Implementation Issues for the Psyche Multiprocessor Operating System

Michael L. Scott, Thomas J. LeBlanc,
Brian D. Marsh, Timothy G. Becker,
Cezary Dubnicki, Evangelos P. Markatos,
and Neil G. Smithline
University of Rochester

ABSTRACT: Psyche is a parallel operating system under development at the University of Rochester. The Psyche user interface is designed to allow programs with widely differing concepts of process, sharing, protection, and communication to run efficiently on the same machine, and to interact productively. In addition, the Psyche development effort is addressing a host of implementation issues for large-scale shared-memory multiprocessors, including the organization of kernel functions, data structures, and address maps for machines with non-uniform memory; the migration and replication of pages to maximize locality; the introduction of user-level device drivers, memory management, and scheduling; and remote source-level kernel debugging. We focus in this paper on our implementation of Psyche for the BBN Butterfly Plus

This work has been supported in part by NSF IIP grant number CDA-8822724, DARPA ETL contract number DACA76-85-C-0001, ONR contract number N00014-87-K-0548, and a DARPA/NASA Graduate Research Assistantship in Parallel Processing.

multiprocessor, though many of the issues we consider apply to any operating system kernel on a large-scale shared-memory machine. We describe our major design decisions, the results of our initial experience with the implementation, and our plans for continued evaluation and experimentation with kernel implementation techniques.

1. Introduction

Parallel processing is in the midst of a transition from special purpose to general purpose systems. Part of the impetus for this transition has been the development of practical, large-scale, shared-memory multiprocessors. To make the most effective use of these machines, an operating system must address two fundamental issues that do not arise on uniprocessors. First, the kernel interface must provide the user with greater control over parallel processing abstractions than is customary in a traditional operating system. Second, the kernel must be structured to take advantage of the parallelism and sharing available in the hardware.

If shared-memory multiprocessors are to be used for day-to-day computing, it is important that users be able to program them with whatever style of parallelism is most appropriate for each particular problem. To do so they must be able to exercise control over concepts traditionally reserved to the kernel of the operating system, including processes, communication, scheduling, sharing, and protection. If shared-memory multiprocessors are to be used efficiently, it is also important that the kernel not define abstractions that hide a significant portion of the hardware's functionality.

The Psyche project is an attempt to design and prototype a high-performance, general-purpose operating system for large-scale shared-memory multiprocessors. The fundamental kernel abstraction, an abstract data object called a *realm*, can be used to implement such diverse mechanisms as monitors, remote procedure calls, buffered message passing, and unconstrained shared memory [Scott et al. 1990]. Sharing is the default in Psyche; protection is provided only when the user specifically indicates a willingness to sacrifice performance in order to obtain it. Sharing also occurs between the user and the kernel, and facilitates explicit, user-level control of process structure and scheduling.

This emphasis on multi-model parallel computing, and on user-level flexibility in general, is the core of the Psyche project. Our intent is to allow the users of a shared-memory multiprocessor to do almost anything for which the underlying hardware is well suited, and to allow applications or application components that use the machine in different ways to coexist and to interact productively. Working with members of the computer vision and planning groups within our department, we have undertaken a large joint project in real-time vision and robotics. Because they must perform a wide variety of reflexive and cognitive tasks (naturally expressed with a wide variety of parallel programming models), the roboticists make an excellent user community.

A glimpse of Psyche from the user's point of view appears in Section 2; more detail can be found in other papers [Scott et al. 1988; 1989; 1990]. In this paper we focus on implementation issues for our prototype of Psyche. Some of these issues, such as the management of virtual address spaces, scheduling, device drivers, and the handling of page faults, are heavily tied to the Psyche kernel interface. Others, such as multi-processor bootstrapping, communication, synchronization, and the division of data and functionality among instances of the kernel, must be addressed in any operating system for a large shared-memory multiprocessor.

Our implementation of Psyche runs on the BBN Butterfly Plus multiprocessor, the hardware base of the GP1000 product line. We began writing code for the kernel in the summer of 1988, building on the bare machine. Our first toy program ran in user space in December of 1988. Our first major application [Yamauchi 1989] was ported to Psyche in November of 1989. It uses video cameras and a robot arm to locate and bat a balloon. Because our work is still in progress, we devote the bulk of our presentation here to unsolved research problems, and to what we consider the most promising ways to address them.

Section 2 presents the Psyche kernel interface, including its rationale, its benefits to users, and its implications for implementation. Sections 3 and 4 describe the structure of our kernel and the management of its resources. Section 5 relates our experience with kernel debugging tools. Section 6 details our status and future plans.

2. *Overview of Psyche*

2.1 Motivation

The Computer Science Department at the University of Rochester acquired its first shared-memory multiprocessor, a 3-node BBN Butterfly machine, in 1984. Since that time, departmental resources have grown to include four distinct varieties of Butterfly (one with 128 nodes) and an IBM 8CE multiprocessor workstation. From 1984 to 1987, our work could best be characterized as a period of experimentation, designed to evaluate the potential of scaleable shared-memory multiprocessors and to assess the need for software support. In the course of this experimentation we ported three compilers to the Butterfly, developed five major and several minor library packages, built two different operating systems, and implemented dozens of applications. A summary of this work can be found in LeBlanc et al. [1988].

As we see it, the most significant strength of a shared-memory architecture is its ability to support efficient implementations of many different parallel programming models, encompassing a wide range of grain sizes of process interaction. Local-area networks and more tightly-coupled multicomputers (the various commercial hypercubes, for example) can provide outstanding performance for message-based models with large to moderate grain size, but they do not admit a reasonable implementation of interprocess sharing at the level of individual memory locations. Shared-memory multiprocessors can support this fine-grained sharing, and match the speed of multicomputers for message passing, too.

We have used the BBN Butterfly to experiment with many different programming models. BBN has developed a model based on fine-grain memory sharing [Thomas & Crowther 1988]. In addition, we have implemented remote procedure calls, an object-oriented encapsulation of processes, memory blocks, and messages, a message-based library package, a shared-memory model with numerous lightweight processes, and a message-based programming language.

Using our systems packages, we have achieved significant speedups (often nearly linear) on over 100 processors with a range of applications that includes various aspects of computer vision, connectionist network simulation, numerical algorithms, computational geometry, graph theory, combinatorial search, and parallel data structure management. In every case it has been necessary to address the issues of locality and contention, but neither of these has proven to be an insurmountable obstacle. Simply put, a shared-memory multiprocessor is an extremely flexible platform for parallel applications. The challenge for hardware designers is to make everything scale to larger and larger machines. The challenge for systems software is to keep the flexibility of the hardware visible at the level of the kernel interface.

A major focus of our experimentation with the Butterfly has been the evaluation and comparison of multiple models of parallel computing [Brown et al. 1986; LeBlanc 1986; LeBlanc et al. 1988]. Our principal conclusion is that while every programming model has applications for which it seems appropriate, no single model is appropriate for every application. In an intensive benchmark study conducted in 1986 [Brown et al. 1986], we implemented seven different computer vision applications on the Butterfly over the course of a three-week period. Based on the characteristics of the problems, programmers chose to use four different programming models, provided by four of our systems packages. For one of the applications, none of the existing packages provided a reasonable fit, and the awkwardness of the resulting code was a major impetus for the development of yet another package. It strikes us as highly unlikely that any predefined *set* of parallel programming models will be adequate for the needs of all user programs.

Other researchers have recognized the need for multiple models of parallel computing. Remote procedure call systems, for example, have often been designed to work between programs written in multiple languages [Bershad et al. 1987; Hayes & Schlichting 1987; Jones et al. 1985; Liskov et al. 1988]. Unfortunately, most RPC-based systems support only one style of process interaction, and are usually intended for a distributed environment; there is no obvious way to extend them to fine-grained process interactions. Synchronization is supported only via client-server rendezvous, and even the most efficient implementations [Bershad et al. 1990] cannot compete with the low latency of direct access to shared memory.

At the operating system level, the Choices project at Illinois [Campbell et al. 1987] allows the kernel itself to be customized through the replacement of C++ abstractions. The University of Arizona's *x*-Kernel [Peterson et al. 1989] adopts a similar approach in the context of communication protocols for message-based machines. Both Choices and the *x*-Kernel are best described as *reconfigurable* operating systems; they provide a single programming model defined at system generation time, rather than supporting multiple models at run time. The Agora project [Bisiani & Forin 1988] at CMU defines new mechanisms for process interaction based on pattern-directed events and a stylized form of shared memory. Its goals are to support parallel AI applications using heterogeneous languages and machines.

Mach [Accetta et al. 1986] is representative of a class of operating systems designed for parallel computing. Other systems in this class include Amoeba [Mullender & Tanenbaum 1986], Chorus [Rozier et al. 1988], Topaz [Thacker & Stewart 1988], and V [Cheriton 1984]. To facilitate parallelism within applications, these systems allow more than one kernel-supported process to run in one address space. To implement minimal-cost threads of control, however, or to exercise control over the representation and scheduling of threads, coroutine packages must still be used within a single kernel process. Psyche provides mechanisms unavailable in existing systems to ensure that threads created in user space can use the full range of kernel services (including those that block), without compromising the operations of their peers. In contrast to existing systems, Psyche also emphasizes data sharing between applications as the default, not the exception, distributes access rights without kernel assistance, checks those rights lazily, and presents an explicit tradeoff between protection and performance.

Washington's Presto system [Bershad et al. 1988] is perhaps the closest relative to Psyche, at least from the point of view of an individual application. Presto runs on a shared-memory machine (the Sequent Symmetry), and allows its users to implement many different varieties of processes and styles of process interaction in the context of a single C++ program. As with Agora, however, Presto is implemented on top of an existing operating system, and is limited by the constraints imposed by that system. Where Agora relies on operations supported across protection boundaries in Mach, Presto works within a single language and protection domain, where a wide variety of parallel programming models can be used. Psyche is designed to provide the flexibility of Presto without its limitations, allowing programs written under different models (e.g. in different languages) to interact while maintaining protection.

2.2 Kernel Interface

Psyche is intended to provide a common substrata for the implementation of parallel programming models. In pursuit of this goal we have adopted a low-level kernel interface. We do not expect application programmers to use our interface directly. Rather, we expect them to depend upon library packages and language run-time systems that implement their favorite programming models. The low-level interface allows new packages to be written on demand, and provides well-defined underlying mechanisms that can be used to communicate between models when desired.

Seen in this light, the kernel exists primarily to implement protection and to perform operations (such as accessing page tables and fielding interrupts) that must occur in a privileged hardware state. We recognize that it may be necessary for the sake of performance to place other functions in the kernel as well, but our philosophy is to err initially on the side of minimality, returning functionality to the kernel only if forced to do so.

Because we are interested in providing as much flexibility as possible to the library and language implementor, we are more interested in minimality of function in the kernel than minimality of the kernel interface itself. We are willing, for example, to make heavy use of data structures shared between the kernel and the user, in order to reduce the number of kernel calls required to implement important functions, or to make some feature of the hardware more readily accessible to user-level code.

The Psyche kernel interface is based on four abstractions: the *realm*, the *protection domain*, the *virtual processor*, and the *process* (see Figure 1). Realms (squares) form the unit of code and data sharing. Protection domains (ellipses) are a mechanism for limiting access to realms. Processes (small circles) are user-level threads of control. Virtual processors (triangles) are kernel-level abstractions of physical processors, on which processes are scheduled. Processes are implemented in user space; the other three abstractions are implemented in the kernel.

Each *realm* consists of code and data. The code usually consists of operations that provide a protocol for accessing the data. Since all code and data is encapsulated in realms, all computation consists of the invocation of realm operations. Interprocess communication is effected by invoking operations of realms accessible to more than one process.

Depending on the degree of protection desired, an invocation of a realm operation can be as fast as an ordinary procedure call, termed *optimized* invocation, or as safe as a remote procedure call between heavyweight processes, termed *protected* invocation.

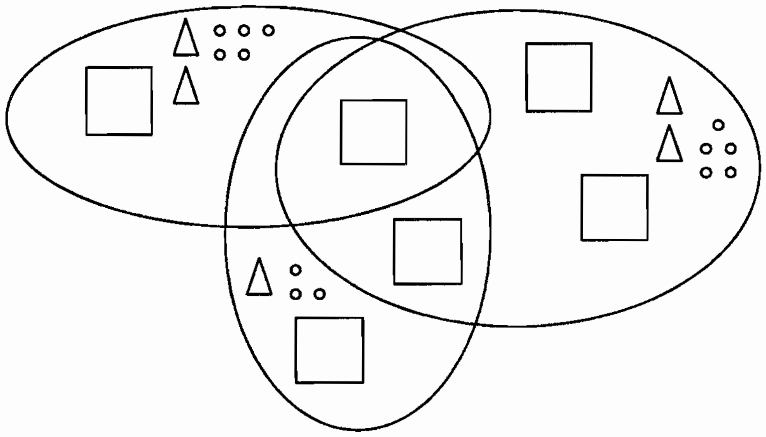


Figure 1: Basic Psyche Abstractions

Unless the caller explicitly asks for protection (by performing an explicit kernel call), the two forms of invocation are initiated in exactly the same way, with the native architecture's jump-to-subroutine instruction. The kernel implements protected invocations by catching and interpreting page faults.

A *process* in Psyche represents a thread of control meaningful to the user. A *virtual processor* is a kernel-provided abstraction on top of which processes are implemented. There is no fixed correspondence between virtual processors and processes. One virtual processor will generally schedule many processes. Likewise, a given process may run on different virtual processors at different points in time. As it invokes protected operations, a process moves through a series of *protection domains*, each of which embodies a set of access rights appropriate to the invoked operation. Each domain has a separate page table, which includes precisely those realms for which the right to perform optimized invocations has been verified by the kernel in the course of some past invocation. In addition to the page table, the kernel also maintains for each protection domain a list of the realms for which the right to perform protected invocations has already been verified.

To facilitate sharing of arbitrary realms at run time, Psyche arranges for every realm to have a unique system-wide virtual address. This *uniform addressing* allows processes to share pointers without worrying about whether they might refer to different data structures or functions in different address spaces. An attempt to touch a realm that is not yet a part of the current protection domain will of course result in a page fault. Given appropriate access rights, the kernel will respond to the fault by *opening* the realm in question for future access from the protection domain – adding it either to the page table of the domain (in the case of optimized access) or to the list of verified targets for protected procedure calls.

Every realm has a distinguished protection domain in which protected calls to its operations should execute. When a process performs a protected invocation of some operation of a realm, it moves to that realm's distinguished domain. The domain therefore contains processes that have moved to it as a result of protected invocations, together with processes that were created in it and have not moved. Processes in different domains may be represented in many different ways – as lightweight threads of various kinds, or requests on the queue of a heavyweight server. The kernel keeps track of the call chains of processes that have moved between protection domains (in order to implement returns correctly), but it knows nothing about how processes are represented or scheduled inside domains, and is not even aware of the existence of processes that have not moved.

In order to execute processes inside a given protection domain, the user must ask the kernel to create a collection of *virtual processors* on which those processes can be scheduled. The number of virtual processors in a domain determines the maximum level of physical parallelism available to the domain's processes. On each physical node of the machine, the kernel time-slices among the virtual processors currently located on that node. A data structure maintained by the user and visible to the kernel contains an indication of which process is being served by the current virtual processor. It is entirely possible (in fact likely) that when execution enters the kernel the currently running process will be different from the one that was running when execution last returned to user space.

To facilitate data sharing between the kernel and the user, the kernel implements a so-called "magic page," which appears to the user as a collection of read-only pseudo-registers. As with the hardware registers, there is a separate magic page on each physical processor, which is mapped into the address space of every resident virtual processor at a well-known virtual address. Contained in the magic page are such kernel-maintained data as the topology of the machine, the local processor number, scheduling statistics, the time of day, and pointers to user-maintained data structures describing the currently-executing virtual processor and its protection domain. These latter two data structures contain such information as the name of the current process, the time at which to deliver the next wall clock timer interrupt, and lists of access rights.

Synchronous communication from the kernel to the virtual processors takes the form of signals that resemble software interrupts. A software interrupt occurs when a process moves to a new protection domain, when it returns, and whenever a kernel-detected error occurs. In addition, user-level code can establish interrupt handlers for wall clock and interval timers.

The interrupt handlers of a protection domain are the entry points of a scheduler for the processes of the domain. Protection domains can thus be used to provide the boundaries between distinct models of parallelism. Each scheduler is responsible for the processes in its domain at the current point in time, managing their representations and mapping them onto the virtual processors of the domain. Realms are the building blocks of domains, and define the granularity at which domains can intersect.

2.3 Advantages for Users

As UNIX-like systems are developed for multiprocessors, a consensus is emerging on multiple kernel-supported processes within an address space. Amoeba, Chorus, Mach, Topaz, and V all take this approach. Most support some sort of memory sharing between processes in different address spaces, but message passing or RPC is usually the standard mechanism for synchronization and communication across address-space boundaries.

On the surface there is a similarity between Psyche and these modern conceptions of UNIX. A protection domain corresponds to an address space. A virtual processor corresponds to a kernel-provided process. Protected procedure calls correspond to RPC. The correspondence breaks down, however, in three important ways.

Ease of Memory Sharing. Uniform addressing in Psyche means that pointers do not have to be interpreted in the context of a particular address map. Without uniform addressing, it is impossible to guarantee that an arbitrary set of processes will be able to place a shared data structure at a mutually-agreeable location at run time. The key and access list mechanism, with user dissemination of keys and lazy checking by the kernel, means that processes do not have to pay for things they don't actually use, nor do they have to realize when they are using something for the first time, in order to ask explicitly for access. Pointers in distributed data structures can be followed without worrying about whether access checking has yet been performed for the portion of the data they reference.

Uniformity of Invocation. Optimized and protected invocations share the same syntax and, with the exception of protection and performance issues, the same semantics. No stub generators or special compiler support are required to implement protected procedure calls. In effect, an appropriate stub is generated by the kernel when an operation is first invoked, and is used for similar calls thereafter. As with the lazy checking of access rights, this late binding of linkage mechanisms facilitates programming techniques that are not possible with remote procedure call systems. Function pointers can be placed in data structures and can then be used by processes whose need for an appropriate stub was not known when the program was written.

First Class User-Level Threads. Because there are no blocking kernel calls, a virtual processor is never wasted while the user-level process it was running waits for some operation to complete. Protected invocations are the only way in which a process can leave its protection domain for an unbounded amount of time, and its virtual processor receives a software interrupt as soon as this occurs. These software interrupts provide user-level code with complete control over the implementation of lightweight processes, while allowing those processes to make standard use of the full set of kernel operations.

2.4 Ramifications for Kernel Implementation

The nature of the Psyche kernel interface poses several unusual challenges for the kernel implementor. Promiscuous sharing in a uniform user address space, for example, means that valid addresses in a given protection domain are likely to be sparse. If the hardware does not support sparse address spaces well, the kernel implementor may be forced to page the page tables, rely exclusively on the TLB, or use hardware page tables to cache a more flexible data structure maintained in software. Because each realm has a unique system-wide virtual location, user-level code must be relocated at load time unless it is position independent. Though this is not strictly a kernel-level issue in Psyche (since the loader is outside the kernel), it has ramifications for a host of user-level tools, and also for code sharing. If users create many realms from the same load image, the kernel can arrange for these realms to share code segments (at a significant savings in aggregate working set size) only if they are position independent.

Uniform addressing also means that large system workloads are unlikely to fit within the 32-bit addressing range of many microprocessors. The need within the kernel to access data structures on a large number of processors creates a competing demand for address space, which only makes matters worse. We expect the 32-bit limit to be lifted by emerging architectures. In the meantime, we have developed techniques to economize on virtual addresses in the kernel (as described in the following section), and have devised (though not implemented) additional techniques for use in user space.

Another area of kernel design that is complicated by Psyche abstractions is the handling of page faults. The default cause of a page fault is an error on the part of the user-level program. Traditional operating systems overload page faults to implement demand paging as well. They may also use them to compensate for missing hardware features, as in the simulation of reference bits on the VAX [Babaoglu & Joy 1981], or the provision of more than one virtual-to-physical mapping on machines with inverted page tables [Rashid et al. 1988]. Page faults may be used for lazy initiation of expensive operations, as in copy-on-write message passing [Fitzgerald & Rashid 1986], time-critical process migration [Theimer et al. 1985; Zayes 1987], or the management of locality in machines with non-uniform memory access times [Bolosky et al. 1989; Cox & Fowler 1989]. In Psyche, page faults are given two more functions: the opening of realms for optimized access and the initiation of protected procedure calls. Implementing these functions efficiently, without compromising the performance of other operations triggered by page faults, is a potentially difficult task.

Perhaps the most important challenge for the Psyche kernel (one not addressed in this paper) is to implement software interrupts and protected procedure calls efficiently. Because these mechanisms lie at the heart of scheduling, device management, and cross-domain communication, it is imperative that they work as fast as possible. It is not yet clear whether techniques similar to those used in LRPC [Bershad et al. 1990] or the Synthesis kernel [Massalin & Pu 1989] can be made to work well without an explicit, user-specified "bind to service" operation. Our hope is that substantial amounts of time can be saved by automatically precomputing linkage mechanisms for protected procedure calls when a realm is first opened for access.

3. Kernel Organization

3.1 Basic Kernel Structure

The Psyche kernel interface is designed to take maximum advantage of shared-memory architectures. Since we are interested in concepts that scale, we assume that Psyche will be implemented on NUMA (non-uniform memory access) machines. A NUMA host is modeled as a collection of *clusters*, each of which comprises processors and memories with identical locality characteristics. A Sequent or Encore machine consists of a single cluster. On a Butterfly, each node is a cluster unto itself. The proposed Encore Gigamax [Wilson 1987] would consist of non-trivial clusters.

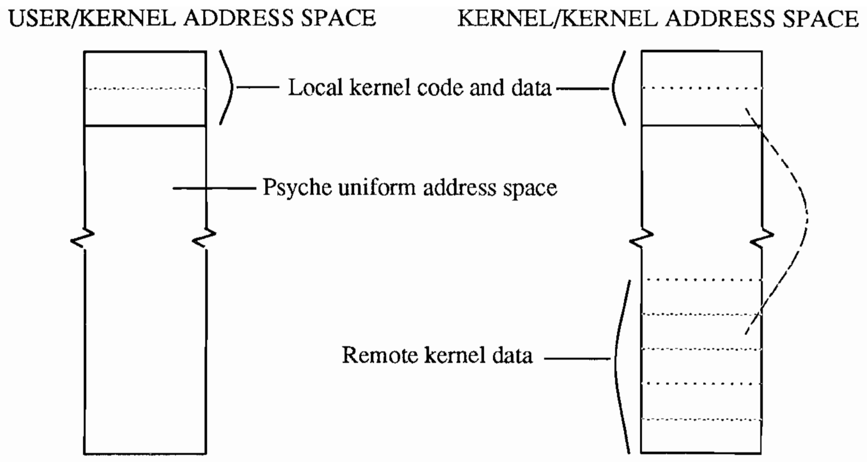
Our most basic kernel design decisions have been adopted with an eye toward efficient use of very large NUMA machines.

1. The kernel is *symmetric*; each cluster contains a separate copy of most of the kernel code, and each processor executes this code independently. The alternative organization, in which particular kernel services would execute on particular processors, does not seem to scale well to different numbers of nodes. We allocate kernel scheduling and memory management data structures on a per-cluster basis. Kernel functions are performed locally whenever possible. The only exceptions are interrupt handlers (which must be located where the interrupts occur) and some virtual memory daemons, which consume fewer resources when run on a global basis.
2. As in most modern operating system implementations, little distinction is made between parallelism in user space and parallelism in the kernel. Kernel resources are represented by parallel-access data structures, not by active processes. A

virtual processor that traps into the kernel enters a privileged hardware state (and begins to execute trusted code, but continues to be the same active entity that it was in user space. This approach to process structure is not motivated by NUMA architecture *per se*, but tends to minimize the cost of simple kernel calls, and simplifies the management and scheduling of virtual processors.

3. The kernel makes extensive use of shared memory to communicate between processors, both within and between clusters. Ready lists, for example, are manipulated remotely in order to implement protected invocations. The alternative, a message-passing scheme in which instances of the kernel would be asked to perform the manipulations themselves [LeBlanc et al. 1989], was rejected as overly expensive. Most modifications to remote data structures can be performed asynchronously; the remote kernel will notice them the next time the data is read. Synchronous inter-kernel interrupts are used for I/O, remote TLB invalidation, and insertion of high-priority processes in ready queues.
4. Kernel data structures do not share the uniform address space with user programs. If individual instances of the kernel are to access the data structures of arbitrary other instances, then the need for scalability will dictate that the space available for kernel data structures be very large. Existing 32-bit architectures provide barely enough room for the user-level uniform address space, and cannot be stretched to accommodate the kernel's needs as well. In order to access arbitrary portions of both the uniform address space and the kernel data structures, each kernel instance must be prepared to remap portions of its address space, or to switch between multiple spaces.

A diagram of our current kernel addressing structure appears in Figure 2. The code and data of the local kernel instance are mapped into the same locations in each address space, making switches between spaces easy. The data of the local kernel includes the "magic page" of kernel-maintained data that is readable by the user. The user/kernel address space also contains the protection domain of the currently running virtual processor (at


USER/KERNEL ADDRESS SPACE
KERNEL/KERNEL ADDRESS SPACE
Local kernel code and data
Psyche uniform address space
Remote kernel data
Figure 2: Kernel Address Spaces

its correct position in the uniform address space), and the kernel/kernel address space contains the data of every kernel instance. Local kernel data structures appear at two different locations in the kernel/kernel space.

When executing in user space, the virtual processors of separate protection domains must have separate page tables. There is therefore a separate user/kernel address space for every protection domain. Address space switches are required in the kernel in order to examine data in user space and in other instances of the kernel. Unfortunately, they are also required in order to examine user data in more than one protection domain, something that is needed for protected invocations of realm operations.

An alternative implementation of the two-address-space structure would employ both a separate page table for each protection domain and a single, universal user/kernel address space that included the code and data of every protection domain. This latter address space would be used only in the kernel. Virtual processors would switch to it when executing any operation that required access to more than one protection domain. It is not yet clear whether the savings in address space changes would exceed the cost of additional page table management. A change to the uniform address space (destruction of a realm, for example) must currently be made only to the page tables of protection domains that include the relevant realm. Depending on whether one shares page tables across clusters (a question that has its own set of tradeoffs), a large number of page tables might be used to represent a universal user/kernel address space, and these would need to be kept consistent as well.

When we designed the two-address-space structure, it was our hope that a typical kernel call would begin operation in the user/kernel space, examining any needed user data. It would then switch to the kernel/kernel space in order to perform the requested operation, and switch back again in order to write results into user space. We have found in practice that the situation is seldom so simple. The implementation of protected procedure calls, for example, must switch back and forth between address spaces several times in order to obtain all the information it needs from user space, and compare it against relevant data in the kernel.

Many address space changes stem from our decision to place any sharable data structure in the kernel/kernel space, even if it is accessed locally most of the time. Data in the local portion of the kernel/kernel space are also visible at the upper end of both address spaces, but at a different virtual address. Accessing them at this location would require repeated address arithmetic, not only to find things initially, but also to follow any pointers found inside. Accessing data at their "official" location, however, frequently requires a switch between user/kernel and kernel/kernel spaces.

An alternative approach to managing kernel data would be to use only the user/kernel address space, but to augment its two kernel segments (code and local data) with (1) a segment shared permanently by all instances of the kernel, containing frequently-used data structures that do not need to scale with the size of the machine, and (2) one or more segments that can be re-mapped dynamically to address the local data structures of some other kernel instance. Other than the few data structures contained in the permanently shared segment, data would be allocated among the local variables of whichever kernel instance is expected to access them most often. Another kernel wishing to gain access would use its temporary segment(s). Since temporary segments appear at a different virtual address than the local data segment, address arithmetic would be required when accessing data of another kernel instance.

This alternative approach is similar to one employed in Chrysalis, the original operating system for the Butterfly [BBN 1988]. We were not happy with the temporary segment mechanism in Chrysalis; it took too long to remap it. Costs would be lower, however, on the current generation of hardware, and could probably be made comparable to the cost of switching between existing address spaces, by preallocating and initializing page table fragments for temporary segments, and patching them into the kernel address space when needed, rather than creating them on the fly. The tradeoff between our current two-address-space structure and the remapping scheme with temporary segments would come down to the choice between easy access to the data of other kernel instances, and the ability to examine kernel and user data simultaneously. We are beginning an audit of kernel data structures and access patterns in an attempt to quantify the difference between these options. We expect to re-consider the design of kernel page tables once we have a better understanding of the relative costs involved.

3.2 Synchronization

A traditional uniprocessor operating system often obtains mutual exclusion for kernel data structures by disabling preemption in the kernel. In a shared-memory multiprocessor, this simple technique no longer works. Data structures can be modified remotely. An inventory of kernel data structures in Psyche reveals that almost none of them (other than those local to a subroutine) are private to a single processor, though most are *usually* accessed locally. Explicit synchronization is almost always required when accessing kernel data. We have therefore opted to allow preemption in the kernel. The overhead incurred by explicit locking remains to be measured. We expect it to be significant, but still less than the cost of message-passing between kernels to avoid the need for locking.

We have found a need in the kernel for four major types of synchronization. (We also have a facility for all-processor barrier synchronization, but this is used only for kernel initialization.)

Disabled preemption. Those few data structures that *are* processor-local (buffers for the per-processor console, for example) can be protected by disabling preemption of virtual processors. To allow nesting of locks, the kernel maintains a "preemption level" that is incremented when entering a critical section and decremented when leaving. At the end of a quantum, the clock handler forces a context switch only if the counter is zero. If the counter is positive, the handler sets a flag and returns. The code that decrements the preemption level counter causes a context switch on behalf of the clock handler if the flag is set and the level has returned to zero.

Locked-out interrupts. Interrupt masking is used solely to synchronize with device handlers. Data structures that can be accessed by interrupt handlers *and* by remote processors must be protected by both a spin lock and locked-out interrupts.

Spin locks. Spin locks are the most frequently-used locks in the kernel. They are used to protect critical sections of small, bounded length. The spin lock implementation disables preemption of virtual processors, to ensure that the bound is not violated by an inopportune context switch.

Semaphores. True blocking semaphores are used when a virtual processor must wait for a condition that may not happen soon. Unlike most other operating systems, the Psyche kernel will rarely block a virtual processor (and all of its processes) for an unbounded length of time. The exceptions are operations (such as demand page-in) that must complete before *any* process can use the virtual processor, and a kernel call whose explicit purpose is to block the current virtual processor (pending interrupts) when it has no processes to run. To block itself, a virtual processor (1) disables preemption, (2) writes its name down where some other virtual processor will find and resume it at an appropriate time, and (3) invokes the kernel scheduler, thereby saving its state, switching to another virtual processor, and re-enabling preemption. The same basic mechanism could be used to implement monitors. To avoid a timing window, anyone who wants to resume a virtual processor must spin until it state is completely saved.

3.3 Bootstrapping

Initialization of Psyche proceeds in three distinct phases. The first phase loads an instance of the kernel onto a single processor of the bare machine and starts it running. The second phase replicates the kernel and starts the remaining processors. The third phase brings up an initial set of user-level programs.

Initial booting of Psyche employs a two-step process. When power-cycled, the Butterfly Plus executes a serial-line loader in ROM. We initially used this facility to load the entire kernel, but found frequent reloads to be increasingly painful as the kernel grew. We now transfer a small bootstrap program that initializes the Ethernet interface and loads the bulk of the kernel using a naive (busy-wait) implementation of UDP. This loader took about a man-month to construct – less than we expected; we wish we had written it sooner. To reduce the need for reloading, we checksum the kernel code and save a copy of its data in memory, so that the current version can be restarted in response to a request on the console line.

The second phase of Psyche initialization must be accomplished on any multiprocessor, but is more difficult on multicomputers and multiprocessors with distributed memory than it is on bus-based machines. In Psyche the kernel is symmetric, but a substantial amount of initialization code is executed only on the initial "king node" processor (see Figure 3).

```
startup:
    initialize local data structures
    if king
        initialize shared data structures
        replicate kernel code and data
        start other processors
    barrier_sync
    write information for other processors
        into shared data structures
    . . .
    barrier_sync
```

Figure 3: Kernel Initialization

The king node begins execution by initializing both its own local data structures and certain of the shared (kernel-kernel) data structures required for basic communication with other instances of the kernel. Among other things it allocates space for a shared heap and creates several objects in that heap. Addresses of these objects, and of the heap itself, are written into variables in the king node's local data. Once this basic initialization is complete, the king node copies its code and data (*in its current state*) into the local memories of all the other processors. It writes into each processor's interrupt vector table the address of the same initialization routine it is itself executing, and delivers remote interrupts that cause each processor to begin execution in that routine. It then falls into a barrier synchronization routine, waiting for the other processors to catch up with it.

On each other processor, the kernel repeats the king node's initialization of its local data structures, then joins the synchronization barrier. Because the king node replicates its own, initialized data structures, rather than a pristine version of the kernel, non-king nodes know from the beginning of execution whatever the king node knew at replication time, including the addresses of various shared data structures. In order to initialize virtual memory, device management, and synchronous inter-kernel interrupts, the various processors then proceed through several additional barrier episodes. Between each pair of barriers, processors read information written into shared data by their peers before the most recent barrier, use this to calculate new information about themselves, and write that new information into shared data for use beyond the next barrier.

For the third phase of Psyche initialization (user-level programs), the kernel on the king node creates a single primordial realm in a single protection domain, containing a single user-level process. This process executes code to create additional realms, resulting in an initial set that currently includes a command-interpreter *shell*, a user-level *loader*, a *name server*, and a simple network *file server*.

In response to a typed command, the shell communicates with the loader to start a user program (see Figure 4). The loader reads the header of the object file to discover the size of the program. It uses a system call to create an empty realm (which the kernel places at an address of its choosing), and then loads the program into the realm, resolving references to a small set of standard external symbols (examples of these include entry points of the general-purpose name server, the file system, the loader itself, and reentrant subroutine libraries). At this point the loader returns to the shell. The realm itself is still passive.

The shell uses another system call to create an initial virtual processor in the new realm's protection domain. The kernel immediately provides this virtual processor with a software interrupt that allows it to initialize its scheduling package. The shell then performs a protected invocation to the program's main entry point, whose address was returned by the loader. In most programs it is expected that the main entry point will be in a library routine that initializes standard I/O and repackages command line arguments before branching to the user's main routine. If desired, the main routine can then register an external interface with the name server.

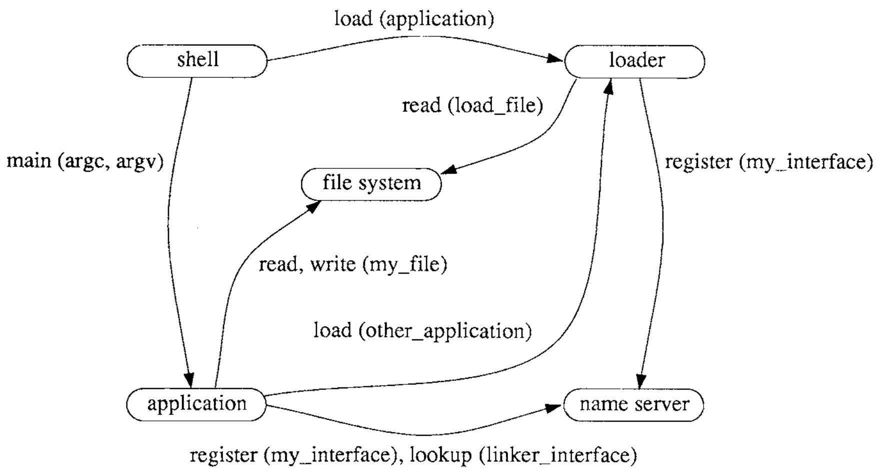


Figure 4: Basic User-level Utilities

4. *Resource Management*

4.1 *Devices*

Consistent with the philosophy of user-level flexibility and kernel minimality, Psyche allows memory-mapped devices to be accessed directly from user space. The `make_realm` system call allows the user, with appropriate access rights, to create a realm at a specified virtual or physical address. On the Butterfly Plus, Multibus devices are accessed at special virtual addresses and VME devices are accessed at physical addresses. By creating a memory-mapped realm, a user-level program obtains the ability to read and write device registers without the assistance of the kernel. For polled I/O devices, this interface limits the kernel's role in device management to that of initialization. In our robotics lab, polled I/O from user space is used to access low-level image processors and robot eye controllers over the VME bus.

Devices that require interrupts are currently implemented in the kernel in Psyche. On the Butterfly Plus, there are currently three types of device in this class:

1. A pair of serial lines connects a Multibus on the king node of the Butterfly to a UNIX host machine.
2. An Ethernet interface can be plugged into the Multibus on the king node, or on any other node.
3. On every node there is a on-board pair of serial lines.

One of the Multibus serial lines on the king node is used by Psyche as a console. The other is used for debugging (see Section 5). Our intent is that both these devices be available only in the kernel. For development purposes we have implemented a set of system calls that allow a user program to read and write from the console. These calls will be removed as soon as we finish a network-based remote login server.

We use the Ethernet interface for network file service, bootstrapping of the kernel, and communication via UDP. Bootstrapping is encapsulated in the kernel. UDP send and receive commands are part of the kernel interface. Simple remote file service is built on UDP. To increase flexibility, we would like to move as much network functionality as possible out of the kernel and into user space. In the limit, one might translate hardware interrupts directly into software interrupts, and deliver them to appropriate virtual processors. It is currently unclear to what extent this goal can be achieved with reasonable performance. Much will depend on the speed with which we can deliver software interrupts. Experience with the Synthesis kernel at Columbia [Massalin & Pu 1989] suggests that very high speed may be possible. Psyche is a more complex system than Synthesis, however, and we may be forced to leave part of the network protocol stack (ARP, IP, UDP, etc.) inside the Psyche kernel.

Our robot arm is currently controlled from Psyche via network communication with a Sun to which the robot is attached. We are in the process of developing device support for the Butterfly's node-local serial lines, in order to control the robot directly. Our facilities for memory-mapped I/O can be used to access device registers from user space, but as with the network there is a need for interrupt management. We are experimenting with direct user-space handling of serial line interrupts, and with a small interrupt handler in the kernel that buffers data for the user. Our experience with the serial lines will be used to drive later experimentation with the Ethernet device.

4.2 Virtual Memory

Simple physics dictates that memory cannot simultaneously be located very close to a very large number of processors. Memory that can be accessed quickly by one node of a large multiprocessor will be distant from many other nodes. Even on a small machine, price/performance may be maximized by an architecture with non-uniform memory access times.

On any Non-Uniform Memory Access (NUMA) machine, performance will depend heavily on the extent to which data reside close to the processes that use them. In order to maximize locality of reference, data replication and migration can be performed in hardware (with consistent caches), in the operating system, in compilers or library routines, or in application-specific user code. The last option can place an unacceptable burden on the programmer and the first, if feasible at all in large machines, will certainly be expensive.

Because of our interest in scaleable machine, much of our early work on the Psyche virtual memory system was aimed at the integration of NUMA management with other kernel functions. We were particularly concerned with the interaction of data replication and migration with demand paging and protected procedure calls, and with the structuring of a VM system that could balance all these concerns without becoming unmaintainable. The design we developed [LeBlanc et al. 1989a] has four distinct abstraction layers. The lowest layer encapsulates physical page frames and page tables. The next layer provides the illusion of uniform memory access time through page replication and migration. The third layer provides a default pager for backing store and a mechanism for user-level pagers. The final layer implements the Psyche uniform address space and protection domains. Page faults may indicate events of interest to any of the layers; they percolate upward until handled.

With the exception of demand paging, most of this VM system was implemented by the summer of 1989. During the implementation process, however, we grew increasingly uncomfortable with the amount of kernel code required, and with the extent to which VM policies were being dictated by the kernel. As with interrupt-driven I/O, our goal is now to move much of this functionality into user space, so that user-supplied replication and migration policies can use application-specific knowledge to build on kernel operations that replicate or migrate pages. Real-time programs will not want a NUMA locality management system to move their pages at unpredictable times. Demand paging may not be needed at all on a multiprocessor with lots of physical memory.

Application-level research on Psyche can proceed for some time without kernel NUMA management (network I/O, by contrast, is crucial). In pursuit of locality management in user space, we have therefore backed the functions of data migration and replication out of the current implementation. In the resulting simple VM system, each virtual address space is represented by a hardware page table on each node that may execute in it. Each hardware page table is a cache for a more compact representation of the virtual address space. To avoid the performance impact of instruction fetching through the Butterfly switch, code is replicated automatically on every processor that executes it. The creator of a realm can specify whether the replication should occur when the realm is created, when it is opened for access in a particular protection domain, or page-by-page on demand. The kernel maintains a mapping from virtual addresses to realms, which is consulted when a page fault occurs, allowing the kernel to determine whether an attempt to touch an inaccessible page constitutes an error, a protected invocation, or an initial use of a realm that should be mapped in for optimized access.

To our simple VM system we plan to re-introduce elements of the more sophisticated system, either in the kernel or (preferably) in user space, as concrete needs emerge. The exact division of labor between the kernel and user has not been determined, but our goal is to strike a balance between efficiency of implementation and user-level flexibility.

4.3 Scheduling

Psyche employs a two-level scheduling system. The kernel scheduler is responsible for multiplexing virtual processors on physical processors. The actual work performed by a user, however, is determined by how processes are allocated to virtual processors. This latter mapping is performed in user space in Psyche, both to allow scheduling mechanisms to accommodate many different process representations, and to allow scheduling policies to benefit from application knowledge.

Scheduling plays an extremely important role in many applications, both for performance reasons, as in the case of computation-intensive multiprocess programs, and for correctness reasons, as in real-time applications. Data and code sharing, synchronization, communication, remote code execution, and process migration can all have serious effects on the performance of the system, which the scheduler can mitigate by using appropriate policies. For example, under round-robin scheduling and without co-scheduling [Ousterhout 1981], a process may find itself unable to run (due to synchronization constraints) when it is allocated the processor, or it may immediately block or spin due to a synchronization constraint. The added overhead caused by extraneous context switches and spinning in this situation slows the system as a whole, which in turn causes more processes to suffer from the same problem. A user-level scheduler, using application-specific knowledge, might make scheduling decisions that avoid this situation. Scheduling policy could be tuned for each application, and better decisions would be likely.

To support user-level scheduling, the Psyche kernel provides the user with virtual processors, software interrupts, and magic pages with system statistics. The kernel schedules virtual processors on each physical processor in a round-robin fashion. The user creates and destroys virtual processors, and can assign them to physical processors. The user creates the processes that run on the virtual processors and has complete control over the scheduling of processes on the virtual processors of each protection domain.

Software interrupts occur when a scheduling decision has to be made, such as when a process leaves a protection domain for a protected invocation, a new process arrives in a protection domain, or a timer expires. The user can define handlers for each type of interrupt. When an interrupt occurs, a data structure shared between the kernel and the user is set to contain the state of the process that was running. The handler can use this information to perform a context switch on the virtual processor, or to make long-term scheduling decisions. In order to facilitate decisions, the kernel maintains statistics in magic pages, which it updates periodically. These statistics include the load (virtual processor run queue length) on each physical processor, the association between virtual processors and physical processors, and the current state and accumulated execution time of each virtual processor. Potentially they could also include information on page faults, migrations, and replications.

Using our mechanisms for user-level scheduling, we have implemented a thread package that allows users to create and schedule lightweight processes in a shared address space. The user can specify the number of virtual processors to be created and can assign processes to run on them. A different scheduler can be used on each virtual processor, and schedulers can be changed on the fly. We plan to use this package to evaluate existing multiprocessor scheduling algorithms and experiment with new ones.

We also plan to introduce real-time scheduling into Psyche. We are aware of the difficulties of adding real-time support to a pre-existing operating system, but we believe that our minimal kernel, which provides the ability to perform user-level scheduling and to define process models in user space, is sufficient to explore real-time issues. In addition, we can segregate real-time processes within a subset of available processors, where they can be managed with different policies, or dedicate the physical memory of a processor (without paging) to a particular application. Given the many real-time process models in existence, each of which incorporates different timing constraints, it would be impractical to expect a single scheduler within the kernel to meet the timing constraints for all models simultaneously. By exploiting Psyche mechanisms, we can achieve predictability for processes such as device handlers, whose computation time, period, or deadline is known, while allowing flexible scheduling polices for processes whose behavior is not as well known.

5. *Kernel Debugging*

The most important tool we have constructed for Psyche is a mechanism for remote, source-level debugging, in the style of the Topaz TeleDebug facility developed at DEC SRC [Redell 1988]. An interactive front end runs on a Sun workstation using the GNU *gdb* debugger. *gdb* comes with a remote debugging facility; relatively minor modifications were required to get it to work with Psyche. The debugger communicates via UDP with a multiplexor running on the Butterfly's host machine. The multiplexor in turn communicates with a low-level debugging stub (*lld*) that underlies the Psyche kernel.

The multiplexor allows many different debugging sessions to be underway simultaneously, each of them talking to a different Psyche node. It communicates with *lld* via one of the serial lines connected to the Butterfly king node. The interrupt handler for the debugging line accumulates input until it recognizes a special debugger packet termination character. It looks inside the packet to determine the node for which the packet is intended, and either wakes up the instance of *lld* on its own node or causes a remote interrupt to effect the same result on another node.

The protocol between *gdb* and *lld* is strictly request-reply, and does not require reliable communication. Since *lld* is stateless and never issues a request, a debugger can be attached to any instance of the kernel at any time. *lld* is also very simple, by design. It was the first portion of the kernel to be written, and has proven extremely useful. With it we are able, for example, to single-step through interrupt drivers using all the facilities of a high-quality source-level debugger.

One question that arises in the design of a remote debugging facility is where to keep track of the instructions that underlie breakpoints. If breakpoint information is kept on the host machine the target system becomes unusable if the debugger crashes. Topaz therefore maintains its breakpoint information in the debugging stub on the target. The guiding philosophy behind this decision is that it should always be possible to debug, so long as the debugging stub remains intact. For the sake of simplicity, we initially kept our breakpoints on the Sun. *lld* tended to break more often than *gdb* anyway, and only infrequently did we find ourselves unable to continue debugging because of lost information. As the kernel has become more stable and our debugging needs more sophisticated, this situation has begun to change. Particularly annoying is the fact that the kernel cannot be restarted if its code has been corrupted by breakpoint trap instructions. We are planning to move breakpoint data into *lld*. Only the underlying instructions will be maintained; associated conditions, commands, enable status, etc. will still be kept in *gdb*.

We have observed one serious interaction between *lld* and the virtual memory system which in hindsight can be used to illustrate a few important lessons. *lld* was designed as a kernel debugger. We are working on the design of a user-space debugger, but until it becomes operational we have been using *lld* in a makeshift fashion to debug user programs. Unfortunately, while the kernel is permanently resident, pages of user-level programs may not be present when the debugger tries to look at them. In particular, we quickly discovered that typed requests from the user during a debugging session could cause *lld* to trigger a page fault, something it was not designed to handle, and which caused the machine to crash. In attempting to enhance *lld* to avoid this common accident, we found it easier to use existing VM code to force a page into memory than to write a new routine to determine whether the page *could* be forced into memory if desired. We therefore produced a version of *lld* that would automatically (and silently!) force user-level pages into memory.

Meanwhile, we had for some time been experiencing intermittent unexpected user-level bus errors. The source of the trouble turned out to be that we had neglected to write the clause of the bus error handler that handles page faults for instructions that span page boundaries (this is a special case because of details of the processor pipeline). We had great trouble finding the bug because it was hidden by *lld*. The natural course of action when the bug occurred was to ask *lld* to print the offending instructions. Unfortunately, *lld* would then fetch the instructions as *data*, silently faulting the page into memory and eliminating evidence of the bug. From this we learned that (1) you can't depend on a debugger to debug something on which the debugger depends; (2) you need to *remember* the things on which the debugger depends; and (3) a kernel debugger probably shouldn't depend on something as complicated as VM.

We intend to use our existing debugging facility as the base for user-level debugging of multi-model programs. As part of a related research project we have developed sophisticated techniques for monitoring and analysis of parallel programs, including deterministic replay of fundamentally non-deterministic programs [Fowler et al. 1988]. We are currently developing extensions to our techniques that will allow the developers of programming language run-time packages and communication libraries to define debugging interfaces that allow a debugger to talk to the user in terms of model-specific, high level abstractions. Our goal is (1) to provide a framework that unifies our toolkit-based approach with the various techniques for program monitoring and visualization that have been described in the literature and (2) to develop a step-by-step methodology and corresponding tools for parallel program analysis over the entire software development cycle, from initial debugging to performance modeling and extrapolation.

6. Status and Future Plans

A full evaluation of our kernel structuring decisions will require more tuning and measurement than we have been able to undertake to date. Though we can not yet quantify the tradeoffs between our current two-address-space structure and the alternative remapping structure, we strongly suspect that kernel interaction through shared memory will be more efficient than message passing, even with the extra locking that memory sharing requires. We believe that the use of many different kinds of locks, each tuned to a particular type of sharing, will help to minimize the cost of synchronization. We recognize that locking constitutes a conceptual burden for kernel programmers, but we have not found that burden to be unreasonable.

We have chosen to share data structures between the kernel and the user whenever synchronous communication is not required. Descriptive data structures, for example, allow the kernel to inspect access rights, process names, software interrupt vectors, and realm operation descriptions without requiring the user to provide them as explicit arguments to system calls. Kernel-maintained data allow the user to read the time, compute load averages, or examine scheduling statistics. Shared flags and timers allow the user to disable software interrupts, ask for timer interrupts, or anticipate virtual processor preemption (to avoid acquiring a spin lock), all without a kernel trap. A modifiable indication of the current process allows the user to switch between processes in the same protection domain without the kernel's intervention. We expect user/kernel sharing to significantly reduce the number of system calls (and related kernel overhead) incurred by typical programs.

We found that the layering of our original VM system made it relatively easy to understand and modify, but were reluctant to keep all its functionality in the kernel. We are not yet sure how much of that functionality can be recreated in user space, nor are we sure how quickly we will be able to propagate faults through the layers. Experience with systems such as Swift [Clark 1985] and the *x*-Kernel [Peterson et al. 1989] indicates that layering need not preclude efficiency. It would be premature, moreover, to use performance as the principal design goal for a multiprocessor VM system, when it is not yet clear how to divide labor between the kernel and the user, or to manage the interactions between VM, protection, and scheduling. In as much as these structural issues form a central focus of our work, we are happy with the clarity and modularity of our layered design.

Many of the open issues in kernel design come down to a question of performance. From the user's point of view, the performance of software interrupts and protected procedure calls will be particularly important. One potential source of overhead is the frequent use of locks for synchronization of access to data structures shared between nodes. Another is the memory management context switches induced by the multiple-address-space structure of the kernel. A third is the propagation of page faults through layers of the VM system. In the course of quantifying these costs (and recoding to reduce them), we also plan to investigate the ramifications of allowing virtual processor preemption in the kernel.

Multi-model parallel programming forms the focus of our user-level work [Scott et al. 1990]. We are evaluating the extent to which Psyche kernel primitives facilitate use of, and interaction between, disparate process and communication models. We are also investigating appropriate user-level tools for multi-model debugging, parallel program configuration, management of access rights, name service, and file service. At the interface between the kernel and the user, we are developing mechanisms for user-level device control, locality management, and adaptive real-time scheduling. With the bulk of the kernel in place, and the first major applications running, we are in a position to address these issues in earnest.

Our evaluation effort will depend in large part on experience with applications, many of which will come from the department's robot lab. The lab includes a custom binocular "head" on the end of a PUMA robot "neck." Images from the robot's "eyes" feed into a MaxVideo pipelined image processor. Higher-level vision, planning, and robot control have been implemented on a uniprocessor Sun. Real-time response, however, will require extensive parallelization of these functions. The Butterfly implementation of Psyche provides the platform for this work. Effective implementation of the full range of robot functions will require several different models of parallelism, for which Psyche is ideally suited. In addition, practical experience in the vision lab will provide feedback on the Psyche design.

Acknowledgments

Neal Gafter and John Mellor-Crummey were the original implementors of our remote debugging system. Sean Colbath, Yenjo Han, Kurt Jones, John Leadley, Dave Tilley, and Jack Veenstra have helped build kernel or user-level software. Bill Bolosky contributed to early design discussions.

References

M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986, pages 93-112.

O. Babaoglu and W. Joy, "Converting a Swap-Based System to do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, December 1981, pages 78-86.

[BBN] BBN Advanced Computers Incorporated, "Chrysalis® Programmers Manual, Version 4.0," Cambridge, MA, 10 February 1988.

B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo and M. Schwartz, "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems," *IEEE Transactions on Software Engineering SE-13*:8 (August 1987), pages 880-894.

B. N. Bershad, E. D. Lazowska, H. M. Levy and D. B. Wagner, "An Open Environment for Building Parallel Programming Systems," *Proceedings of the ACM/SIGPLAN PPEALS 1988 – Parallel Programming: Experience with Applications, Languages and Systems*, 19-21 July 1988, pages 1-9. In *ACM SIGPLAN Notices 23*:9.

B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, "Lightweight Remote Procedure Call," *ACM TOCS 8*:1 (February 1990), pages 37-55. Originally presented at the *Twelfth ACM Symposium on Operating Systems Principles,* 3-6 December 1989.

R. Bisiani and A. Forin, "Multilanguage Parallel Programming of Heterogeneous Machines," *IEEE Transactions on Computers 37*:8 (August 1988), pages 930-945. Originally presented at the *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, CA, 5-8 October 1987.

W. J. Bolosky, R. P. Fitzgerald and M. L. Scott, "Simple But Effective Techniques for NUMA Memory Management," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989, pages 19-31. In *ACM Operating Systems Review 23*:5.

C. M. Brown, R. J. Fowler, T. J. LeBlanc, M. L. Scott, M. Srinivas and others, "DARPA Parallel Architecture Benchmark Study," BPR 13, Computer Science Department, University of Rochester, October 1986.

R. Campbell, G. Johnston and V. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)," *ACM Operating Systems Review 21*:3 (July 1987), pages 9-17.

D. Cheriton, "The V Kernel – A Software Base for Distributed Systems," *IEEE Software 1*:2 (April 1984), pages 19-42.

D. Clark, "The Structuring of Systems Using Upcalls," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pages 171-180. In *ACM Operating Systems Review 19*:5.

A. L. Cox and R. J. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989, pages 32-44. In *ACM Operating Systems Review 23*:5.

R. Fitzgerald and R. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM TOCS 4*:2 (May 1986), pages 147-177. Originally presented at the *Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985.

R. J. Fowler, T. J. LeBlanc and J. M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," *Proceedings, ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 5-6 May 1988, pages 163-173. In *ACM SIGPLAN Notices 24*:1 (January 1989).

R. Hayes and R. D. Schlichting, "Facilitating Mixed Language Programming in Distributed Systems," *IEEE Transactions on Software Engineering SE-13*:12 (December 1987), pages 1254-1264.

M. B. Jones, R. F. Rashid and M. R. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing," *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985, pages 225-235.

T. J. LeBlanc, "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986, pages 463-466.

T. J. LeBlanc, M. L. Scott and C. M. Brown "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proceedings of the ACM SIGPLAN PPEALS 1988 – Parallel*

Programming: Experience with Applications, Languages, and Systems, 19-21 July 1988, pages 161-172.

T. J. LeBlanc, J. M. Mellor-Crummey, N. M. Gafter, L. A. Crowl and P. C. Dibble, "The Elmwood Multiprocessor Operating System," *Software – Practice and Experience 19*:11 (November 1989), pages 1029-1056.

T. J. LeBlanc, B. D. Marsh and M. L. Scott, "Memory Management for Large-Scale NUMA Multiprocessors," TR 311, Computer Science Department, University of Rochester, March 1989a.

B. Liskov, R. Bloom, D. Gifford, R. Scheifler and W. Weihl, "Communication in the Mercury System," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988, pages 178-187.

H. Massalin and C. Pu, "Threads and Input/Output in the Synthesis Kernel," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989, pages 191-201. In *ACM Operating Systems Review 23*:5.

S. J. Mullender and A. S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," *The Computer Journal 29*:4 (1986), pages 289-299.

J. K. Ousterhout, *Medusa, A Distributed Operating System*, UMI Press, 1981.

L. Peterson, N. Hutchinson, S. O'Malley and M. Abbott, "RPC in the *x*-Kernel: Evaluating New Design Techniques," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989, pages 91-101. In *ACM Operating Systems Review 23*:5.

R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers 37*:8 (August 1988), pages 896-908. Originally presented at the *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, 5-8 October 1987.

D. Redell, "Experience with Topaz TeleDebugging," *Proceedings, ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 5-6 May 1988, pages 35-44. In *ACM SIGPLAN Notices 24*:1 (January 1989).

M. Rozier and others, "Chorus Distributed Operating Systems," *Computing Systems 1*:4 (Fall 1988), pages 305-370.

M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proceedings of the 1988 International Conference on Parallel Processing*, V. II – Software, 15-19 August 1988, pages 255-262.

M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors," TR 309, Computer Science Department, University of Rochester, March 1989.

M. L. Scott, T. J. LeBlanc and B. D. Marsh, "Multi-Model Parallel Programming in Psyche," *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 15-16 March, 1990, pages 70-78.

C. P. Thacker and L. C. Stewart, "Firefly: A Multiprocessor Workstation," *IEEE Transactions on Computers 37*:8 (August 1988), pages 909-920. Originally presented at the *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, 5-8 October 1987.

M. Theimer, K. Lantz and D. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pages 2-12. In *ACM Operating Systems Review 19*:5.

R. H. Thomas and W. Crowther, "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors," *Proceedings of the 1988 International Conference on Parallel Processing*, V. II – Software, 15-19 August 1988, pages 245-254.

A. W. Wilson, Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Fourteenth Annual International Symposium on Computer Architecture*, 2-5 June 1987, pages 244-252.

B. Yamauchi, "Juggler: Real-Time Sensorimotor Control Using Independent Agents," *Optical Society of America Image Understanding and Machine Vision Conference*, 1989 Technical Digest Series, V. 14, June 1989, pages 6-9.

E. Zayas, "Attaching the Process Migration Bottleneck," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 8-11 November 1987, pages 13-24. In *ACM Operating Systems Review 21*:5.