Process Synchronization in the UTS Kernel

Lawrence M. Ruane AT&T Bell Laboratories

ABSTRACT: Any operating system kernel has some form of *process synchronization*, allowing a process to wait for a particular condition. The traditional choice for UNIX systems, the event-wait mechanism, leads to race conditions on multiprocessors.

This problem was initially solved in Amdahl's UTS multiprocessing kernel by replacing the event-wait mechanism with Dijkstra semaphores. The kernel, however, became noticeably more complicated and less reliable when based on semaphores. This has led us to develop a race-free multiprocessor event-wait mechanism with some novel properties. A few common synchronization techniques have emerged, the most complex of which was verified correct with the supertrace protocol validation system *spin*. A new scheduling approach with per-CPU run queues reduces the number of unnecessary context switches due to awakening all waiting processes.

The overall approach is claimed to be simple, efficient, and reliable.

1. Introduction

Broadly, *process synchronization* refers to processes blocking themselves until the system changes state in some way. In the original UNIX kernel, processes wait for *events* [Thompson 1978]. An event is represented by an arbitrary integer (the *channel*), chosen by convention to be the address of the kernel data structure whose state-change the process is interested in.

For example, suppose a resource, represented by a data structure pointed to by `r`, must be used by only one process at a time. Here is the standard protocol to lock the resource:[1]

```
while (r->lock)
    sleep(r);
r->lock = 1;
```

If the resource is *locked* `lock`(is nonzero), the process suspends itself until the resource becomes *free* `lock`(is zero). The process then locks the resource and proceeds to use it. The `lock` flag is often called a *sleep lock*.

Since the kernel is *non-preemptable* (the kernel will not switch to another process unless explicitly requested via `sleep()`), there is no *race condition* in the test and set of the sleep lock: it is impossible for two processes to see the lock cleared and then both set the lock and use the resource simultaneously.

Non-preemptability also simplifies the kernel tremendously, since it implies that sleep locks are only needed when a process might sleep at a "lower level," during which no other process should access the resource. For example, while a process waits for I/O completion for a particular *inode* (file), the process holds the inode's sleep lock.

After changing the state of the system in a way that might be relevant to sleeping processes, a process must wake them up using the agreed-upon channel:

```
r->lock = 0;
wakeup(r);
```

1. For simplicity, we ignore process scheduling priority upon waking up, the second argument to `sleep()`.

An awakened process must treat the wakeup as *advisory*; it must re-evaluate the wait condition – thus the `while` instead of an `if` – for two reasons. First, any number of the processes might have run since the the wakeup was issued, any of which might have changed the wait condition. Second, more than one event can map to the same channel, so the wait condition might not have changed at all.

Our experience has been that while event-wait is not as efficient as some other synchronization methods (that wake only one waiting process, for example), it puts the least burden on kernel algorithms because it closely approximates the busy-wait model. Under pure busy-wait, there are no process blocking or resumption concerns at all – every process has its own processor. Algorithms can use "polling" loops that are much more complex than the simple `while` statement above; then the polling is slowed down by slipping in calls to `sleep()` and `wakeup()` at the right places, without changing the algorithms. Appendix *A* compares event-wait to other forms of monitors.

2. Race Conditions

If the resource is released by an interrupt handler, a race condition can cause the wakeup to be lost. A process wishing to use the resource tests the lock, finds it set, and commits to go to sleep. But before actually calling `sleep()`, an interrupt occurs. The interrupt handler frees the lock and does the `wakeup()` (which has no effect), and returns to the interrupted process, which goes to sleep, possibly forever.

Disabling interrupts between the test of `r->lock` and the transition into the wait state eliminates the race condition:

```
spl6(); /*disable interrupts*/
while (r->lock)
    sleep(r);
r->lock = 1;
spl0(); /*enable interrupts*/
```

Interrupts are automatically enabled after the process is switched out, and are disabled again when the process switches back in.

On multiprocessing systems, disabling interrupts on the current CPU is not sufficient – the interrupt handler can run on another CPU. Also, the race condition prevented by non-preemptability on a uniprocessor can now occur: two processes on different CPUs attempt to lock the same resource at about the same time; the processes both see `lock` set to zero, set `lock`, and simultaneously use the resource – practically a guaranteed disaster [Bach 1986].

3. Semaphores

The problems with the event-wait method on multiprocessors have led to several implementations that substitute *semaphores* `P()`(and `V()` routines) [Dijkstra 1968]. Multiprocessing kernels based on semaphores exist for the IBM/370 architecture (UTS and the UNIX System for System/370 [Felton et al. 1984]), the AT&T 3B20A computer, and the Sequent systems [Beck et al. 1987], among others.

In supporting and developing new kernel features for UTS over the years, we have found semaphores troublesome. There are the obvious portability problems of converting algorithms based on event-wait to a different synchronization model. But more importantly, we have encountered many difficulties due to the nature of the semaphore mechanism itself. Sometimes semaphores simplify kernel algorithms by providing just the right abstraction, but often their use leads to significantly more complicated algorithms.

What is the root of the problems with semaphores? Semantically, if not in actual implementation, the semaphore mechanism encapsulates the common

```
while (resource is locked)
    sleep();
set resource lock;
```

synchronization paradigm we have already seen (or a slight generalization if the semaphore is initialized to a value greater than one). There are really three separate parts to this: testing for resource availability; blocking; and resource allocation.

But when the algorithms are complex, we would often like to have direct control over the individual parts. For example, the buffer cache `getblk()` routine requires sophisticated "polling" loops [Bach 1986]. In other situations, we would like to (indivisibly) test for the availability of several different resources before allocating any of them, to simplify the handling of a non-available resource. In cases like these, which are all too common in the real world, event-wait works better.

By analogy, consider the `printf()` routine. It really does two separate things – formatting a string and writing it to standard output – that often happen to be needed together. But if only `printf()` were available, the programmer's life would be difficult indeed. This is why `sprintf()` and `puts()`, the two components of `printf()`, are made available. Providing high-level primitives such as `printf()` and `P()` is fine, as long as one allows access to the lower level ones as well.

A case study showing some of the problems with semaphores in actual use is given in Appendix *B*.

Due to these problems, we decided to develop a modified event-wait mechanism for UTS.

4. Spinlocks

Before getting into the multiprocessor event-wait implementation, we must review spinlocks. All multiprocessing kernels use spinlocks to provide low-level *processor* (as opposed to *process*) synchronization.

A *spinlock* is a variable that takes on values 0 and 1. The `spinlock()` routine indivisibly changes a given spinlock variable from 0 to 1 (retrying if necessary, without giving up the CPU). The UTS implementation of spinlocks is typical – a busy-wait loop around an indivisible test-and-set instruction. The `freelock()` routine simply sets the given spinlock to 0. (Spinlocks are known by other names, such as *primitive semaphores* in Bach [1986].)

Spinlocks provide short-term exclusion; they must never be held across sleeps (otherwise every processor may try to get the spinlock at about the same time, causing system-wide deadlock since the process holding the lock cannot run to free it).

Similarly, interrupts must be disabled while holding any spinlocks (at least any that an interrupt handler might try to get), lest the interrupt handler deadlock on a held spinlock. This requirement is met in UTS in the simplest possible way: the entire kernel is non-interruptible. (Another simple way is to increment a per-CPU counter in `spinlock()`, decrement it in `freelock()`, and only enable interrupts if the counter is zero.)

The sleep queue is a typical example. In UTS, the sleep queue is hashed by channel number, with one spinlock per hash chain. While manipulating or referencing a hash chain, one must hold its spinlock:

```
struct hsque {
    long lk;
    struct proc *proc;
} hsque[64];
#define sqhash(c) (&hsque[(c>>3)&63])

sleep(chan)
{
    struct hsque *hp = sqhash(chan);

    spinlock(&hp->lk);
    link current proc onto hp->proc;
    freelock(&hp->lk);
    swtch();
}

wakeup(chan)
{
    struct hsque *hp = sqhash(chan);

    spinlock(&hp->lk);
    move all procs waiting on chan
      from hp->proc to run queue;
    freelock(&hp->lk);
}
```

5. *A Multiprocessor Event-Wait Mechanism*

The modified event-wait mechanism extends the use of spinlocks to prevent the race conditions described earlier. A new routine, `sleepl()` (for "sleep while holding a (spin)lock"), takes the address of a spinlock as an additional argument. `wakeup()` is unchanged. `sleepl()` has the following form:

```
sleepl(chan, lp)
    long *lp;
{
    struct hsque *hp = sqhash(chan);

    spinlock(&hp->lk);
    freelock(lp);
    link current proc onto hp->proc;
    freelock(&hp->lk);
    swtch();
    spinlock(lp);
}
```

This is just `sleep()` except that after locking the sleep queue, the argument lock `lp` is freed. We will see below that this is the earliest possible time `lp` can be freed, minimizing contention. As a matter of convenience, it is relocked as late as possible before returning.

Here is the multiprocessor protocol for acquiring a resource:

```
spinlock(&r->lk);
while (r->lock)
    sleepl(r, &r->lk);
r->lock = 1;
freelock(&r->lk);
```

(The resource structure now includes a spinlock `lk`.) It is easy to see how the race condition in which two processes use the resource at the same time is prevented: the test and set of the sleep lock (`lock`) is made indivisible by the use of the spinlock (`lk`).

A new primitive, waitlock(), solves the lost wakeup race condition in a particularly elegant and (to our knowledge) unique way. As the name implies, it waits until the given spinlock is free, but does not lock it:

```
waitlock(lp)
    long *lp;
{
    while (*lp)
        ;
}
```

This primitive is used when releasing the resource:

```
r->lock = 0;
waitlock(&r->lk);
wakeup(r);
```

To see how the wakeup cannot be lost, suppose process *A*, trying to lock the resource, has seen lock set, but has not yet called sleepl(). Now process *B* on another CPU, releasing the resource, sets lock to zero. At this point *B* spins in waitlock() until *A* gets the sleep queue hash spinlock and releases lk, at which point *B* can proceed to wakeup(). The first thing wakeup() does is get the (same) sleep queue hash spinlock, which it cannot do until *A* is queued to the sleep hash chain. Therefore, *B* is sure to find *A* and move it to the run queue.

It is not necessary, as we first thought, to hold the spinlock throughout the entire sequence to release the resource. Using waitlock() minimizes contention on lk.

The rule for preventing a lost wakeup is that the spinlock must be held, even if momentarily, between the time the awaited condition is made true and the wakeup. waitlock() fulfills this requirement since it is equivalent to a spinlock() followed by a freelock().

Interestingly, no race results from lock being inspected at the same time it is being set (to zero) on another CPU. Also, the fact that lock is set to zero without holding any spinlock provides a simple counter-example to our early impression that a spinlock protects data structures – that if a spinlock needs to be held for any change of a variable, it needs to be held for every change. Here we see that a spinlock actually protects *transitions* – the spinlock must be held while changing `lock` from zero to one, but not the reverse. This sort of thing occurs in other places.[2]

6. *Combining the Two Uses of a Spinlock*

Often, the above rule is satisfied without using `waitlock()`. This happens when the same spinlock is used to both protect a data structure and prevent a lost wakeup.

For example, suppose we have a pool of resources (data structures), with the free ones on a singly linked list:

```
struct {
    long lk;
    struct resource *head;
} freelist;
```

When a process wants one of the resources and there is none available, it sleeps:

```
spinlock(&freelist.lk);
while (freelist.head == NULL)
    sleepl(&freelist, &freelist.lk);
r = freelist.head;
freelist.head = r->next;
freelock(&freelist.lk);
return(r);
```

Putting `r` back on the freelist is done with:

```
spinlock(&freelist.lk);
r->next = freelist.head;
freelist.head = r;
freelock(&freelist.lk);
wakeup(&freelist);
```

2. For example, in UTS it is necessary to hold the per-process spinlock for some process state (sleep, run, zombie, etc.) transitions, but not for others.

Here, `freelist.lk` is used both to protect the freelist and to prevent a lost wakeup. When returning a resource to the freelist, there is naturally a moment at which both the wait condition is true (`freelist != NULL`) and the spinlock is held, so no `waitlock()` is needed.

7. "Wanted" Flags

To improve performance, a "wanted" flag is sometimes added to the protocol to avoid unnecessary wakeups. The uniprocessor method of allocating the resource becomes:

```
while (r->lock) {
    r->wanted = 1;
    sleep(r);
}
r->lock = 1;
```

The resource is released with:

```
r->lock = 0;
if (r->wanted) {
    r->wanted = 0;
    wakeup(r);
}
```

Using multiprocessor event-wait, the resource is allocated with:

```
spinlock(&r->lk);
while (r->lock) {
    r->wanted = 1;
    sleepl(r, &r->lk);
}
r->lock = 1;
freelock(&r->lk);
```

and released with:

```
r->lock = 0;
waitlock(&r->lk);
```

```
    if (r->wanted) {
        r->wanted = 0;
        waitlock(&r->lk);
        wakeup(r);
    }
```

It is easy to see that since the wakeup is conditioned on the test of `wanted`, a `waitlock()` must be done *before* the test. It is harder to see the need for the second `waitlock()`, but one can imagine that without it process *A* is about to clear `wanted`; process *B* sets `wanted` and is about to go to sleep; process *A* clears `wanted` and does the `wakeup()`, which *B* misses.

A very complex sequence with this ending, requiring three processors, was indeed found by the *spin* supertrace protocol verifier [Holzmann 1990]. With the second `waitlock()` in place, *spin* has proven this protocol safe for at least 5 processors. The *spin* model is given in Appendix *C*.

Surprisingly, `wanted` can be getting turned on and off by different processors at the same time (with an indeterminate result), yet there is no race condition. This is unusual in a multiprocessing kernel.

This protocol provides very low contention on spinlocks, and especially fast resource releases. Normally, `lk` is not held and `wanted` is not set, in which case no spinlocks at all are needed to release the resource.

8. Conditionally Acquiring a Sleep Lock

Another interesting protocol used in uniprocessing kernels involves waiting for a sleep lock to become available, but not locking the sleep lock unless sleep is required. Here is an outline of `free()`, which puts a given disk block on the free list of a given filesystem. The in-memory part of the free list and the sleep lock are kept in the filesystem's superblock, pointed to by `fp`. If `free()` discovers that the in-memory free list is full, it synchronously writes the free list to disk and resets the list to empty. Then `free()` adds the given disk block the free list.

```
while (fp->lock)
    sleep(fp);
if (fp->free is full) {
    fp->lock = 1;
    synchronously write fp->free to disk;
    set fp->free to "empty";
    fp->lock = 0;
    wakeup(fp);
}
add block being freed to fp->free;
```

This routine doesn't bother to change the sleep lock or do a `wakeup()` in the normal case that the free list is not full.

We can get the same advantage in the multiprocessing version of this routine:

```
spinlock(&fp->lk);
while (fp->lock)
    sleepl(fp, &fp->lk);
if (fp->free is full) {
    fp->lock = 1;
    freelock(&fp->lk);
    synchronously write fp->free to disk;
    set fp->free to "empty";
    spinlock(&fp->lk);
    fp->lock = 0;
    wakeup(fp);
}
add block being freed to fp->free;
freelock(&fp->lk);
```

The spinlock is held across the `wakeup()`, which could be prevented in most cases by adding a `wanted` flag. Notice that the `wakeup()` happens before we are done using the resource, which is fine since we hold the spinlock in the interval.

9. *What Provides Processor Mutual Exclusion?*

The previous example provides a good lead-in to the general question of what type of lock provides *processor* mutual exclusion. At one extreme, such as with the sleep queue hash chains, spinlocks provide all of it – there are no sleep locks. This applies to many other parts of UTS as well: the run queue, the free list of process structures, the hash lists of the buffer cache, etc.

In other cases, such as the most recent example, the responsibility is split between a spinlock and a sleep lock. The spinlock provides the processor mutual exclusion for the statement

add block being freed to fp->free;

(we hold the spinlock but not the sleep lock) while the sleep lock protects

set fp->free to "empty";

(we hold the sleep lock but not the spinlock).

At the other extreme, spinlocks only provide synchronization for manipulating the sleep lock (including preventing missed wakeups); the data structure is manipulated with just the sleep lock held. This is also very common, applying to individual buffer cache headers, inodes, etc.

Of course sleep locks provide all *process* mutual exclusion.

10. *Scope of Sleep Locks*

When a sleep lock provides processor mutual exclusion, it must be held, not just across sleep, but across all manipulations of the resource. We have discovered parts of the standard kernel where the scope of a sleep lock is too small for a multiprocessor. In a uniprocessing kernel, an inode can be unlocked and then still manipulated; since the kernel is non-preemptable, no other process can use the inode until the current process gives up the CPU.

On a multiprocessor, this doesn't work. From the instant the sleep lock is released, a process on another processor can get the lock and start to use the inode, even though the first process is still running. So, in a few cases, the unlock of the resource had to be moved (delayed) to the point at which the resource was really finished being used.

11. Indivisibly Freeing a Sleep Lock

There are many parts of the kernel where a process gets the sleep lock of a resource, but then discovers that the resource is "not ready" in some sense, requiring the process to wait. Before blocking, however, it must free the sleep lock to allow another process to change the state of the resource to "ready."

One place this occurs is in the IPC message facility. Each message queue has a sleep lock. Suppose process *R* wishes to receive a message from a certain queue, but after locking the queue, discovers that it is empty. It then frees the sleep lock and sleeps on the address of a particular member (qp->qnum) of that queue structure (not the address of the overall structure, since that is used for the sleep lock). When another process *S* sends (enqueues) a message to the queue, it does a wakeup on the address of qp->qnum.

On a multiprocessor, without any precautions, *R* can miss the wakeup, since as soon as it frees the sleep lock in preparing to sleep, *S* can get the sleep lock, enqueue the message, and do the wakeup – before *R* is asleep. The solution for *S* is:

```
/* get sleep lock: */
spinlock(&msg_lk);
while (qp->lock)
    sleepl(qp, &msg_lk);
qp->lock = 1;
freelock(&msg_lk);
/* deposit the message: */
...
wakeup(&qp->qnum);
/* free the sleep lock: */
qp->lock = 0;
```

```
    waitlock(&msg_lk);
    wakeup(qp);
```

and for *R*:

```
    loop:
    /* get sleep lock: */
    spinlock(&msg_lk);
    while (qp->lock)
        sleepl(qp, &msg_lk);
    qp->lock = 1;
    freelock(&msg_lk);
    ...
    /* no messages to read;
     * free sleep lock and
     * wait for a message:
     */
    spinlock(&msg_lk);
    qp->lock = 0;
    wakeup(qp);
    sleepl(&qp->qnum, &msg_lk);
    freelock(&msg_lk);
    goto loop;  /*re-evaluate*/
```

(There is one global spinlock for the entire IPC message facility, rather than one spinlock per resource, although it could have been done the other way.) The important point is that *R* gets the spinlock before it frees the sleep lock, so there is no way *S* can lock the sleep lock until *R* is asleep, since *S* must get the spinlock before getting the sleep lock. This means the wakeup cannot be lost. (The spinlock is be held across a `wakeup()`; as before, this can be prevented in most cases by use of a `wanted` flag.)

Notice that after *S* gets the sleep lock and queues a message, it does not have to do a `waitlock()` before the `wakeup()`; *R* cannot be about to go to sleep, or *S* would not have been able to get the spinlock, which is needed to get the sleep lock.

12. Semaphores on Top of Event-Wait

A *semaphore* is a counter initialized to the number of some class of resources (or one for mutual exclusion). When nonnegative, the semaphore gives the number of available resources; when negative, the absolute value of the semaphore gives the number of processes waiting for resources. P() is called when allocating a unit of the resource; it decrements the counter and blocks if negative. A call to V() releases a unit by incrementing the counter and, if processes are waiting, unblocking one of them.

Here is the old UTS implementation:

```
struct semaphore {
    long lk;
    int count;
    struct proc *p;
};

P(s)
    struct semaphore *s;
{
    spinlock(&s->lk);
    if (--s->count < 0) {
        queue curproc onto s;
        freelock(&s->lk);
        swtch();
    } else
        freelock(&s->lk);
}

V(s)
    struct semaphore *s;
{
    struct proc *p;
    spinlock(&s->lk);
    if (s->count++ < 0) {
        p = dequeue one process from s;
```

```
            freelock(&s->lk);
            setrq(p);
        } else
            freelock(&s->lk);
    }
```

Since semaphores can easily be implemented using event-wait, this is what we did in UTS, replacing the "primitive" semaphore implementation above. Here is the event-wait version of semaphores:[3]

```
    struct semaphore {
        long lk;
        int count;
    };

    P(s)
        struct semaphore *s;
    {
        spinlock(&s->lk);
        while (--s->count < 0)
            sleepl(s, &s->lk);
        freelock(&s->lk);
    }

    V(s)
        struct semaphore *s;
    {
        spinlock(&s->lk);
        if (s->count < 0) {
            s->count = 1;
            freelock(&s->lk);
            wakeup(s);
        } else {
            ++s->count;
            freelock(&s->lk);
        }
    }
```

3. For simplicity, we ignore the case of *interruptible* semaphores, although they are easy to implement.

(Since a negative count indicates one or more processes are sleeping, it is functionally a "wanted" flag.) This reimplementation of semaphores changed their semantics slightly, and unfortunately broke some things. (It is unclear if the original programmers were aware they were depending on such detailed semantics of semaphores.)

The old semaphores were "strict": if process *A* blocks on a `P()`, and process *B* executes `V()` on the same semaphore, *A* is guaranteed to get the semaphore next, even if some other process, *C*, tries to do a `P()` before *A* runs. *C* blocks; *A* switches in and gets the semaphore.

The new semaphores, shown above, are "lazy": in the previous example *C* gets the semaphore; *A* would switch in only to find the semaphore still locked, and go back to sleep.

One part of the kernel that broke is the named pipe open protocol. A per-inode (per-pipe) semaphore, `i_fwcnt`, maintains the number of writers. When a process tries to open a named pipe for read, it does a `P()` on that semaphore, which blocks if there are no writers. What can happen is that the writer comes along; increments the writer count (does a `V()`, which awakens the reader); quickly writes a small amount of data; and closes the pipe, which decrements the writer count (with `P()`) to zero. *Then*, the reader gets the CPU, but finds the semaphore count is still zero, and goes back to sleep! The reader remains stuck in the open protocol.

This was easily fixed by porting the standard event-wait based code, which is simpler anyway. Other problems of this sort were fixed the same way.

13. Convoys

In going from semaphores to event-wait, two performance factors work in opposite directions. Semaphores have the advantage that processes never wake up unnecessarily. That is, once a process is on the run queue, it is guaranteed to obtain the resource.

Arising from this very fact, however, are semaphore *convoys*, which hurt performance [Lee et al. 1987]. Convoys can occur when processes lock and release a semaphore repeatedly without sleeping. Such high-contention semaphores are typical of database environments, for example.

If process *q* holds a semaphore, and process *p* blocks on that semaphore, a convoy begins. When *q* releases the semaphore, *p* is put on the run queue. Then *q* loops back to re-obtain the semaphore, but blocks because the semaphore is owned by *p*. Process *p* runs, uses the resource, and releases the semaphore, giving it back to *q*, and so on. The two processes alternate on every use of the resource, dramatically increasing the number of context switches.

Convoys tend to be self-perpetuating, because once started, the resource-locked duration per use increases dramatically. It now includes the time for the new owner process to be scheduled to run, in addition to the time the process actually uses the resource. This in turn increases the probability that another process will block on the semaphore.

Once identified, convoys can usually be solved by redesigning the kernel algorithms, but at the expense of simplicity.

With event-wait, convoys are not possible, because even though a process has been awakened and is on the run queue, expecting to use the resource, it does not yet *own* the resource, so the current process can lock it again without blocking.

14. The "Thundering Herd" Problem

This delightful expression refers to a problem that derives from the fact that `wakeup()` makes *all* processes waiting on the event runnable, not just one. They race to lock the resource, one gets it, and the others go back to sleep. Semaphores have the advantage that processes never wake up unnecessarily. Once a process is on the run queue, it is guaranteed to obtain the resource.

However, Beck et al. [1987] point out that there usually isn't a problem with event-wait on uniprocessors, since by the time each process runs, the resource is generally available. For example, suppose that while a buffer is busy being read into from disk, five processes block trying to read it. When the I/O is done, all five go on the run queue. Each one in turn finds the data in the buffer valid, copies it to user space, frees the buffer, and gives up the CPU for an unrelated reason (e.g., sleeping on another resource). As long as each process does not sleep while holding the resource, there are no unnecessary context switches.

On a multiprocessor, this condition is not sufficient. While the five processes in the example are waiting on the run queue, they are available to be run by other CPUs. This problem becomes worse as CPUs are added to the system.

We addressed this problem by giving a private run queue to each CPU. `wakeup()` (actually, `setrq()`) puts processes on the current CPU's run queue. The global run queue remains, but only holds processes with user priority (that is, processes ready to return to the user program that have deferred to a better priority process); thus user priorities are honored.

When a CPU wants a process to run (in `swtch()`), it first looks in its own run queue, then in the user priority queue, and finally, since it doesn't make sense for a CPU to go idle if there is really work to do somewhere, it "raids" other CPU's run queues.

The result is that the number of unnecessary context switches is reduced to about the level of a uniprocessor, as long as there is sufficient work in the system to prevent a lot of raiding (and if there isn't that much work, we probably don't care about unnecessary context switches anyway). Another advantage is that the contention on the run queue spinlock, usually the worst in the system, is far lower.

To demonstrate the full effect of this approach, we started four identical processes, each reading the first 64 blocks of the same file forever:

```
fd = open("bigfile", 0);
for (;;) {
     lseek(fd, 0, 0);
     read(fd, buf, 64*4096);
}
```

Then we started another four reading a different file. The system had two processors.

Using the new run queue scheme and scheduler, the system did 90% fewer context switches per second, and got 50% more "real work" done (bytes read) per second. The effect on a typical workload is not yet known.

What happens is that each resource becomes informally associated, for perhaps a few seconds at a time, with a particular CPU; each CPU becomes a defacto *server* for a set of resources. If process *a*, running on CPU 1, blocks while trying to access a resource that is being used by process *b*, running on CPU 2, then when *b* releases the resource, *a* goes on CPU 2's run queue. Thus, the users of a certain resource tend to migrate to the same CPU as other users of the same resource. This may also improve the processor cache efficiency. The global user-priority run queue provides load balancing among CPUs.

15. Conclusions

Our experience with event-wait on UTS has clearly been positive. The code, except for per-CPU run queues, has been incorporated into the UTS product beginning with release 2.1. It is conceptually easy to understand and leads to simpler and more reliable kernel algorithms. There are just two new primitives, `sleepl()` and `waitlock()`, with trivial implementations.

The importance of easing porting from the standard uniprocessing kernel is hard to overstate, especially as the kernel (unfortunately) becomes more complicated. Porting the uniprocessor buffer cache code, for example, was almost as easy as adding `spinlock()` to the top of each function, `freelock()` to the bottom, and replacing each `sleep()` with `sleepl()` (a single spinlock protects the entire buffer cache). None of the algorithms needed modification. (Of course, it's not always quite that easy!)

Our performance measurements so far have been encouraging. Without the per-CPU run queues (which is still being evaluated), the throughput of the event-wait kernel was consistently measured to be 0.8% to 1.2% better than the semaphore based kernel on a typical mix (compiles, *greps*, *troffs*, etc.), depending on the load. The only difference was that in the event-wait kernel, semaphores were implemented using event-wait rather than as primitives; none of the other algorithms were changed (i.e., they used semaphores).

A greater relative improvement can be expected as more of the kernel is converted to use event-wait directly, and as a result of per-CPU run queues.

This approach might provide a good basis for future multiprocessing kernels, since it is in no way specific to the S/370 architecture or to UTS.

Acknowledgements

Thanks to Joan Kalmanek and Craig Harmer for their contributions to this project. Gerard Holzmann provided lots of help with using his protocol verifier.

Appendix A: Event-wait and Monitors

The event-wait mechanism is actually a variant of the traditional *monitor* synchronization model as described in Ben-Ari [1990]. A particular implementation of monitors is defined by several independent characteristics, many of which are described in the literature as if they were dependent. What all forms of monitors have in common is the existence of two routines, `wait()` and `signal()` (sometimes with different names). The `signal()` primitive *always* suspends the current process (unlike the semaphore `P()` operation); `wait()` unblocks one or more processes.

Each of the following sections describes a separate monitor characteristic.

1. implicit vs. explicit mutual exclusion

Traditionally, a monitor is described as consisting of a set of procedures and data, encapsulated in a `monitor` construct provided by the language. In some cases, a monitor is a *class*, from which many instances of the encapsulated data can be created. The language provides implicit per-instance mutual exclusion among the procedures within the monitor, ensuring single-threaded execution.

The alternative is explicit mutual exclusion of code segments using calls to primitives such as `spinlock()` and `freelock()` on a multiprocessor or `spl6()` and `spl0()` on a uniprocessor. This is how most "real systems" are written, not only because C does not support monitors, but because of the added flexibility of being able to apply the mutual exclusion exactly where needed, without having to pull code segments into separate monitor functions. (Of course, the encapsulation approach can be followed by convention.)

2. blocking vs. non-blocking mutual exclusion

If a process finds the mutual exclusion lock unavailable, the system can either suspend the process, or spin until the lock is free. In the practical world, spinning is almost always best, because the lock duration should be small (otherwise a design problem is indicated).

Also, blocking mutual exclusion presents difficulties for interrupt handlers, which cannot sleep. The only use of a blocking mutual exclusion lock in UTS is in the memory manager (which should probably be redesigned to use spinlocks). When a disk paging request completes, but the disk interrupt handler cannot get the memory manager lock, it queues a data structure to a special list (protected by a spinlock) of "paging operations completed but not yet processed through the memory manager." Then, whenever the memory manager lock is freed, any items on this list are processed. This obviously complicates the memory manager.

3. condition variables vs. wait channels

In the usual description of monitors, `wait()` is passed a pointer to a *condition variable*. The process is queued on the condition variable, to be later dequeued by `signal()`. There is no state associated with a condition variable except the list of processes.

In the UNIX kernel, the argument to `wait()`'s equivalent (-`sleep()`) is an arbitrary integer, the channel, which usually is hashed to one of many sleep queues for speed.

The difference is almost purely cosmetic. The arbitrary integer approach simplifies matters by not requiring variables to be allocated (and named!). Also, condition variables might use a significant amount of space if they are members of highly replicated structures.

4. immediate resumption vs. normal scheduling

In the usual description of monitors, if `signal()` finds a process waiting, that process is made runnable; but more than that, it is sure to be the next process to run in the monitor.

The advantage is that the condition which was made true by the process doing the `signal()` need not be reevaluated by the awakened process – it is guaranteed to be true. Usually `signal()` is required to be the last statement of the monitor routine so that the signaling process can not change the state of the monitor before the awakened process runs.

The alternative is that `signal()` simply puts the process on the run queue, and an arbitrary amount of time might pass until it runs. Since the state of the monitor might change between the signal and when the awakened process runs, it must reevaluate the condition that caused it to block. This is the approach taken in Mesa [Lampson & Redell 1980].

How is immediate resumption implemented? If the mutual exclusion lock is the blocking type, then `signal()` can simply *not* free the lock (and not have `wait()` try to get the lock before returning). But this approach won't work if the mutual exclusion lock is a spinlock, unless the run queue is bypassed entirely, with the signaling process directly resuming the process being signaled.

Immediate resumption can lead to convoys (discussed elsewhere in this paper), since a process cannot reenter the monitor after doing a `signal()` without giving up the CPU.

5. *wake one vs. wake all*

This alternative is actually a subset of the "normal scheduling" case above; it does not apply to immediate resumption. If `signal()` finds more than one process waiting, should it make just one process runnable, or all waiting processes?

Waking just one process is sometimes more efficient (however, see the "Thundering Herd" section of this paper), but adds to the complexity of the monitor. When the process resumes inside the monitor, it bears a heavy burden of responsibility: if the process finds the condition true (i.e., this was not an unnecessary wake up), but decides not to make the sleep condition false for any reason, it must signal the next waiting process. For example, after sleeping on a busy buffer in the `getblk()` buffer cache routine (see Appendix B), the process cannot blithely `goto loop` as if it has just come into the routine for the first time; it might choose a different buffer and thus fail to make the original buffer busy. That could cause processes waiting for the original buffer to wait forever.

Also, the mapping from conditions to events must be unique across the entire system.

Waking everyone, also called a *broadcast* signal, allows each awakened process to worry only about itself with regard to the event that has been signaled. It minimizes complexity by allowing algorithms to be written in the busy-wait model, adding process synchronization at the last minute.

Monitors Summary

Characteristics 1 through 3 are mostly superficial, and do not affect the structure of kernel algorithms much. Characteristics 4 and 5 can have a significant impact on algorithms.

The standard uniprocessing UNIX kernel uses explicit mutual exclusion, wait channels, normal scheduling, and wake all. UTS uses in addition non-blocking mutual exclusion (spinlocks).

Appendix B: Problems with Semaphores

One of the areas in which semaphores have given us serious problems is the section of the kernel that implements the buffer cache. By looking at how the design evolved as problems were discovered, we can gain insight into the question of when semaphores are appropriate and when they are not. (While the story of this evolution is not historically accurate, it's not far off.)

Although the following discussion applies to both uniprocessing and multiprocessing kernels, we will ignore the multiprocessing issues; the structures of the algorithms are the same.

The `getblk()` routine finds a buffer in the disk buffer cache [Bach 1986]. Here is an outline of `getblk()` using event-wait (for simplicity, we assume there are no delayed-write (dirty) buffers):

```
loop:
if (find buffer in cache) {
    if (buffer BUSY) {
        sleep(&buffer);
        goto loop;
    }
    set buffer BUSY;
    freecnt--;
    take buffer off freelist;
    return(buffer);
}
if (freecnt == 0) {
    sleep(&freelist);
    goto loop;
}
freecnt--;
set first buffer on freelist BUSY;
take first buffer off freelist;
reassign buffer;
return(buffer);
```

There are two "polling" loops, corresponding to the two goto statements. In the first loop, the cache hit case, the current process *p* polls the buffer it wants until it is not busy. In the second loop, for a cache miss, *p* polls for the freelist to be non-empty. The loops must enclose the search as well as the test for BUSY because anything can happen while *p* sleeps.

For the cache hit case, when the buffer is no longer busy, *p* sets it busy, unlinks it from somewhere in the freelist (the buffer must be there since it wasn't busy) and returns.

In the case of a cache miss, when the freelist becomes non-empty, *p* takes an old buffer from the freelist and reassigns it (i.e., change the disk block this buffer will henceforth be associated with).

Converting this routine to use semaphores seems straightforward. Since *p* might wait for two kinds of resources (a particular buffer, and any buffer on the freelist), we replace each buffer's BUSY bit with a semaphore, and replace `freecnt` with a semaphore, `freesema`, initialized to the number of buffers. Here is a first try using semaphores:

```
if (find buffer in cache) {
     P(buffer);
     P(freesema);
     take buffer off freelist;
     return(buffer);
}
P(freesema);
P(first buffer on freelist);
take first buffer off freelist;
reassign buffer;
return(buffer);
```

It's pretty easy to see why this version doesn't work. If *p* finds the buffer, and blocks while locking it (`P(buffer)`), the buffer might be reassigned, causing *p* to return the wrong buffer. Similarly, if the process doesn't find the buffer, it might block on the freelist semaphore, allowing the buffer it is looking for to appear in the cache. This results in two buffers corresponding to the same disk block – bad news.

Here is a fix for the first problem:

```
loop:
if (find buffer in cache) {
     P(buffer);
     if (wrong buffer) {
          V(buffer);
          goto loop;
     }
     P(freesema);
     take buffer off freelist;
     return(buffer);
}
```

After finding the buffer, *p* locks it. Since in doing so *p* might have blocked, it then verifies that this buffer is still the one it wants. If not, *p* releases the buffer and re-evaluates.

Since in practice the process usually *doesn't* block in `P()`, verifying the match every time seems wasteful. Even worse, the analogous solution to the second problem (the case of a cache miss) requires that *p* make a second search for the buffer every time, since it might have blocked on the freelist semaphore.

The problem seems to be that *p* doesn't know when it blocks in `P()`; that's hidden from it. We can make blocking explicit with a new primitive, `CP()`, the conditional version of `P()` `CP()`(is available in UTS). `CP()` attempts to lock the semaphore but will not block, returning zero instead.

Here is a version using `CP()`:

```
loop:
if (find buffer in cache) {
     if (!CP(buffer)) {
          P(buffer);
          V(buffer);
          goto loop;
     }
     P(freesema);
     take buffer off freelist;
     return(buffer);
}
```

```
if (!CP(freesema)) {
    P(freesema);
    V(freesema);
    goto loop;
}
P(first buffer on freelist);
take first buffer off freelist;
reassign buffer;
return(buffer);
```

Suppose *p* finds the buffer. If the buffer is available, CP() succeeds, and since *p* hasn't blocked, verifying the match is unnecessary – the normal case is fast. If CP() fails, *p* sleeps until the buffer is free, and re-evaluates. Similarly, *p* knows it must re-evaluate if it blocks on the freelist.

This version does not work (with the usual "strict" semaphores). Suppose two processes, *p* and *q*, block while trying to access the same buffer. When the buffer is released, *p* runs, does a V() (which puts *q* on the run queue), loops back, and finds the same buffer. But the attempt to lock the buffer fails (the semaphore count is zero) because the buffer is owned by *q* (who is waiting on the run queue). So *p* blocks, *q* runs, and the cycle repeats forever.

The solution is to re-evaluate only if the buffer has been reassigned:

```
loop:
if (find buffer in cache) {
    if (!CP(buffer)) {
        P(buffer);
        if (wrong buffer) {
            V(buffer);
            goto loop;
        }
    }
    P(freesema);
    take buffer off freelist
    return(buffer);
}
```

In the cache miss case, the freelist semaphore can cause the same kind of livelock, but no analogous solution exists. Although keeping track of the number of items on the freelist seems such a natural use of semaphores, we know of no versions that actually do. Instead, a semaphore that never goes positive is used to block processes, in an event-wait style.

There is yet another problem with all the above semaphore-based versions (assuming strict semaphores). Just because a buffer is on the freelist, a `P()` operation on it might still block. The buffer may be owned by a process on the run queue, who is about to take the buffer off the freelist. So in the cache miss case, the process must *scan* the freelist until it finds a buffer that can be locked.

More generally, it is harder to enforce invariants (such as "a buffer is on the freelist if and only if it is not locked") because of the existence of an interval between when a resource (a buffer) becomes owned by someone and when that process actually runs and changes the state of the system accordingly (takes the buffer off the freelist).

Appendix C: Spin Model for Sleep/Wakeup with "Wanted" Flag

Using BITSTATE, this model needed 3 minutes of CPU time and 32MB on an IBM 3090 running UTS to get a coverage factor over 100 (see Holzmann [1990]). No errors were found.

```
/**
get resource:
    spinlock(&r->lk);
    while (r->lock) {
        r->wanted = 1;
        sleepl(r, &r->lk);
    }
    r->lock = 1;
    freelock(&r->lk);

use resource... free resource:
    r->lock = 0;
    waitlock(&r->lk);
    if (r->wanted) {
        r->wanted = 0;
        waitlock(&r->lk);
        wakeup(r);
    }
**/

/* number of cpus (processes) */
#define N   5
#define RUN 0
#define SLEEP   1
/* resource wanted flag and sleep lock */
byte r_wanted, r_lock;
/* resource spinlock, 0 or 1 */
byte r_lk;
/* sleep queue spinlock, 0 or 1 */
byte sq_lk;
```

```
/* process state, SLEEP or RUN */
byte state[N];

proctype cpu(int i)
{
    int j;
    /* get resource: */
    atomic {
        (r_lk == 0) -> r_lk = 1
    };
    do
    :: (r_lock == 1) ->
        r_wanted = 1;
        /* inside sleepl() */
        atomic {
            (sq_lk == 0) -> sq_lk = 1
        };
        r_lk = 0;
        state[i] = SLEEP;
        sq_lk = 0;
        /* wait to be awakened */
        (state[i] == RUN);
        atomic {
            (r_lk == 0) -> r_lk = 1
        }
    :: (r_lock ==  0) ->
        break
    od;
    assert(r_lock == 0);
    r_lock = 1;
    r_lk = 0;

    /* use resource ... free resource */
    assert(r_lock == 1);
    r_lock = 0;
    (r_lk == 0);
    if
    :: (r_wanted == 1) ->
        r_wanted = 0;
        (r_lk == 0);
```

```
                /* inside wakeup() */
                atomic {
                    (sq_lk == 0) -> sq_lk = 1
                };
                /* wakeup everyone */
                atomic {
                    do
                    :: (j < N) ->
                        state[j] = RUN;
                        j = j+1
                    :: (j >= N) -> break
                    od
                };
                sq_lk = 0
            :: (r_wanted == 0) ->
                skip
            fi
        }

        init {
            int i;
            atomic {
                do
                :: (i < N) ->
                    run cpu(i);
                    i = i+1
                :: (i >= N) -> break
                od
            }
        }
```

References

M. J. Bach, *The Design of the UNIX Operating System*, Englewood Cliffs, NJ: Prentice-Hall, 1986.

B. Beck, B. Kasten, and S. Thakkar, VLSI Assist for a Multiprocessor, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems,* pages 10-20, October 1987.

M. Ben-Ari, *Principles of Concurrent and Distributed Programming,* Englewood Cliffs, NJ: Prentice Hall, 1990.

E. W. Dijkstra, Cooperating Sequential Processes, in *Programming Languages*, ed. F. Genuys, pages 43-112, New York: Academic Press, 1968.

W. A. Felton, G. L. Miller, and J. M. Milner, A UNIX System Implementation for System/370, *AT&T Bell Laboratories Technical Journal,* 63(8), Part 2, pages 1751-1768, October 1984.

Gerard Holzmann, Algorithms for Automated Protocol Validation, *AT&T Technical Journal,* 69(1):32-59, January/February 1990.

B. W. Lampson, D. D. Redell, Experience With Processes and Monitors, *Communications of the ACM*, 23(2):105-117, Feb. 1980.

T. P. Lee, M. W. Luppi, and R. E. Menninger, Solving Performance Problems on a Multiprocessor UNIX System, *Proceedings of the USENIX Conference*, pages 399-405, Summer, 1987.

K. Thompson, UNIX Implementation, *The Bell System Technical Journal,* 57(6) Part 2, pages 1931-1946, July-August 1978.

[*submitted April 9, 1990; revised June 14, 1990; accepted June 20, 1990*]