1);
```

New:

```
msleep(10);
```

Figure 7: Two possible ways to sleep

kernel. Is such a change feasible? More importantly, what are the potential performance impacts of such a change? How should the interfaces be modified to accomodate the new system?

3.1 The Status Quo in Soft-Timers

A full exposition of the current soft-timer subsystem is beyond the scope of this paper. Instead, we will give a rough overview of the important terms necessary to understand the changes we hope to make. Additionally, keeping our comments in mind while examining this complex code should make the details easier to see and understand. Like much of the kernel, the soft-timer subsystem is defined by its data structures.

3.1.1 Buckets and Bucket Entries

There are five "buckets" in the soft-timer subsystem. Bucket one is special and designated the "root bucket." Each bucket is actually an array of `struct timer_list`s. The root bucket contains 256 bucket entries, while the remaining four buckets each contain 64.[3] Each entry represents an interval of jiffies; all soft-timers in a given entry have `expires` values in that entry's interval. Thus, when a timer in a particular entry is expired, all timers in that entry are expired, i.e. sorting is not necessary. In bucket one, each entry represents one jiffy. In bucket two, each entry represents a range of 256 jiffies. In bucket three, each entry represents a range of $64 \times 256 = 16384$ jiffies. Buckets four and five's entries represent intervals of $64 \times 64 \times 256 = 1048576$ and $64 \times 64 \times 64 \times 256 = 6467108864$ jiffies, respectively.

Imagine that we fix the first entry in bucket one (which we will designate `tv1`[4]) to be the initial value of `jiffies` (which we can pretend is zero). Then, treating the buckets like the arrays they are, `tv1[7]` represents the timers set to expire seven jiffies from now. Similarly, `tv3[4]` represents the timers with `expires` values satisfying $82175 \leq$ `expires` < 98559. The perceptive reader may have noticed a very nice relationship between bucket `tv[n]` and `tv[n+1]`: a single entry of `tv[n+1]` represents a span of jiffies exactly equal to the total span of all of `tv[n]`'s entries. Thus, once `tv[n]` is empty, we can refill it by pulling, or "cascading," an appropriate entry from `tv[n+1]` down.[5]

[3] This was the only possibility before `CONFIG_SMALL_BASE` was introduced. If `CONFIG_SMALL_BASE`=y, then bucket one is 64 entries wide and the other four buckets are each 16. See `kernel/timer.c`.

[4] The name given to the buckets in the code is "time vector."

[5] See `kernel/timer.c::cascade()` if you have any doubts.

3.1.2 Adding Timers

Imagine we keep a global value, `timer_jiffies`, indicating the `jiffies` value the last time timers were expired. Then take the `expires` value stored in the `timer_list`, which is in absolute jiffy units, and subtract `timer_jiffies`, thus giving a relative jiffy value.[6] Then determine into which entry the timer should be added by simple comparisons.

Keep in mind that for each bucket, we know the exact value of the least significant *X* bits, i.e. for all entries in `tv2`, the bottom eight bits are zero. Therefore, we can throw away those bits when indexing the bucket. Similarly, we also know the maximum value of any timer's `expires` field in a given bucket. Thus, we can ignore the top 18 bits in `tv2`. We are now at a six-bit value, which exactly indexes our 64-entry wide bucket! Similar logic holds true for the remaining buckets. All the gory details are available in `kernel/timer.c::internal_add_timer()`.

3.1.3 Expiring Timers

Expiration follows the addition algorithm pretty closely. Compare `timer_jiffies` to `jiffies`: if `jiffies` is greater, then we know that time has elapsed since we last expired timers and there might be timers to expire. We then search through `tv1`, beginning at the index corresponding to the lower eight bits of `timer_jiffies`, which would be timers added immediately after the last time we added expired timers. We expire all those timers and then increment `timer_jiffies`. This process repeats until either `timer_jiffies` = `jiffies` or we have reached the end of `tv1`. In the former case, we are done expiring timers and we can exit the expiration routine. In the latter case, we need to cascade an appropriate entry from a higher bucket down into tv1.[7] The strategy is to figure out which interval we are currently in relative to our system-wide initial value and re-add the corresponding timers to the system. This forces those timers which should be expired now into `tv1`. Thus, we only ever need to consider `tv1` when expiring timers. Timer expiration is accomplished by invoking the timer's callback function and removing the timer from the bucket.

[6] Relative to `timer_jiffies`, not `jiffies`.

3.2 What To Keep?

Our proposal is simple: keep the data structures and the algorithms for addition and expiration. Rather than fix the entry width to be one jiffy in `tv1`, we define a new unit: the `timerinterval`. This unit represents the best resolution of soft-timer addition and expiration. To convert from human-time units, we use a new `nsecs_to_timerintervals()` function. This allows us to preserve the algorithmic design of the soft-timer subsystem, which expects the timer's `expires` field to be an `unsigned long`. Correspondingly, we do not base our last expiration time (now stored in `last_timer_time` instead of `timer_jiffies`) and current expiration time on `jiffies`, but on the new `timeofday` subsystem's `do_monotonic_clock()`. Finally, we store the last expiration time in a more sensibly named variabled, `last_timer_time`, rather than `timer_jiffies`.

We actually require two conversion functions. On addition of soft-timers, we use `nsecs_to_timerintervals_ceiling()`, and on expiration of soft-timers, we use `nsecs_to_timerintervals_floor()`. This insures

[7] I hope all of you were highly suspicious of my claims and took a look at `kernel/timer.c::cascade()`.

that timers are not expired early. In the simplest case, where we wish to approximate the current millisecond granularity of `HZ` = 1000, the pseudocode shown in Figure 8 achieves the conversion.

In short, `timerintervals`, not `jiffies` are now the units of the soft-timer subsystem. The new system is extremely flexible. By changing the previous example's value of `TIMER_INTERVAL_BITS`, we are able to change the overall resolution of the soft-timer subsystem. We have made the soft-timer subsystem independent of the periodic timer tick.

3.3 New Interfaces

As was already mentioned, the new human-time infrastructure enables several new human-time interfaces. The reader should be aware that existing interfaces will continue to be supported, although they will be less precise as jiffies and human-time do not directly correspond to one another.

3.3.1 `add_timer`, `mod_timer`

After a careful review of the code, we believe the `add_timer()` interface should be deprecated. It duplicates the code in `mod_timer()`, using `timer->expires` as the `expires` value. Since we are moving away from the current use of `mod_timer()`, where the parameter is in jiffies, to a system using nanoseconds (a 64-bit value), we would like to avoid reworking the `timer_list` structure. `mod_timer()` is also deprecated with the new system, as we provide one clear interface to both add and modify timers, `set_timer_nsecs` (see §3.3.2).

3.3.2 `set_timer_nsecs`

This function accepts, as parameters, a `timer_list` to modify and an absolute number of nanoseconds, modifying the `timer_list`'s `expires` field accordingly. This is, in our new code, the preferred way to add and modify timers.

3.3.3 `schedule_timeout_nsecs`

`schedule_timeout_nsecs()` allows for relative timeouts, e.g. 10,000,000 nanoseconds (10 milliseconds) or 100 nanoseconds. The soft-timer subsystem will convert the relative human-time value to an appropriate absolute `timerinterval` value.

3.4 Future Direction and Enhancements

One area which has not received sufficient attention is the setting of timers using a relative `expires` parameter. That is, we should be able to specify `set_timer_rel_nsecs(timer, 10)` and `timer`'s `expires` value should be modified to 10 nanoseconds from now. Due to the higher precision of `do_monotonic_clock()` in contrast to `jiffies`, we must be careful to pick an appropriate and consistent function to determine when "now" is.

4 Dynamic Interrupt Source Management

4.1 Interrupt Source Management

With the changes to the soft-timer subsystem (see §3.2), we can address issues related to the

```
#define TIMER_INTERVAL_BITS 20
nsecs_to_timerintervals_ceiling(nsecs):
    return (((nsecs-1) >> TIMER_INTERVAL_BITS) & ULONG_MAX)+1

nsecs_to_timerintervals_floor(nsecs):
    return (nsecs >> TIMER_INTERVAL_BITS) & ULONG_MAX
```

Figure 8: Approximating HZ = 1000 with the new soft-timer subsystem

reliance on a periodic tick. With power constrained devices, we want to avoid unneccesary interrupts, keeping the processor in a low power mode as long as possible. In a virtual environent, it is useful to know how long between events a guest OS can be off the CPUs.

Many people have attacked these various problems individually and have been met with some success and some resistance. The new time system discussed in §1.5 enables the time system to do without the periodic tick and, therefore, frees the soft-timer system to follow suit. Currently, when the system timer interrupt is raised, its interrupt handler is run and all the expired timers are executed—which could be a lot, a few, or none at all. This polled approach to timer management is not very efficient and hinders improvements for virtualization and power constrained devices.

4.2 A New Approach

By changing the soft-timer implementation to schedule interrupts as needed, we can have a more efficient event based (rather than polled) soft-timer system. Our proposed changes leverage the existing NO_IDLE_HZ code to calculate when the next timer is due to expire and schedule an interrupt accordingly. This frees soft-timers from the periodic system tick and the associated overhead it imposes. Unfortunately, some decisions still have to be made as to how often we are willing to stop what we are doing and expire the timers. This period of time, the timerinterval, is configurable (see §3.2). The length of a timerinterval unit places the lower bound on the soft-timer resolution, while the hard-timer defines the upper bound of how long we can wait between expiring timers. The default of our proposed changes places the lower bound at about one millisecond, and the upper bound of the PIT, for example, would be 55 milliseconds. Hardware permitting, higher resolution timers are acheived by simply reducing the timerinterval unit length and we get the functionality of NO_IDLE_HZ for free!

4.3 Implementation

At the time of this writing, the implementation is undergoing development. This section outlines our proposed changes with some extra detail given to portions that are already implemented.

The new interrupt source management system consists of a slightly modified version of the NO_IDLE_HZ code, arch specific set_next_timer_interrupt() and a new next_timer_interrupt() routines, and calls to these routines in __run_timers() and set_timer_nsecs() (see §3.3.2).

The former next_timer_interrupt() routine has been renamed to next_timer_expires() to avoid confusion between the next timer's expires and when the the next interrupt is due to fire. The routine was updated to use the slightly modified soft-timer structures discussed in §3.2.

The arch-specific `next_timer_interrupt()` routine returns the time in absolute nanoseconds of when the next hard-timer interrupt will fire.

The arch-specific `set_next_timer_interrupt()` routine accepts an absolute nanosecond parameter specifying when the user would like the next interrupt to fire. Depending on the hard-timer being used, the routine calculates the optimal time to fire the next interrupt and returns that value to the caller. Because interrupt sources vary greatly in their implementation (counters vs. decrementers, memory mapped vs. port I/O vs. registers, etc.), each source must be treated individually. For example, older hardware that is dependant on the PIT as an interrupt source will not get higher resolution soft-timers or very long intervals between interrupts simply because the PIT is painfully slow to program (about 5.5 microseconds in our tests), and only 16 bits wide. At about 1.2 MHz the PIT's maximum delay is only 55 milliseconds. Fortunately, systems that must use the PIT can do so without incurring a penalty since the PIT interrupt scheduling function is free to reprogram the hardware only when it makes sense to do so. We have discussed the specifics of the PIT, but other interrupt sources such as local APICs, HPETs, decrementers, etc. provide more suitable interrupt sources. Since `set_next_timer_interrupt()` is arch specific, it can be `#defined` to do nothing for those archs that would prefer to rely on a periodic interrupt.

Projects such as Dynamic Ticks, Variable System Tick, High Res Timers, `NO_IDLE_HZ`, etc. attempt to solve the limitations of the current soft-timer system. They approach each problem individually by adding code on top of the existing tick based system. In contrast, by integrating dynamically scheduled interrupts with the new time and soft-timer systems discussed earlier, we create a clean, simple solution that avoids the overhead of periodic ticks and provides similar functionality.

Conclusion

We have reimplemented the `timeofday` subsystem to be independent of `jiffies`, thus resolving a number of outstanding bugs and limitations. We have also demonstrated how these changes facilitate cleanups and new features in the soft-timer subsystem. We have reoriented the time and timer subsystems to be human-time based, thus improving flexibility, readability, and maintainability.

Legal Statement

Automated BoardFarm: Only Better with Bacon

Christian Höltje
TimeSys Corp.
christian.holtje@timesys.com
Bryan Mills
TimeSys Corp.
bryan.mills@timesys.com

Abstract

In this presentation, we introduce the concept of a BoardFarm, a tool to aid in the development and support of embedded systems. TimeSys had an opportunity to save time and energy that was being spent juggling a limited number of embedded boards among our support staff and developers who are spread throughout the world. We decided to build a system to provide remote access to the boards and to automate many of the tedious tasks such as running tests, booting the boards and installing software including the operating systems, board support packages and toolchains. This allows the developers and support gurus at TimeSys to concentrate on specific problems instead of how each board boots or how a specific board needs to be set up.

We talk about why the BoardFarm was built, how to use it, how it works, and what it's being used for. We also talk about ideas that we have for future improvements. Pigs were harmed in the making of this BoardFarm and were delicious.

1 Intro

Let's talk about embedded system boards. In the fast paced hi-tech world of embedded development, we just call them "boards." These little gizmos are developer prototypes, used to design software that goes in your MP3 player, routers, your new car, and Mars rovers.

These boards usually look like motherboards with the normal ports and such, but not always. Some are huge and others are itsy-bitsy. They all have one thing in common, though. They are expensive. These boards are usually bleeding edge technology and are only made available so that developers can get software out the door before the boards become obsolete. This means that usually a company can only afford one (or maybe two) of a board that they are working on.

The BoardFarm is an online tool that gives users remote access to all the boards. It is an interface to the boards and includes automation and testing. The goal is make the boards available to all users, everywhere, all the time.

2 Problems

Fred and Sue have problems.

The first problem is finding a board. “Fred had that last week,” but Fred says that Sue has it now, but she’s out of the office. Where *is* that board? This is asset management, but it is complicated by having the boards move between developers and with no central repository. It is even more complicated if your coworkers are not in the building or worse, in a different town or state! “Oh, yeah, Sue took that with her to California, did you need that?”

What if Fred and Sue need the same board at the same time? “Board snatching” is the common name for a form of stealing (not usually investigated by Scotland Yard) and is quite common when two people have deadlines involving the same board. If they’re lucky, they can cooperate and maybe work at different hours to share the board.

Finally, and possibly the most frustrating, is having to rediscover how to boot and configure a board. “Didn’t Fred work on that board a year ago?” A lot of the boards are unlabeled, only configurable through arcane methods or require sacrifices (under a full moon) to enter the boot loader. Sure, the developer should write it down. Fred usually does. But writing down a process is never as good as having functioning code that does the process. And the code is well exercised which means you know it works.

3 The BoardFarm: Grab a hoe and get to it

So what do we do? Panic? Sure! Run around in circles? Why not? Write code? Nah... No, wait, that was right answer.

So now, instead of investing in sharp cutlery, Fred and Sue can use our BoardFarm. The boards are safely locked away where they will no longer be stolen or used as blunt weapons, yet are easily accessible and even more useful than before.

What was our secret? We used the awesome power of python and bacon to create the BoardFarm. The BoardFarm acts as resource manager, automating some tasks and scheduling when Fred and Sue can use boards. It knows how to configure the board, install a board support package (BSP), boot the board using the BSP, and even how to run tests. This means that Fred and Sue can worry about their own problems, not who has the board or how the thing boots.

3.1 A day in the life of Fred and Sue

Sue has been working on a really tough spurious interrupt problem in some kernel driver. Fred has been running various tests in the PPC7xx compiler, using the same board as Sue because someone discovered a bug.

So Sue sits down at her desk to start a BoardFarm session. She uses ssh to connect to one of the BoardFarm testroot servers. Once there, she reserves a testroot and specifies the BSP that she is going to use. The BoardFarm creates a clean testroot, a virtual machine, and installs the selected BSP. This virtual machine is called the testroot. She logs into the testroot and starts her work.

Since the BSP is already installed and the BSP has the kernel sources and proper cross compilers, she is ready to start working on her problem. As she progresses, she compiles the kernel then tells the BoardFarm to run a portion of a testsuite which has been really good at triggering the system-crippling bug. The BoardFarm

grabs an available board, boots the board using her *new* kernel, compiles the tests in the testroot and then runs the tests on the board. She checks the results and keeps on working on her bug. She can repeat these steps as often as she wants.

Fred is working his way through the test-results that show his bug. He got the test results from an automated test that ran last night. The automated tests run every time a change happens to a major component of the BSP. Last night, it detected problems with the toolchain. Since Fred broke it, he gets to fix it.

Fred sits at his desk. Instead of requesting a testroot to work in, Fred uses his own workstation, preferring to submit test requests via the web interface. Using the test results, he figures out where the problem probably is and starts working on his bug. After an hour or two, he has what he thinks is a good rough fix. He compiles the toolchain and submits it to the BoardFarm web page to automatically test. He tells the BoardFarm to use specific testsuites that are good for testing compilers. The BoardFarm reserves the appropriate board when it's available, sets up a testroot, installs a BSP and his *new* toolchain, boots the board, compiles the tests and then runs them. When the tests finish, the BoardFarm saves the results, destroys the testroot and unreserves the board. Fred uses the results from the tests to analyze his work.

This works even if Fred and Sue are using the same board. If Fred and Sue submit requests that need the board at the same time, then the BoardFarm schedules them and runs them one after another. Either Fred or Sue's results might take a little longer, but neither has to hover around the desk of the other, waiting to snatch the board. Instead, if their jobs conflict, they can just go grab a bacon sandwich from the company vending machine while the BoardFarm does the needful.

By the end of the day, Fred feels confident that he has throughly squished the PowerPC bug. To be sure, he should really check his new and improved toolchain out on more than just the one board. So he kicks off full test suite runs, using his new toolchain, on all the PPC7xx boards and then goes home. While each test suite can take 8 hours or more, they can all run in parallel, automatically, while Fred relaxes in a lawn chair with his Bermuda shorts and a good Belgian beer, out front of his south-side Pittsburgh town home.

We'd like to take this opportunity to point out some interesting facts. Neither Fred nor Sue had to worry about booting the boards or installing the BSPs. The BoardFarm knew how to boot the boards, install the BSPs and did it without human primate intervention. This is a real time-saver as well as a great way to prevent Sue and especially Fred from getting distracted.

4 Architecture: The design, layout, blueprint, and plan

How is the BoardFarm put together? The BoardFarm is made up of four parts: a web server, a database, testroot systems, and the boards themselves. There is also some other support hardware: an SMNP-controlled power strip, a network switch, and a serial terminal server.

The web server is running Apache2 [2]. The web pages are powered by PSE [7] which uses `mod_python` [1] to get the job done. We chose this combination to allow fast development and to harness the power of Python's [6] extensive library.

The database system is powered by PostgreSQL [5]. It's otherwise just a normal humble box. The database itself isn't that radical.

It's mainly used to track who is using a board and what architecture and model a board is. We need this information to match a given BSP to one or more boards.

The testroot systems are normal boxes running Python and have a standardized OS image that is used to build a testroot. The OS image must have the components needed to compile, remotely test and debug problems on the boards. The testroot is an isolated environment that could be implemented as a chroot or UML [8] environment.

The boards themselves are all different. This is one of the hard parts of dealing with embedded developer boards. Some of them even require special hardware tricks just to power on.

All boards are powered via a network-controlled power strip. This allows us to reboot a system remotely. Usually. Some boards require a button to be pressed which requires interesting solutions ranging from relays to LEGO[3] Mindstorms[4].

A serial terminal server is needed to connect all the serial ports on the boards to the test servers. Since the serial terminal servers are networked, we can share one board among many testroot servers.

5 Show me the BoardFarm

Let's talk about the user interface. The BoardFarm has two user interfaces: a web page and a remote shell.

The web page is the easiest to use. Fred likes it because it's simple, direct, and has pretty colors. To run a test, Fred chooses the BSP, picks some tests and then clicks the "Test" button. After the BoardFarm is done, Fred gets an email with a link to the web page with all the test results.

Sue preferred the more powerful interface: the testroot command line shell. The shell is connected to via ssh and looks remarkably like a bash shell prompt, mainly because it is. Within the testroot, Sue has a series of BoardFarm shell commands.

These shell commands are very powerful. They provide control of the testroot, the BSPs, kernel, and the board itself. Using these commands, Sue can clean out the testroot, reinstall the BSP, choose a custom kernel, reserve a board, boot the board, power-cycle the board, run tests on the board, connect to the board's serial console, unreserve the board and save the test results. Furthermore, since these are normal shell commands, Sue can write custom batch scripts.

6 Successes so far

This system is actually a **fully** operational death star… It has been running now for over 6 months inside TimeSys. The BoardFarm is used on a daily basis. TimeSys and the BoardFarm are located in Pittsburgh, Pennsylvania, and we have remote users in California and even India.

We have used the BoardFarm to successfully troubleshoot problems, help develop new BSPs and to test old BSPs. The support team uses the BoardFarm to reproduce and troubleshoot customer problems. The documentation team uses the BoardFarm to confirm their manuals and to gather screen shots.

In addition to manual tasks, the automated testing is used with various TimeSys test suites to test BSPs on demand (just one click!^{TM}). In

addition, new BSPs are automatically tested as soon as they are successfully built in our build system.

It is only fair to point out that the BoardFarm is actually a part of TimeSys's in-house build environment. This integration makes manual usage easy and provides the starting point for the automated testing. As the build system successfully finishes a build, the BoardFarm queues the build to run tests when the appropriate boards are available. This makes the "change, compile, test" cycle much shorter.

7 Known Limitations

We knew from the beginning that we couldn't have everything (at least not right away). The original plan for the BoardFarm was to only provide an automated testing environment. Since then we have added the ability to do actual development using the BoardFarm. Since these goals have evolved, we have run into some limitations with our original design.

In general though, these limitations boil down to one real problem. These are developer boards, not meant to be production level devices, that at times require someone to actually go and visit the board. For example, a board might need a button pressed to power on. Or certain error situations can only be diagnosed by looking at a pattern of LEDs.

Another aspect of the "no access" problem is developing peripheral device support. To troubleshoot USB, you need to be able to plug-in and remove devices. To check that PCMCIA is working, you have to try various classes of devices. And so on.

The only other limitation isn't a technical problem, it's a social one. Developers are the ultimate power users. Most developers hate having something between them and what they're working on. Some developers appreciate the advantages of having the BoardFarm help them. Others try to work around the BoardFarm however they can. And a few of the extremists just demand that the board be handed over to them.

8 The future's so bright, we gotta wear shades

Like most projects we have grandiose plans for the future. We have plans to make the BoardFarm do test analysis, boot boards that require a button press, and integrate with our project management tools.

The BoardFarm collects all test results but it doesn't understand them. Some tests are more important than others and sometimes multiple failures can be due to just one root problem. The BoardFarm can't make this distinction. We would like the BoardFarm to help us understand our trends in failures and help recognize where we need to focus our efforts.

Some boards require a button to be pressed. Despite experimentation with electrodes and lawyers, we haven't found a sufficiently reliable solution. Plus, the lawyers were expensive. We have our crack electrical engineering team working on a solution.

The BoardFarm is just a piece of a larger system. Even though it works with the large system, it isn't feeding back information into all the systems. Ideally, we'd like it to file bugs and test things that have we have bugs open for.

9 Conclusion

In conclusion, we only have this to say: “Bacon Rocks.”

Good Night!

References

[1] Apache python module. `http://www.modpython.org/`.

[2] Apache web server. `http://httpd.apache.org/`.

[3] Lego. `http://www.lego.com/`.

[4] Lego mindstorms. `http://www.legomindstorms.com/`.

[5] Postgres sql database. `http://www.postgres.org/`.

[6] Python language. `http://www.python.org/`.

[7] Python servlet engine. `http://nick.borko.net/pse/`.

[8] User mode linux. `http://user-mode-linux.sourceforge.net/`.

The BlueZ towards a wireless world of penguins

Marcel Holtmann
BlueZ Project
marcel@holtmann.org

Abstract

The Bluetooth wireless technology is getting more and more attention. There are a lot of devices available and most of them are working perfect with Linux, because Linux has the BlueZ. This is the codename of the official Bluetooth protocol stack for Linux. It is possible to use Bluetooth for simple cable free serial connections, dialup networks, TCP/IP networks, ISDN networks, human interface devices, printing, imaging, file transfers, contact and calendar synchronization etc. All these services are designed to integrate seamlessly into existing and established parts of Linux, like the kernel TTY layer, the network subsystem, the CUPS printing architecture, the OpenOBEX library and so on.

1 Introduction

The Bluetooth technology was announced in May 1998 with the goal to create an easy usable cable replacement. Therefor it uses radio transmission within the 2.4 GHz ISM band to connect mobile devices like mobile phones, handhelds, notebooks, printer etc. from different manufactures without any cables. But Bluetooth is more than a simple cable replacement technology and with more and more devices using Bluetooth we see scenarios that are now making perfect sense. Examples for this are the communication of mobile phones with handhelds for exchanging contact and calendar information. Also the wireless printing of pictures without the interaction of a desktop computer.

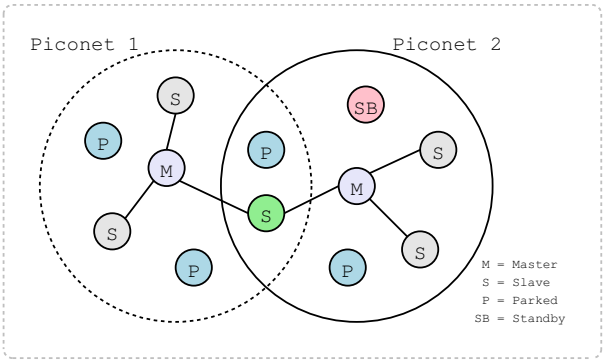


Figure 1: Bluetooth topology

Many of these applications are also possible with other technologies like IrDA or IEEE 802.11 (WiFi), but Bluetooth make the use a lot easier and defines clear application profiles.

The first steps into supporting Bluetooth with Linux are done by Axis Communications and they released their OpenBT Bluetooth Stack in April 1999. Also IBM released its BlueDrekar which was only available as binary modules. The problem of both stacks was that they are character device driven, but the Bluetooth technology is for connecting devices. So it is bet-

ter to intergrate it into the Linux network layer and to use the socket interface as primary API. On May 3, 2001, the Bluetooth protocol stack called BlueZ which was written by Qualcomm was released under GPL. This new stack followed the socket based approach. One month later it was picked up by Linus Torvalds and integrated into the Linux 2.4.6-pre2 kernel. Another Bluetooth stack for Linux was released by Nokia Research Center in Helsinki and it is called Affix. The open source community already decided to support BlueZ as official Bluetooth protocol stack Linux and it became one of the best implementations of the Bluetooth specification.

2 Bluetooth architecture

The Bluetooth architecture separates between three different core layers; hardware, host stack and applications. The hardware consists of radio, baseband and the link manager and this will be found in Bluetooth chips, dongles and notebooks. The control of the hardware is done via the host controller interface (HCI) and for the communication between the host stack and the Bluetooth hardware a hardware specific host transport driver is used. For the USB and UART transports it is possible to use general drivers, because these host transport are part of the Bluetooth specification. For PCMCIA or SDIO vendor specific driver are needed.

BlueZ implements the host stack and also the applications. The lowest layer is the logical link control and adaptation protocol (L2CAP). This protocol uses segmentation and reassembly (SAR) and protocol and service multiplexing (PSM) to abstract from the Bluetooth packet types and low-level connection links. Starting with Bluetooth 1.2 this protocol layers was extended with retransmission and flow control (RFC).

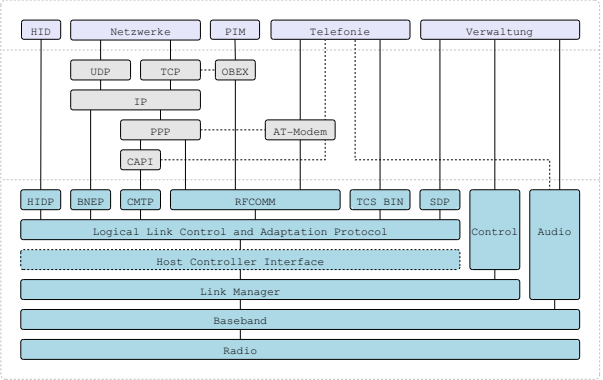


Figure 2: Bluetooth architecture

All protocol layers above L2CAP are presenting the interaction with the applications. In general it is possible to use L2CAP directly (at least with Linux), but this is not specified within the Bluetooth specification. The real Bluetooth parts of the host stack are only the L2CAP layer and the adaption layer which integrates it into other subsystems or protocol suites, like TCP/IP and OBEX.

3 Design of BlueZ

The main components of BlueZ are integrated into the Linux kernel as part of the network subsystem. It provides its own protocol family and uses the socket interface. This basic design makes it easy for application to adapt the Bluetooth technology and the integration is simple and straight forward. The use of different Bluetooth hardware is handled by the hardware abstraction inside the kernel. The BlueZ core supports the usage of 16 Bluetooth adapters at the same time. The list of supported devices is growing every day and currently over 300 different working adapters are known.

Besides the BlueZ core and the hardware abstraction also the L2CAP layer is running inside the kernel. It provides a socket interface

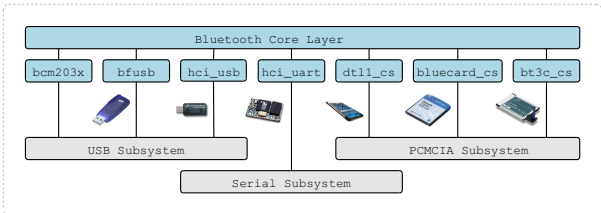


Figure 3: BlueZ core

with sequential packet characteristics and in future the RFC feature will add the stream interface.

With RFCOMM it is possible to emulate terminal devices. It is called cable replacement for legacy applications. With BlueZ it is possible to access this protocol layer from two levels. One is again a socket interface and this time with stream characteristics and the second is through the Linux TTY layer. RFCOMM emulates a full serial port and it is possible to use the point-to-point protocol (PPP) over it to create dialup or network connections.

The Bluetooth network encapsulation protocol (BNEP), the CAPI message transport protocol (CMTP) and the human interface device protocol (HIDP) are transport protocols for the network layer, the CAPI subsystem and the HID driver. Their main job is to shrink the protocol overhead and keep the latency low.

All of these protocols are implemented inside the kernel. Other Bluetooth protocols are implemented as libraries, like the service discovery protocol (SDP) or the object exchange protocol (OBEX). Some of them are also directly integrated into applications. For example the hardcopy cable replacement protocol (HCRP) and the audio video distribution transport protocol (AVDTP).

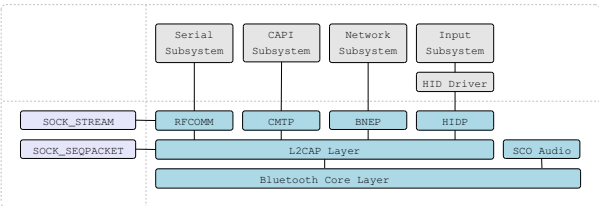


Figure 4: BlueZ protocols

Almost every Linux distribution contains a Bluetooth enabled kernel and a decent version of the BlueZ library and the BlueZ utilities.

4 Bluetooth configuration

After plugging in a Bluetooth USB dongle or inserting a Bluetooth PCMCIA card a call to `hciconfig` will show the new device.

This device is still unconfigured and like a network card it needs to be activated first. This can be done via `hciconfig hci0 up` or in the background by `hcid`.

The detailed output shows the Bluetooth device address (BD_ADDR) and additional information like name, class of device and manufacturer specific details. With `hciconfig` all of these settings can be changed.

Now it is possible to scan for other Bluetooth devices in range. For this and some other actions `hcitool` is used.

```
# hciconfig -a
hci0: Type: USB
      BD Address: 00:02:5B:01:66:F5 ACL MTU: 384:8 SCO MTU: 64:8
      UP RUNNING PSCAN ISCAN
      RX bytes:3217853 acl:79756 sco:0 events:199989 errors:0
      TX bytes:77188889 acl:294284 sco:0 commands:206 errors:0
      Features: 0xff 0xff 0x8f 0xfe 0x9b 0xf9 0x00 0x80
      Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
      Link policy: RSWITCH HOLD SNIFF PARK
      Link mode: SLAVE ACCEPT
      Name: 'Casira BlueCore4 module'
      Class: 0x3e0100
      Service Classes: Networking, Rendering, Capturing
      Device Class: Computer, Laptop
      HCI Ver: 2.0 (0x3) HCI Rev: 0x77b LMP Ver: 2.0 (0x3)
        LMP Subver: 0x77b
      Manufacturer: Cambridge Silicon Radio (10)
```

Figure 5: Local Bluetooth adapter

```
# hcitool scan
Scanning ...
  00:80:37:25:55:96 Pico Plug
  00:E0:03:04:6D:36 Nokia 6210
  00:90:02:63:E0:83 Bluetooth Printer
  00:06:C6:C4:08:27 Anycom LAP  00:06:C6:C4:08:27
  00:04:0E:21:06:FD Bluetooth ISDN Access Point
  00:0A:95:98:37:18 Apple Wireless Keyboard
  00:A0:57:AD:22:0F ELSA Vianect Blue ISDN
  00:80:37:06:78:92 Ericsson T39m
  00:01:EC:3A:45:86 HBH-10
```

Figure 6: Scanning for Bluetooth devices

After the scan the program `sdptool` can be used to retrieve the available services of a remote Bluetooth device. The service list identifies the supported Bluetooth profiles and reveals protocol specific information that are used by other tools. The example shows a dialup networking service and a fax service which both are using the RFCOMM channel 1.

Some tools have integrated SDP browsing support and will determine the needed service information by themself. Others don't have this capability, because it is not always useful. Every SDP request involves the creation of a piconet and this can fail or timeout. So for all Bluetooth tools running at boot time this is not a desired behavior.

With the Bluetooth device address and the channel number it is possible to setup a RFCOMM TTY terminal connection for using AT commands or PPP for Internet access. The command `rfcomm bind 0 00:E0:03:04:6D:36 1` creates the device `/dev/rfcomm0` which is connected to the RFCOMM channel 1 on the mobile phone with the Bluetooth address `00:E0:03:04:6D:36`. The connection itself is not created by this command. It will first established when an application, like `pppd`, opens this device node and terminated when the last user closes it.

```
# sdptool browse 00:E0:03:04:6D:36
Browsing 00:E0:03:04:6D:36 ...
Service Name: Dial-up networking
Service RecHandle: 0x10000
Service Class ID List:
  "Dialup Networking" (0x1103)
  "Generic Networking" (0x1201)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 1
Profile Descriptor List:
  "Dialup Networking" (0x1103)
    Version: 0x0100

Service Name: Fax
Service RecHandle: 0x10001
Service Class ID List:
  "Fax" (0x1111)
  "Generic Telephony" (0x1204)
Protocol Descriptor List:
  "L2CAP" (0x0100)
  "RFCOMM" (0x0003)
    Channel: 1
Profile Descriptor List:
  "Fax" (0x1111)
    Version: 0x0100
```

Figure 7: Requesting service information

5 Bluetooth networks

For creating network connection over Bluetooth the preferable method is using a personal area network (PAN) with BNEP. The old method was called LAN access using PPP and it used PPP over RFCOMM. This was a bad decision for the performance and now this profile is deprecated. A PAN connection can be created with the command `pand --connect 00:06:C6:C4:08:27` and after a successful connect `ifconfig` will show a `bnep0` device with the same MAC address as the BD_ADDR of the remote device.

This network device is a virtual network card,

```
# ifconfig -a
bnep0 Link encap:Ethernet  HWaddr 00:06:C6:C4:08:27
      BROADCAST MULTICAST MTU:1500 Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:100
      RX bytes:4 (4.0 b) TX bytes:0 (0.0 b)
```

Figure 8: Bluetooth network device

but not limited in its functionality. It is possible to use all Ethernet related commands on it and besides IPv4 and IPv6 it can also used inside an IPX network. Even methods like bridging or network address translation (NAT) are working without any problems.

Another possibility to create network connections is using ISDN and the CAPI subsystem. The Bluetooth part is called common ISDN profile (CIP) and it uses CMTP. Once a connection to an ISDN access point with `ciptool connect 00:04:0E:21:06:FD` has been created, a virtual ISDN card will be presented by the CAPI subsystem and the standard tools can be used.

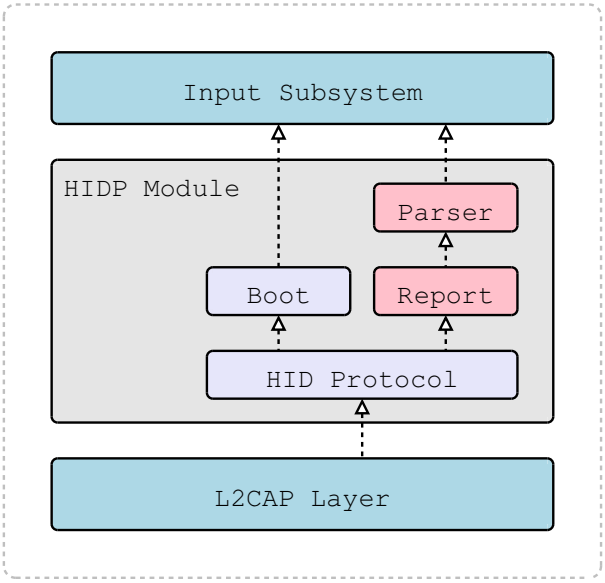


Figure 9: HID architecture

6 Printing over Bluetooth

For accessing printers over Bluetooth it is possible to do this via RFCOMM or HCRP. The Bluetooth CUPS backend supports both methods and is able to choose the best one by itself. The setup of a Bluetooth printer is very easy. The only thing that it is needed is an URI and this is created from its BD_ADDR by removing the colons. For accessing a printer with the Bluetooth device address `00:90:02:63:E0:83` the URI `bluetooth://00900263E083/` should be given to CUPS.

7 Bluetooth input devices

With the human interface device specification for Bluetooth it is also possible to use wireless mice and keyboards. Since HID devices can disconnect and reconnect at any time it is necessary to run `hidd --server` to handle such events. To bind a mouse or keyboard to your system it is only needed to contact it once. This initial connection can be done with the command `hidd --connect 00:0A:95:98:37:18`. All further connects are initiated by the device.

8 Bluetooth audio

The Bluetooth technology can be used for data communication, but it also support audio connections. For example headsets for voice connection and headphones for high-quality stereo transmission. For both device types an integration into the ALSA sound system is planned. Like all other subsystem or library integrations done by BlueZ so far, this will be almost invisible for the end user. First beta versions of the ALSA plugins exists by now and a final version is expected very soon.

9 Other applications

Many more applications started to integrate native Bluetooth support. The popular examples are the contact and calendar synchronization program MultiSync and the Gnokii tool for accessing Nokia mobile phones. Both programs can also use IrDA or cable connections as transport and Bluetooth is only another access method. In most programs the lines of Bluetooth specific code are very small, but the mobility increases a lot.

10 Conclusion

Since 2001 a lot of things have been improved and the current Bluetooth subsystem is ready for every day usage. But the development is not finished and the end user experience can be still be improved. The GNOME Bluetooth subsystem and the KDE Bluetooth framework are two projects to integrate Bluetooth into the desktop.

With Bluetooth the need of cables is decreasing and BlueZ tries to paint the Linux world blue.

References

[1] Special Interest Group Bluetooth: *Bluetooth Core Specification v1.2*, 2003.

[2] Special Interest Group Bluetooth: *Bluetooth Network Encapsulation Protocol Specification*, 2003.

[3] Special Interest Group Bluetooth: *Common ISDN Access Profile*, 2002.

[4] Special Interest Group Bluetooth: *Human Interface Device Profile*, 2003.

[5] Infrared Data Association (IrDA): *Object Exchange Protocol OBEX, Version 1.3*, 2003.

On faster application startup times: Cache stuffing, seek profiling, adaptive preloading

bert hubert
Netherlabs Computer Consulting BV
bert.hubert@netherlabs.nl

Abstract

This paper presents data on current application start-up pessimizations (on-demand loading), relevant numbers on real-life harddisk seek times in a running system (measured from within the kernel), and shows and demonstrates possible improvements, both from userspace and in the kernel. On a side note, changes to the GNU linker are discussed which might help. Very preliminary experiments have already shown a four-fold speedup in starting Firefox from a cold cache.

1 1980s and 1990s mindset

The cycle of implementations means that things that were slow in the past are fast now, but that things that haven't gotten any faster are perceived as slow, and relatively speaking, are.

CPUs used to be slow and RAM was generally fast. These days, a lot of CPU engineering goes into making sure we do not mostly wait on memory.

Something like this has happened with hard disks. In the late 1980s, early 90s, there was a lot of attention for seek times, which was understandable as these were in the order of 70ms. These have been reduced, but not as much as disk throughput has increased. Typical measured seek times on laptop hard disks are still in the 15-20ms region, while 20 megabytes/second disk speeds would allow the disks of old to be read in their entirety in under 10 seconds.

2 Some theory

To retrieve data from disk, four things must happen:

1. The instruction must be passed to the drive
2. The drive positions its reading head to the proper position
3. We wait until the proper data passes under the disk
4. The drive passes the data back to the computer

It is natural to assume that seeking to locations close to the current location of the head is faster, which in fact is true. For example, current Fujitsu MASxxxx technology drives specify the 'full stroke' seek as 8ms and track-to-track latency as 0.3ms.

However, for many systems the actual seeking is dwarfed by the rotational latency. On average, the head will have to wait half a rotation for the desired data to pass by. A quick calculation shows that for a 5400RPM disk, as commonly found in laptops, this wait will on average be 5.6ms.

This means that even seeking a small amount will at least take 5.6ms.

The news gets worse—the laptop this article is authored on has a Toshiba MK8025GAS disk, which at 4200RPM claims to have an average seek time of 12ms. Real life measurements show this to be in excess of 20ms.

3 What this means, what Linux does

That one should avoid seeking by all means. Given a 20ms latency penalty, it is cheaper to read up to 5 megabytes speculatively to get to the desired location.

In Linux, on application startup, the relevant parts of binaries and libraries get mmapped into memory, and the CPU starts executing the program. As the instructions are not loaded into memory as such, the kernel encounters page faults when data is missing, leading to disk reads to fill the memory with the executable code.

While highly elegant, this leads to unpredictable seek behaviour, with occasional hits going *backward* on disk. The author has discovered that if there is one thing that disks don't do well, it is reading backwards.

Short of providing a 'reverse' setting to the disk's engine, the onus is on the computer to optimize this away.

4 How to measure, how to convert

As binary loading is "automatic," userspace has a hard time seeing page faults. However, the recently implemented 'laptop mode' not only saves batteries, it also allows for logging of actual disk accesses.

At the level of logging involved, only PID, device id, and sector are known, which is understandable as the logging infrastructure of laptop mode is mostly geared towards figuring out which process is keeping the disk from spinning down.

Typical output is:

```
bash(261): READ block 11916
  on hda1
bash(261): READ block 11536
  on hda1
bash(261): dirtied inode 737
  (joe) on hda1
bash(261): dirtied inode 915
  (ld-linux.so.2) on hda1
```

Short of dragging around a lot more infrastructure than is desireable, the kernel is in no position to help us figure out which files correspond to these blocks.

Furthermore, there is no reverse map in any sane fs to tell us which block belongs to which file.

Luckily, another recent development comes to our rescue: syscall auditing. This can be thought of as a global strace, allowing designated system calls to be logged, whatever their origin. This generates a list of files which might have caused the accesses.

This combined with the forward logical mapping facility used by lilo to determine the sector

locations of files allows us to construct a partial reverse map that should include all block accesses logged by the kernel.

From this we gather information which parts of which positions in whic files will be accessed or system or application startup.

5 Naïvely using the gathered information

Using the process above, a program was written which gathers the data above for a typical Debian Sid startup, up to and including the launch of Firefox. On next startup, a huge shell script used 'dd' to read in all relevant blocks, sequentially. Even without merging nearby reads, or or utilizing knowledge of actual disk layout, this sped up system boot measureably. Most noticeable was the factor of four improvement in startup times of Firefox.

In this process a few things have become clear:

- There are a lot of reads which cannot be connected to a file
- The 'dd' read script is very inefficient
- The kernel has its own ideas on cache maintenance and throws out part of the data
- Reads are spread over a large number of files

The reads which cannot be explained are in all likelihood part of either directory information or filesystem internals. These are of such quantity that directory traversal appears to be a major part of startup disk accesses.

It is interesting to note that only in the order of 40 megabytes of disk is touched on booting, leading to the tentative conclusion that all disk access could conceivably be completed in 2 seconds or less.

However, it is also clear that reads are spread over a large number of files, making naïve applications of `readahead(2)` less effective.

6 More sophisticated ways of benefiting from known disk access patterns

Compiler, assmbler and linker work together in laying out the code of a program. Andi Kleen has suggested storing in an ELF header which blocks are typically read during startup, allowing the dynamic linker to touch these blocks sequentially. Howver, this idea is not entirely relevant anymore as most time is spent touching libraries, which will have differing access patterns for each file program using them.

Linus Torvalds has suggested that the only way of being really sure is to stuff the page cache with a copy of the data we know that is needed, and that we store that data in a sequential slab on disk so as to absolutely prevent having to seek.

The really dangerous bit is that we need to be very sure our sequential slab is still up to date. It also does not address the dentry cache, which appears to be a dominant factor in bootup.

Another less intrusive solution is to use a syscall auditing daemon to discover which application is being started and touch the pages that were read last time this binary was being started. During bootup this daemon might get especially smart and actually touch pages that it knows will be read a few seconds from now. The hard time is keeping this in sync.

7 Conclusions

Currently a lot of time is wasted during application and system startup. Actual numbers appear to indicate that the true amount of data read during startup is minimal, but spread over a huge number of files.

The kernel provides some infrastructure which, through convoluted ways, can help determine seek patterns that userspace might employ to optimize itself.

A proper solution will address both directory entry reads as well as bulk data reads.

Building Linux Software with Conary

Michael K. Johnson *rpath, Inc.*
ols2005@rpath.com

Abstract

This paper describes best practices in Conary packaging: writing recipes that take advantage of Conary features; avoiding redundancy with recipe inheritance and design; implementing release management using branches, shadows, labels, redirects, and flavors; and designing and writing dynamic tag handlers. It describes how Conary policy prevents common packaging errors. It provides examples from our rpath Linux distribution, illustrating the design principles of the Conary build process. It then describes the steps needed to create a new distribution based on the rpath Linux distribution, using the distributed branch and shadow features of Conary.

Conary is a distributed software management system for Linux distributions. Based on extensive experience developing Linux distributions and package management tools, it replaces traditional package management solutions (such as RPM and dpkg) with one designed to enable loose collaboration across the Internet. It enables sets of distributed and loosely connected repositories to define the components which are installed on a Linux system. Rather than having a full distribution come from a single vendor, it allows administrators and developers to branch a distribution, keeping the pieces which fit their environment while grabbing components from other repositories across the Internet.

If you do not have a basic working knowledge of Conary terminology and design, you may want to read the paper *Repository-Based System Management Using Conary,* in *Proceedings of the Linux Symposium, Volume Two, 2004*, kept updated at `http://www.rpath.com/technology/techoverview/`, which introduces Conary's design and vocabulary in greater detail. Terms called out in **boldface** in this paper without an explicit definition are defined in that overview.

1 Conary Source Management

Unlike legacy package management tools, Conary has integral management for source code and binaries, and the binaries are directly associated with the source code from which they have been built.

Conary stores source files in **source components**, and then uses a **recipe** (described later) to build **binary components** that it can install on a system. While most of the code that handles the two kinds of components is actually the same, the interface is different. The source components are managed using a Software Configuration Management (SCM) model, and the binary components are managed using a system management model.

The SCM model for managing source components is sufficiently familiar to experienced Concurrent Versioning System (cvs) or Subversion (svn) users; you can create a new source

component, check out an existing source component, add files to a source component, remove files from a source component (this is a single action in Conary, unlike cvs's two-step operation), rename files in a source component (like svn, but unlike cvs), and commit the current set of changes since the last commit (like svn, and like cvs except that the commit is atomic).

Conary has not been optimized as a complete SCM system. For example, we do not use it to manage subdirectories within a source component, and instead of importing source code into vendor branches, we import archives and apply patches to them. Conary may someday support a richer SCM model, since there are no significant structural or design barriers within Conary. It was a choice made for the sake of simplicity and convenience, and to focus attention on Conary as a system management tool rather than as an SCM tool—to encourage using Conary to track upstream development, rather than encourage using it to create forks.

In addition, Conary stores in the repository (without cluttering the directory in which the other files are stored) all files that are referenced by URL. When they are needed, they are downloaded (from the repository if they have been added to the repository; otherwise, via the URL) and stored in a separate directory. Then, when committing, Conary stores those automatically added source files (**auto-source files**) in the repository so that the exact same source is always available, enabling repeatable builds.

Like cvs, Conary can create branches to support divergent development (forks); unlike cvs, those branches can span repositories, and the repository being branched from is not modified, so the user only needs write privileges in the repository in which the branch is created. Unlike cvs (and any other SCM we are aware of), Conary also has two features that support converging source code bases: **shadows** that act like branches but support convergent instead of divergent development by providing intentional tracking semantics, and **redirects** that allow redirecting from the current head of the branch (or shadow) to any other branch (or shadow), including but not limited to that branch's parent. The redirect is not necessarily permanent; the branch can be revived. A redirect can even point to a different name package entirely, and so is useful when upstream names change, or when obsoleting one package in favor of another.

1.1 Cooking with Conary

Using a recipe to turn source into binary is called **cooking**.

The exact output produced by building source code into a binary is defined by several factors, among them the instruction set (or sets) that the compiler emits, and the set of features selected to be built. Conary encodes each combination of configuration and instruction set as a **flavor**. The configuration items can be system-wide or package-local. When cooking, Conary builds a **changeset** file that represents the entire contents of the cooked package.

There are three ways to cook:

- A **local** cook builds a changeset on the special `local@local:COOK` branch. It loads the recipe from the local filesystem, and can cook with recipes and sources that are not checked into the repository. It will download any automatic sources required to build.
- A **repository** cook builds a transient changeset on the same branch as the source component, and then commits it to

the repository. It loads the source component (including the recipe and all sources) from the repository, not the local filesystem. It finds all automatic sources in the repository. The same version can be built into the repository multiple times with different flavors, allowing users to receive the build that best matches their system flavor when they request a trove.

- An **emerge** builds a transient changeset on the special `local@local:EMERGE` branch, and then commits it to (that is, installs it on) the local system. Like a repository cook, it takes the recipe and all sources from the repository, not the filesystem. (This is the only kind of cook that Conary allows to be done as the root user.)

2 The Conary Recipe

All software built in Conary is controlled through a **recipe**, which is essentially a Python module with several characteristics. Here is an example recipe, which has a typical complexity level:[1]

```
class MyProgram(PackageRecipe):
  name = 'myprogram'
  version = '1.0'
  def setup(r):
    r.addArchive(
      'http://example.com/%(name)s-%(version)s.tar.gz')
    r.Configure()
    r.Make()
    r.MakeInstall()
```

The goal of Conary's recipe structure is not to make all packaging trivial, but to make it possible to write readable and maintainable complex recipes where necessary, while still keeping the great majority of recipes extremely simple—and above all, avoiding boilerplate that needs to be copied from recipe to recipe. This example is truly representative of the the most common class of recipes; the great majority of packaging tasks do not require any further knowledge of how recipes work. In other words, this example is representative, not simplistic. New packagers tend to find it easy to learn to write new Conary packages.

However, some programs are not designed for such easy packaging, and many packagers have become used to the extreme complexity required by some common packaging systems. This experience can lead to writing needlessly complex and thereby hard-to-maintain recipes. This, in turn, means that while reading the RPM spec file or Debian rules for building a package can be an easy way to find a resolution to a general packaging problem when you are writing a Conary recipe, trying to translate either of them word-by-word is likely to lead to a poor Conary package.

The internal structure of objects that underlie Conary recipes makes them scale gracefully from simple recipes (as in the example) to complex ones (the kernel recipe includes several independent Python classes that make it easier to manage the kernel configuration process). Some of the more complex recipe possibilities require a deeper structural understanding.

- The recipe module contains a class that is instantiated as the **recipe object** (`MyProgram` in the example above). This class declares, as class data, a `name` string that matches the name of the module, a `version` string, and a `setup()` method. This class must be a subclass of one of a small family of abstract superclasses (such as `PackageRecipe`).
- Conary calls the recipe object's `setup()` method, which populates lists of things to do; each to-do item is represented by

[1] No, I do not like two-column formatting for technical papers, either.

an object. There are **source objects**, which represent adding an archive, patch, or source file; and **build objects**, which represent actions to take while building and installing the software. Additionally, there are pre-existing lists of **policy objects** to which you can pass extra information telling them how to change from their default actions. The `setup()` function returns after preparing the lists, before any build actions take place.

- Conary then processes the lists of things to do; first all the source objects, then all the build objects, and finally all the policy objects.

It is important to keep in mind that unlike RPM spec files and portage ebuild scripts (processed in read-eval-print loop style by a shell process) or Debian rules (processed by make), a Conary recipe is processed in two passes (three, if you count Python compiling the source into bytecode), because it both constrains the actions you can or should take and makes Conary more powerful. For example, you should not add sources inside a Python conditional (instead, you unconditionally add them but can choose not to apply them based on a conditional), but this constraint allows Conary to always automatically store copies of all sources that it has fetched by URL instead of being explicitly committed locally.

Another important data structure in a recipe is the macros object, an enhanced dictionary object that is an implicit part of every recipe. Almost every string used by any of the different kinds of objects in the recipe—including the strings stored in the macros object itself—is automatically evaluated relative to the contents of the macros object, meaning that standard Python string substitution is done. Thus, you do not have to type `%r.macros` after every string; the substitution is done within the functions you call. It also means that macros can reference each other. Be aware that changes to the macros object all take place before any list processing. This means that an assignment or change to the macros object at the end of the recipe will affect the use of the macros object at the beginning of the recipe. This is an initially non-obvious result of the multi-pass recipe processing.

The string items contained in the macros object are colloquially referred to by the Python syntax for interpolating dictionary items into a string. Thus, `r.macros.foo` is usually referred to as `%(foo)s`, because that is the way you normally see it used in a recipe.

The macros object contains a lot of immediately useful information, including the build directory (`%(builddir)s`), the destination directory (`%(destdir)s`) that is the proxy for the root directory (`/`) when the software is installed, many system paths (`%(sysconfdir)s` for `/etc` and `%(bindir)s` for `/usr/bin`), program names (`%(cc)s`), and arguments (`%(cflags)s`).

3 Recipe Inheritance and Reference

Conary recipes can reference each other, which makes it easier to use them to create a coherent system.

When many packages are similar, it is easy to end up with boilerplate text that is copied between packages to make the result of cooking them reflect that similarity. That boilerplate can be encoded in `PackageRecipe` subclasses, stored in recipes that are normally never cooked because they function as abstract superclasses. The recipes containing

those abstract superclasses are loaded with the `loadSuperClass()` function, which loads the latest version of the specified recipe from the repository into the current module's namespace. The main class in the recipe then descends from that abstract superclass. (The inheritance is pure Python, so it is possible to use multiple inheritance if that is useful.) This mechanism serves two purposes: it reduces transcription errors in what would otherwise be boilerplate text, and it reduces the effort required to build similar packages. It also allows bug fixes that are generic to be made in the superclass and thus automatically apply to all the subclasses.

Sometimes, you want to reference a recipe without inheriting from it. In that case, you use a similar function called `loadInstalled()`, which loads a recipe while preferring the version that is installed on your system, if any version is installed on your system. (Otherwise, it acts just like `loadSuperClass()`.) For example, you can load the perl recipe in order to programmatically determine the version of perl included in the distribution, without actually requiring that perl even be installed on the system.

4 Dynamic Tag Handlers

Conary takes a radically different approach to install-time scripts than legacy package management tools do. Typical install-time scripts are package-oriented instead of file-oriented, primarily composed of boilerplate, and often clash with rollback (for those package management tools that even try to provide rollback functionality). Conary tags individual files, instead; this file is a shared library, that file is an init script, another file is an X font. Then, at install time, once for each transaction (which may be many troves all together), it calls **tag handler** scripts to do whatever is required with the tagged files.

Bugs in tag handlers demonstrate some of the good characteristics of this system. When a bug in a tag handler is fixed, that bug is fixed for all packages in one place, without any need to copy code or data around. It is fixed even for older versions of packages built before the tag handler was fixed (as long as the package in question is not the one that implements the tag handler). Also, tag handlers can be called based not only on changes to the tagged files, but also changes to the tag handler itself, including bug fixes.

While tag handlers are clearly an improvement over legacy install-time scripts, it is still possible to make many of the same mistakes.

Overuse It is always slower to run a script than to package the results of running a script. Therefore, if you can perform the action at packaging time instead, do so. For example, put files in the `/etc/cron.d/` directory instead of calling the `crontab` program or editing the `/etc/crontab` file.

Excessive dependencies The more programs a script calls, the more complex the dependency set needed to run it. Circular dependencies are worse; they will break, one way or another. Finally, calling more programs increases the risk of accidental circular dependencies.

Inappropriate changes Modifying file data or metadata that is under package management and storing backup copies of files are generally inappropriate for any package manager, not just Conary. With Conary, though, a few more things are inappropriate, including adding users and groups to the system (it is too late; the files have already been created).

Poor error handling Scripts that do not check for error return codes can easily wreak havoc by assuming that previous actions succeeded. (We have discovered that if you are having difficulty managing error handling in a tag handler, you may be taking an approach that is needlessly complex. Look for a simpler way to solve the problem.)

It is also easy to misapply assumptions developed while writing legacy install-time scripts to tag handler scripts. The most important thing to remember is that everything in Conary, including tag handlers, is driven by changes. Therefore, if a package includes five files with the `foo` tag, and none of those tagged files is affected by an update, the `foo` tag handler will simply not be called. If only two of the five files is modified in the update, the `foo` tag handler will be called, but asked to operate only on the two modified files.

Write tag handlers with rollbacks in mind. This means that if the user does the inverse operation in Conary, the effect of the tag handler should be inverted as well. Most post-installation tasks merely involve updating caches, and often the list of affected files is not even required in order to update the cache. These cases are easy; just run the program which regenerates the cache.

When inventing new tag names, keep the tag mechanism in mind. `mypackage-script` is a horrible name for a tag handler, because it initiates or perpetuates the wrong idea about what it is and how it works. The name of a tag handler should describe the files so tagged. The sentence "*file* is a(n) *tag name*" should sound sensible, as in "`/lib/libc-2.3.2.so` is a `shlib`" or "`/usr/share/info/gawk.info.gz` is an `info-file`". Following this rule carefully has helped produce clean, simple, fast, relatively bug-free tag handlers.

5 Policy

After unpacking sources, building binaries, and installing into the `%(destdir)s`, Conary invokes an extensive set of policy objects to normalize file names, contents, locations, and semantics, and to enforce inter-package and intra-package consistency.

Policy objects are invoked serially and have access to the `%(destdir)s` as well as the `%(builddir)s`. Early policy objects can modify the `%(destdir)s`. After all `%(destdir)s` modification is finished, Conary creates a set of objects that represents the set of files in the destdir. Later policy then has that information available and can modify the packaging, including marking configuration files, marking tagged files, setting up dependencies, and setting file ownership and permissions.

The `policy.Policy` abstract superclass implements much of the mechanism that policies need. Many policies can simply list some regular expressions that specify (positively and negatively) the files to which they apply by default, and then implement a single method which is called for each matching file. The superclass provides a rich generic exception mechanism (also based on regular expressions) and a file tree walker that honors all the regular expressions. Policies are not required to use that superstructure; they can implement their own actions wherever necessary.

Policies can take additional information, as well as exceptions. For example, policy divides the files in a package into components automatically, but when the automatic assignment makes a bad choice, you can pass additional information to the `ComponentSpec` policy to change its behavior. Whenever you want to create multiple packages from a single recipe,

you have to give the `PackageSpec` policy information on which files belong to what package. When you use non-root users and groups, you need to provide user information using the `User` policy and group information using the `Group` policy.

This object-oriented approach is fundamentally different from RPM policy scripts. In contrast to Conary policy, RPM policy scripts are shell scripts (or are driven by shell scripts), and have an all-or-nothing model. If they do not do exactly what you want, you have to disable them and do by hand every action that the scripts would have done. This means that if you do not keep up with changes in the RPM policy scripts, your re-implementation may not keep up with changes in RPM. Also, because this restriction is onerous, RPM policy scripts cannot be very strict or very useful; they have to be limited in power and scope to the least common denominator. By contrast, the rich exception mechanism in Conary policy allows the policy to be quite strict by default, allowing for explicit exceptions where appropriate. This allows Conary to enable a rich array of tests that enable packaging quality assurance, with the tests run before any cooked troves are committed to the repository or even placed in a changeset.

Policies have access to the recipe object, mainly for the macros and a cache of content type identification that works like an object-oriented version of the `file` program's magic database and is therefore called the **magic cache**. This makes it easy to predicate policy action on ELF files, ar files, gzip files, bzip2 files, and so forth. The magic objects (one per file) sometimes contain information extracted from the files, such as ELF sonames and compression levels in gzip and bzip2 files.

Some policy, such as packaging policy, is intended to remain a part of the Conary program per se. However, many policies will eventually be defined (or expanded) outside of Conary, by the distribution, via **pluggable policy**. This will be a set of modules loaded from the filesystem, probably with one policy object per module, and a system to ensure that ordering constraints are honored.

5.0.1 Policy Examples

One of the best ways to describe what policy can do is to describe some examples of what it does do. As of this writing, there are 58 policy modules, so these few examples are by no means exhaustive; they are merely illustrative.

The `FilesInMandir` policy is very simple, and demonstrates how one line of code and several lines of data can implement effective policy. It looks only in `%(mandir)s`, `%(x11prefix)s/man`, and `%(krbprefix)s/man`, does not recurse through subdirectories, and within those directories only considers file entries, not subdirectory entries. Any recipe that needs to actually store a file in one of those directories can run `r.FilesInMandir(exceptions='%(mandir)s/somefile')` to cause the policy to ignore that file. All of this action specified so far requires only three simple data elements to be initialized as class data in the policy. Then a single-line method reports an error if the policy applies to any files, automatically ignoring any exceptions that have been applied from the recipe.

This simple policy effectively catches a common disagreement about what the `--mandir` configure argument or the `MANDIR` make variable specifies; the autotools de-facto standard is `/usr/share/man/` but some upstream packages set it instead to be a subdirectory thereof, such as `/usr/share/man/man1`, which would cause all the man pages to go in the wrong place by default. Having this policy

to catch errors makes it feasible for Conary to set `--mandir` and `MANDIR` by default, and fix up the exceptional cases when they occur.

The `RemoveNonPackageFiles` policy modifies the `%(destdir)s` and is even simpler in implementation than `FilesInMandir`. It lists as class data a set of regular expressions defining files that (by default) should not be included in the package, such as .cvsignore files, .orig files, libtool .la files, and so forth. It then has a single-line method that removes whatever file it is applied to. Again like `FilesInMandir`, a simple exception overrides the defaults; a recipe for a package that actually requires the libtool .la files (there are a few) can avoid having them removed by calling `RemoveNonPackageFiles(exceptions=r'\.la$')`—note the leading `r`, which tells Python that this is a "raw" string and that it should not interpret any `\` characters in the string.

Both `RemoveNonPackageFiles` and `FilesInMandir` use the built-in directory walking capabilities of the `Policy` object. Most policy does, but it is not required. The `NormalizeManPages` policy is different. It implements its own walking over each man directory (the same ones that the `FilesInMandir` policy looks at), and removes any accidental references to `%(destdir)s` in the man pages (a common mistake), makes maximal use of symlinks, makes sure that all man pages are compressed with maximal compression, and then makes sure that all symlinks point to the compressed man pages. It uses almost none of the built-in policy mechanism; it merely asks to be called at the right time.

The `NormalizeCompression` policy ignores man pages and info pages, since they have their compression normalized intrinsically via other policy. It automatically includes all files that end in `.gz` and `.bz2`. Then, for each file, it looks in the magic cache (which is self-priming; if no entry exists, it will create one), and if it really is a gzip or bzip2 file and is not maximally compressed, it recompresses the file with maximal compression.

Finally, the `DanglingSymlinks` policy uses packaging information, looking at which component each file is assigned to. It is not enough to test whether all symlinks in a package resolve; it is also important to know whether a symlink resolves to a different component, since components can be installed separately. There are also special symlinks that are allowed to point outside the package, including conshelper symlinks (which create an automatic requirement for the `usermode:runtime` component) and symlinks into the /proc filesystem. The `DanglingSymlinks` policy simply warns about symlinks that point from one component into another component built from the same package (except for shared libraries, where `:devel` components are expected to provide symlinks that cross components); symlinks that are not resolved within the package and are not explicitly provided as exceptions cause the cook to fail.

6 Recipe Writing Best Practices

Disclaimer: Not all recipes written by rpath follow these best practices. We learned many of these best practices by making mistakes, and have not systematically cleaned up every recipe. So the first rule is probably not to worry about mistakes; Conary is pretty forgiving as a packaging system, and we have tried to make it fix mistakes and warn about mistakes. You absolutely do not need to memorize this list to be a Conary packager. It's quite possible that the majority of new Conary recipes are fewer than ten lines of code. Relax, everything is going to be all right!

The best practices are somewhat arbitrarily divided into general packaging policy suggestions, conventions affecting building sources into binaries, and conventions affecting the `%(destdir)s`.

6.1 General Packaging Policy

Before worrying about packaging details, start working on consistency at a higher level. Names of packages, structure of recipes, and versions are best kept consistent within a repository, and between repositories.

6.1.1 Simple Python Modules

Starting simple: recipes are Python modules. Follow Python standards as a general rule. In particular, do not use any tabs for indentation. Tabs work fine, but you will be running the `cvc diff` command many times, and tabs make diff output look a little odd because the indentation levels are not all even, and when you mix leading tabs and leading spaces, the output looks even weirder.

Second, follow Conary standard practice. Conary standard practice has one significant difference from Python standard practice: the self-referential object is called `r` (for recipe) instead of `self` because it is used on practically every line.

To get the most benefit from Conary, write your recipes to make it easy to maintain a unified patch that modifies them. That way, someone who wants to shadow your recipe to make (and maintain) a few small changes will not be stymied. The most basic way to do this is to keep your recipe as simple as possible. Don't do anything unnecessary. Don't do work that you can count on policy to do for you, such as recompressing gzip or bzip2 files with maximal compression turned on, or moving files from `/etc/rc.d/init.d` to `%(initdir)s`, unless policy can't do the job quite the right way—and in that case, add a comment explaining why, so that the person shadowing your recipe does not walk into a trap.

Not doing lots of make-work has another important benefit. The less you do in the recipe, the less likely you are to clash with future additions to policy, and the more likely you are to benefit from those additions. Policy will continue to grow to solve packaging problems as we continue to find ways to reduce packaging problems to general cases that have solutions which we can reliably (partially or completely) automate.

6.1.2 Follow Upstream

Whenever possible, follow upstream conventions. Use the upstream name, favoring lower case if upstream sometimes capitalizes and sometimes does not, and converting "`-`" to "_" because "`-`" is reserved as a separator character. Do not add version numbers to the name; use version numbers in the name only if the upstream project indisputably uses the version number in the name of the project. Conary can handle multiple versions simultaneously just fine by using branches; no need to introduce numbers into the name. The branches can be created quite arbitrarily; they do not need to be in strict version order. Just avoid clashing with the label used for release stages by choosing a very package-specific tag. Example tags rpath has used so far are `sqlite2`, `gnome14`, and `cyrus-sasl1`. The head of `conary.rpath.com@rpl:devel` for sqlite is sqlite version 3, and the branch `conary.rpath.com@rpl:devel/2.8.15-1/sqlite2/` contains sqlite version 2, as of this writing version 2.8.16, giving a full version string

of `/conary.rpath.com@rpl:devel/2.8.15-1/sqlite2/2.8.16-1`

When possible and reasonable, one upstream package should produce one Conary package. Sometimes, usually to manage dependencies (such as splitting a text-mode version of a program from a graphical version so that the text-mode version can be installed on a system without graphical libraries installed) or installed features (such as splitting client from server programs), it is reasonable to have one upstream package produce more than one Conary package. Rarely, it is appropriate for two upstream packages to be combined; this is generally true only when the build instructions for a single package require multiple archives to be combined to complete the build, and all the archive files really are notionally the same project; they aren't just a set of dependencies.

If you have to convert "`-`" to "`_`" in the name, the following convention may be helpful:

```
r.macros.ver = \
  r.version.replace('_', '-')
r.mainDir('%(name)s-%(ver)s')
```

6.1.3 Redirects

Finally, if the upstream name changes, change the name of the package as well. This means creating a new package with the new name, and then changing the old package into a redirect that points to the new package. Users who update the package using the old name will automatically be updated to the new package.

Alternatively, if you change the package that provides the same functionality, you can do the exact same thing; from Conary's point of view there is no difference. For example, rpath Linux used to use the old mailx program to provide `/usr/bin/mail`, but switched to the newer nail program for that task. The mailx recipe was then changed to create a redirect to the newer nail package. Anyone updating the mailx package automatically got the nail package instead.

A redirect is not necessarily forever. The old recipe for mailx could be restored and a new version cooked; the nail recipe could even be changed to a redirect back to mailx. If that happened, an update from the older version of mailx would completely ignore the temporary appearance of the redirect to nail. (Not that this is likely to happen—it just would not cause a problem if it did.)

Redirects pointing from the old troves to the new troves solve the "obsoletes wars" that show up with RPM packages. In the RPM universe, two packages can each say that they obsolete each other. In the Conary world, because redirects point to specific versions and never just to a branch or shadow, this disagreement is not possible; the path to a real update always terminates, and terminates meaningfully with a real update. There are no dead ends or loops.

6.2 Build Conventions

Compiling software using consistently similar practices helps make developers more productive (because they do not have to waste time figuring out unnecessarily different and unnecessarily complex code) and Conary more useful (by enabling some of its more hidden capabilities).

6.2.1 Use Built-in Capabilities

Use build functions other than `r.Run` whenever possible, especially when modifying the `%(destdir)s`.

Build functions that do not have to start a shell are faster than `r.Run`. Using more specific functions enables more error checking; for example, `r.Replace` not only is faster than `r.Run('sed -i -e ...')` but also defaults to raising an exception if it cannot find any work to do. These functions can also remove build requirements (for example, for `sed: runtime`), which can make bootstrapping simpler and potentially faster.

Most build functions have enough context to prepend `%(destdir)s` to absolute paths (paths starting with the `/` character) but in `r.Run` you have to explicitly provide `%(destdir)s` whenever it is needed. For example, `r.Replace('foo', 'bar', '/baz')` is essentially equivalent (except for error checking) to `r.Run("sed -i -e 's/foo/bar/g' %(destdir)s/baz")` in function, but the `r.Replace` is easier to read at a glance.

Many build functions automatically make use of Conary configuration, including macros. `r.Make` automatically enables parallel make (unless parallel make has been disabled for that recipe with `r.disableParallelMake`), and automatically provides many standard variables.

A few build functions can actually check for missing `buildRequires` items. For example, if you install desktop files into the `/usr/share/applications/` directory using the `r.Desktopfile` build command, it will ensure that `desktop-file-utils:runtime` is in your `buildRequires` list, and cause the build to fail if it is not there.

In particular, `r.Configure`, `r.Make`, and `r.MakeParallelSubdirs` provide options and environment variables that the autotools suite, and before that, the default make environment, have made into de-facto standards, including names for directories, tools, and options to pass to tools. Consistency isn't just aesthetic here; it also enhances functionality. It enables `:debuginfo` components that include debugging information, source code referenced by debugging information, and build logs. It allows consistently rebuilding the entire operating system with different compiler optimizations or even different compilers entirely. This is useful for testing compilers as well as customizing distributions.

6.2.2 Macros

Use macros extensively. In general, macros allow recipes to be used in different contexts, allow changes to be made in one place instead of all through a recipe, and can make the recipe easier to read.

Using macros for filenames means that a single recipe can be evaluated differently in different contexts. If you refer to the directory where initscripts go as `%(initdir)s`, the same recipe will work on any distribution built with Conary, whether it uses `/etc/rc.d/init.d/` or `/etc/init.d/`.

Using macros such as `%(cc)s` for program names (done implicitly with make variables when calling the `r.Make*` build actions) means that the recipe will adapt to using different tools, whether that is for building an experimental distribution with a new compiler, or for using a cross-compiler to build for another platform, or any other similar purpose.

Use the macros that define standard arguments to pass to programs, such as `%(cflags)s`, and modify them intelligently. Instead of overwriting them, just modify them, like `r.macros.cflags += ' -fPIC'` or `r.macros.cflags = r.macros.cflags.replace('-O2', '-Os')`

or `r.macros.dbgflags = r.macros.dbgflags.replace('-g', '-ggdb')`

Creating your own macros for text that you would otherwise have to repeat throughout your recipe makes the recipe more readable, less susceptible to bugs from transcribing existing errors and making errors in transcription, and easier to modify. You might, for example, do things like `r.macros.prgdir = '%(datadir)s/%(name)'`

Creating your own macros can also help you make your recipes fit in 80 columns, for easier reading in the majority of terminal sessions.

```
r.macros.url = 'http://reallyLongURL/'
r.addArchive('%(url)s/%(name)s-%(version)s.tar.bz2')
```

6.2.3 Flavored Configuration

When configuring software (generally speaking, before building it, but it is also possible for configuration to control what gets installed rather than what is built), make sure that the configuration choices are represented in the flavor. When a configuration item depends on the standard set of **Use flags** for your distribution, use those. If there is no system-wide Use flag that matches that configuration item, you can create a **local flag** instead.

A lot of configuration is encoded in the arguments to `r.Configure`. We commonly use the variable `extraConfig` to hold those. There are two reasonable idioms:

```
extraConfig = ''
if Use.foo:
  extraConfig += ' --foo'
r.Configure(extraConfig)
```

and

```
extraConfig = []
if Use.foo:
  extraConfig.append('--foo')
r.Configure(
  ' '.join(extraConfig))
```

In either case, referencing `Use.foo` will cause the system-wide Use flag named "foo" to be part of the package's flavor, with the value that is set when the recipe is cooked.

If you need to create a local flag, you do it with the package metadata (like `name`, `version`, and `buildRequires`):

```
class Asdf(PackageRecipe):
  name = 'asdf'
  version = '1.0'
  Flags.blah = True
  Flags.bar = False
  def setup(r):
    if Flags.blah:
      ...
```

Now the `asdf` package will have a flavor that references `asdf.blah` and `asdf.bar`. The values provided as metadata are defaults that are overridden (if desired) when cooking the recipe.

6.3 Destdir Conventions

Choosing how to install files into `%(destdir)s` can determine how resilient your recipe is to changes in Conary and in upstream packaging, and how useful the finished package is.

6.3.1 Makefile Installation

Using `r.Make('install')` would not work very well, because it would normally cause the Makefile to try to install the software directly

onto the system, and you would soon see the install fail because of permission errors. Instead, use `r.MakeInstall()`. It works if the Makefile defines one variable which gives a "root" into which to install, which by default is called `DESTDIR` (thus the `%(destdir)s` name). If that does not work, read the Makefile to see if it uses another make variable (common names are `BUILDROOT` and `RPM_BUILD_DIR`), and pass that in with the `rootVar` keyword: `r.MakeInstall(rootVar='BUILDROOT')`

Sometimes there is no single variable name you can use. In these cases, there is a pretty powerful "shotgun" available: `r.MakePathsInstall`. It re-defines all the common autotools-derived path names to have the `%(destdir)s` prepended. This works for most of the cases without an install root variable. Sometimes you will find (generally from a permission error, less commonly from reviewing the changeset) that you need to pass an additional option: `r.MakePathsInstall('WEIRDDIR=%(destdir)s/path/to/weird/dir')`

For a few packages, there is no Makefile, just a few files that you are expected to copy into place manually. Use `r.Install`, which knows when to prepend `%(destdir)s` to a path, and knows that source files with any executable bit set should default to mode 0755 when packaged, and source files without any executable bit set should default to 0644; specify other modes like `mode=0600`—do not forget the leading 0 that makes the number octal. (Conary does look for mode values that are nonsensical modes and look like you left the 0 off, and warns you about them, but try not to depend on it; the testing is heuristic and not exhaustive.) Like other build actions, `r.Install` will create any necessary directories automatically. If you want to install a file into a directory, make sure to include a trailing `/` character on the directory name so that Install knows that it is intended to be a directory, not a file. (It is this requirement that allows it to make directories automatically.)

6.3.2 Multi-lib Friendliness

Conary does its best to make all packages multi-lib aware. Practically all 32-bit x86 libraries are available to work on the 64-bit x86_64 platform as well, making Conary-based distributions for x86_64 capable of running practically any 32-bit x86 application, not just a restricted set that uses some of the most commonly-used libraries.

The first thing to do is to always use `%(libdir)s` and `%(essentiallibdir)s` instead of `/usr/lib` and `/lib`, respectively. Furthermore, for any path that is not in one of those locations, but still has a directory named "lib" in it, you should use `%(lib)s` instead of `lib`. Conveniently, for programs that use the autotools suite, Conary does this for you, but when you are reduced to choosing directory names or installing files by hand, follow this rule.

On 64-bit platforms on which `%(lib)s` resolves to `lib64`, Conary tries to notice library files that are in the wrong place, and will even move 64-bit libraries where they belong, while warning that this should be done in the packaging rather than as a fixup, because there is probably other work that also needs to be done. Conary warns about errors that it cannot fix up, and causes the cook to fail.

Conary specifically ensures that `:python` and `:perl` components are multi-lib friendly, since there are special semantics here; some `:python` or `:perl` packages have only interpreted files and so should be found in the 32-bit library directory even on 64-bit platforms; others have libraries as well, and should be in

the 64-bit library on 64-bit platforms. Putting the 64-bit object files in one hierarchy and interpreted files in another hierarchy would create path collisions between 32-bit and 64-bit `:python` or `:perl` components.

6.3.3 Direct to Destdir

Occasionally, an upstream project will include a package of data files that is intended to be unpacked directly into the filesystem. Instead of unpacking it into the build directory with `r.addSource` and then copying it to `%(destdir)s` with `r.Copy` or `r.Install`, use the `dir` argument to `r.addArchive`. Normally, `dir` is specified with a relative path and thus is relative to the build directory, but an absolute path is relative to `%(destdir)s`. So something like `r.addArchive('http://example.com/foo.tar.bz2', dir='%(datadir)s/%(name)s/')` will do what you want in just one line. Not only does it make for a shorter recipe with less potentially changing text to cause shadow merges to require manual conflict resolution, it is also faster to do.

6.3.4 Absolute Symlinks

Most packaging rules tell you to use relative symlinks (`../...`) instead of absolute (`/...`) symlinks, because it allows the filesystem to continue to be consistent even when the root of the filesystem is mounted as a subdirectory rather than as the system root directory; for example, in some "rescue disk" situations.

This rationale is great, but Conary does something even better. It automatically converts all absolute symlinks not just to relative symlinks, but to *minimal* relative symlinks. That is, if you create the absolute symlink `/usr/bin/foo -> /usr/bin/bar`, Conary will change that to `/usr/bin/foo -> bar`, and `/usr/bin/foo -> /usr/lib/foo/bin/foo` to `/usr/bin/foo -> ../lib/foo/bin/foo`. Therefore, for Conary, it is best to use absolute symlinks in your `%(destdir)s` and let Conary change them to minimal relative symlinks for you.

6.4 Requirements

Conary has a very strong dependency system, but it is a bit different from legacy dependency systems. The biggest difference is that depending on versions is very different from any other packaging system. Because Conary (by design) does not have a function that tries to guess which upstream versions might be newer than another upstream version, you cannot have a dependency that looks like "upstream version 1.2.3 or greater."

Because Conary has the capability for a rich branching structure, trying to do version comparisons even on Conary versions for the purposes of satisfying dependencies fails utility tests. If you say that a shadow does not satisfy a dependency that its parent satisfies, then shadows are almost useless for creating derivative distributions. However, if you say that a shadow does satisfy a dependency that its parent satisfies, then a shadow that intentionally removes some particular capability relative to its parent will falsely satisfy versioned dependencies. Trying to do strict linear comparisons in the Conary version tree universe just does not work.

Conary separates dependencies into different spaces that are provided with individual semantics. Each ELF shared library provides a **soname** dependency that includes the ABI (for example, `SysV`), class (`ELFCLASS32` or `ELFCLASS64` encoded as `ELF32` or `ELF64`, respectively), and instruction set (`x86`, `x86_64`,

ppc, and so forth) as well as any symbol versions (GLIBC_2.0, GLIBC_2.1, ACL_1.0 and so forth). The elements are stored as separate **flags.** Programs that link to the shared libraries have a dependency with the same format. These dependencies (requirements or provisions) are coded explicitly as a soname dependency class. The order in which the flags are mentioned is irrelevant.

Trove dependencies are limited to components, since they are the only normal troves that directly reference the files needed to satisfy the dependencies. (Filesets also contain files, but they are always files pulled from troves, so they are not the primary sources of the files, and they are not intended for this use.) By default, a trove dependency is just the name of the trove, but it can also include **capability flags**, whose names are arbitrary and not interpreted by Conary except checking for equality (just like upstream versions).

This provides the solution to the version comparison problem. Trove A's recipe does not really require upstream version 1.2.3 or greater of trove B:devel in order to build. Instead, it requires some certain functionality in trove B:devel. The solution, therefore, is for package B to provide a relevant capability flag describing the necessary interface, and for trove A's recipe to require trove B:devel with that capability flag. The capability flag could be as simple as 1.2.3, meaning that it supports all the interfaces supported by upstream version 1.2.3 (the meaning of any package's capability flag is relative to only that package). So package B's recipe would have to call r.ComponentProvides('1.2.3') and trove A's recipe would have to require 'B:devel(1.2.3)'.

This solution does require cooperation between the packagers of A and B, but only in respect to a single context. This means that you may choose to shadow trove B in order to add this capability flag in the context of your derived distribution, if your upstream distribution does not provide the capability your package requires.

Do not add trove capability flags without good reason, especially for build requirements. They add complexity that is not always useful. Usually, the development branch for a distribution just needs to be internally consistent, and adding lots of capability flags will just make it harder for someone else to make a derivative work from your distribution, particularly if they are deriving from multiple distributions at once (a reasonable thing to do in the Conary context).

6.4.1 Build Requirements

Conary's build requirements are intentionally limited to trove requirements.

In general, there are two main kinds of build requirements: :runtime components (and their dependencies) for programs that need to run at build time, and :devel components (and their dependencies) for libraries to which you need to link.

Build requirements need to be added to a list that is part of recipe metadata. Along with name and version, there is a list called buildRequires, which is simply a list of trove names (including, if necessary, flavors, branch names, and capability flags, but not versions). It can be extended conditionally based on flavors.

```
buildRequires = [
  'a:devel(A_CAPABILITY)',
  'gawk:runtime',
  'pam:devel[!bootstrap]',
  'sqlite:devel=:sqlite2',
]
```

```
if Use.gtk:
  buildRequires.append(
    'gtk:devel')
```

The `buildRequires` list does not have to be exhaustive; you can depend on transitive install-time dependency closure for the troves you list. That is to say, in the example above, you do not have to explicitly list `glib:devel`, because `gtk:devel` has an install-time requirement for `glib:devel`. (The `buildRequires` lists do not themselves have transitive closure, as that would be meaningless; you never require a `:source` component in a `buildRequires` list, and the dependencies that the other components carry are install-time dependencies.)

Build requirements for `:runtime` components can be a little bit hard to find if you already have a complete build environment, because some of them are deeply embedded in scripts. It is possible to populate a changeroot environment with only those packages listed in the `buildRequires` list and their dependencies, then chroot to that environment and build in it and look for failures, but it is not likely to be a very useful exercise. The best approach here is to add items to address known failure cases.

Build requirements for `:devel` components are much simpler. Cook the recipe to a local changeset, and then use `conary showcs --deps foo-1.0.ccs` to show the dependencies. (Better yet, use `--all` instead of `--deps` and review the sanity of the entire changeset.) Then, for each soname requirement listed under each `Requires` section, add the associated component to the list. (Right now, this takes too many steps; you need to look for the library, then use `conary q --path /path/to/library` to find the name of the component. In the future, there will be a simple command for looking these up, and we are considering automating the whole process of resolving soname requirements to `buildRequires` list entries.)

6.4.2 Runtime Requirements

The best news about runtime requirements is that you can almost ignore the whole problem. The automatic soname dependencies handle almost everything for you without manual intervention.

There are also some automatic file dependencies, which present a little bit of an asymmetry. Script files automatically require their interpreters. That is, if a file starts with `#! /bin/bash` that file (and thereby its component) automatically has a requirement for `file: /bin/bash` added to it. However, there is no automatic provision of file paths. This is because files are not primarily accessed by their paths, but rather by a long numeric identifier (rather like an inode number in a filesystem, but much longer, and random rather than sequential in nature). Files can be tagged as providing their path, but this must be done manually. In practice, this is not a big problem; most programs that normally act as script interpreters are already tagged as providing their paths, and so the exceptions tend to exist within a single trove. Those cases are easy to fix; Conary refuses to install a trove saying that it cannot resolve a `file: /usr/bin/foo` dependency, but the trove itself contains the /usr/bin/foo file. Just add `r.Provides('file', '%(bindir)s/foo')` to the recipe.

The hard job with any dependency system is working out the dependencies for shell scripts. It is not practical to make shell dependencies automatic for a variety of reasons (including the fact that shell scripts could generate additional dependencies from the text of their in-

put), and so it remains a manual process. If you are lucky, the package maintainer has listed the requirements explicitly in an INSTALL or README file. If not, you need to glance through shell scripts looking for programs that they call. Since this is not a new problem, you can in practice (for some packages, at least), find the results of other people's efforts in this direction by reading RPM spec files and dpkg `debian/rules`. This also tends to be an area where dependencies accrete as a result of bug reports.

There is one place where you need to be much more careful about listing the requirements of shell scripts: you must explicitly list all the requirements of the tag handlers you write. This should not be a great burden; most tag handlers are short and call only a few programs. But if you do not list them, Conary cannot ensure that the tag handlers can always be run, which can jeopardize not only successful installation but also rollback reliability.

7 Release Management

Building software into a repository is already an improvement over legacy package management, but release and deployment need more management and process than just building software into a versioned repository. Several of Conary's features are useful for managing release and deployment; groups, branches, shadows, redirects, and labels can all help.

Different release goals or deployment needs will result in different policies and processes. This paper uses some concrete examples to demonstrate how Conary features can support a release management process, but the mechanisms are flexible and can support diverse processes. A release can go through one QA step or ten separate QA steps without changing the fundamental processes. Release management and deployment have many of the same needs, so this paper will refer generally to release management except when it is useful to distinguish between the two.

The capabilities needed for release management include:

Staging Collecting troves (including locally modified versions) to create a coherent set for promotion to the next step in the process.

Access Control Mandatory or advisory controls on who can or should access a set of troves.

Maintenance Controlled updates for sets of troves.

In addition, the jargon for talking about Linux distributions is somewhat vague and used in conflicting ways. The following definitions apply to this discussion.

Distribution A notionally-connected set of products consisting of an operating system and related components. A distribution might last for years, going through many major release cycles. Examples include rpath Linux, Foresight Linux, Red Hat Linux, Fedora Core, Debian, Mandrake (now Mandriva) Linux, CentOS, cAos, and Gentoo.

Version One instance of a distribution product, encompassing the entire "release cycle," which might include steps like alpha, beta, release candidate, and general availability. Examples include rpath Linux 1, Red Hat Linux 7.3, Fedora Core 2, etc.

Stage A working space dedicated to a task.

Release An instance of any step in the distribution release management process. (This is a slightly unusual meaning for "release;" "version" and "release" are often used almost interchangeably, but for the purposes of this discussion, we need to differentiate these two meanings.) This might be alpha 1 release candidate 1, alpha 1 release candidate 2, alpha 1, beta 1, beta 2, release candidate 1, general availability, and each individual maintenance update.

A release of a version of a distribution is defined (in Conary terms) by a unique version of an inclusive group that defines the contents of the distribution. In rpath Linux, that group is called `group-os`.

7.1 Example Release Management Process

The policy and much of the process in this example is synthetic, but the version tree structure it demonstrates (including the names for labels in the example) is essentially the one that we have defined for rpath Linux.

The development branch called `/conary.rpath.com@rpl:devel` (hence, `:devel`) is where the latest upstream versions are committed. At some point, a group defining a distribution is shadowed to create a **base** stage, `/conary.rpath.com@rpl:devel//rel-base` (hence, `//rel-base`) allowing unfettered development to continue on the `:devel` development branch, while controlled development (a state sometimes called "slush," by analogy from "freeze") is now possible on `//rel-base`.

Given a very simple, informal release management process—say, one where only one person is doing all the work, following all the process from the time that the initial release stage is created, and in which maintenance does not need to be staged—this single shadow creating a single stage might be sufficient. However, in order to allow any controlled development to happen in parallel with the full release engineering process, and in order to allow maintenance work to be staged, a two-level stage structure is necessary.

Therefore, when the controlled development has reached the point where an alpha release is appropriate, another shadow is created on which to freeze that release. This allows controlled development to continue on the release base stage: `/conary.rpath.com@rpl:devel//rel-base//rel-alpha` (hence, `//rel-alpha`). Build the shadowed `group-os` (or whatever you have called your inclusive group), and the version you have just created is a candidate alpha release. Cycle through your test, fix, rebuild process until you have a version of `group-os` that meets your criteria for release as an alpha. At this point, that specific version of `group-os`, say `group-os=/conary.rpath.com@rpl:devel//rel-base//rel-alpha/1.0-1-2.3.1-35`, is your alpha 1 release.

Note that during the test, fix, rebuild process for alpha 1, development work aimed at alpha 2 can already be progressing on the base stage. Fixes that need to be put on the alpha stage for alpha 1 can either be committed to the base stage and thence shadowed to the alpha stage, or if further development has happened on the base stage that could destabilize the alpha stage, or the immediate fix is a workaround or hack and the right fix has not yet been committed to the base stage, the fix, workaround, or hack can be committed directly to the alpha stage.

Then for the alpha 2 cycle, you re-shadow everything from the base stage to the alpha stage, and start the test, fix, rebuild process over again. When you get to betas, you just create a beta stage: `/conary.rpath.com@rpl:`

devel//rel-base//rel-beta (hence, //rel-beta) and work with it exactly as you worked with the alpha stage. Finally, when you are ready to prepare release candidates, build them onto the final release stage /conary.rpath.com@rpl:devel//rel-base//rel (hence, //rel) in the same way.

Note that it is possible to do all your release staging from first alpha to ongoing maintenance onto the release stage //rel. However, using separate named stages for alpha, beta, and general availability can be a useful tool for communicating expectations to users. It is your choice from a communications standpoint; it is not a technical decision.

During maintenance, do all of your maintenance candidates on the base stage, and promote the candidate inclusive group to the release stage by shadowing it when all the components have passed all necessary tests.

All the stages are really **labels**, as well as shadows. You can shadow any branch you need to onto the base stage, and you will probably want to shadow troves from several branches. Not just /conary.rpath.com@rpl:devel but also branches like /conary.rpath.com@rpl:devel/1.5.28-1-0/cyrus-sasl1/ (hence, :cyrus-sasl1) for different versions where both versions should be installed at once. With that :cyrus-sasl1 branch and cyrus-sasl 2 from the :devel branch both shadowed onto the base stage and thence to the release stages, the command conary update cyrus-sasl will put both versions on your system.

When the release stage is no longer maintained, you might choose to cook redirects (perhaps only for your inclusive group, perhaps for all the packages) to another, still-maintained release. This is purely a matter of distribution policy.

8 Derived Distributions

The possibilities for creating derived distributions are immense, but a few simple examples can show some of the power of creating derived distributions.

The simplest example of a derived distribution is the 100% derived distribution. If you merely want control over deployment of an existing distribution, just treat the parent distribution's release stage (//rel) as your base stage, and create a shadow of it in your own repository. You will end up with something like: /conary.rpath.com@rpl:devel//rel-base//rel//conary.example.com@rpl:myrel (hence, //myrel). Then, whenever a group on //rel passes your acceptance tests, you shadow it onto //myrel.

If you are doing anything more complicated, you may want to set up two stages; your own base stage and your own release stage. If you are doing this, you probably do not want to shadow a release stage as your base stage; you will end up with very long version numbers like 1.2.0-2.0.4.0-1.0.1.0; each shadow adds a "." character with a trailing number. You probably want either to shadow the parent distribution's base stage, or even create your own base stage. To create your own base stage, create your own shadow of the parent distribution's inclusive group and make your own changes to it. Those changes might be adding references to some unique troves from your own repository, or to shadows in your repository from other repositories.

You could create a private corporate distribution, with your repository inaccessible from

outside, that contains your internally developed software, or third-party proprietary software to which you have sufficient license. (A source trove doesn't necessarily have to contain source files; it could contain an archive of binary files which are installed into the `%(destdir)s`.) You could create a distribution in which everything is identical to the parent, except that you have your own kernel with special patches that support hardware you use locally that is not yet integrated into the standard kernel, and it has two extra packages which provide user-space control for that hardware.

It is also possible to make significant changes. For example, Foresight Linux[2] is built from the same development branch as rpath Linux, but about 20% of its troves are either specific to Foresight Linux or are shadows that are changed in some way in order to meet Foresight's different goals as a distribution; rpath Linux is meant to be very "vanilla," with few patches relative to upstream packages and therefore easy to branch from, while Foresight Linux is intended to provide the latest innovations in GNOME desktop technology and optimize the rest of the distribution to support this role.

Conclusion

Conary combines system management and software configuration management, sharing features between the two models and implementing them both using a distributed repository that combines source code and the binaries built from that source code. It brings a unique set of features that simplify and unify system management, software configuration management, and release management. This new model drastically reduces the cost and complexity of creating customized Linux distributions. The best practices discussed in this paper help you take advantage of this new paradigm most effectively.

[2] `http://www.foresightlinux.com/`

Profiling Java on Linux

John Kacur
IBM
jekacur@ca.ibm.com

Abstract

In this paper, I will examine two profilers. IBM's Open Source Performance Inspector and OProfile which contains code that has been officially accepted into the Linux Kernel. Currently OProfile doesn't work with programs that dynamically generate code, such as Python and Java JITs. Various people have proposed patches that record events in anonymously mapped memory regions as raw virtual addresses, instead of the usual tuple of binary image and offset. This information can be post-processed by matching it with the output generated by running a Java program with Performance Inspector's JPROF which uses JVMPI to record addresses of JITted methods. In this paper, I will discuss the details of profiling Java, specifically looking at the inner workings of OProfile and Performance Inspector. I will discuss problems that we have encountered with both tools and our attempts to resolve them. Finally, I will demonstrate profiling a java program to show the kind of information that can be obtained.

Check Online

This author did not provide the body of this paper by deadline. Please check online for any updates.

Testing the Xen Hypervisor and Linux Virtual Machines

David Barrera
IBM Linux Technology Center
dbarrera@us.ibm.com

Li Ge
IBM Linux Technology Center
lge@us.ibm.com

Stephanie Glass
IBM Linux Technology Center
sglass@us.ibm.com

Paul Larson
IBM Linux Technology Center
plars@us.ibm.com

Abstract

Xen is an interesting and useful technology that has made virtualization features, normally found only in high-end systems, more widely available. Such technology, however, demands stability, since all virtual machines running on a single system are dependent on its functioning properly. This paper will focus on the methods employed to test Xen, and how it differs from normal Linux® testing. Additionally, this paper discusses tests that are being used and created, and automation tools that are being developed, to allow testers and developers working on Xen to easily run automated tests.

1 Testing Linux vs. Testing Linux Under Xen

Xen, which provides a high performance resource-managed virtual machine monitor (VMM) [2], is one of several open-source projects devoted to offering virtualization software for the Linux environment. As virtualization is rapidly growing in popularity, Xen has recently gained a lot of momentum in the open-source community and is under active development. Therefore, the need to test Xen becomes a critical task to ensure its stability and reliability. Most often, people run tests on Linux under Xen in order to exercise Xen code and to test its functionalities, as Xen is the hypervisor layer that is below the Linux and above the hardware.

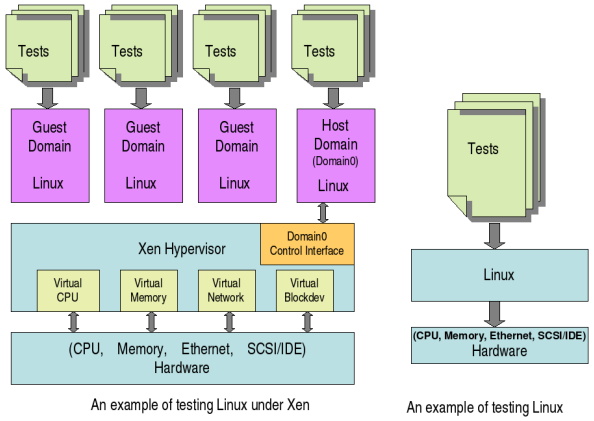


Figure 1: Testing Linux With and Without Xen

1.1 Similarities

Testing Linux under Xen and testing Linux itself are very much alike. Those traditional testing scenarios used to test Linux can also be applied to testing Linux under Xen. The most

common testing done on Linux is testing different kernels and kernel configurations to uncover any regressions or new bugs. To help insure binary compatibility, different versions of glibc may also be used. Another big chunk of tests done is a wide range of device I/O tests including networking and storage tests. Also, hardware compatibility testing is very important to insure reliability across a broad range of hardware such as x86, x86-64, UP, SMP, and Blades.

The ABI for running Linux under Xen is no different than running under Linux on bare hardware, there is no change needed for user-space applications when running on Linux under Xen. In general, all user-space applications that can be used to test Linux can be used to test Linux on Xen. For example, memory intensive web serving application and real world large database applications are very good tools to create high stress workload for both Xen and the guest Linux OS.

1.2 Differences

Although testing Linux under Xen and testing Linux are very similar, there are still some fundamental differences. First, Xen supports many virtual machines, each running a separate operating system instance. Hence, on one physical machine, testing Linux under Xen involves testing multiple versions of Linux, which can be different on multiple domains, including host domain and guest domain. The Linux distribution, library versions, compiler versions, and even the version of the Linux kernel can be different on each domain. Furthermore, each domain can be running different tests without disturbing the tests running on other domains. The beneficial side of this is that you can use a single machine to enclose and test upgrades or software as if they were running in the existing environment, but without disturbing the other domains [3].

Second, running tests on Linux under Xen guest domain actually accesses hardware resources through Xen virtual machine interfaces, while running tests on Linux accesses physical hardware resources directly. Xen virtual machine interfaces have three aspects: memory management, CPU, and device I/O [2]. In order to achieve virtualization on these three aspects, Xen uses synchronous hypercalls and an asynchronous event mechanism for control transfer, and uses the I/O rings mechanism for data transfer between the domains and the underlying hypervisor. Therefore, even though the Xen hypervisor layer appears to be transparent to the application, it still creates an additional layer where bugs may be found.

Third, Xen requires modifications to the operating system to make calls into the hypervisor. Unlike other approaches to virtualization, Xen uses para-virtualization technology instead of full virtualization to avoid performance drawbacks [2]. For now, Xen is a patch to the Linux kernel. Testing Linux on Xen will be testing the modified Linux kernel with this Xen patch. As Xen matures though, it may one day be part of the normal Linux kernel, possibly as a sub-architecture. This would simplify the process of testing Xen and make it much easier for more people to become involved.

2 Testing Xen With Linux

One useful and simple approach to testing Xen is by running standard test suites under Linux running on top of Xen. Since Xen requires no user space tools, other than the domain management tools, this is a very straightforward approach. The approach to test Linux under Xen described here is patterned after the approach

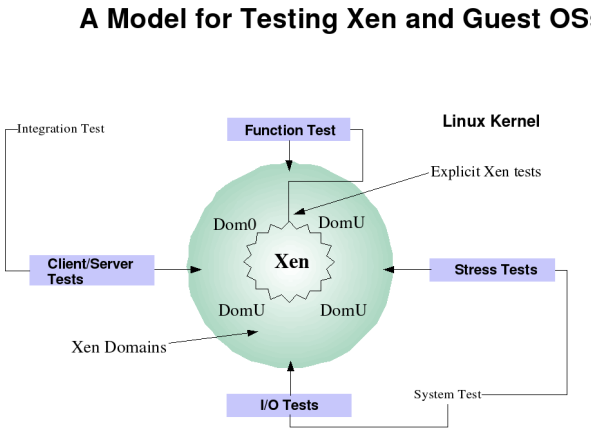


Figure 2: Xen Testing Model

taken to test the development Linux kernels. The traditional testing model used to test the Linux kernel involves function, system, and integration testing. One clear advantage to this approach is that results can easily be checked against results of the same tests, running on an unmodified kernel of the same version, running on bare hardware.

The Linux Test Project (LTP) test suite is the primary tool we used in function testing. LTP is a comprehensive test suite made up of over two thousand individual test cases that test such things as system calls, memory management, inter-process communications, device drivers, I/O, file systems, and networking. The LTP is an established and widely used test suite in the open source community, and has become a de facto verification suite used by developers, testers, and Linux distributors who contribute enhancements and new tests back to the project.

The system testing approach involves running workloads that target specific sub-systems. For example, workloads that are memory intensive or drive heavy I/O are used to stress the system for a sustained period of time, say 96 hours. These tests are performed after function tests have successfully executed; thus, defects that manifest only under stressful conditions are discovered. For example, in past system test efforts testing development Linux kernels, a combination of I/O heavy and file system stress test suites have been used such as IOZone, Bonnie, dbench, fs_inode, fs_maim, postmark, tiobench, fsstress, and fsx_linux. The tests are executed on a given file system, sustained over a period of time to expose defects. The combination of these tests have proven themselves particularly useful in exposing defects in many parts of the kernel.

Integration testing is done after function and system testing have been successfully executed. This type of testing involves the running of multiple, varied workloads that exercise most or all subsystems. A database workload, for example, is used to insert, delete, and update millions of database rows, stressing the I/O subsystem, memory management, and networking if running a networked application. Additionally, other workloads are run in parallel to further stress the system. The objective is to create a realistic scenario that will expose the operating systems to interactions that would otherwise not be exercised under function or system test.

Figure 3, Sample Network Application, illustrates an integration test scenario where a database application, the Database Open source Test Suite (DOTS), is used to create a pseudo-networked application running both the clients and the server on virtual machines running on the same hardware under Xen. Obviously, this is an unlikely scenario in the real world, but it is useful in test to induce a workload in a test environment.

3 Testing Xen More Directly

While much of the functionality of Xen can be tested using Linux and standard tests, there

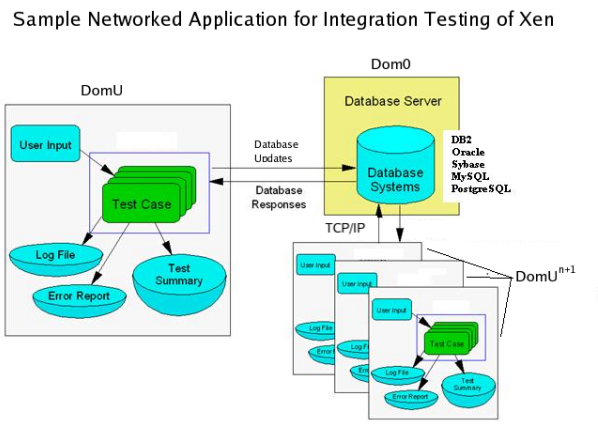


Figure 3: Sample Network Application

are many features that are very specific to Xen. Such features often require careful attention to insure they are adequately tested. A couple of examples include privileged hypercalls, and the balloon driver. A testing strategy for each is briefly outlined here to illustrate why simply running Linux as a guest OS under Xen and running standard tests does not suffice for testing Xen as a whole.

3.1 Testing Privileged Hypercalls

Domain 0 in Xen is considered to be a privileged domain. As the privileged domain, there are certain operations that can only be performed from this domain. A few of these privileged operations include:

1. `DOM0_CREATEDOMAIN` – create a new domain
2. `DOM0_PAUSEDOMAIN` – remove a domain from the scheduler run queue
3. `DOM0_UNPAUSEDOMAIN` – mark a paused domain as schedulable again
4. `DOM0_DESTROYDOMAIN` – deallocate all resources associated with a domain
5. `DOM0_IOPL` – set I/O privilege level
6. `DOM0_SETTIME` – set system time
7. `DOM0_READCONSOLE` – read console content from the hypervisor buffer ring

These are just a few of the privileged operations available only to domain 0. A more complete list can be found in `xen/include/public/dom0_ops.h` or in the Xen Interface Manual [4].

Many of these operations perform actions on domains such as creating, destroying, pausing, and unpausing them. These operations can easily be tested through the Xen management tools. The management tools that ship with Xen provide a set of user space commands that can be scripted in order to exercise these operations.

Other operations, such as `DOM0_SETTIME`, can be exercised through the use of normal Linux utilities. In the case of `DOM0_SETTIME`, something like `hwclock --systohc` may be used to try to set the hardware clock to that of the current system time. The return value of that command on domain 0 is 0 (pass) while on an unprivileged domain it is 1 (fail). This simple test not only verifies that it succeeds as expected on domain 0, but also sufficiently shows that the operation fails as expected on an unprivileged domain.

For something like IOPL, there are tests in LTP that exercise the system call. These tests are expected to pass on domain 0, but fail on unprivileged domains. This is an example where a the results of a test may be unintuitive at first glance. The iopl test in LTP will prominently display a failure message in the resulting test output, but context must be considered as a “FAIL” result would be considered passing in unprivileged domains.

Still other operations such as DOM0_READCONSOLE are probably best, and easiest to test in an implicit manner. The functionality of readconsole may be exercised by simply booting Xen and watching the output for obviously extraneous characters, or garbage coming across the console. Moreover, features of the console can be tested such as pressing Control-A 3 times in a row to switch back and forth from the domain 0 console to the Xen console.

3.2 Testing the Xen Balloon Driver

Another feature of Xen that warrants attention is the balloon driver. The balloon driver allows the amount of memory available to a domain to dynamically grow or shrink. The current balloon information for a domain running Linux can be seen by looking at the contents of /proc/xen/balloon. This is an example of the resulting output:

```
# cat /proc/xen/balloon
Current allocation:   131072 kB
Requested target:     131072 kB
Low-mem balloon:           0 kB
High-mem balloon:          0 kB
Xen hard limit:          ??? kB
```

This feature is wide open to testing possibilities. Some of the possible test scenarios for the balloon driver include:

1. Read from /proc/xen/balloon.
2. Echo a number higher than current ram to balloon, cat balloon and see that requested target changed.
3. Echo a number lower than current ram to balloon, cat balloon and see that requested target and current allocation changed to that number.
4. Allocate nearly all available memory for the domain, then use /proc/xen/balloon to reduce available memory to less than what is currently allocated.
5. Try to give /proc/xen/balloon a value larger than the available RAM in the system.
6. Try to give /proc/xen/balloon a value way too low, say 4k for instance.
7. Write something to /proc/xen/balloon as a non-root user, expect -EPERM.
8. Write 1 byte to /proc/xen/balloon, expect -EBADMSG.
9. Write >64 bytes to /proc/xen/balloon, expect -EFBIG.
10. Rapidly write random values to /proc/xen/balloon.

Many of the above tests may also be performed by using an alternative interface for controlling the balloon driver through the domain management tools that come with Xen. Scripts are being written to automate these tests and report results.

4 Xentest

In the process of testing Xen, occasionally a patch will break the build, or a shallow bug will get introduced from one day to the next. These kinds of problems are common, especially in large projects with multiple contributers, but they are also relatively easy to look for in an automated fashion. So, a decision was made to develop an automated testing framework centered around Xen. This automated testing framework is called *Xentest*.

There are, of course, several test suites already available that may be employed in the testing of Xen. It should be made clear that Xentest is not a test suite, but rather an automation framework. The main purpose of Xentest is to provide automated build services, start the execution of tests, and gather results. That being said, the build and boot part of Xentest can be considered a build verification test (BVT) in its own right.

Our hope is that Xentest can be used by anyone with a spare machine to execute nightly tests of Xen. It was designed to be simple and unobtrusive, while still providing the basic functionality required in an automated testing framework. Our goals were:

1. Use existing tools in Xen wherever possible.
2. Simple and lightweight design, requires only a single physical machine to run.
3. Supports reusable control files.
4. Tests running under Xentest are easily extended by just adding lines to the control file.

At the time this is being written, Xentest is composed of three main scripts named xenbuild, xenstartdoms, and xenruntests. There is also a small init.d script, and a control file is used to describe information such as: where to pull Xen from, which virtual machines to launch, and which tests to run on which virtual machines. A shared directory must also be created and defined in the control file. The shared directory is used for communicating information down to the virtual machines, and for storing results for each virtual machine. Usually, something like NFS is used for the shared directory.

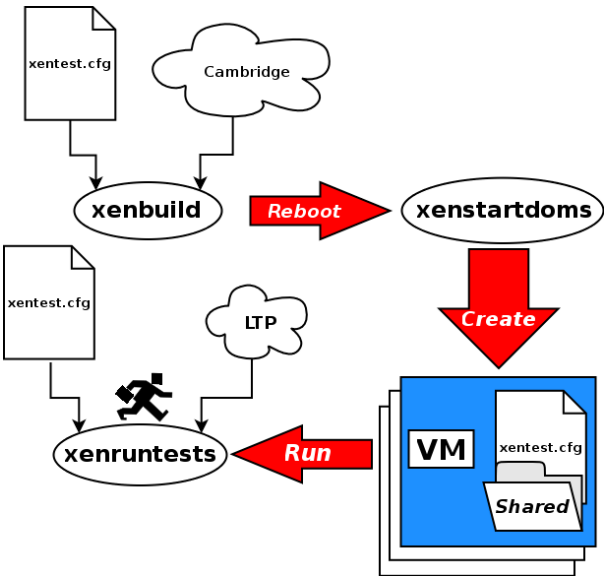


Figure 4: Xentest process

The xenbuild script takes a single argument, the name of control file to use for this test. That control file is first copied to `/etc/xen/xentest.conf`. The xenbuild script is responsible for downloading the appropriate version of Xen, building it, and rebooting the system. Before the system reboot occurs, a file is created in `/etc` called `xen_start_tests`. The `init.d` script checks for the existence of this file to signify that it should launch the remaining scripts at boot time.

If the `init.d` script has detected the existence of `/etc/xen_start_tests`, the next script to be executed after a successful reboot is xenstartdoms. The xenstartdoms script reads `/etc/xentest.conf` and calls `xm create` to create any virtual machines defined in the control file. The xenstartdoms script also creates subdirectories for each virtual machine in the shared directory for the purpose of storing test results. For now though, `/etc/xentests.conf`, which is a copy of the original control file passed to xenbuild, is copied into that directory.

The xenruntests script looks for a directory matching its hostname in the shared directory. In this directory it expects to find a copy of `xentests.conf` that was copied there by xenstartdoms. All domains, including dom0, look for `xentests.conf` there in the xenruntests scripts, so that no special handling is needed for domain 0. Xenruntests is the only script executed in all domains. After reading the control file in, xenruntests finds the section corresponding to the virtual machine it is running on, and reads a list of tests that it needs to execute. A section corresponding to each test is then located in the control file telling it where to download the test from, how to build and install the test, how to run the test, and where to pick up logs from. After performing all these tasks for each test, xenruntests removes its own copy of the control file stored in the shared directory. This signifies that it is complete, and prevents the possibility of it from interfering with future runs.

5 Xentest control file format

The Xentest control file structure is simple and easy to read, but it is also highly configurable. It allows tests to be easily defined, and executed independent of one another on multiple guests. The ConfigParser class in python is used to implement Xentest control files, so the control file structure adheres to RFC 822 [1]. Let's take a look at a basic control file.

```
[Preferences]
xen-tree=xen-unstable
shared_dir=/xentest/shared
```

This section defines the tree you want downloaded for testing, and the shared directory to use. Remember that these config files are reusable, so it's easy to set up a control file for any given machine to explicitly run tests on the stable, testing, and unstable Xen builds. The other variable here is the shared directory, which was discussed previously and is usually mounted over something like NFS. The `/etc/fstab` should be configured to automatically mount the shared directory for every domain configured for testing under Xentest.

```
[Locations]
xen-2.0=http://www.where/to/
        download/xen-stable.tgz
xen-2.0-testing=http://www.where/
        to/download/xen-testing.tgz
xen-unstable=http://www.where/to/
        download/xen-unstable.tgz
```

The locations in this section simply describe where to download each of the Xen nightly snapshot tarballs. More can be added if it ever becomes necessary. To work properly, the value for xen-tree above must simply match the variable name of one of these locations.

```
[LTP]
source_url=http://where.to.download/
        ltp/ltp-full-20050207.tgz
build_command=make
install_command=make install
test_dir=ltp-full-20050207
log_dir=logs/ltp
run_command=./runltp -q > \
        ../logs/ltp/runltp.output
```

A bit more information is require to describe a specific test to Xentest. First, `source_url` describes where to get the tarball for the test from. Currently gzip and bzip2 compressed tar files are supported.

The `test_dir` variable tells Xentest the name of the directory that will be created when it extracts the tarball. After changing to that directory, Xentest needs to know how to build the test. The command used for building the test, if

any command is needed, is stored in `build_command`. Likewise, if any commands are needed for installing the test before execution, Xentest can determine what to run by looking at the value of `install_command`.

The value of `log_dir` is used to tell Xentest where to pick up the test output from, and `run_command` tells it how to run the test. This will be enough or more than enough to handle a wide variety of tests, but for especially complex tests, you might consider writing a custom script to perform complex setup tasks beyond the scope of what is configurable here. Then all that would need to be defined for the test is `source_url`, `test_dir`, and `run_command`.

Since Xentest relies on the ConfigParser class in python to handle control files, variables may be used and substituted, but only within the same section, or if they are defined in the [DEFAULT] section. For instance, if temporary directory was defined in this section as `tempdir`, then variables like `log_dir` can be specified as:

```
log_dir=%(tempdir)s/logs/ltp
```

Since the temporary directory is more appropriately defined under the domain section (described below), a variable substitution cannot be used here. It is for this reason that all directories in the test section are relative to the path of the temporary directory on the domain being tested. Even though the variable substitution provide by the python ConfigParser class is not available for use in this case, there may be other situations where a Xentest user can define variables in [DEFAULT], or in the same section that would be useful for substitution. This allows for a great range of configuration possibilities for different environments.

```
[XENVM0]
tempdir=/tmp
config=none
name=bob13
test1=LTP

[XENVM1]
tempdir=/tmp
config=/home/plars/xen/xen-sarge/
        bob13-vm1.config
name=bob13-vm1
test1=LTP

[XENVM2]
tempdir=/tmp
config=/home/plars/xen/xen-sarge/
        bob13-vm2.config
name=bob13-vm2
```

These three sections describe the domains to be started and tested by Xentest. The only requirement for these section names is that they start with the string `XENVM`. That marker is all Xentest needs in order to understand that it is dealing with a domain description, anything after that initial string is simply used to tell one from another.

The `config` variable sets the config file that will be used to start the domain, if any. If this variable is set, that file will be passed to `xm create -f` in order to start the domain running. In the case of domain 0, or in the event that the domain will already be started by some other mechanism before Xentest is started, the field may be left blank.

The `tempdir` variable is used to designate the temporary directory that will be used on that domain, since you may want a different directory for every one of them. The `name` variable should match the hostname of the domain it is running on. Remember that this file is going to get copied into the shared directory for every domain to look at. In order to figure out where its `tempdir` is, each domain will find its section in the control file by simply looking for its own hostname in one of the XENVM sections.

Notice that Xentest does not try to understand whether a test passes or fails. Determination of test exit status is best left up to post-processing scripts that may also contain more advanced features specific to the context of an individual test. Such features may include:

1. Results comparisons to previous runs
2. Nicer output of test failures
3. Graphing capabilities
4. Test failure analysis and problem determination
5. Results summaries for all tests

No post-processing scripts are currently provided as part of Xentest, but as more tests are developed for testing Xen, they would be a useful enhancement.

Xenfc is an error or negative path test for the domain 0 hypercalls in Xen, and was originally written by Anthony Liguori. What that means is that xenfc attempts to make calls into the hypervisor, most of which are expected to fail, and checks to see that it received the expected error back for the data that was passed to the hypercall. Furthermore, xenfc does not systematically test all of the error conditions, but rather generates most of its data randomly. It is probabilistically weighted towards generating valid hypercalls, but still with random data.

Xenfc generates a random interface version 1% of the time, the other 99% of the time it uses the correct interface version. 80% of the time, a valid hypercall is generated, 20% of the time, it is a random hypercall. The random nature of this test accomplishes three important goals:

1. Stress testing the error code path in Xen hypercalls
2. Consistency checking in error handling with different data
3. Bounds checking, as often the data is on the edge, far off from expected limits

Valid commands currently tested by xenfc are:

1. `DOM0_CREATEDOMAIN`
2. `DOM0_PAUSEDOMAIN`
3. `DOM0_UNPAUSEDOMAIN`
4. `DOM0_DESTROYDOMAIN`

These are only a few of the domain 0 hypercalls currently available in Xen, and more tests are being added to cover these in xenfc. Even in its current state though, xenfc has turned up some interesting results, and uncovered bugs in Xen not yet seen in any other tests. Tests such as xenfc are highly effective at uncovering corner cases that are hard to reproduce by conventional means. Even though bugs like this are difficult to find in normal use, that does not make them any less serious. Even though xenfc relies heavily on randomization, the seed is reported at the beginning of every test run so that results can be reproduced.

Xenfc currently supports the following options:

1. `s` – specify a seed rather than generating one for reproducing previous results
2. `l` – specify a number of times to loop the test, new random data and calls are generated in each loop

Here is some sample output of xenfc in its current form:

```
Seed: 1114727452
op
  .cmd = 9
  .interface_version = 2863271940
  .u.destroydomain
    .domain = 41244
Expecting -3
PASS: errno=-3 expected -3
```

In this example, xenfc is calling the `DOM0_DESTROYDOMAIN` hypercall. The interface version is valid, but the domain it's being told to destroy is not, so `-ESRCH` is expected. Before attempting to execute the hypercall, the `dom0_op_t` structure is dumped along with the relevant fields for this particular call. This can help debug the problem in the event of a failure.

6 Legal Statement

References

[1] *The Python Library Reference*, March 2005. `http://www.python.org/doc/2.4.1/lib/module-ConfigParser.html`.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles (SOSP 2003)*. ACM, October 2003.

[3] Bryan Clark. A moment of xen: Virtualize linux to test your apps. `http://www-128.ibm.com/developerworks/linux/library/l-xen/`, March 2005.

[4] The Xen Team. *Xen Interface Manual - Xen v2.0 for x86*, 2004. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/readmes/interface.pdf`.

Accelerating Network Receive Processing

Intel I/O Acceleration Technology

Andrew Grover
Intel Corporation
andrew.grover@intel.com

Christopher Leech
Intel Corporation
christopher.leech@intel.com

Abstract

Intel® I/O Acceleration Technology (I/OAT) is a set of features designed to improve network performance and lower CPU utilization. This paper discusses the implementation of Linux support for the three features in the network controller and platform silicon that make up I/OAT. It also covers the bottlenecks in network receive processing that these features address, and describes I/OAT's impact on the network stack.

1 Introduction

As network technology has improved rapidly over the past ten years, a significant gap has opened between the CPU overhead for sending and for receiving packets. There are two key technologies that allow the sending of packets to be much less CPU-intensive than receiving packets.

First, TCP segmentation offload (TSO) allows the OS to pass a buffer larger than the connection's Maximum Transmission Unit (MTU) size to the network controller. The controller then segments the buffer into individual Ethernet packets, attaches the proper protocol headers, and transmits. Without TSO, each MTU-sized data buffer must be passed to the controller individually, which is more CPU-intensive.

Second, data to be transmitted need not even be touched by the CPU, allowing zero-copy operation. Using the `sendfile()` interface, the kernel does not need to copy the user data into networking buffers, but can point to pages pinned in the page cache as the source of the data. This also does not pollute the CPU cache with data that is not likely to be used again, and lowers the CPU cycles needed to send a packet.

However, neither of the above optimizations can be applied to improve receive performance. I/OAT attempts to alleviate the additional overhead of receive packet processing with the addition of three additional features:

1. Split headers
2. Multiple receive queues
3. DMA copy offload engine

Each of these is targeted to solve a particular bottleneck in receive processing. They should help to alleviate receive processing overhead issues by allowing better network receive throughput and/or lower CPU utilization. Each can be implemented without requiring radical changes to the way the Linux network stack currently works.

2 Split Headers

For transmission over the network, several layers of headers are attached to the actual application data. One common example consists of a TCP header, an IP header, and an Ethernet header, the former each in turn wrapped by the latter. (This is a gross simplification of the varieties of network headers, made for the sake of convenience.) When receiving a packet, at a minimum the network controller only must examine the Ethernet header, and all the rest of the packet can be treated as opaque data. Therefore, when the controller DMA transfers a received packet into a buffer, it typically transfers the TCP/IP and Ethernet headers along with the actual application data into a single buffer.

Recognizing higher-level protocol headers can allow the controller to perform certain optimizations. For example, all modern controllers recognize TCP and IP headers and use this knowledge to perform checksum validation, offloading this task from the OS's network stack.

I/OAT adds support for split headers. Using this capability, the controller can partition the packet between the headers and the data, and copy these into two separate buffers. This has several advantages. First, it allows both the header and the data to be optimally aligned. Second, it allows the network data buffer to consist of a small slab-allocated header buffer plus a larger, page-allocated data buffer. Surprisingly, making these two allocations is faster than one large slab allocation, due to how the buddy allocator works. Third, split header suppoer results in better cache utilization by not polluting the CPU's cache with any application data during network processing.

3 Multiple Receive Queues

While the processing of large MTU-sized packets is not generally CPU limited, receiving many small packets requires additional processing that can fully tax the CPU, resulting in a bottleneck. Even on a system with many CPUs, this may limit throughput, since processing for a network controller occurs on a single CPU—the one which handled the controller's interrupt.

Multiple receive queues allow network processing to be distributed among more than one CPU. This improves utilization of the available system resources, and results in higher small-packet throughput by alleviating the CPU bottleneck.

The next-generation Intel network controller has multiple receive queues. The queue for a given packet is chosen by computing a hash of certain fields in its protocol headers. This results in all packets for a single TCP stream being placed on the same queue.

After packets are received and an interrupt generated, the interrupt service routine uses a newly-added function called `smp_call_async_mask()` to send inter-processor interrupts (IPIs) to the two CPUs that have been configured to handle the processing for each queue:

```
struct call_async_data_struct {
  void (*func) (void *info);
  void *info;
  cpumask_t cpumask;
  atomic_t count;
  struct list_head node;
};

int smp_call_async_mask(
  struct call_async_data_struct
         *call_data);
```

The IPI runs a function that starts NAPI polling on each CPU, using a hidden polling netdev.

The existing mechanism for running a function on other CPUs, `smp_call_function()`, cannot be called from interrupt context; waits for the function to complete; and runs the function on all CPUs, instead of allowing the called CPUs to be specified. These shortcomings were addressed by adding `smp_call_async_mask()`.

The overhead of using IPIs is minimized because of NAPI. An IPI is only needed to enter NAPI polling mode for the two queues. Once in NAPI mode, the two CPUs independently process packets on their queues without any additional overhead.

On single-processor systems, a single receive queue is used, since there are no additional CPUs to perform packet processing. In addition, on multi-CPU systems that also support HyperThreading, we ensure that the two CPU threads targeted for receive processing do not share the same physical core.

Preliminary benchmark results show this implementation results in greater small-packet throughput.

4 DMA Copy Offload Engine

As shown in Table 1, the most time during receive processing is spent copying the data. While a modern processor can handle these copies for a single gigabit connection, when multiple gigabit links, or a ten-gigabit connection is present, the processor may be swamped. All the cycles spent copying incoming packets are cycles that prevent the CPU from performing more demanding computations.

Samples	Percent	Function
48876	18.1772	__copy_user_intel
10382	3.8611	tcp_v4_rcv
10206	3.7957	e1000_intr
7640	2.8414	schedule
7130	2.6517	e1000_irq_enable
6965	2.5903	eth_type_trans
6355	2.3635	default_idle
6300	2.3430	find_busiest_group
6231	2.3173	packet_rcv_spkt

Table 1: oprofile data taken during netperf TCP receive test (TCP_MAERTS), e1000

I/OAT offloads this expensive data copy operation from the CPU with the addition of a DMA engine—a dedicated device to do memory copies. While the DMA engine performs the data copy, the CPU is free to proceed with processing the next packet, or other pending task.

4.1 The DMA Engine

The DMA engine is implemented as a PCI-enumerated device in the chipset, and has multiple independent DMA channels with direct access to main memory. When the engine completes a copy, it can optionally generate an interrupt.[1]

4.2 The Linux DMA Subsystem

The I/OAT DMA engine was added specifically to benefit network-intensive server loads, but its operation is not coupled tightly with the network subsystem, or the network controller driver.[2] Therefore, support is implemented as a "DMA subsystem." This subsystem exports a

[1] Further hardware details will be available once platforms with the DMA engine are generally available.

[2] Future generations may be more tightly integrated.

generic async-copy interface that may be used by other parts of the kernel if modified to use the subsystem interface. It should be easy for other subsystems to make use of the DMA capability, so we made async memcpy look as much like normal memcpy as possible. This abstraction also gives hardware designers the freedom to develop new DMA engine hardware interfaces in the future.

The first step for kernel code to use the DMA subsystem is to register, using `dma_client_register()`, and request one or more DMA channels:

```
typedef void
(*dma_event_callback)(
  struct dma_client *client,
  struct dma_chan *chan,
  enum dma_event_t event);

struct dma_client *
dma_client_register(
  dma_event_callback
event_callback);

void
dma_client_chan_request(
  struct dma_client *client,
  unsigned int number);
```

Depending on where in the kernel init process this is done, DMA channels may be already available for allocation, or may be enumerated later, at which point clients who have asked for but not yet received channels will have their callback called, indicating the new channel may be used.[3] Clients need to handle the failure to receive a DMA channel gracefully. This is usually easy to do, as the client can fall back to non-offloaded copying.

(The initial need to make channel allocation asynchronous was driven by the desire to use it in the net stack. The net stack initializes very early, before PCI devices are enumerated, so use of a synchronous allocation method would result in the net stack asking for DMA channels before any were available, and then never getting any, once they were.)

Once a client has a DMA channel, it can start using copy offload functionality:

```
dma_cookie_t
dma_memcpy_buf_to_buf(
    struct dma_chan *chan,
    void *dest,
    void *src,
    size_t len);

dma_cookie_t
dma_memcpy_buf_to_pg(
    struct dma_chan *chan,
    struct page *page,
    unsigned int offset,
    void *kdata,
    size_t len);

dma_cookie_t
dma_memcpy_pg_to_pg(
    struct dma_chan *chan,
    struct page *dest_pg,
    unsigned int dest_off,
    struct page *src_pg,
    unsigned int src_off,
    size_t len);
```

Notice that in addition to a version that takes kernel virtual pointers for source and destination, there are also versions to copy from a buffer to a page, as well as from page to page.[4]

[3] The initial need to make channel allocation asynchronous was driven by the desire to use it in the net stack. The net stack initializes very early, before PCI devices are enumerated, so use of a synchronous allocation method would result in the net stack asking for DMA channels before any were available, and then never getting any, once they were.

[4] Many parts of the kernel use a pointer to a buffer's struct page instead of a pointer to the memory itself, since on systems with highmem, not all physical memory is directly addressable by the kernel.

These operations are asynchronous and the copy is not guaranteed to be completed when the function returns. It is necessary to use another function to wait for the copies to complete. These functions return a non-negative "cookie" value on success, which is used as a token to wait on:

```
enum dma_status_t
dma_wait_for_completion(
    struct dma_chan *chan,
    dma_cookie_t cookie);

enum dma_status_t
dma_memcpy_complete(
    struct dma_chan *chan,
    dma_cookie_t cookie);
```

Typically, a client has a series of copy operations it can offload, but there comes a point when it cannot continue until all the copy operations are guaranteed to have been completed. At this point, the client can use the above functions with the last cookie value returned from the memcpy functions. If the copy operations have been properly parallelized they may already be complete. If not, the client uses one of the above functions, depending on if it wants to sleep, or not.

4.3 Net Stack Changes Required for Copy Offload

The Linux network stack's basic copy-to-user operation is from a series of struct skbuffs (also known as SKBs) each generally containing one network packet, to an array of struct iovecs each describing a user buffer.[5] Both these data structures are rather complex, which complicates matters.

In addition, final TCP processing and the copy-to-user operation must happen in the context of the process for the following reasons:

1. The user buffer (described by the iovec) is pageable. If it is paged out when written, it will generate a page fault. Page faults can only be handled in process context.

2. If the network controller does not implement TCP checksum capability, it is possible to do the copy-to-user and checksum in one step. However, almost all modern controllers support hardware TCP checksum.

3. ACK generation. Waiting until in the process context to generate TCP ACKs ensures that the ACKs represent the actual rate that the process is getting scheduled and receiving packets. If the stack ACKed as soon as the packet was received, this might cause the receiver to be overwhelmed[DM].

These three reasons drove the implementation of the changes to the network stack. In order to achieve proper parallelism, it is crucial that we begin the copy as soon as possible, from bottom half or interrupt context, and not wait until after the return to process context. Therefore, we:

1. Lock down the user buffer, using `get_user_pages()`. There is a real performance penalty associated with doing this (measured ~6800 cycles to pin, ~5800 to un-pin) that must be saved via copy parallelism before we achieve a benefit.

2. Do not initiate engine-assisted copies on non-HW-checksummed data.

3. Wait until we are in the process context to generate ACKs.

While this code is still under development, the current sequence of events is:

[5] Usually the array will contains only one entry, but if `readv()` is used, it will contain more.

1. When entering `tcp_recvmsg()` as a result of a `read()` system call, the iovec is pinned in memory. This generates a list of pages that map to the iovec, which we save in a secondary structure called the `locked_list`.
2. The process sleeps.
3. Packets arrive and an interrupt is generated. NAPI polling starts, and packets are run up the net stack to `tcp_v4_rcv()`.
4. Normally the packet is placed on the prequeue so TCP processing is completed in the process context. However, `tcp_prequeue` also tries doing fastpath processing on the packet, and if successful, starts the copy to the user buffer. Even though it is executing from a bottom half and copying to a user buffer, it will not take a page fault, since the pages are pinned in memory. For each such packet, we set a flag in the SKB, `copied_early`.
5. The process wakes up, and checks the prequeue for packets to process. For any packets with the `copied_early` flag set, fastpath checks are skipped, and ACK-generation starts.
6. Normally at the end of `tcp_rcv_established()` the skb is freed by calling `__kfree_skb()`. However, the DMA engine may still be copying data, so it is necessary to wait for copy completion. Instead of being freed, the SKB is placed on another queue, the `async_wait_queue`.
7. The process waits for the last cookie to be to be completed, using `dma_wait_for_completion`.
8. The iovec is un-pinned and its pages are marked dirty.
9. All SKBs in the `async_wait_queue` are freed.
10. The system call is completed.

Using this mechanism packet processing by the CPU and the DMA engine's data copies take place in parallel. Of course, for a user buffer to be available for the early async copy to commence, the user process must make a buffer available prior to packet reception by using `read()`. If the process is using `select()` or `poll()` to wait for data, user buffers are not available until data has already arrived. This reduces the parallelism possible, although reduced CPU utilization should still be attainable. Further work into asyncronous network interfaces may allow better utilization of the DMA engine.

5 Conclusion

Each of these new features targets a specific bottleneck in the flow of handling received packets, and we believe they will be effective in alleviating them. However, more development, testing, and benchmarking is needed. This paper is meant to be a starting point for further discussions in these areas—we look forward to working with the Linux community to support these features.

References

[LDD] J. Corbet, Alessandro Rubini, Greg Kroah-Hartman, *Linux Device Drivers, 3rd Edition*, 2005

[DM] David Miller, *Re: prequeue still a good idea?*, `http://marc.theaimsgroup.`

com/?l=linux-netdev&m=111471509704660&w=2, Apr 28, 2005

[LWN] LWN, *Driver porting: Zero-copy user-space access*, http://lwn.net/Articles/28548/, Nov 2003

dmraid - device-mapper RAID tool

Supporting ATARAID devices via the generic Linux device-mapper

Heinz Mauelshagen
Red Hat Cluster and Storage Development
mauelshagen@redhat.com

Abstract

Device-mapper, the new Linux 2.6 kernel generic device-mapping facility, is capable of mapping block devices in various ways (e.g. linear, striped, mirrored). The mappings are implemented in runtime loadable plugins called mapping targets.

These mappings can be used to support arbitrary software RAID solutions on Linux 2.6, such as ATARAID, without the need to have a special low-level driver as it used to be with Linux 2.4. This avoids code-redundancy and reduces error rates.

Device-mapper runtime mappings (e.g. map sector N of a mapped device onto sector M of another device) are defined in mapping tables.

The dmraid application is capable of creating these for a variety of ATARAID solutions (e.g. Highpoint, NVidia, Promise, VIA). It uses an abstracted representation of RAID devices and RAID sets internally to keep properties such as paths, sizes, offsets into devices and layout types (e.g. RAID0). RAID sets can be of arbitrary hierarchical depth in order to reflect more complex RAID configurations such as RAID10.

Because the various vendor specific metadata formats stored onto ATA devices by the ATARAID BIOS are all different, metadata format handlers are used to translate between the ondisk representation and the internal abstracted format.

The mapping tables which need to be loaded into device-mapper managed devices are derived from the internal abstracted format.

My talk will give a device-mapper architecture/feature overview and elaborate on the dmraid architecture and how it uses the device-mapper features to enable access to ATARAID devices.

1 ATARAID

Various vendors (e.g. Highpoint, Silicon Image) ship ATARAID products to deploy Software RAID (Redundant Array Of Inexpensive Disks) on desktop and low-end server system. ATARAID essentially can be characterized as:

- 1-n P/SATA or SCSI interfaces
- a BIOS extension to store binary RAID set configuration metadata on drives attached
- a BIOS extension to map such RAID sets in early boot and access them as a single device so that booting off them is enabled

- an operating system driver (typically for Windows) which maps the RAID sets after boot and updates the vendor specific metadata on state changes (e.g. mirror failure)
- a management application to deal with configuration changes such as mirror failures and replacements

Such ATARAID functionality can either be provided via an additional ATARAID card or it can be integrated on the mainboard as with solutions from NVidia or VIA. It enables the user to setup various RAID layouts, boot off them and have the operating system support them as regular block devices via additional software (hence the need for Windows drivers). Most vendors do RAID0 and RAID1, some go beyond that by offering concatenation, stacked RAID sets (i.e. RAID10) or higher RAID levels (i.e. RAID3 and RAID5).

1.1 Some metadata background

The vendor on-disk metadata keeps information about:

- the size(s) of the areas mapped onto a disk
- the size of the RAID set
- the layout of the RAID set (e.g. RAID1)
- the number of drives making up the set
- a unique identifier (typically 1 or 2 32 bit numbers) for the set
- the state of the set (e.g. synchronized for RAID1) so that the driver can start a resynchronization if necessary

What it usually doesn't keep is a unique human readable name for the set which means, that dmraid needs to derive it from some available unique content and make a name up. The tradoff is, that names are somewhat recondite.

Some vendors (i.e. Intel) retrieve ATA or SCSI device serial numbers and store them in the metadata as RAID set identifiers. Others just make unique numbers using random number generators.

1.2 Support in Linux

Linux 2.4 supported a limited list of products via ATARAID specific low-level drivers.

Now that we have the very flexible device-mapper runtime in the Linux 2.6 kernel series, this approach is no longer senseful, because an application (i.e. dmraid) can translate between all the different on-disk metadata formats and the information device-mapper needs to activate access to ATRAID sets. This approach avoids the overhead of seperate low-level drivers for the different vendor solutions in the Linux kernel completely.

2 Device-Mapper

The device-mapper architecture can be delineated in terms of these userspace and kernel components:

- a core in the Linux kernel which maintains mapped devices (accessible as regular block devices) and their segmented mappings definable in tuples of offset, range, target, and target-specific parameters. Offset and ranges are in units of sectors of 512 bytes. Such tuples are called targets (see examples below). An arbitrary length list of targets defining segments in the logical address space of a mapped device make up a device mapping table.

- a growing list of kernel modules for plugable mapping targets (e.g. linear, striped, mirror, zero, error, snapshot, cluster mapping targets...) which are responsible for (re)mapping IOs to a sector address range in the mapped devices logical address space to underlying device(s) (e.g. to mirrors in a mirror set).
- an ioctl interface module in the kernel to communicate with userspace which exports functionality to create and destroy mapped devices, load and reload mapping tables, etc.
- a device-mapper userspace library (libdevmapper) which communicates with the kernel through the ioctl interface accessing the functions to create/destroy mapped devices and load/reload their ASCII formated mapping tables. This library is utilized by a couple applications such as LVM2 and dmraid.
- a dmsetup tool which uses libdevmapper to manage mapped devices with their mapping tables and show supported mapping targets, etc.

2.1 Mapping table examples

1. 0 1024 linear /dev/sda 0
2. 0 2048 striped 2 64 /dev/sda 1024 /dev/sdb 0
3. 0 4711 mirror core 2 64 nosync 2 /dev/sda 2048 /dev/sdb 1024
4. 0 3072 zero
   3072 1024 error

Example 1 maps an address range (segment) starting at sector 0, length 1024 sectors linearly (*linear* is a keyword selecting the linear mapping target) onto /dev/sda, offset 0. `/dev/sda 0` (a device path and an offset in sectors) are the 2 target-specific parameters required for the linear target.

Example 2 maps a segment starting at sector 0, length 2048 sectors striped (the *striped* keyword selects the striping target) onto /dev/sda, offset 1024 sectors and `/dev/sdb`, offset 0 sectors. The striped target needs to know the number of striped devices to map to (i.e. '2') and the stride size (i.e. '64' sectors) to use to split the IO requests.

Example 3 maps a segment starting at sector 0, length 4711 sectors mirrored (the *mirror* keyword selects the mirror target) onto `/dev/sda`, offset 2048 sectors (directly after the striped mapping from before) and `/dev/sdb`, offset 1024 sectors (after the striped mapping).

Example 4 maps a segment starting at sector 0, length 3072 sectors using the 'zero' target, which returns success on both reads and writes. On reads a zeroed buffer content is returned. A segment beginning at offset 3072, length 1024 gets mapped with the 'error' target, which always returns an error on reads and writes. Both segments map a device size of 4096 sectors.

As the last example shows, each target line in a mapping table is allowed to use a different mapping target. This makes the mapping capabilities of device-mapper very flexible and powerfull, because each segent can have IO optimized properties (e.g. more stripes than other segments).

Note: Activating the above mappings at once is just for the purpose of the example.

2.2 dmsetup usage examples

By putting arbitrary mapping tables like the above ones into files readable by the dmsetup tool (which can read mapping tables from standard input as well), mapped devices can be created or removed and their mappings can be loaded or reloaded.

1. dmsetup create ols filename
2. dmsetup reload ols another_filename
3. dmsetup rename ols OttawaLinuxSymposium
4. dmsetup remove OttawaLinuxSymposium

Example 1 creates a mapped device named 'ols' in the default device-mapper directory `/dev/mapper/`, loads the mapping table from 'filename' (e.g. '0 3072 zero') and activates it for access.

Example 2 loads another mapping table from 'another_filename' into 'ols' replacing any given previous one.

Example 3 renames mapped device 'ols' to 'OttawaLinuxSymposium'.

Example 4 deactivates 'OttawaLinuxSymposium', destroys its mapping table in the kernel, and removes the device node.

3 dmraid

The purpose of dmraid is to arbitrate between the ATARAID on-disk metadata and the device-mapper need to name mapped devices and define mapping tables for them.

Because the ATARAID metadata is vendor specific and the respective formats therefore all differ, an internal metadata format abtraction is necessary to translate into and derive the mapped device names and mapping tables content from.

The 'translators' between the vendor formats and the internal format are called 'metadata format handlers.' One of them is needed for any given format supported by dmraid.

An activation layer translates from there into mapping tables and does the libdevmapper calls to carry out device creation and table loads to gain access to RAID sets.

3.1 dmraid components

- the dmraid tool which parses the command line and calls into
- the dmraid library with:
  - a device access layer to read and write metadata from/to RAID devices and to retrieve ATA and SCSI serial numbers
  - a metadata layer for the internal metadata format abstraction with generic properties to describe RAID devices and RAID sets with their sizes and offsets into devices, RAID layouts (e.g. RAID1) including arbitrary stacks of sets (e.g. RAID10)
  - metadata format handlers for every vendor specific solution (e.g. Highpoint, NVidia, VIA, ...) translating between those formats and the internal generic one
  - an activation layer doing device-mapper library calls
  - a display layer to show properties of block devices, RAID devices and RAID sets
  - a logging layer which handles output for verbosity and debug levels
  - a memory management layer (mainly for debugging purposes)
  - a locking layer to prevent parallel dmraid runs messing with the metadata

3.2 Command line interface

The dmraid CLI comprehends options to:

- activate or deactivate ATARAID sets
- select metadata formats
- display properties of
  - block devices
  - RAID devices
  - RAID sets
  - vendor specific metadata
- display help (command synopsis)
- list supported metadata formats
- dump vendor metadata and locations into files
- display the dmraid, dmraid library and the device-mapper versions

The command synopsis looks like:

```
dmraid: Device-Mapper Software RAID tool

* = [-d|--debug]... [-v|--verbose]...

dmraid  {-a|--activate} {y|n|yes|no} *
        [-f|--format FORMAT]
        [-p|--no_partitions]
        [-t|--test]
        [RAID-set...]

dmraid  {-b|--block_devices} *
        [-c|--display_columns]...

dmraid  {-h|--help}

dmraid  {-l|--list_formats} *

dmraid  {-n|--native_log} *
        [-f|--format FORMAT]
        [device-path...]

dmraid  {-r|--raid_devices} *
        [-c|--display_columns]...
        [-D|--dump_metadata]
        [-f|--format FORMAT]
        [device-path...]

dmraid  {-r|--raid_devices} *
        {-E|--erase_metadata}
        [-f|--format FORMAT]
        [device-path...]

dmraid  {-s|--sets}...[a|i|active|inactive] *
        [-c|--display_columns]...
        [-f|--format FORMAT]
        [-g|--display_group]
        [RAID-set...]

dmraid  {-V/--version}
```

3.3 dmraid usage examples

List all available block devices:

```
# dmraid -b
/dev/sda:     72170879 total, "680631431K"
/dev/sdb:      8887200 total, "LG142316"
/dev/sdc:     72170879 total, "680620811K"
```

List all discovered RAID devices:

```
# dmraid -r
/dev/dm-14: hpt45x, "hpt45x_dbagefdi", \
 stripe, ok, 320172928 sectors, data@ 0
/dev/dm-18: hpt45x, "hpt45x_dbagefdi", \
 stripe, ok, 320172928 sectors, data@ 0
/dev/dm-22: hpt45x, "hpt45x_bhchfdeie", \
 mirror, ok, 320173045 sectors, data@ 0
/dev/dm-26: hpt45x, "hpt45x_bhchfdeie", \
 mirror, ok, 320173045 sectors, data@ 0
/dev/dm-30: hpt45x, "hpt45x_edieecfd", \
 linear, ok, 320173045 sectors, data@ 0
/dev/dm-34: hpt45x, "hpt45x_edieecfd", \
 linear, ok, 320173045 sectors, data@ 0
/dev/dm-38: hpt45x, "hpt45x_chidjhaiaa-0", \
 stripe, ok, 320172928 sectors, data@ 0
/dev/dm-42: hpt45x, "hpt45x_chidjhaiaa-0", \
 stripe, ok, 320172928 sectors, data@ 0
/dev/dm-46: hpt45x, "hpt45x_chidjhaiaa-1", \
 stripe, ok, 320172928 sectors, data@ 0
/dev/dm-50: hpt45x, "hpt45x_chidjhaiaa-1", \
 stripe, ok, 320172928 sectors, data@ 0
```

List all discovered RAID sets:

```
# dmraid -cs
hpt45x_dbagefdi
hpt45x_bhchfdeie
hpt45x_edieecfd
hpt45x_chidjhaiaa
```

Show mapped devices and mapping tables for RAID sets discovered:

```
# dmraid -tay
hpt45x_dbagefdi: 0 640345856 striped \
 2 128 /dev/dm-14 0 /dev/dm-18 0
hpt45x_bhchfdeie: 0 320173045 mirror \
 core 2 64 nosync 2 /dev/dm-22 0 /dev/dm-26 0
hpt45x_edieecfd: 0 320173045 \
 linear /dev/dm-30 0
hpt45x_edieecfd: 320173045 320173045 \
 linear /dev/dm-34 0
hpt45x_chidjhaiaa-0: 0 640345856 striped \
 2 128 /dev/dm-38 0 /dev/dm-42 0
hpt45x_chidjhaiaa-1: 0 640345856 striped \
 2 128 /dev/dm-46 0 /dev/dm-50 0
hpt45x_chidjhaiaa: 0 640345856 mirror \
 core 2 256 nosync 2 \
 /dev/mapper/hpt45x_chidjhaiaa-0 0 \
 /dev/mapper/hpt45x_chidjhaiaa-1 0
```

Activate particular discovered RAID sets:

```
# dmraid -ay hpt45x_dbagefdi hpt45x_bhchfdeie
```

3.4 Testbed

It is too costly to keep a broad range of ATARAID products in a test environment for regression tests. This would involve plenty of different ATARAID cards and ATARAID-equipped mainboards. Even worse, multiple of each of those would be needed in order to keep various configurations they support acessible for tests in parallel (e.g. Highpoint 47x type cards support RAID0, RAID1, RAID 10 and drive concatination). Not to mention the amount of disks needed to cover those *and* a couple of different sizes for each layout. For the formats already supported by dmraid, the costs easily sum up to a couple of USD 10K.

Because of that, the author created a testbed which utilizes device-mapper to 'fake' ATARAID devices via sparse mapped devices (that's why the examples above show `/dev/dm-*` device names). A sparse mapped device is a stack of a zero and a snapshot mapping on top. The snapshot redirects all writes to the underlying device and keeps track of those redirects while allowing all reads to not-redirected areas to hit the underlying device. In case of the zero mapping, success and a zeroed buffer will be returned to the application. The space where the snapshot redirects writes to (called exception store) can be way smaller than the size of the zero device. That in turn allows the creation of much larger sparse than available physical storage.

The testbed is a directory structure containing subdirectory hierarchies for every vendor, adaptor type and configuration (images of the metadata on each drive and drive size). The top directory holds setup and remove scripts to create and tear down all sparse mapped devices for the drives invloved in the configuration which get called from configuration directory scripts listing them.

A typical subdirectory (e.g. `/dmraid/ataraid.data/hpt/454/raid10`) looks like:

hde.size hdg.size hdi.size hdk.size
hde.dat hdg.dat hdi.dat hdk.dat
setup remove

`dmraid -rD` is able to create the .dat and .size files for supported formats for easy addition to the testbed. Users only need to tar those up and send them to the author on request.

4 dmraid status and futures

dmraid and device-mapper are included in various distributions such as Debian, Fedora, Novell/SuSE, and Red Hat.

Source is available at
`http://people.redhat.com/`

`heinzm/sw/dmraid`
for dmraid and
`http://sources.redhat.com/dm`
for device-mapper.

The mailing list for information exchange on ATARAID themes including dmraid is `ataraid-list@redhat.com`. If you'd like to subscribe to that list, please go to `https://www.redhat.com/mailman/listinfo/ataraid-list`.

Work is in progress to add Fedora installer support for dmraid and device-monitoring via an event daemon (dmeventd) and a libdevmapper interface extension to allow registration of mapped devices for event handling. dmeventd loads application specific dynamic shared objects (e.g. for dmraid or lvm2) and calls into those once an event on a registered device occurs. The DSO can carry out appropriate steps to change mapped device configurations (e.g. activate a spare and start resynchronization of the mirrored set).

Additional metadata format handlers will be added to dmraid including one for SNIA DDF.

The author is open for any proposals which other formats need supporting...

Usage of Virtualized GNU/Linux for Binary Testing Across Multiple Distributions

Gordon McFadden
Intel corporation
gordon.mcfadden@intel.com

Michael Leibowitz
Intel Corporation
michael.leibowitz@intel.com

Abstract

In this paper, we will discuss how we created a test environment using a single high-end test host that implemented multiple test hosts. The test environment enabled the testing of software running on different Linux distributions with different kernel versions. This approach improved test automation, avoided capital expenditures and saved on desktop real-estate. We employed a version of Gentoo Linux with a modified 2.6 kernel, along with multiple instances of different distributions and version of Linux running on User Mode Linux (UML). The particular tests involved are related to the Linux Standards Base, but the concept is applicable to many different environments.

We will describe how we improved aspects of the Gentoo kernel to improve performance. We will describe the methods used to affect a lightweight inter UML communications mechanism. We will also talk about the file systems chosen for both the host OS and the UML. Finally, we will have a brief discussion around the benefits and limitations of this type of test environment, and will discuss plans for future test environments.

1 Introduction

While setting up a test environment to execute tests to verify compliance of various Linux distributions as part of the testing of the Linux Standard Base (LSB) [1] 3.0 Specification, it was noted that both time and capital expense could be saved if the host running the tests could be effectively reused.

In the context of executing LSB conformance tests, it is the cass that many of the tests can be readily executed in an automated and autonoumous manner. Only some tests are manual in nature and require the attendance of a test operator. It was also noted that the tests require different distributions, including different kernel versions.

Given the fact that the test cycle was not expected to last indefinitely, and that the number of distributions under test was likely to increase, it did not make sense to attempt to allocate one host to each distribution.

Additionally, it was important to the test philosophy that the distributions be available at all times, allowing tests to run independently of each other. If a multi-boot system, such as GRUB or LILO were employed, then testing could only proceed sequentially.

Another required aspect of the test environment

is the ability to instantiate tests without impacting other running tests.

The solution that is employed is the use of a host operating system running Guest Operating Systems (GOS) in User Mode Linux (UML) [2]. Gentoo [3] release 2.6.11-R-6 was chosen as the host operating system because it is very configurable in the areas of file systems, how many process are running and other areas. The intent is to keep the host of installed software on the host operating system very small. It is very easy to install a minimal set of packages in a GenToo build. While any distribution provides the ability to configure installed packages, and allows modifications to the kernel, the GenToo distributions seems to be geared toward allowing installers to make the types of modifications needed to encompass the solution.

It makes a great deal of sense from the perspective of cost and space to arrange the test environment to use one host per architecture.

1.1 Changes to the kernel

1.1.1 Elevators

The processes of optimizing the kernel started with the 2.6.11.6-vanilla Linux Kernel and involved modifications to the elevator to increase performance of spawning UML instances. The Linux kernel implements a disk I/O scheduling system referred to as the elevator. The name elevator comes from the conceptual model of the disk drive as a linear array with a single read/write head. The head moves up and down the disk, as an elevator moves in an elevator shaft, and the blocks that are read or written to as the call buttons on various floors. As in the real world, the algorithm for moving the elevator in response to floor requests is a non-trivial dining philosophers type of problem. Responsiveness, repeatability, equity, and aggregate bandwidth must all be carefully balanced. No algorithm can always maximize all of these needs, but any suitable algorithm should be able to avoid starvation in all circumstances. User Mode Linux also has an elevator, which operates in the same way as the host elevator does. Because several elevators may be in use on multiple encapsulated operating systems in parallel (see Figure 1), they can effectively "collude" to starve one or more processes of disk access. The elevator of the host kernel was modified to better deal with this situation.

1.2 File System Issues

XFS was chosen as the host file system. When XFS was devised by SGI, it was designed to be able to give high throughput for media applications. Filesystem-within-filesystem applications are similar to media, in that both involve contiguous large files that are accessed in regular ways. In non-linear editing applications, files are written to and read not in a strictly linear fashion, but in large linear blocks. File system access from an guest operating system in similar due to the elevator inside the encapsulated kernel.

1.3 Execution Environment

There are two modes of executing kernels in a UML environment. The first is refered to as Tracing Threads (TT) mode. The second mode is Separate Kernel Address Space (SKAS) mode. In the TT mode, the processes and the kernel of the GOS all exist in the user space of the host kernel. In SKAS mode, the kernel is mapped into its own address space. The advantage to Tracing Thread mode is that there is support for Symetric Multi-Processor

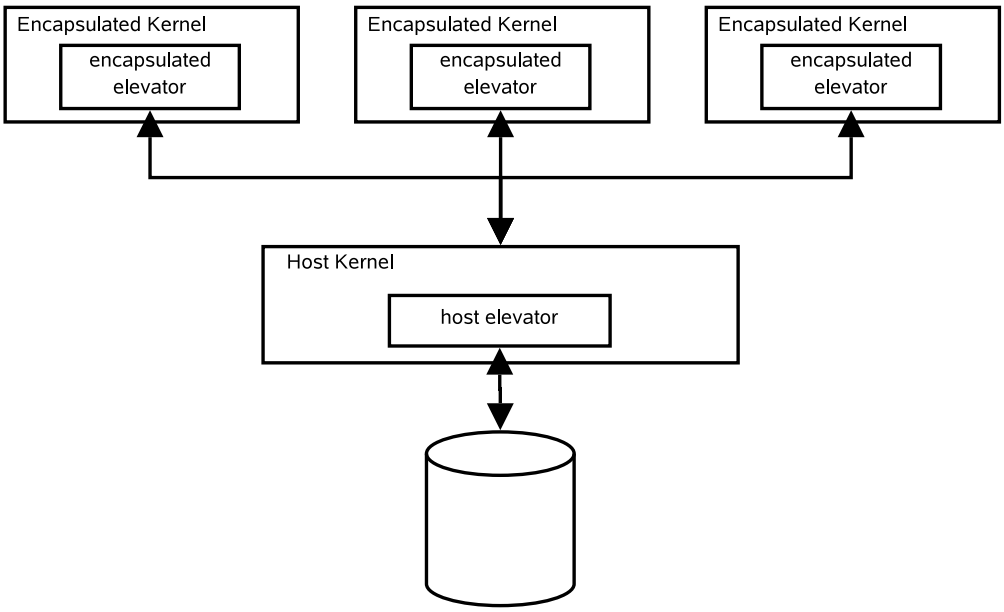


Figure 1: Nested Elevators

(SMP) based platforms. The most compelling reason to consider using the SKAS mode is the perfomance advatage it holds over TT mode. This advanage is most noticable in applications that are fork() intensive.

To work in SKAS mode requires a minor patch to the host OS kernel. This patch was examined and it was determined that for the purposes of the specific LSB tests being considered, the patch did not affect the viability of the test.

It was not nessesary to have SMP available to the GOS kernels in order to run the tests, and the host kernel effectively makes use of the SMP platform. Therefore, the desision was made to employ the SKAS patch.

1.4 Distributions Tested

The following represents a sample of the distributions required for the LSB 3.0 testing:

- Novell Linux Distribution 10
- RedHat Enterprise Linux 3
- RedHat Enterprise Linux 4
- Red Flag

1.5 Intra-UML communications

In the deployed environment, even though each GOS has it own IP address and stack and is connecte via a virtual switch, there was no requirement for communications between individual GOSs. In the future, it is foreseen that extending the test environment to allow Client/Server style tests on separate GOSs may provide value. In this environment, GOSs could communicate in one of three methods. First of all, given the fact that there is an IP stack running on each GOS, then socket based communications are available. This would include direct sockets, ssh, ftp, rsh, and other well known IP-based communications methods. Second, it would also be possible to use semaphore files between GOSs in a manner similar to that described in this paper. Finally, it is theoretically possible

to attach TTY/PTY devices between GOSs, allowing character based traffic to be passed between two GOSs. This approach would have very little overhead, and may be very attractive as a management conduit for test control. More research is needed in this area.

2 Concurrent Test Limitations

It is important to understand the limitations that exist when running encapsulated or virtualized test environments. These limitations associated with running concurrent tests on a UML based system include:

- hardware abstraction—it may not be appropriate to test the hardware and hardware abstraction layers since some aspects of the encapsulated operating systems are abstraced. Example of this are the apparent memory size of which the encapsulated system is aware, the block I/O systems, etc.
- resource sharing—it is possible for a test to have different resources available for different invocations of the test. This may produce different results in the area of execution time, CPU utilization, and other similar measurements.
- inter-client communication—it adds value to the test cycle for the host operating system to be able to communicate with the guest operating systems for the purpose of kicking off tests and recovering test results. In the future it may also be useful to enable communication between encapsulated systems.

For the purposes of the LSB testing, these limitations are not onerous, and the test environment is sensible.

3 The Cost of UML

As with any system of emulation, encapsulation, or virtualization, there is some performance penalty to be expected. Because the LSB compliance testing takes such a long time to execute, two synthetic benchmarks were chosen that isolate particular areas of system performance that have a large impact on our tests. Of principal interest is file system performance and scheduler performance.

3.1 File System Test results

One of the benchmark tools employed was Bonnie++ (1.03a) a widely accepted file system throughput test. The Bonnie++ benchmark was used the in the development of the ReiserFS file system. Even though a HyperThreaded machine is used as the test host, it was decided to use single threaded mode for bonnie++ because the primary interest is in the performance that *one* process would receive, rather than trying to approximate a full running system in some way. Bonnie++ was configured to choose the set size, which for the host was 2G. The encapsulated operating systems have varying quantities of memory, so the same set size as the host was not used. The intent of this study was to compare file system performance in the encapsulated operating systems, rather than comparing encapsulated performance to the host. See Table 1 for details on file system throughput.

3.2 Scheduler Performance

Because the LSB tests require a large number of sequential and concurrent operations, scheduler performance inside the encapsulated operating systems is of interest. There are two factors of interest here, the process creation overhead and the context switching time. To measure the former, the `spawn` test program of the

	Sequential Output			Sequential Input	
	Per Char	Block	Rewrite	Per Char	Block
Host Kernel	KB/s	KB/s	KB/s	KB/s	KB/s
2.6.11.6-skas3-v9-pre1	30775	64928	22433	14864	53974
2.6.11.6	30726	65639	22906	15159	54649
	.16%	–1.08%	–2.06%	–1.95%	–1.24%

Table 1: Host disk throughput comparison

unixbench-4.1.0 test suite was used. To measure the latter, the context switching measurements of the lmbench-3.0-a4 test suite was observed.

4 Testing on UML

One of the factors that influenced the design of the test environment was the relative execution time of the tests in questions. Generally, the compliance tests take in the order of an hour to execute. The Application Battery suite of tests for LSB certification takes in the order of 3 hours per distribution, and is a very manual operation. The validation of the Sample Implementation takes about 30 minutes. The full testing of the distribution using the runtime library is documented to take approximately seven hours on a uni-processor host.

Since the tests take so long to execute, there is no need to launch the tests instantly. If it takes one or two seconds to cause the testing to begin, this will cause no appreciable difference in overall test execution time. Accordingly, a file based system was developed based on an NFS file system. Each GOS exports a directory to be used for testing. The host OS mounts a directory for each GOS. The fact that test take a long time to execute also means that that the residual files will persist for quite some time. It is for this reason that `.ini` and `.fini` files are used in the scripts to indicate when a test is ready to start, and when it has completed. This approach also allows multiple tests to be run concurrently.

The appropriateness of running concurrent tests must be determined by examining the many aspects associated with backgrounding tests. It would be possible for one GOS to over consume CPU and disk resources by instantiating many tests. The GOS side of the test environment does not put any restrictions on the number of concurrent tests that may be run.

For a general purpose test environment, it would not be difficult to modify the scripts so they kept track of the number of outstanding test—those actively executing tests—and throttle the arrival rate of tests according to some high water mark.

4.1 Launching a test

Table 2 describes the steps taken by the host OS to launch a test on GOS 2.

For the purposes of this example, it is assumed the test to be run exists in local directory `/opt/test1` in the form of an executable and some supporting files. The tests are to be executed on GOS #2.

Note that the executable test and any supporting data must be transferred before the `.ini` file used to kick off the test is created.

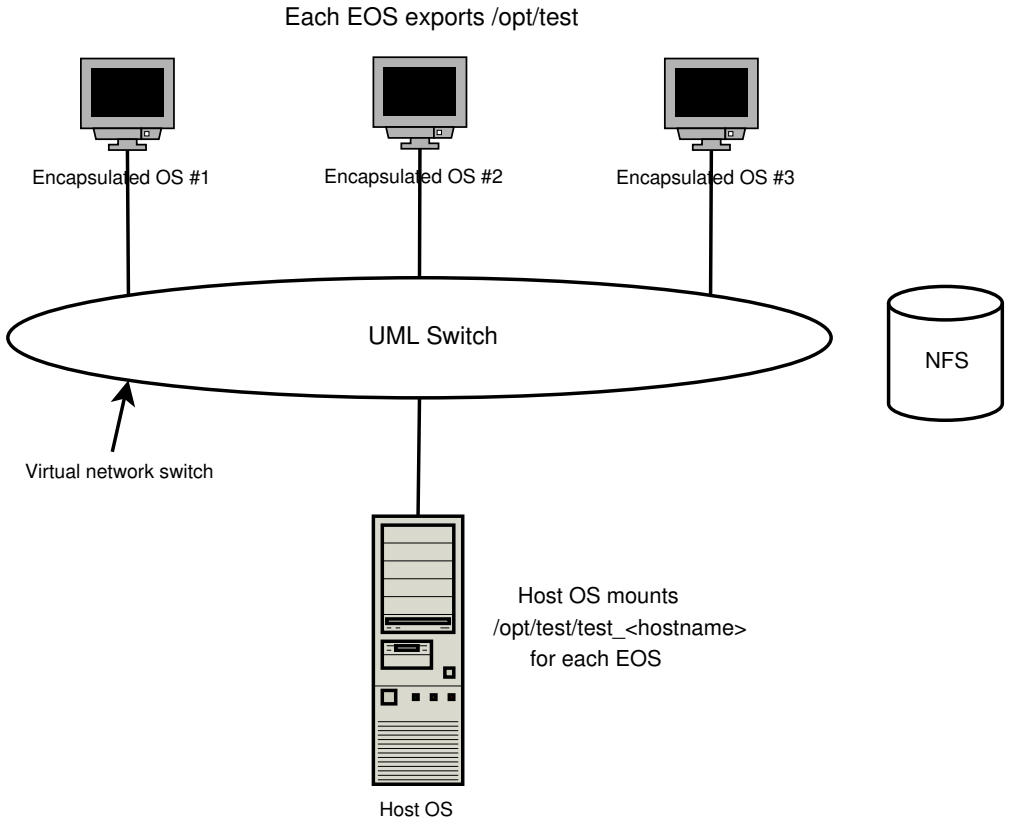


Figure 2: UML Figure

```
 $ host: ls /opt/test1
CVS bin result data

 $ host: ls /mnt/GOS2
 $ host: mkdir /mnt/GOS2/bin
 $ host: mkdir /mnt/GOS2/results
 $ host: mkdir /mnt/GOS2/data
 $ host: cp -R /opt/test1/data /mnt/GOS2/data
 $ host: cp /opt/test1/bin/test_exec /mnt/GOS2/bin/test_exec
 $ host: touch /mnt/GOS2/bin/test_exec.ini
```

Table 2: Script Used to Launch Tests

4.2 Test execution

On each GOS, there is a script running that periodically checks for the existence of a test. The script is presented in Table 3.

If can be noted that this script tests for the existences of `.ini` files in the `.bin` directory, executes the tests redirecting stdin and stderr to files based on the test name in a results directory. When the test has completed, the script create a `.fini` file, which is a flag to indicate the test is complete.

4.3 Getting results

Obtaining the results of the test are reasonably trivial. The test application on the host OS waits for the creation of a `.fini` file in the results directory. Once this empty file is created, then the stdout and stderr of the test can be evaluated to determine the success of failure of the test. If the test generates any log files, then these too can be evaluated.

5 Futures

One of the major drawbacks of the method employed in the test environment described in this paper is the fact that the system resources are not protected. The memory associated with one encapsulated may be partially swapped out in the host operating system. Disk file systems need to be carefully planned, and can not change with out adversely affecting disk performance. The overhead of the host OS also reduces the resources available to the encapsulating OSes.

A solution that addresses the shortcoming is the newly released Virtualization Technology (VT) platform. This platform would a allowed a much faster deployment of the test environment. It also has a much faster transition between guest operating systems due to the hardware assisted switching. A VT platform also allows the operating systems that run on them to have protected hardware resources.

Although not available for this test environment, UML is being ported to a VT technology platform, allowing a more efficient use of system resources from within an GOS. UML is also being ported to x86-64 architures, as well as PPC and S390 processors.

As the testing for the Linux Standards Base continues, it is fully anticipated that this test environment will be migrated to a VT-enabled platform in the very near term.

6 Conclusion

The test environment described in this paper was designed to facilitate simultaneous or near simultaneous testing of different distributions. The nature of the testing involved was well suited for the type of environment available from a UML based test platform. Performance slowdowns were not an issue.

7 Acknowledgments

The authors would like to express their thanks to Jeffery Dike, one of the developers and maintainers of UML, for taking the time to answer questions and for providing information on the future of UML.

```
#!/bin/sh

export TD=/mnt/test
while [ 1 -gt 0 ] ; do
  for test in ‘ls $TD/bin/*.ini‘ ; do
    echo TEST is $test
    export fex=‘basename $test .ini‘
    echo $fex
    if [ -x $TD/bin/$fex ] ; then
      ($TD/bin/$fex > $TD/results/$fex.out 2> $TD/results/$fex.err;
        touch $TD/results/$fex.fini;
        rm $TD/bin/$fex) &
    fi
    rm $test
  done
  sleep 5
done
```

Table 3: Script Used to Execute Tests

8 References

[1] Linux Standard Base at
`http://www.linuxbase.org/`

[2] Read more about User Mode Linux at
`http://usermodelinux.org/`

[3] Read more about Gentoo at
`http://www.gentoo.org/`

DCCP on Linux

Arnaldo Carvalho de Melo
Conectiva S.A.
acme@{conectiva.com.br,mandriva.com,ghostprotocols.net}

Abstract

In this paper I will present the current state of DCCP for Linux, looking at several implementations done for Linux and for other kernels, how well they interoperate, how the implementation I'm working on took advantage of the work presented in my OLS 2004 talk ("TCP-fying the poor cousins") and ideas about plugabble congestion control algorithms in DCCP, taking advantage of recent work by Stephen Hemminger on having a IO scheduller like infrastructure for congestion control algorithms in TCP.

1 What is DCCP?

The Datagram Congestion Control Protocol is a new Internet transport protocol to provide unreliable, congestion controlled connections, providing a blend of characteristics not available in other existing protocols such as TCP, UDP, or SCTP.

There has been concern that the increasing use of UDP in application such as VoIP, streaming multimedia and massively online games can cause congestion collapse on the Internet, so DCCP is being designed to provide an alternative that frees the applications from the complexities of doing congestion avoidance, while providing a core protocol that can be extended with new congestion algorithms, called CCIDs, that can be negotiated at any given time in a connection lifetime, even with different algorithms being used for each direction of the connection, called Half Connections in DCCP drafts.

This extensibility is important as there are different sets of requirements on how the congestion avoidance should be done, while some applications may want to grab as much bandwidth as possible and accept sudden drops when congestion happens others may want not to be so greedy but have fewer oscillations in the average bandwidth used through the connection lifetime.

Currently there are two profiles defined for DCCP CCIDs: CCID2, TCP-Like Congestion Control[3], for those applications that want to use as much as possible bandwidth and are able to adapt to sudden changes in the available bandwidth like those that happens in TCP's Additive Increase Multiplicative Decrease (AIMD) congestion control; and CCID3, TCP Friendly Congestion Control (TFRC)[4], that implements a receiver-based congestion control algorithm where the sender is rate-limited by packets sent by the receiver with information such as receive rate, loss intervals, and the time packets were kept in queues before being acknowledged, indented for applications that want a smooth rate.

There are a number of RFC drafts covering aspects of DCCP to which interested people should refer for detailed information about the many aspects of this new protocol, such as:

- *Problem Statement for DCCP[1]*
- *Datagram Congestion Control Protocol (DCCP)[2]*
- *Profile for DCCP Congestion Control ID 2[3]*
- *Profile for DCCP Congestion Control ID 3[4]*
- *Datagram Congestion Control Protocol (DCCP) User Guide[5]*
- *DCCP CCID 3-Thin[6]*
- *Datagram Congestion Control Protocol Mobility and Multihoming[7]*
- *TCP Friendly Rate Control (TFRC) for Voice: VoIP Variant and Faster Restart[8]*

This paper will concentrate on the the current state of the author's implementation of DCCP and its CCIDs in the Linux Kernel, without going too much into the merits of DCCP as a protocol or its adequacy to any application scenario.

2 Implementations

DCCP has been a moving target, already in its 11th revision, with new drafts changing protocol aspects that have to be tracked by the implementators, so while there has been several implementations written for Linux and the BSDs, they are feature incomplete or not compliant with latest drafts.

Patrick McManus wrote an implementation for the Linux Kernel version 2.4.18, Implementing only CCID2, TCP-Like Congestion Control, but has not updated it to the latest specs and also has bitrotted, as the Linux kernel networking core has changed in many aspects in 2.6.

Another implementation was made for FreeBSD at the Luleå University of Technology, Sweden, that is more complete, implementing even the TFRC CCID. This implementation has since been merged in the KAME Project codebase, modified with lots of ifdefs to provide a single DCCP code base for FreeBSD, NetBSD, and OpenBSD.

The WAND research group at the University of Waikato, New Zealand also has been working on a DCCP implementation for the Linux kernel, based on the stack written by Patrick McManus, combining it with the the Luleå FreeBSD CCID3 implementation.

The DCCP home page at ICIR also mentions a user-level implementation written at the Berkeley University, but the author was unable to find further details about it.

The implementation the author is writing for the Linux Kernel is not based on any of the DCCP core stack implementations mentioned, for reasons outlined in the "DCCP on Linux" section later in this paper.

3 Writing a New Protocol for the Linux Kernel

Historically when new protocols are being written for the Linux kernel existing protocols are used as reference, with code being copied to accelerate the implementation process.

While this is a natural way of writing new code it introduces several problems when the reference protocols and the core networking infrastructure is changed, these problems were

discussed in my "TCPfying the poor Cousins" [10] paper presented in 2004 at the Linux Symposium, Ottawa.

This paper will describe the design principles and the refactorings done to the Linux kernel networking infrastructure to reuse existing code in the author's DCCP stack implementation to minimise these pitfalls.

4 DCCP on Linux

The next sections will talk about the design principles used in this DCCP implementation, using the main data structures and functions as a guide, with comments about its current state, how features were implemented, sometimes how missing features or potential DCCP APIs that are being discussed in the DCCP community could be implemented and future plans.

5 Design Principles

1. Make it look as much as possible as TCP, same function names, same flow.
2. Generalise as much as possible TCP stuff.
3. Follow as close as possible the pseudocode in the DCCP draft[2], as long as it doesn't conflicts with principle 1.
4. Any refactoring to existing code (TCP, etc.) has to produce code that is as fast as the previous situation—if possible faster as was the case with TCP's `open_request` generalization, becoming `struct request_sock`. Now TCP v4 syn minisocks use just 64 bytes, down from 96 in stock Linus tree; lmbench shows performance improvements.

Following these principles the author hopes that the Linux TCP hackers will find it easy to review this stack, and if somebody thinks that all these generalisations are dangerous for TCP, so be it, its just a matter of reverting the TCP patches and leaving the infrastructure to be used only by DCCP and in time go on slowly making TCP use it.

6 Linux Infrastructure for Internet Transport Protocols

It is important to understand how the Linux kernel internet networking infrastructure supports transport protocols to provide perspective on the refactorings done to better support a DCCP implementation.

A `AF_INET` transport protocol uses the `inet_add_protocol` function so that the IP layer can feed it packets with its protocol identifier as present in the IP header, this function receives the protocol identifier and a `struct net_protocol` where there has to be a pointer for a function to handle packets for this specific transport protocol.

The transport protocol also has to use the `inet_register_protosw` function to tell the inet layer how to create new sockets for this specific transport protocol, passing a `struct inet_protosw` pointer as the only argument, DCCP passes this:

```
struct inet_protosw
dccp_v4_protosw = {
.type     = SOCK_DCCP,
.protocol= IPPROTO_DCCP,
.prot     = &dccp_v4_prot,
.ops      = &inet_dccp_ops,
};
```

So when applications use `socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP)` the

inet infrastructure will find this struct and set `socket->ops` to `inet_dccp_ops` and `sk->sk_prot` to `dccp_v4_prot`.

The `socket->ops` pointer is used by the network infrastructure to go from a syscall to the right network family associated with a socket, DCCP sockets will be reached through this struct:

```
struct proto_ops inet_dccp_ops = {
.family    = PF_INET,
.owner     = THIS_MODULE,
.release   = inet_release,
.bind      = inet_bind,
.connect   = inet_stream_connect,
.socketpair= sock_no_socketpair,
.accept    = inet_accept,
.getname   = inet_getname,
.poll      = sock_no_poll,
.ioctl     = inet_ioctl,
.listen    = inet_dccp_listen,
.shutdown  = inet_shutdown,
.setsockopt=
sock_common_setsockopt,
.getsockopt=
sock_common_getsockopt,
.sendmsg   = inet_sendmsg,
.recvmsg   = sock_common_recvmsg,
.mmap      = sock_no_mmap,
.sendpage  = sock_no_sendpage,
};
```

Looking at this struct we can see that the DCCP code shares most of the operations with the other `AF_INET` transport protocols, only implementing the `.listen` method in a different fashion, and even this method is to be shared, as the only difference it has with `inet_listen`, the method used for TCP is that it checks if the socket type is `SOCK_DGRAM`, while `inet_listen` checks if its `SOCK_STREAM`.

Another point that shows that this stack is still in development is that at the moment it doesn't supports some ot the `struct proto_ops` methods, using stub routines that return appropriate error codes.

One of these methods, `.mmap`, is implemented in the Waikato University DCCP stack to provide transmission rate information when using the TFRC DCCP CCID, and can be used as well to implement an alternative sending API that uses packet rings in an mmaped buffer as described the paper "A Congestion-Controlled Unreliable Datagram API" by Junwen Lai and Eddie Kohler[12].

To go from the common `struct proto_ops` `AF_INET` methods to the DCCP stack the `sk->sk_prot` pointer is used, and in DCCP case it is set to this struct:

```
struct proto dccp_v4_prot = {
.name        = "DCCP",
.owner       = THIS_MODULE,
.close       = dccp_close,
.connect     = dccp_v4_connect,
.disconnect  = dccp_disconnect,
.ioctl       = dccp_ioctl,
.init        = dccp_v4_init_sock,
.setsockopt  = dccp_setsockopt,
.getsockopt  = dccp_getsockopt,
.sendmsg     = dccp_sendmsg,
.recvmsg     = dccp_recvmsg,
.backlog_rcv = dccp_v4_do_rcv,
.hash        = dccp_v4_hash,
.unhash      = dccp_v4_unhash,
.accept      = inet_csk_accept,
.get_port    = dccp_v4_get_port,
.shutdown    = dccp_shutdown,
.destroy     =
dccp_v4_destroy_sock,
.max_header  = MAX_DCCP_HEADER,
.obj_size    = sizeof(struct
dccp_sock),
.rsk_prot    =
&dccp_request_sock_ops,
.orphan_count= &dccp_orphan_count,
};
```

Two of these methods bring us to a refactoring done to share code with TCP, denounced by the `.accept` method, `inet_csk_accept`, that previously was named `tcp_accept`, and as will be described in the next section could be made generic because most of the TCP infrastructure to handle SYN packets was generalised so as to be used by DCCP and other protocols.

7 Handling Connection Requests

DCCP connection requests are done sending a packet with a specific type, and this shows an important difference with TCP, namely that DCCP has an specific field in its packet header to indicate the type of the packet, whereas TCP has a flags field where one can use differenc combinations to indicate actions such as the beginning of the 3way handshake to create a connection, when a SYN packet is sent, while in DCCP a packet with type REQUEST is sent.

Aside from this difference the code to process a SYN packet in TCP fits most of the needs of DCCP to process a REQUEST packet: to create a mini socket, a structure to represent a socket in its embryonic form, avoiding using too much resources at this stage in the socket lifetime, and also to deal with timeouts waiting for TCP's SYN+ACK or DCCP's RESPONSE packet, synfloods (requestfloods in DCCP).

So the `struct open_request` TCP specific data structure was renamed to `struct request_sock`, with the members that are specfic to TCP and TCPv6 were removed, effectively creating a class hierarchy similar to the `struct sock` one, with each protocol using this structure creating a derived struct that has a `struct request_sock` as its first member, so that the functions that aren't protocol specific could be moved to the networking core, becoming a new core API usable by other protocols, not even necessarily an `AF_INET` protocol.

Relevant parts of `struct request_sock`:

```
struct request_sock {
struct request_sock *dl_next;
u8                  retrans;
u32                 rcv_wnd;
unsigned long       expires;
struct request_sock_ops *rsk_ops;
struct sock         *sk;
};
```

The `struct request_sock_ops` data structure is not really a new thing, it already exists in the stock kernel sources, within the TCP code, named as `struct or_calltable`, introduced when the support for IPv6 was merged. At that time the approach to make this code shared among TCPv6 and TCPv4 was to add an union to `struct open_request`, leaving this struct with this layout (some fields suppressed):

```
/* this structure is too big */
struct open_request {
struct open_request *dl_next;
u8                  retrans;
u32                 rcv_wnd;
unsigned long       expires;
struct or_calltable *class;
struct sock         *sk;
union {
  struct tcp_v4_open_req v4_req;
#if defined(CONFIG_IPV6) || defined
(CONFIG_IPV6_MODULE)
  struct tcp_v6_open_req v6_req;
#endif
  } af;
};
```

So there is no extra indirection added by this refactoring, and now the state that TCPv4 uses to represent syn sockets was reduced significantly as the TCPv6 state is not included,

being moved to `struct tcp6_request_sock`, that is derived in an OOP fashion from `struct tcp_request_sock`, that has this layout:

```
struct tcp_request_sock {
struct inet_request_sock req;
u32                      rcv_isn;
u32                      snt_isn;
};
```

That is, derived from another new data structure, `struct inet_request_sock`, that has this layout:

```
struct inet_request_sock {
 struct request_sock req;
 u32                 loc_addr;
 u32                 rmt_addr;
 u16                 rmt_port;
 u16                 snd_wscale:4,
                     rcv_wscale:4,
                     tstamp_ok:1,
                     sack_ok:1,
                     wscale_ok:1,
                     ecn_ok:1;
struct ip_options    *opt;
};
```

Which bring us back to DCCP, where sockets in the first part of the 3way handshake, the ones created when a DCCP REQUEST packet is received, are represented by this structure:

```
struct dccp_request_sock {
struct inet_request_sock
dreq_inet_rsk;
u64                      dreq_iss;
u64                      dreq_isr;
};
```

This way TCP's `struct open_request` becomes a class hierarchy, with the common part (`struct request_sock`) becoming available for use by any connection oriented protocol, much in the same way `struct sock` is common to all Linux network protocols.

References

[1] Sally Floyd, Mark Handley, and Eddie Kohler, 2002 "Problem Statement for DCCP" draft-ietf-dccp-problem-00.txt

[2] Eddie Kohler, Mark Handley and Sally Floyd, 2005 "Datagram Congestion Control Protocol (DCCP)" draft-ietf-dccp-spec-11.txt

[3] Sally Floyd, Eddie Kohler, 2005 "Profile for DCCP Congestion Control ID 2: TCP-like Congestion Control" draft-ietf-dccp-ccid2-10.txt

[4] Sally Floyd, Eddie Kohler and Jitendra Padhye, 2005 "Profile for DCCP Congestion Control ID 3: TFRC Congestion Control" draft-ietf-dccp-ccid3-11.txt

[5] Tom Phelan, 2005 "Datagram Congestion Control Protocol (DCCP) User Guide" draft-ietf-dccp-user-guide-04.txt

[6] Eddie Kohler, 2004 "DCCP CCID 3-Thin" draft-ietf-dccp-ccid3-thin-01.txt

[7] Eddie Kohler, 2004 "Datagram Congestion Control Protocol Mobility and Multihoming" draft-kohler-dccp-mobility-00.txt

[8] Sally Floyd, Eddie Kohler, 2005 "TCP Friendly Rate Control (TFRC) for Voice: VoIP Variant and Faster Restart" draft-ietf-dccp-tfrc-voip-01.txt

[10] Arnaldo Carvalho de Melo, 2004 "TCPfying the Poor Cousins" Ottawa Linux Symposium, 2004

[11] Patrick McManus DCCP implementation for Linux 2.4.18

[12] Junwen Lai and Eddie Kohler, "A Congestion-Controlled Unreliable Datagram API" http://www.icir.org/kohler/dcp/nsdiabstract.pdf

The sysfs Filesystem

Patrick Mochel
mochel@digitalimplant.org

Abstract

sysfs is a feature of the Linux 2.6 kernel that allows kernel code to export information to user processes via an in-memory filesystem. The organization of the filesystem directory hierarchy is strict, and based the internal organization of kernel data structures. The files that are created in the filesystem are (mostly) ASCII files with (usually) one value per file. These features ensure that the information exported is accurate and easily accessible, making sysfs one of the most intuitive and useful features of the 2.6 kernel.

Introduction

sysfs is a mechanism for representing kernel objects, their attributes, and their relationships with each other. It provides two components: a kernel programming interface for exporting these items via sysfs, and a user interface to view and manipulate these items that maps back to the kernel objects which they represent. The table below shows the mapping between internel (kernel) constructs and their external (userspace) sysfs mappings.

Internal	**External**
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationships	Symbolic Links

sysfs is a core piece of kernel infrastructure, which means that it provides a relatively simple interface to perform a simple task. Rarely is the code overly complicated, or the descriptions obtuse. However, like many core pieces of infrastructure, it can get a bit too abstract and far removed to keep track of. To help alleviate that, this paper takes a gradual approach to sysfs before getting to the nitty-gritty details.

First, a short but touching history describes its origins. Then crucial information about mounting and accessing sysfs is included. Next, the directory organization and layout of subsystems in sysfs is described. This provides enough information for a user to understand the organization and content of the information that is exported through sysfs, though for reasons of time and space constraints, not every object and its attributes are described.

The primary goal of this paper is to provide a technical overview of the internal sysfs interface—the data structures and the functions that are used to export kernel constructs to userspace. It describes the functions among the three concepts mentioned above—Kernel Objects, Object Attributes, and Object Relationships—and dedicates a section to each one. It also provides a section for each of the two additional regular file interfaces created to simplify some common operations—Attribute Groups and Binary Attributes.

sysfs is a conduit of information between the kernel and user space. There are many op-

portunities for user space applications to leverage this information. Some existing uses are the ability to I/O Scheduler parameters and the udev program. The final section describes a sampling of the current applications that use sysfs and attempts to provide enough inspiration to spawn more development in this area.

Because it is a simple and mostly abstract interface, much time can be spent describing its interactions with each subsystem that uses it. This is especially true for the kobject and driver models, which are both new features of the 2.6 kernel and heavily intertwined with sysfs. It would be impossible to do those topics justice in such a medium and are left as subjects for other documents. Readers still curious in these and related topics are encouraged to read [4].

1 The History of sysfs

sysfs is an in-memory filesystem that was originally based on ramfs. ramfs was written around the time the 2.4.0 kernel was being stabilized. It was an exercise in elegance, as it showed just how easy it was to write a simple filesystem using the then-new VFS layer. Because of its simplicity and use of the VFS, it provided a good base from which to derive other in-memory based filesystems.

sysfs was originally called *ddfs* (Device Driver Filesystem) and was written to debug the new driver model as it was being written. That debug code had originally used procfs to export a device tree, but under strict urging from Linus Torvalds, it was converted to use a new filesystem based on ramfs.

By the time the new driver model was merged into the kernel around 2.5.1, it had changed names to *driverfs* to be a little more descriptive. During the next year of 2.5 development, the infrastructural capabilities of the driver model and driverfs began to prove useful to other subsystems. kobjects were developed to provide a central object management mechanism and driverfs was converted to sysfs to represent its subsystem agnosticism.

```
mount -t sysfs sysfs /sys
```

Table 1: A sysfs mount command

```
sysfs  /sys  sysfs  noauto  0 0
```

Table 2: A sysfs entry in /etc/fstab

2 Mounting sysfs

sysfs can be mounted from userspace just like any other memory-based filesystem. The command for doing so is listed in Table 1.

sysfs can also be mounted automatically on boot using the file `/etc/fstab`. Most distributions that support the 2.6 kernel have entries for sysfs in `/etc/fstab`. An example entry is shown in Table 2.

Note that the directory that sysfs is mounted on: `/sys`. That is the de facto standard location for the sysfs mount point. This was adopted without objection by every major distribution.

3 Navigating sysfs

Since sysfs is simply a collection of directories, files, and symbolic links, it can be navigated and manipulated using simple shell utilities. The author recommends the `tree(1)` utility. It was an invaluable aide during the development of the core kernel object infrastructure.

```
/sys/
|--  block
|--  bus
|--  class
|--  devices
|--  firmware
|--  module
‘--  power
```

Table 3: Top level sysfs directories

At the top level of the sysfs mount point are a number of directories. These directories represent the major subsystems that are registered with sysfs. At the time of publication, this consisted of the directories listed in Table 3. These directories are created at system startup when the subsystems register themselves with the kobject core. After they are initialized, they begin to discover objects, which are registered within their respective directories.

The method by which objects register with sysfs and how directores are created is explained later in the paper. In the meantime, the curious are encouraged to meander on their own through the sysfs hierarchy, and the meaning of each subsystem and their contents follows now.

3.1 block

The `block` directory contains subdirectories for each block device that has been discovered in the system. In each block device's directory are attributes that describe many things, including the size of the device and the dev_t number that it maps to. There is a symbolic link that points to the physical device that the block device maps to (in the physical device tree, which is explained later). And, there is a directory that exposes an interface to the I/O scheduler. This interface provides some statistics about about the device request queue and some tunable features that a user or administrator can use to optimize performance, including the ability to dyanmically change the I/O scheduler to use.

```
bus/
|--  ide
|--  pci
|--  scsi
‘--  usb
```

Table 4: The bus directory

Each partition of each block device is represented as a subdirectory of the block device. Included in these directories are read-only attributes about the partitions.

3.2 bus

The `bus` directory contains subdirectories for each physical bus type that has support registered in the kernel (either statically compiled or loaded via a module). Partial output is listed in Table 4.

Each bus type that is represented has two subdirectories: `devices` and `drivers`. The `devices` directory contains a flat listing of every device discovered on that type of bus in the entire system. The devices listed are actually symbolic links that point to the device's directory in the global device tree. An example listing is shown in Table 5.

The `drivers` directory contains directories for each device driver that has been registered with the bus type. Within each of the drivers' directories are attributes that allow viewing and manipulation of driver parameters, and symbolic links that point to the physical devices (in the global device tree) that the driver is bound to.

```
bus/pci/devices/
|-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|-- 0000:00:01.0 -> ../../../devices/pci0000:00/0000:00:01.0
|-- 0000:01:00.0 -> ../../../devices/pci0000:00/0000:00:01.0/0000:01:00.0
|-- 0000:02:00.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:00.0
|-- 0000:02:00.1 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:00.1
|-- 0000:02:01.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:01.0
'-- 0000:02:02.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:02.0
```

Table 5: PCI devices represented in `bus/pci/devices/`

```
class/
|--  graphics
|--  input
|--  net
|--  printer
|--  scsi_device
|--  sound
'--  tty
```

Table 6: The class directory

3.3 class

The `class` directory contains representations of every device class that is registered with the kernel. A device class describes a functional type of device. Examples of classes are shown in Table 6.

Each device class contains subdirectories for each class object that has been allocated and registered with that device class. For most of class device objects, their directories contain symbolic links to the device and driver directories (in the global device hierarchy and the bus hierarchy respectively) that are associated with that class object.

Note that there is not necessarily a 1:1 mapping between class objects and physical devices; a physical device may contain multiple class objects that perform a different logical function. For example, a physical mouse device might map to a kernel mouse object, as well as a generic "input event" device and possibly a "input debug" device.

Each class and class object may contain attributes exposing parameters that describe or control the class object. The contents and format, though, are completely class dependent and depend on the support present in one's kernel.

3.4 devices

The `devices` directory contains the global device hierarchy. This contains every physical device that has been discovered by the bus types registered with the kernel. It represents them in an ancestrally correct way—each device is shown as a subordinate device of the device that it is physically (electrically) subordinate to.

There are two types of devices that are exceptions to this representation: platform devices and system devices. Platform devices are peripheral devices that are inherent to a particular platform. They usually have some I/O ports, or MMIO, that exists at a known, fixed location. Examples of platform devices are legacy x86 devices like a serial controller or a floppy controller, or the embedded devices of a SoC solution.

System devices are non-peripheral devices that are integral components of the system. In many ways, they are nothing like any other device. They may have some hardware register access for configuration, but do not have the capability to transfer data. They usually do not have

drivers which can be bound to them. But, at least for those represented through sysfs, have some architecture-specific code that configures them and treats them enough as objects to export them. Examples of system devices are CPUs, APICs, and timers.

3.5 firmware

The `firmware` directory contains interfaces for viewing and manipulating firmware-specific objects and attributes. In this case, 'firmware' refers to the platform-specific code that is executed on system power-on, like the x86 BIOS, OpenFirmware on PPC platforms, and EFI on ia64 platforms.

Each directory contains a set of objects and attributes that is specific to the firmware "driver in the kernel." For example, in the case of ACPI, every object found in the ACPI DSDT table is listed in `firmware/acpi/namespace/` directory.

3.6 module

The `module` directory contains subdirectories for each module that is loaded into the kernel. The name of each directory is the name of the module—both the name of the module object file and the internal name of the module. Every module is represented here, regardless of the subsystem it registers an object with. Note that the kernel has a single global namespace for all modules.

Within each module directory is a subdirectory called `sections`. This subdirectory contains attributes about the module sections. This information is used for debugging and generally not very interesting.

Each module directory also contains at least one attribute: `refcnt`. This attributes displays the current reference count, or number of users, of the module. This is the same value in the fourth column of `lsmod(8)` output.

3.7 power

The `power` directory represents the underused power subsystem. It currently contains only two attributes: `disk` which controls the method by which the system will suspend to disk; and `state`, which allows a process to enter a low power state. Reading this file displays which states the system supports.

4 General Kernel Information

4.1 Code Organization

The code for sysfs resides in `fs/sysfs/` and its shared function prototypes are in `include/linux/sysfs.h`. It is relatively small (~2000 lines), but it is divided up among 9 files, including the shared header file. The organization of these files is listed below. The contents of each of these files is described in the next section.

- `include/linux/sysfs.h` - Shared header file containing function prototypes and data structure definitions.
- `fs/sysfs/sysfs.h` - Internal header file for sysfs. Contains function definitions shared locally among the sysfs source.
- `fs/sysfs/mount.c` - This contains the data structures, methods, and initialization functions necessary for interacting with the VFS layer.

- `fs/sysfs/inode.c` - This file contains internal functions shared among the sysfs source for allocating and freeing the core filesystem objects.
- `fs/sysfs/dir.c` - This file contains the externally visible sysfs interface responsible for creating and removing directories in the sysfs hierarchy.
- `fs/sysfs/file.c` - This file contains the externally visible sysfs interface responsible for creating and removing regular, ASCII files in the sysfs hiearchy.
- `fs/sysfs/group.c` - This file contains a set of externally-visible helpers that aide in the creation and deletion of multiple regular files at a time.
- `fs/sysfs/symlink.c` - This file contains the externally- visible interface responsible for creating and removing symlink in the sysfs hierarchy.
- `fs/sysfs/bin.c` - This file contains the externally visible sysfs interface responsible for creating and removing binary (non-ASCII) files.

4.2 Initialization

sysfs is initialized in `fs/sysfs/mount.c`, via the `sysfs_init` function. This function is called directly by the VFS initialization code. It must be called early, since many subsystems depend on sysfs being initialized to register objects with. This function is responsible for doing three things.

- **Creating a `kmem_cache`**. This cache is used for the allocation of `sysfs_dirent` objects. These are discussed in a later section.
- **Registering with the VFS**. `register_filesystem()` is called with the `sysfs_fs_type` object. This sets up the appropriate super block methods and adds a filesystem with the name `sysfs`.
- **Mounts itself internally.** This is done to ensure that it is always available for other kernel code to use, even early in the boot process, instead of depending on user interaction to explicitly mount it.

Once these actions complete, sysfs is fully functional and ready to use by all internal code.

4.3 Configuration

sysfs is compiled into the kernel by default. It is dependent on the configuration option `CONFIG_SYSFS`. `CONFIG_SYSFS` is only visible if the `CONFIG_EMBEDDED` option is set, which provides many options for configuring the kernel for size-constrained envrionments. In general, it is considered a good idea to leave sysfs configured in a custom-compiled kernel. Many tools currently do, and probably will in the future, depend on sysfs being present in the system.

4.4 Licensing

The sysfs code is licensed under the GPLv2. While most of it is now original, it did originate as a clone of ramfs, which is licensed under the same terms. All of the externally-visible interfaces are original works, and are of course also licensed under the GPLv2.

The external interfaces are exported to modules, however only to GPL-compatible modules, using the macro `EXPORT_SYMBOL_GPL`. This is done for reasons of maintainability and derivability. sysfs is a core component

of the kernel. Many subsystems rely on it, and while it is a stable piece of infrastructure, it occasionally must change. In order to develop the best possible modifications, it's imperative that all callers of sysfs interfaces be audited and updated in lock-step with any sysfs interface changes. By requiring that all users be licensed in a GPL manner, and hopefully merged into the kernel, the level of difficulty of an interface change can be greatly reduced.

Also, since sysfs was developed initially as an extension of the driver model and has gone through many iterations of evolution, it has a very explicit interaction with its users. To develop code that used sysfs but was not copied or derived from an existing in-kernel GPL-based user would be difficult, if not impossible. By requiring GPL-compatibility in the users of sysfs, this can be made explicit and help prevent falsification of derivability.

5 Kernel Interface Overview

The sysfs functions visible to kernel code are divided into three categories, based on the type of object they are exporting to userspace (and the type of object in the filesystem they create).

- Kernel Objects (Directories).
- Object Attributes (Regular Files).
- Object Relationships (Symbolic Links).

There are also two other sub-categories of exporting attributes that were developed to accomodate users that needed to export other files besides single, ASCII files. Both of these categories result in regular files being created in the filesystem.

- Attribute Groups
- Binary Files

The first parameter to all sysfs functions is the kobject (hereby referenced as `k`), which is being manipulated. The sysfs core assumes that this kobject will remain valid throughout the function; i.e., they will not be freed. The caller is always responsible for ensuring that any necessary locks that would modify the object are held across all calls into sysfs.

For almost every function (the exception being `sysfs_create_dir`), the sysfs core assumes that `k->dentry` is a pointer to a valid dentry that was previously allocated and initialized.

All sysfs function calls must be made from process context. They should also not be called with any spinlocks held, as many of them take semaphores directly and all call VFS functions which may also take semaphores and cause the process to sleep.

6 Kernel Objects

Kernel objects are exported as directories via sysfs. The functions for manipulating these directories are listed in Table 7.

`sysfs_create_dir` is the only sysfs function that does not rely on a directory having already been created in sysfs for the kobject (since it performs the crucial action of creating that directory). It does rely on the following parameters being valid:

- k->parent
- k->name

```
int sysfs_create_dir(struct kobject * k);

void sysfs_remove_dir(struct kobject * k);

int sysfs_rename_dir(struct kobject *, const char *new_name);
```

Table 7: Functions for manipulating sysfs directories.

6.1 Creating Directories

These parameters control where the directory will be located and what it will be called. The location of the new directory is implied by the value of `k->parent`; it is created as a subdirectory of that. In all cases, the subsystem (not a low-level driver) will fill in that field with information it knows about the object when the object is registered with the subsystem. This provides a simple mechanism for creating a complete user-visible object tree that accurately represents the internal object tree within the kernel.

It is possible to call `sysfs_create_dir` without `k->parent` set; it will simply create a directory at the very top level of the sysfs filesystem. This should be avoided unless one is writing or porting a new top-level subsystem using the kobject/sysfs model.

When `sysfs_create_dir()` is called, a dentry (the object necessary for most VFS transactions) is allocated for the directory, and is placed in `k->dentry`. An inode is created, which makes a user-visible entity, and that is stored in the new dentry. sysfs fills in the `file_operations` for the new directory with a set of internal methods that exhibit standard behavior when called via the VFS system call interface. The return value is 0 on success and a negative errno code if an error occurs.

6.2 Removing Directories

`sysfs_remove_dir` will remove an object's directory. It will also remove any regular files that reside in the directory. This was an original feature of the filesystem to make it easier to use (so all code that created attributes for an object would not be required to be called when an object was removed). However, this feature has been a source of several race conditions throughout the years and should not be relied on in the hopes that it will one day be removed. All code that adds attributes to an object's directory should explicitly remove those attributes when the object is removed.

6.3 Renaming Directories

`sysfs_rename_dir` is used to give a directory a new name. When this function is called, sysfs will allocate a new dentry for the kobject and call the kobject routine to change the object's name. If the rename succeeds, this function will return 0. Otherwise, it will return a negative errno value specifying the error that occurred.

It is not possible at this time to move a sysfs directory from one parent to another.

7 Object Attributes

Attributes of objects can be exposed via sysfs as regular files using the `struct attribute`

```
int sysfs_create_file(struct kobject *, const struct attribute *);

void sysfs_remove_file(struct kobject *, const struct attribute *);

int sysfs_update_file(struct kobject *, const struct attribute *);
```

Table 8: Functions for manipulating sysfs files

```
struct device_attribute {
  struct attribute        attr;
  ssize_t (*show)(struct device *dev, char *buf);
  ssize_t (*store)(struct device *dev, const char *buf, size_t count);
};

int device_create_file(struct device *device,
                       struct device_attribute *entry);
void device_remove_file(struct device *dev,
                        struct device_attribute *attr);
```

Table 10: A wrapper for `struct attribute` from the Driver Model

```
struct attribute {
    char            *name;
    struct module   *owner;
    mode_t          mode;
};
```

Table 9: The `struct attribute` data type

data type described in Table 9 and the functions listed in Table 8.

7.1 Creating Attributes

`sysfs_create_file()` uses the `name` field to determine the file name of the attribute and the `mode` field to set the UNIX file mode in the file's inode. The directory in which the file is created is determined by the location of the kobject that is passed in the first parameter.

7.2 Reference Counting and Modules

The `owner` field may be set by the the caller to point to the module in which the attribute code exists. This should **not** point to the module that owns the kobject. This is because attributes can be created and removed at any time. They do not need to be created when a kobject is registered; one may load a module with several attributes for objects of a particular type that are registered after the objects have been registered with their subsystem.

For example, network devices have a set of statistics that are exported as attributes via sysfs. This set of statistics attributes could reside in an external module that does not need to be loaded in order for the network devices to function properly. When it is loaded, the attributes contained within are created for every registered network device. This module could be unloaded at any time, removing the

attributes from sysfs for each network device. In this case, the `module` field should point to the module that contains the network statistic attributes.

The `owner` field is used for reference counting when the attribute file is accessed. The file operations for attributes that the VFS calls are set by sysfs with internal functions. This allows sysfs to trap each access call and perform necessary actions, and it allows the actual methods that read and write attribute data to be greatly simplified.

When an attribute file is opened, sysfs increments the reference count of both the kobject represented by the directory where the attribute resides, and the module which contains the attribute code. The former operation guarantees that the kobject will not be freed while the attribute is being accessed. The latter guarantees that the code which is being executed will not be unloaded from the kernel and freed while the attribute is being accessed.

7.3 Wrappable Objects

One will notice that `struct attribute` does not actually contain the methods to read or write the attribute. sysfs does not specify the format or parameters of these functions. This was an explicit design decision to help ensure type safety in these functions, and to aid in simplifying the downstream methods.

Subsystems that use sysfs attributes create a new data type that encapsulates `struct attribute`, like in Table 10. By defining a wrapping data type and functions, downstream code is protected from the low-level details of sysfs and kobject semantics.

When an attribute is read or written, sysfs accesses a special data structure, through the kobject, called a kset. This contains the base operations for reading and writing attributes for kobjects of a particular type. These functions translate the kobject and attribute into higher level objects, which are then passed to the `show` and `store` methods described in Table 10. Again, this helps ensure type safety, because it guarantees that the downstream function receives a higher-level object that it use directly, without having to translate it.

Many programmers are inclined to cast between object types, which can lead to hard-to-find bugs if the position of the fields in a structure changes. By using helper functions within the kernel that perform an offset-based pointer subtraction to translate between object types, type safety can be guaranteed, regardless if the field locations may change. By centralizing the translation of objects in this manner, the code can be easier to audit in the event of change.

7.4 Reading and Writing Attributes

sysfs attempts to make reading and writing attributes as simple as possible. When an attribute is opened, a `PAGE_SIZE` buffer is allocated for transferring the data between the kernel and userspace. When an attribute is read, this buffer is passed to a downstream function (e.g., `struct device_attribute::show()` which is responsible for filling in the data and formatting it appropriately. This data is then copied to userspace.

When a value is written to a sysfs attribute file, the data is first copied to the kernel buffer, then it is passed to the downstream method, along with the size of the buffer in bytes. This method is responsible for parsing the data.

It is assumed that the data written to the buffer is in ASCII format. It is also implied that the size of the data written is less than one page in

size. If the adage of having one value per file is followed, the data should be well under one page in size. Having only one value per file also eliminates the need for parsing complicated strings. Many bugs, especially in text parsing, are propogated throughout the kernel by copying and pasting code thought to be bug-free. By making it easy to export one value per file, sysfs eliminates the need for copy-and-paste development, and prevents these bugs from propagating.

7.5 Updating an attribute

If the data for an attribute changes, kernel code can notify a userspace process that may be waiting for updates by modifying the timestamp of the file using `sysfs_update_file()`. This function will also call dnotify, which some applications use to wait for modified files.

8 Object Relationships

A relationship between two objects can be expressed in sysfs by the use of a symbolic link. The functions for manipulating symbolic links in sysfs are shown in Table 11. A relationship within the kernel may simply be a pointer between two different objects. If both of these objects are represented in sysfs with directories, then a symbolic link can be created between them and prevent the addition of redundant information in both objects' directories.

When creating a symbolic link between two objects, the first argument is the kobject that is being linked *from*. This represents the directory in which the symlink will be created. The second argument is the kobject which is being linked *to*. This is the directory that the symlink will point to. The third argument is the name of the symlink that will appear in the filesystem.

To illustrate this, consider a PCI network device and driver. When the system boots, the PCI device is discovered and a sysfs directory is created for it, long before it is bound to a specific driver. At some later time, the network driver is loaded, which may or may not bind to any devices. This is a different object type than the physical PCI device represents, so a new directory is created for it.

Their association is illustrated in Table 12. Shown is the driver's directory in sysfs, which is named after the name of the driver module. This contains a symbolic link that points to the devices to which it is bound (in this case, just one). The name of the symbolic link and the target directory are the same, and based on the physical bus ID of the device.

9 Attribute Groups

The attribute group interface is a simplified interface for easily adding and removing a set of attributes with a single call. The `attribute_group` data structure and the functions defined for manipulating them are listed in Table 13.

An attribute group is simply an array of attributes to be added to an object, as represented by the `attrs` field. The `name` field is optional. If specified, sysfs will create a subdirectory of the object to store the attributes in the group. This can be a useful aide in organizing large numbers of attributes.

Attribute groups were created to make it easier to keep track of errors when registering multiple attributes at one time, and to make it more compelling to clean up all attributes that a piece of code may create for an object. Attributes can

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target,
                      char *name);

void sysfs_remove_link(struct kobject *, char *name);
```

Table 11: Functions for manipulating symbolic links in sysfs

```
$ tree -d bus/pci/drivers/e1000/
bus/pci/drivers/e1000/
'-- 0000:02:01.0 -> ../../../../devices/pci0000:00/0000:00:1e.0/0000:02:01.0
```

Table 12: An example of a symlink in sysfs.

be added and removed from the group without having to change the registration and unregistration functions.

When a group of attributes is added, the return value is noted for each one. If any one fails to be added (because of e.g. low memory conditions or duplicate attribute names), the previously added attributes of that group will be removed and the error code will be returned to the caller. This allows downstream code to retain a simple and elegant error handling mechanism, no matter how many attributes it creates for an object.

When an attribute group is removed, all of the attributes contained in it are removed. If a subdirectory was created to house the attributes, it is also removed.

Good examples of attribute groups and their uses can be found in the network device statistics code. Its sysfs interface is in the file `net/core/net-sysfs.c`.

10 Binary Attributes

Binary files are a special class of regular files that can be exported via sysfs using the data structure and functions listed in Table 14. They exist to export binary data structures that are either best left formatted and parsed in a more flexible environment, like userspace process context because they have a known and standard format (e.g., PCI Configuration Space Registers); or because their use is strictly in binary format (e.g., binary firmware images).

The use of binary files is akin to the procfs interface, though sysfs still traps the read and write methods of the VFS before it calls the methods in `struct bin_attribute`. It allows more control over the format of the data, but is more difficult to manage. In general, if there is a choice over which interface to use—regular attributes or binary attributes, there should be no compelling reasons to use binary attributes. They should only be used for specific purposes.

11 Current sysfs Users

The number of applications that use sysfs directly are few. It already provides a substantial amount of useful information in an organized format, so the need for utilities to extract and parse data is minimal. However, there are a few users of sysfs, and the infrastructure to support more is already in place.

```
struct attribute_group {
    char                *name;
    struct attribute   **attrs;
};

int sysfs_create_group(struct kobject *,
                       const struct attribute_group *);
void sysfs_remove_group(struct kobject *,
                        const struct attribute_group *);
```

Table 13: Attribute Groups

```
struct bin_attribute {
    struct attribute        attr;
    size_t                  size;
    void                    *private;
    ssize_t (*read)(struct kobject *, char *, loff_t, size_t);
    ssize_t (*write)(struct kobject *, char *, loff_t, size_t);
    int (*mmap)(struct kobject *, struct bin_attribute *attr,
                struct vm_area_struct *vma);
};

int sysfs_create_bin_file(struct kobject * kobj,
                          struct bin_attribute * attr);
int sysfs_remove_bin_file(struct kobject * kobj,
                          struct bin_attribute * attr);
```

Table 14: Binary Attributes

`udev` was written in 2003 to provide a dynamic device naming service based on user/administrator/distro-specified rules. It interacts with the `/sbin/hotplug` program, which gets called by the kernel when a variety of different events occur. udev uses information stored in sysfs about devices to name and configure them. More importantly, it is used as a building block by other components to provide a feature-rich and user-friendly device management environment.

Information about udev can be found at kernel.org [2]. Information about **HAL**—an aptly named hardware abstraction layer—can be found at freedesktop.org [1].

udev is based on **libsysfs**, a C library written to provide a robust programming interface for accessing sysfs objects and attributes. Information about libsysfs can be found at SourceForge [3]. The udev source contains a version of libsysfs that it builds against. On some distributions, it is already installed. If so, header files can be found in `/usr/include/sysfs/` and shared libraries can be found in `/lib/libsysfs.so.1`.

The **pciutils** package has been updated to use sysfs to access PCI configuration information,

instead of using `/proc/bus/pci/`.

A simple application for extracting and parsing data from sysfs called **si** has been written. This utility can be used to display or modify any attribute, though its true benefit is efforts to aggregate and format subsystem-specific information into an intuitive format. At the time of publication, it is still in early alpha stages. It can be found at kernel.org [5].

12 Conclusion

sysfs is a filesystem that allows kernel subsystems to export kernel objects, object attributes, and object relationships to userspace. This information is strictly organized and usually formatted simply in ASCII, making it very accessible to users and applications. It provides a clear window into the kernel data structures and the physical or virtual objects that they control.

sysfs provides a core piece of infrastructure in the much larger effort of building flexible device and system management tools. To do this effectively, it retains a simple feature set that eases the use of its interfaces and data representations easy.

This paper has described the sysfs kernel interfaces and the userspace representation of kernel constructs. This paper has hopefully demystified sysfs enough to help readers understand what sysfs does and how it works, and with a bit of luck, encouraged them to dive head first into sysfs, whether it's from a developer standpoint, a user standpoint, or both.

References

[1] freedesktop.org. hal, 2005. `http://hal.freedesktop.org/wiki/Software_2fhal`.

[2] Greg Kroah-Hartman. udev, 2004. `http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html`.

[3] The libsysfs Developers. libsysfs, 2003. `http://linux-diag.sourceforge.net/Sysfsutils.html`.

[4] LWN.net. 2.6 driver porting series, 2003. `http://lwn.net/Articles/driver-porting/`.

[5] Patrick Mochel. si, 2005. `http://kernel.org/pub/linux/kernel/people/mochel/tools/si/`.

Using genetic algorithms to autonomically tune the kernel

Jake Moilanen
IBM
moilanen@austin.ibm.com

Peter Williams
Aurema Pty Ltd.
pwil3058@bigpond.net.au

Abstract

One of the next obstacles in autonomic computing is having a system self-tune for any workload. Workloads vary greatly between applications and even during an application's life cycle. It is a daunting task for a system administrator to manually keep up with a constantly changing workload. To remedy this shortcoming, intelligence needs to be put into a system to autonomically handle this process. One method is to take an algorithm commonly used in artificial intelligence and apply it to the Linux® kernel.

This paper covers the use of genetic-algorithms to autonomically tune the kernel through the development of the genetic-library. It will discuss the overall designed of the genetic-library along with the hooked schedulers, current status, and future work. Finally, early performance numbers are covered to give an idea as towards the viability of the concept.

1 What is a Genetic Algorithm

A genetic algorithm, or GA, is a method of searching a large space for a solution to a problem by making a series of educated guesses. This search is done by using the mathematical equivalent of biology's natural selection process. The values of the parameters to the solution are analogous to biology's genes. The genes/values that perform well will survive, while the ones that under perform are pruned from the gene pool. Over time these genes/values evolve towards an optimal solution for the current environment.

1.1 Genetic Algorithm terms

The term *gene* refers to a variable in the problem that is being solved. These variables can be for anything as long as changing their value causes a measurable outcome. A gene is a piece of the solution.

All of the different genes comprise a *child*. Each child normally has different values for their genes, which makes each child unique. These different value and combinations allow some children to perform better than others in a given environment. A single child is a single possible solution to the given problem.

All of the children make up a *population*. A population is a set of solutions to the given problem.

When a set of children are put together, they create a *generation*. A generation is the time

that all children perform before the natural selection process prunes some children. The remaining children become *parents* and create children for the next generation.

The measure of how well a child is performing is a *fitness* measure. This is the numerical value assigned to each child at the end of a generation.

A *phenotype* is the end result of the genes interaction. In biology, an example would be eye color. There are a number of genes that affect eye color, but only one color as an end result. In a genetic algorithms specific genes impact specific fitness outcomes.

Much how evolution works in the wild, a genetic algorithm takes advantage of *mutations* to introduce new genes into the gene pool. This is to combat a limited set of genes that may have worked well in the old environment, but does not have the optimal result in a changing environment. Mutations also aid in premature convergence on less-than-optimal solutions.

2 Genetic-Library

As the name implies, the genetic-library is a library where components in the kernel can plug into to take advantage of a genetic algorithm. The advantage of the genetic-library is that components do not have to create their own method of self-tuning. The genetic-library creates a unified path that is flexible enough to handle almost any tuning that a component has need for.

2.1 Registering

Before the genetic-library is used, components first must register with it. When registering, state must be given to the genetic-library. For instance, the plugins need to give `genetic_ops`, which are implementation specific callback functions that the genetic-library uses. The child lifetime, the number of genes, and the number of children must also be included for each phenotype.

2.2 Genetic library life-cycle

An implementation of a genetic algorithm can vary, but the genetic-library uses the following one:

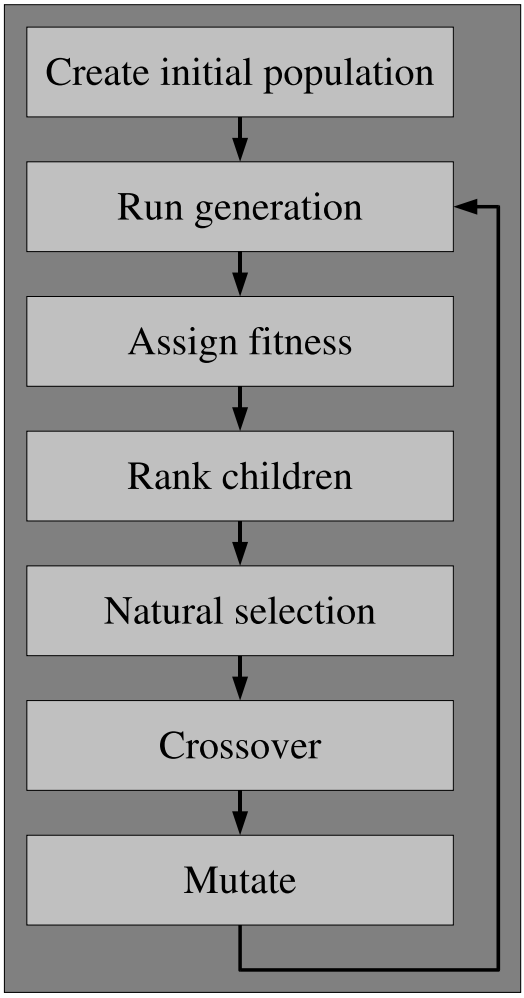


Figure 1: Life Cycle

2.2.1 Create the initial population

The first step in a genetic algorithm is to create an initial population of children with their own set of genes. Usually, the children's genes are given values that spread across the entire search space. This helps facilitate the survival of the fittest.

The genetic-library makes calls into the components through the registered `genetic_ops`. For each phenotype, all of the children are created through the `create_child()` callback. This callback can initialize genes in a number of ways. The most common is by spreading the gene values across the entire search space.

2.2.2 Run generation

In a genetic algorithm, all the children in the current generation are run in serial. The children plug their genes into the system and run for a slice of time. Once all of the children in the generation have completed their run, the generation is over.

In the genetic-library, the first child in every phenotype calls `genetic_run_child()` to kick off the generation. This function sets the genes to be used with the `set_child_genes()` callback. Next, it takes a snapshot of performance counters for the fitness measurement to determine how well this child performed. Finally, a timeout is set that will conclude the child's lifetime. That timer function is used to switch to the next child through `genetic_switch_child()`.

2.2.3 Assign fitness to children

One of the most difficult pieces of a genetic algorithm is assigning an accurate fitness number to a child. This fitness value is used to rank the children against each other. Depending on implementation, the fitness calculation is either done at the completion of a generation, or at the end of a child's lifetime.

For the genetic-library, the fitness calculation is done at the conclusion of a child's lifetime through the `calc_fitness()` callback. This function looks at the snapshot of the performance counters from the beginning of the child's lifetime, and takes the delta of the counters at the end of the lifetime. Since these number are usually normalized between all the children, the delta is usually all that is needed.

There are certain other phenotypes where the fitness calculation must be done at the end of a generation. This is usually when the phenotype contains general tunables that affect other phenotype's outcome. In this case `calc_post_fitness()` is used. This routine normalizes all the different fitness values by taking the average ranking of all the children in the affected phenotypes. The average ranking is used as a fitness measure.

2.2.4 Rank children

Using the fitness value assigned, children are then ranked in order of their performance. Children with well-performing genes get a higher ranking.

In the genetic-library's `genetic_split_performers()`, a bubble sort is used to order the children according to their fitness.

2.2.5 Natural selection operation

The same way Darwin's natural-selection process works in the wild, it works in the genetic algorithm. Those genes that perform well in

the given environment, will survive, and those that perform poorly will not. This enables the strongest genes to carry on to the next generation.

In the genetic-library, the bottom half of the population that under performs is removed. This replacing of part of the population is known as a steady-state type of algorithm. There is also a generational type of algorithm where the entire population is replaced. For instance, in implementations that make use of a roulette wheel algorithm, the whole population is replaced, but the children that have higher fitness have a proportionally higher chance of their genes being passed on.

2.2.6 Crossover Operation

This operation is the main distinguishing factor between a genetic algorithm and other optimization algorithms. The children that survived the natural selection process now become parents. The parents mate and create new children to repopulate the depleted population.

There are a number of methods for crossover, but the most common one in the genetic-library is similar to the blending method. For all of the phenotypes that have genes to combine (some phenotypes are just placeholders for fitness measures and their child's rankings are used to determine fitness for another phenotype), each gene receives X% of parent A's gene value and add in 100-X% of parent B's gene value. X is a random percentage between 0 and 100. The end result is that the child has a gene value that is somewhere randomly in the middle of parent A's, and parent B's genes.

2.2.7 Mutation Operation

To combat premature convergence on a solution, a small number of mutations are introduced into the population. These mutations also aid in changing environments where the current gene pool performs less-than-optimal.

After the new population is created, genes are picked randomly and randomly modified. These mutations keep the population diverse. Staying diverse makes the algorithm perform a global search.

In the genetic-library, mutation is done on some percentage of all the genes. Mutations are randomly done on both new children, and parents. Once the individual from the population is picked, a gene is randomly selected to be mutated. The gene either has a new value picked at random, or else is iteratively modified by having a random percentage increase or decrease in the gene's value.

On a system, workloads are always changing. So the population needs to always be changing to cover the current solution search space. To counteract this moving target, the genetic-library varies the rate of mutation depending on how well the current population is performing. If the average fitness for the population decreases past some threshold, then it appears as if the workload is changing and the current population is not performing as well. To counteract this new problem/workload, the mutation rate is increased to widen the search space and find the new optimal solution. There is a limit on the mutation rate, so not to have the algorithm go spiraling out of control with mutations bringing the population further and further away from the solution. Conversely, if the fitness is increasing, then it appears that the population is converging on an optimal solution, so the mutation rate decreases to not introduce excessive bad genes.

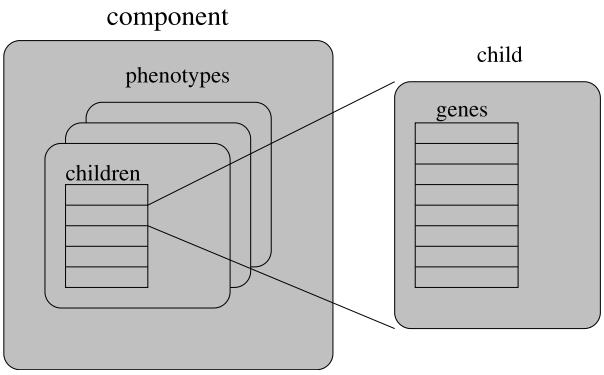


Figure 2: Structure Layout

2.3 Framework

The `struct genetic_s` is the main struct that contains the state for each component plugged into the genetic-lib.

This general structure contains all of the phenotypes in the `struct phenotype_s`. A phenotype is created for each specific measurable outcome.

Within each phenotype, is an array of `struct genetic_child_s` or children. Each child will contain an array of genes that are specific to that phenotype. Since some genes may affect multiple fitness measures, those genes are usually put into a phenotype that encapsulates other phenotypes. This will be discussed further in the next section.

Each gene has a `struct gene_param` associated with it. In this structure, the gene's properties are given. The minimum and the maximum value for a gene, along with its default value are given. If a gene has a specific function to mutate it, that can also be provided.

2.4 Phenotypes

Some other genetic algorithms refer to phenotypes as something comparable to what the genetic-library calls a child. However in the genetic library context, it refers to a population of children that affect a specific fitness measure. Phenotypes were introduced into the genetic library to increase granularity of what could be tuned in a component. Before phenotypes there was one fitness routine per component. This fitness function could look at multiple performance metrics, but all the genes for the component would be affected regardless if they had nothing to do with some of the performance metrics. For example, some of the genes that impact real-time process scheduling were being judged by fitness metrics that looked at throughput. With the introduction of phenotypes, the fitness measure of real-time performance would only affect the genes that impacted real-time.

The next problem that came about with phenotypes was what to do with the genes that affect a number of fitness metrics. For example, time-slice affects fitness measures like number of context switches, and total delay. The solution lies with adding a hierarchy of phenotypes that affect other phenotypes. This is done by assigning a unique ID's, or *uid*, to each phenotype. A *uid* is really a bitmask of phenotypes that affect it.

3 Hooked components

The genetic-library can be hooked into pretty much any component that can be tuned. For the initial implementation, the Zaphod CPU scheduler and the Anticipatory I/O scheduler were picked.

The Zaphod CPU scheduler was attractive to use because of its heavy integration with sched stats. Having extensive scheduler statistics made it much easier to create good fitness routines.

The Anticipatory I/O scheduler was also desirable because modifying the tunables could affect the schedulers performance greatly.

3.1 Zaphod CPU scheduler

The Zaphod CPU scheduler emerged from the CPU scheduler evaluation work. It is a single priority array O(1) with interactive response bonuses, throughput bonuses, soft and hard CPU rate caps and a choice of priority based or entitlement based interpretation of "nice."

3.1.1 Configurable Parameters

The behavior of this scheduler is controlled by a number of parameters and since there was no *a priori* best value for these parameters they were designed to be runtime configurable (within limits) so that experiments could be conducted to determine their best values.

`time_slice` One of the principal advantages of using a single priority array is that a task's time slice is no longer tied up controlling its movement between the active and expired arrays. Therefore all tasks are given a new time slice every time they wake and when they finish their current time slice. This parameter determines the size of the time slice given to `SCHED_NORMAL` tasks.

`sched_rr_time_slice` This parameter determines the size of the time slice given to `SCHED_RR` tasks.

`base_prom_interval` The single priority array introduces the possibility of starvation and to handle this Zaphod includes an O(1) promotion mechanism. When the number of runnable tasks on a run queue is greater than 1, Zaphod periodically moves all runnable `SCHED_NORMAL` tasks with a `prio` value greater than `MAX_RT_PRIO` towards the head of the queue. This variable controls the interval between promotions and its ratio to the value of `time_slice` can be thought of as controlling the severity of "nice."

`bgnd_time_slice_multiplier` Tasks with a soft CPU rate cap are essentially background tasks and generally only run when there are no other runnable tasks on their run queue. These tasks are usually batch tasks that benefit from longer time slices and Zaphod has a mechanism to give them time slices that are an integer multiple of `time_slice` and this variable determines that multiple.

`max_ia_bonus` In order to enhance interactive responsiveness, Zaphod attempts to identify interactive tasks and give them priority bonuses. This attribute determines the largest bonus that Zaphod awards. Setting this attribute to zero is recommended for servers.

`initial_ia_bonus` When interactive tasks are forked on very busy systems it can take some time for Zaphod to recognize them as interactive. Giving all tasks a small bonus when they fork can help speed up this process and this attribute determines the initial interactive bonus that all tasks receive.

`ia_threshold` When Zaphod needs to determine a dynamic priority (i.e., a `prio` value) it calculates the recent average *sleepiness* (i.e., the ratio of the time spends sleeping to the sum of the time spent on the CPU or sleeping) and if this is greater than the value of `ia_threshold` it increases the proportion (`interactive_`

bonus) of max_ia_bonus that it will award this task asymptotically towards 1.

cpu_hog_threshold At the same time it calculates the tasks *CPU usage rate* (i.e. the ratio of the time spent on the CPU to the sum of the time spent on a run queue waiting for CPU access or sleeping) and if this is greater than the value of cpu_hog_threshold it decreases the task's interactive_bonus asymptotically towards zero. From this it can be seen that the size of the interactive bonus is relatively permanent.

max_tpt_bonus Zaphod also has a mechanism for awarding throughput bonuses whose purpose is (as the name implies) to increase system throughput by reducing the total amount of time that tasks spend on run queues waiting for CPU access. These bonuses are ephemeral and once granted are only in force for one task scheduling cycle. The size of the throughput bonus awarded to a task each scheduling cycle is decided by comparing the recent average *delay time* that the task has been suffering to the expected delay time based on how busy the system is, the task's usage patterns and static priority. It will be a proportion of the value of max_tpt_bonus. This bonus is generally only effective when the system is less than fully loaded as once the system is fully loaded it is not possible to reduce the total delay time of the tasks on the system.

current_zaphod_mode As previously mentioned, Zaphod offers the choice of a priority based or an entitlement based interpretation of "nice." This attribute determines which of those interpretations is in use.

3.1.2 Scheduling Statistics

As can be seen from the above description of Zaphod's control attributes, Zaphod needs data on the amount of time tasks spend on run queues waiting for CPU access in order to compute task bonuses. The kernel does not currently provide this data so Zaphod maintains its own scheduling statistics (in nanoseconds) for both tasks and run queues. The scheduling statistics of interest to this paper are the run queue statistics as they are an indication of the overall system performance. The following statistics are kept for each run queue in addition to those already provided in the vanilla kernel:

total_idle The total amount of time (since boot) that the CPU associated with the run queue was idle. This is actually derived from the total CPU time for the run queue's idle thread.

total_busy The total amount of time (since boot) that the CPU associated with the run queue was busy. This is actually derived from the total time that the run queue's idle thread spent off the CPU.

total_delay The total amount of time (since boot) that tasks have spent on this run queue waiting for access to its CPU.

total_rt_delay The total amount of time (since boot) that real time tasks have spent on this run queue waiting for access to its CPU.

total_intr_delay The total amount of time (since boot) that tasks awoken to service an interrupt have spent on this run queue waiting for access to its CPU.

total_rt_intr_delay The total amount of time (since boot) that real time tasks awoken to service an interrupt have spent

on this run queue waiting for access to its CPU.

`total_fork_delay` The total amount of time (since boot) that tasks have spent on this run queue waiting for access to its CPU immediately after forking.

`total_sinbin` The total amount of time (since boot) that tasks associated with this run queue have spent cooling their heels in the *sin bin* as a consequence of exceeding their CPU usage rate hard cap.

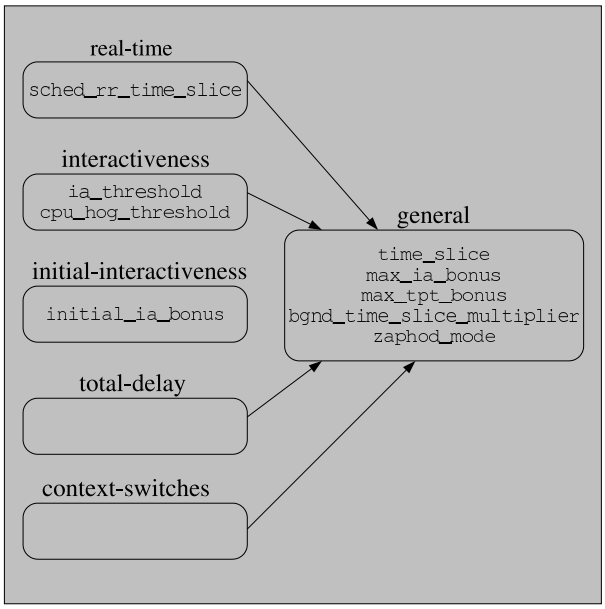


Figure 3: Zaphod Phenotypes

3.1.3 Phenotypes

In Figure 3, there are six phenotypes listed along with the genes that exist within them. All of the phenotypes have their own fitness measures. For example, `real-time`'s fitness measures takes the delta of `total_rt_delay` for each child. The fitness measure not only affects `sched_rr_time_slice`, but also affects all of the genes in the `general` phenotype.

Phenotypes might not always have genes in their children. This is done in when the phenotypes are just being used for their fitness measures. The children that perform well are ranked accordingly. The `general` phenotype looks at the average ranking of the children of all the phenotypes under it.

The `general` phenotype also has a weights associated with each of its subsidiary phenotypes. The phenotypes that have a greater impact on the general phenotype gets higher weights associated with them. For instance, the `total-delay` phenotype is three times more important than the `real-time` phenotype.

To actually calculate the fitness for the `general` phenotype's children, the first child looks at what place it ranked in each phenotype, between `1-NUM_CHILDREN`, and then multiply its place by the weight associated with that phenotype to get a final fitness number for that child. A quick example would be if a child was ranked as the worst performing in `real-time`, it would receive 1 point (top rank gets the most points, lowest gets the least) times the `real-time` weight, which is 1. However, in the `total-delay` phenotype, it was the second best performer, and receives `NUM_CHILDREN-1` points. Assume that there are 8 children. The child would receive 7 points (ranked second best), times `total-delay`'s weight, which is 3.

The final fitness number would be:

```
real-time:              1 * 1
total-delay:          + 7 * 3
                        ------
final fitness:             22
```

3.2 Anticipatory IO scheduler

The anticipatory I/O scheduler, or AS attempts to reduce the disk seek time by using a heuris-

tic to anticipate getting another read request in close proximity. This is done by delaying pending I/O with the expectation that the delay of servicing a request will be made up for by reducing the number of times the disk has to seek.

The anticipatory I/O scheduler was developed on one large assumption, that there was only one outstanding I/O on the bus, and only one head to seek. In other words, it assumed that the disk was an IDE drive. This works very well on most desktops, however, in most server environments, they have SCSI disks, which can handle many outstanding I/Os, and many times these disks are setup in a RAID environment and have many disk heads.

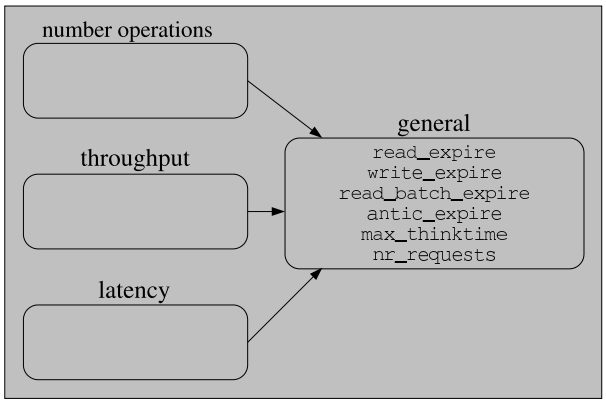


Figure 4: Anticipatory I/O Scheduler Phenotypes

3.2.1 Phenotypes

Figure 4 shows how three of the four phenotypes are just placeholders for fitness measures. Only the `general` phenotype contains genes.

The three phenotypes that are just fitness measures are in place to make sure all workloads are considered, and not favor one type of workload over another. They are agnostic towards the type of I/O, whether it is a read or write.

The `num_ops` phenotype only exists for fitness measurements. The fitness routine looks at the delta of number of I/O operations completed during a child's lifetime. This fitness routine helps balance out the idea of pure throughput. This gives a small fitness bonus to a large number small I/O's.

In the `throughput` phenotype, the fitness simply looks at the number of sectors read or written in during a child's lifetime. This phenotype makes sure data is actually moving, and not just servicing a lot a small requests.

The `latency` phenotype measures the time all requests sit in the queue. This should help combat I/O starvation.

4 Performance numbers

The main goal of the genetic-library is to increase performance through autonomically tuning components of the kernel. The performance gain offered by the genetic-library must outweigh the cost of adding more code into the kernel. While there is no hard-and-fast rule towards what percentage increase is worth adding X number lines of code, gains should be measurable.

In the performance evaluation, an OpenPower 710 system, with 2 CPUs, and 1.848 gigabytes of RAM was used. The benchmarks were conducted on a SLES 9 SP1 base install with a 2.6.11 kernel. More system details can be found in Appendix A.

The base benchmarks were conducted on a stock 2.6.11 kernel, with the PPC64 default config. On the benchmarking of each component utilizing the genetic-library, only the genetic-library patches for the component being exercised at that time were in the kernel.

4.1 Zaphod CPU scheduler

To benchmark the Zaphod CPU Scheduler, SPECjbb2000® was used. This benchmark is a good indicator of scheduler performance. Due to these runs being unofficial, their formal numbers cannot be published. However, a percentage difference should be sufficient for what this paper intends to look at.

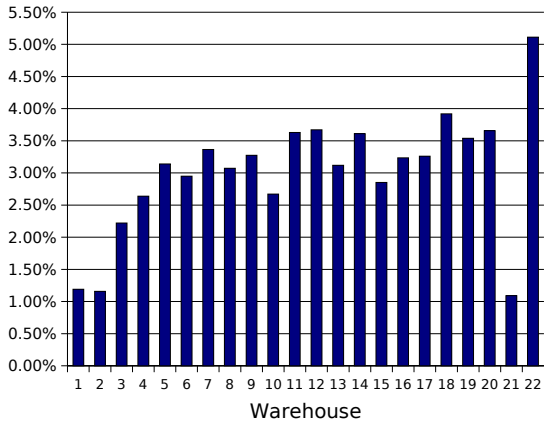


Figure 5: SPECjbb results—GA plugin to Zaphod

The performance improvement ranged from 1.09% to 5.11% in all of the warehouses tested. There is a trend towards a larger improvement as the number of warehouses increase. This indicates that the genetic-library helped Zaphod scale as the load increased on the system. The warehouses averaged an improvement of 3.04%, however the SPECjbb peak performance throughput only showed a 1.52% improvement. The peak performance throughput difference may not be valid due to the performance peaking in different warehouses. That difference makes the two throughput numbers unable to be measured directly against one another.

4.2 Anticipatory I/O scheduler

The Anticipatory I/O scheduler is tuned to do sequential reads very well[1]; however, it has problems with the other types of I/O operations such as sequential writes and random reads. These other I/O types of operations can do better when the AS is tuned for them. If the genetic-library did its tuning correctly, there should be a performance increase across all types of workloads.

To generate these workloads, the flexible file system benchmark, or FFSB was used. The FFSB is a versatile benchmark that gives the ability to simulate any type of workload[2].

For the benchmarking the AS genetic-library plugin, FFSB sequentially went through a series of workload simulations and returned the number of transactions-per-second and the throughput. This experiment was conducted on a single disk ext3 file system.

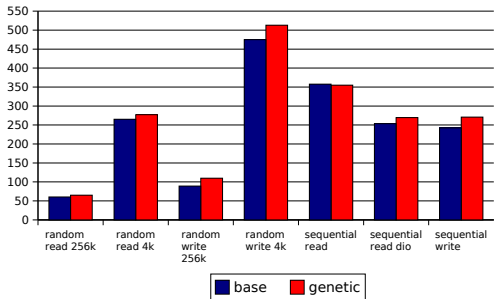


Figure 6: FFSB transactions per second—Anticipatory Plugin

With the exception of one workload, there were performance improvements across the board. The largest increase was in the 256K random write workload. The genetic-library version had a 23.22% improvement over a stock AS. The average improvement of all tested workloads was 8.72%.

The one workload where the genetic-library degraded performance was the sequential read workload at –0.74%. This is not surprising because the AS is optimized specifically for this workload, and the genetic-library's tunings might not get any better than the default settings. The performance loss can be attributed

to the genetic library's attempts at finding better tunings. When the new tuning solution is attempted it will probably be less-than-optimal.

5 Conclusion & Future work

5.1 Kernel Inclusion viability

At the present time, the Anticipatory I/O Scheduler sees large enough improvements, that a strong argument can be made to add the extra complexity into the kernel. The GA plugin to Zaphod also sees substantial gains in performance, especially when the system is under a high load. If only throughput was a concern on the CPU scheduler, then inclusion into the kernel should be considered. However, there are number of other factors that must be looked at. The biggest one is, how well the system also maintains interactiveness, which is very subjective.

In the near-term, the GA plugin to Zaphod should only be used in a server environment. This is because a desktop environment is particularly malicious for the genetic-library. There are numerous CPU usage spikes that can skew performance results. On top of the CPU usage, there are the interactiveness concerns. A user expects to see instant reaction in their interactive applications. If the genetic library goes off on a tangent to try finding a new optimal tuning, a time-slice may go much longer than is acceptable by a desktop user. New features are planned for the genetic-library to help converge quicker on changing workloads.

5.2 Future work

There are other areas of the kernel that are being investigated for their viability of being tuned with the genetic-library. Some of them include scheduler domain reconfiguration, full I/O scheduler swapping, plugsched CPU scheduler swapping, packet scheduling, and SMT scheduling.

The next major feature of the genetic-library will be workload fingerprinting. The idea is to bring back the top-performing genes for certain workloads. By being able to identify a particular workload, the history of optimal tunings for that workload can be saved. These optimal genes will be reintroduced into the population when the current workload matches a fingerprinted workload. This will enable faster convergence when workloads change.

5.3 Conclusion

The genetic-library has the ability to put intelligence into the kernel, and gracefully handle even the most malevolent of workloads. Hopefully, it will pave the way towards a fully autonomic system and the elimination of the system admin dependency.

Legal Statement

References

[1] Pratt, S., Heger, D., *Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers*, 2004 Linux Symposium

[2] `http://sourceforge.net/projects/ffsb/`

Appendix A. Performance System

IBM OpenPower 710 System
2-way 1.66 Ghz Power5 Processors
1.848 GB of memory
2 15,000 RPM SCSI drives
SLES 9 SP1
2.6.11 Kernel