7

Synchronization and Multiprocessors

7.1 Introduction

The desire for more processing power has led to several advances in hardware architectures. One of the major steps in this direction has been the development of multiprocessor systems. These systems consist of two or more processors sharing the main memory and other resources. Such configurations offer several advantages. They provide a flexible growth path for a project, which may start with a single processor and, as its computing needs grow, expand seamlessly by adding extra processors to the machine. Systems used for compute-intensive applications are often CPU-bound. The CPU is the main bottleneck, and other system resources such as the I/O bus and memory are underutilized. Multiprocessors add processing power without duplicating other resources, and hence provide a cost-effective solution for CPU-bound workloads.

Multiprocessors also provide an extra measure of reliability; if one of the processors should fail, the system could still continue to run without interruption. This, however, is a double-edged sword, since there are more potential points of failure. To ensure a high *mean time before failure* (MTBF), multiprocessor systems must be equipped with fault-tolerant hardware and software. In particular, the system should recover from the failure of one processor without crashing.

Several variants of UNIX have evolved to take advantage of such systems. One of the earliest multiprocessing UNIX implementations ran on the AT&T 3B20A and the IBM 370 architectures [Bach 84]. Currently, most major UNIX implementations are either native multiprocessing systems (DECUNIX, Solaris 2.*x*) or have multiprocessing variants (SVR4/MP, SCO/MPX).

Ideally, we would like to see the system performance scale linearly with the number of processors. Real systems fall short of this goal for several reasons. Since the other components of the system are not duplicated, they can become bottlenecks. The need to synchronize when accessing shared data structures, and the extra functionality to support multiple processors, adds CPU overhead and reduces the overall performance gains. The operating system must try to minimize this overhead and allow optimal CPU utilization.

The traditional UNIX kernel assumes a uniprocessor architecture and needs major modifications to run on multiprocessor systems. The three main areas of change are synchronization, parallelization, and scheduling policies. Synchronization involves the basic primitives used to control access to shared data and resources. The traditional primitives of sleep/wakeup combined with interrupt blocking are inadequate in a multiprocessing environment and must be replaced with more powerful facilities.

Parallelization concerns the efficient use of the synchronization primitives to control access to shared resources. This involves decisions regarding lock granularity, lock placement, deadlock avoidance, and so forth. Section 7.10 discusses some of these issues. The scheduling policy also needs to be changed to allow the optimal utilization of all processors. Section 7.4 analyzes some issues related to multiprocessor scheduling.

This chapter first describes the synchronization mechanisms in traditional UNIX systems and analyzes their limitations. It follows with an overview of multiprocessor architectures. Finally the chapter describes synchronization in modern UNIX systems. The methods describe work well on both uniprocessor and multiprocessor platforms.

In traditional UNIX systems, the process is the basic scheduling unit, and it has a single thread of control. As described in Chapter 3, many modern UNIX variants allow multiple threads of control in each process, with full kernel support for these threads. Such multithreaded systems are available both on uniprocessor and multiprocessor architectures. In these systems, individual threads contend for and lock the shared resources. In the rest of this chapter, we refer to a thread as the basic scheduling unit since it is the more general abstraction. For a single-threaded system, a thread is synonymous with a process.

7.2 Synchronization in Traditional UNIX Kernels

The UNIX kernel is reentrant—several processes may be executing in the kernel at the same time, perhaps even in the same routine. On a uniprocessor only one process can actually execute at a time. However, the system rapidly switches from one process to another, providing the illusion that they are all executing concurrently. This feature is usually called *multiprogramming*. Since these processes share the kernel, the kernel must synchronize access to its data structures in order to avoid corrupting them. Section 2.5 provides a detailed discussion of traditional UNIX synchronization techniques. In this section, we summarize the important principles.

The first safeguard is that the traditional UNIX kernel is nonpreemptive. Any thread executing in kernel mode will continue to run, even though its time quantum may expire, until it is ready to leave the kernel or needs to block for some resource. This allows kernel code to manipulate sev-

eral data structures without any locking, knowing that no other thread can access them until the current thread is done with them and is ready to relinquish the kernel in a consistent state.

7.2.1 Interrupt Masking

The no-preemption rule provides a powerful and wide-ranging synchronization tool, but it has certain limitations. Although the current thread may not be preempted, it may be interrupted. Interrupts are an integral part of system activity and usually need to be serviced urgently. The interrupt handler may manipulate the same data structures with which the current thread was working, resulting in corruption of that data. Hence the kernel must synchronize access to data that is used both by normal kernel code and by interrupt handlers.

UNIX solves this problem by providing a mechanism for blocking (masking) interrupts. Associated with each interrupt is an *interrupt priority level* (*ipl*). The system maintains a *current ipl* value and checks it whenever an interrupt occurs. If the interrupt has a higher priority than the *current ipl*, it is handled immediately (preempting the lower-priority interrupt currently being handled). Otherwise, the kernel blocks the interrupt until the *ipl* falls sufficiently. Prior to invoking the handler, the system raises the *ipl* to that of the interrupt; when the handler completes, the system restores the *ipl* to the previous value (which it saves). The kernel can also explicitly set the *ipl* to any value to mask interrupts during certain critical processing.

For example, a kernel routine may want to remove a disk block buffer from a buffer queue it is on; this queue may also be accessed by the disk interrupt handler. The code to manipulate the queue is a *critical region*. Before entering the critical region, the routine will raise the *ipl* high enough to block disk interrupts. After completing the queue manipulation, the routine will set the *ipl* back to its previous value, thus allowing the disk interrupts to be serviced. The *ipl* thus allows effective synchronization of resources shared by the kernel and interrupt handlers.

7.2.2 Sleep and Wakeup

Often a thread wants to guarantee exclusive use of a resource even if it needs to block for some reason. For instance, a thread wants to read a disk block into a block buffer. It allocates a buffer to hold the block and then initiates disk activity. This thread needs to wait for the I/O to complete, which means it must relinquish the processor to some other thread. If the other thread acquires the same buffer and uses it for some different purpose, the contents of the buffer may become indeterminate or corrupted. This means that threads need a way of locking the resource while they are blocked.

UNIX implements this by associating *locked* and *wanted* flags with shared resources. When a thread wants to access a sharable resource, such as a block buffer, it first checks its *locked* flag. If the flag is clear, the thread sets the flag and proceeds to use the resource. If a second thread tries to access the same resource, it finds the *locked* flag set and must block (go to sleep) until the resource becomes available. Before doing so, it sets the associated *wanted* flag. Going to sleep involves linking the thread onto a queue of sleeping threads, changing its state information to show that it is sleeping on this resource, and relinquishing the processor to another thread.

When the first thread is done with the resource, it will clear the *locked* flag and check the *wanted* flag. If the *wanted* flag is set, it means that at least one other thread is waiting for (blocked

on) this resource. In that case, the thread examines the *sleep queue* and wakes up all such threads. Waking a thread involves unlinking it from the sleep queue, changing its state to *runnable*, and putting it on the scheduler queue. When one of these threads is eventually scheduled, it again checks the *locked* flag, finds that it is clear, sets it, and proceeds to use the resource.

7.2.3 Limitations of Traditional Approach

The traditional synchronization model works correctly for a uniprocessor, but has some important performance problems that are described in this section. In a multiprocessor environment, the model breaks down completely, as will be shown in Section 7.4.

Mapping Resources to Sleep Queues

The organization of the sleep queues leads to poor performance in some situations. In UNIX, a thread blocks when waiting for a resource lock or an event. Each resource or event is associated with a *sleep channel*, which is a 32-bit value usually set to the address of the resource. There is a set of sleep queues, and a hash function maps the channel (hence, maps the resource) to one of these queues as shown in Figure 7-1. A thread goes to sleep by enqueuing itself onto the appropriate sleep queue and storing the sleep channel in its `proc` structure.

This approach has two consequences. First, more than one event may map to the same channel. For instance, one thread locks a buffer, initiates I/O activity to it, and sleeps until the I/O completes. Another thread tries to access the same buffer, finds it locked, and must block until it becomes available. Both events map to the same channel, namely the address of that buffer. When the I/O completes, the interrupt handler will wake up both threads, even though the event the second thread was waiting for has not yet occurred.

Second, the number of hash queues is much smaller than the number of different sleep channels (resources or events); hence, multiple channels map to the same hash queue. A queue thus contains threads waiting on several different channels. The `wakeup()` routine must examine each of them and only wake up threads blocked on the correct channel. As a result, the total time taken by `wakeup()` depends not on the number of processes sleeping on that channel, but on the total number

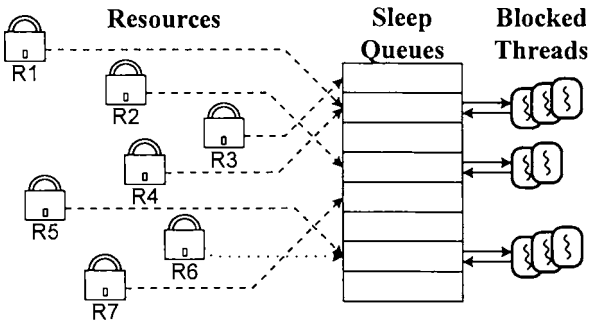

Figure 7-1. Mapping resources to global sleep queues.

of processes sleeping on that queue. This sort of unpredictable delay is usually undesirable and may be unacceptable for kernels that support real-time applications requiring bounded dispatch latency (see Section 5.5.4).

One alternative is to associate a separate sleep queue for each resource or event (Figure 7-2). This approach would optimize the latency of the wakeup algorithm, at the expense of memory overhead for all the extra queues. The typical queue header contains two pointers (forward and backward) as well as other information. The total number of synchronization objects in the system may be quite large and putting a sleep queue on each of them may be wasteful.

Solaris 2.*x* provides a more space-efficient solution [Eykh 92]. Each synchronization object has a two-byte field that locates a *turnstile* structure that contains the sleep queue and some other information (Figure 7-3). The kernel allocates turnstiles only to those resources that have threads blocked on them. To speed up allocation, the kernel maintains a pool of turnstiles, and the size of this pool is greater than the number of active threads. This approach provides more predictable real-time behavior with minimal storage overhead. Section 5.6.7 describes turnstiles in greater detail.

Shared and Exclusive Access

The sleep/wakeup mechanism is adequate when only one thread should use the resource at a time. It does not, however, readily allow for more complex protocols such as readers-writers synchronization. It may be desirable to allow multiple threads to share a resource for reading, but require exclusive access before modifying it. File and directory blocks, for example, can be shared efficiently using such a facility.

7.3 Multiprocessor Systems

There are three important characteristics of a multiprocessor system. The first is its memory model, which defines the way in which the processors share the memory. Second is the hardware support for synchronization. Finally, the software architecture determines the relationships between the processors, kernel subsystems, and user processes.

7.3.1 Memory Model

From a hardware perspective, multiprocessor systems can be divided into three categories (see Figure 7-4), depending on their coupling and memory access semantics:

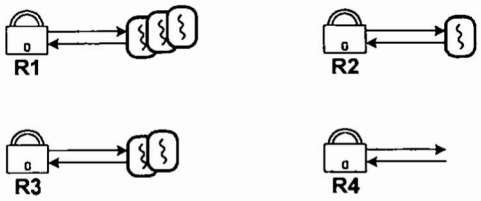

Figure 7-2. Per-resource blocked-thread queues.

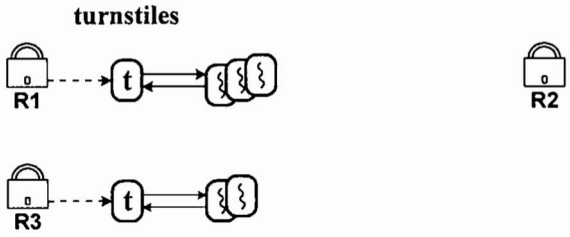

Figure 7-3. Queuing blocked threads on turnstiles.

- *Uniform Memory Access* (UMA)
- *Non-Uniform Memory Access* (NUMA)
- *No Remote Memory Access* (NORMA)

The most common system is the UMA, or shared memory, multiprocessor (Figure 7-4(a)). Such a system allows all CPUs equal access to main memory[1] and to I/O devices, usually by having

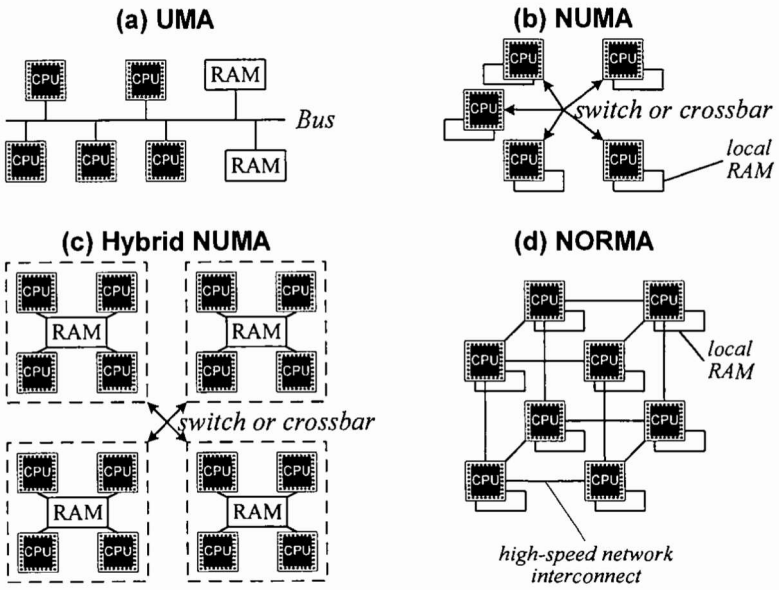

Figure 7-4. UMA, NUMA, and NORMA systems.

[1] However, the data, instruction, and address translation caches are local to each processor.

everything on a single system bus. This is a simple model from the operating system perspective. Its main drawback is scalability. UMA architectures can support only a small number of processors. As the number of processors increases, so does the contention on the bus. One of the largest UMA systems is the SGI Challenge, which supports up to 36 processors on a single bus.

In a NUMA system (Figure 7-4(b)), each CPU has some local memory, but can also access memory local to another processor. The remote access is slower, usually by an order of magnitude, than local access. There are also hybrid systems (Figure 7-4(c)), where a group of processors shares uniform access to its local memory, and has slower access to memory local to another group. The NUMA model is hard to program without exposing the details of the hardware architecture to the applications.

In a NORMA system (Figure 7-4(d)), each CPU has direct access only to its own local memory and may access remote memory only through explicit message passing. The hardware provides a high-speed interconnect that increases the bandwidth for remote memory access. Building a successful system for such an architecture requires cache management and scheduling support in the operating system, as well as compilers that can optimize the code for such hardware.

This chapter restricts itself to UMA systems.

7.3.2 Synchronization Support

Synchronization on a multiprocessor is fundamentally dependent on hardware support. Consider the basic operation of locking a resource for exclusive use by setting a *locked* flag maintained in a shared memory location. This may be accomplished by the following sequence of operations:

1. Read the flag.
2. If the flag is 0 (hence, the resource is unlocked), lock the resource by setting the flag to 1.
3. Return TRUE if the lock was obtained, or else return FALSE.

On a multiprocessor, two threads on two different processors may simultaneously attempt to carry out this sequence of operations. As Figure 7-5 shows, both threads may think they have exclusive access to the resource. To avoid such a disaster, the hardware has to provide a more powerful primitive that can combine the three subtasks into a single indivisible operation. Many architectures solve this problem by providing either an atomic test-and-set or a conditional store instruction.

Atomic Test-and-Set

An atomic test-and-set operation usually acts on a single bit in memory. It tests the bit, sets it to one, and returns its old value. Thus at the completion of the operation the value of the bit is one (locked), and the return value indicates whether it was already set to one prior to this operation. The operation is guaranteed to be atomic, so if two threads on two processors both issue the same instruction on the same bit, one operation will complete before the other starts. Further, the operation is also atomic with respect to interrupts, so that an interrupt can occur only after the operation completes.

Such a primitive is ideally suited for simple locks. If the test-and-set returns one, the calling thread owns the resource. If it returns zero, the resource is locked by another thread. Unlocking the resource is done by simply setting the bit to zero. Some examples of test-and-set instructions are

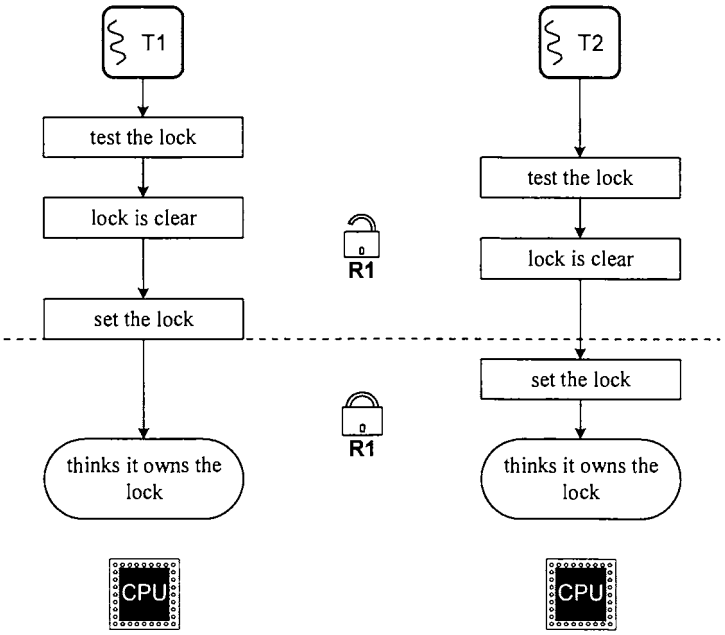

Figure 7-5. Race condition if test-and-set is not atomic.

BBSSI (**B**ranch on **B**it **S**et and **S**et **I**nterlocked) on the VAX-11 [Digi 87] and LDSTUB (**L**oa**D** and **ST**ore **U**nsigned **B**yte) on the SPARC.

Load-Linked and Store-Conditional Instructions

Some processors such as the MIPS R4000 and Digital's Alpha AXP use a pair of special load and store instructions to provide an atomic read-modify-write operation. The *load-linked* instruction (also called the *load-locked* instruction) loads a value from memory into a register and sets a flag that causes the hardware to monitor the location. If any processor writes to such a monitored location, the hardware will clear the flag. The *store-conditional* instruction stores a new value into the location provided the flag is still set. In addition, it sets the value of another register to indicate if the store occurred.

Such a primitive may be used to generate an atomic increment operation. The variable is read using load-linked, and its new value is set using store-conditional. This sequence is repeated until it succeeds. Event counters in DG/UX [Kell 89] are based on this facility.

Some systems such as the Motorola MC88100 use a third approach based on a *swap-atomic* instruction. This method is explored further in the exercises at the end of this chapter. Any of these hardware mechanisms becomes the first building block for a powerful and comprehensive synchronization facility. The high-level software abstractions described in the following sections are all built on top of the hardware primitives.

7.3.3 Software Architecture

From a software perspective, again there are three types of multiprocessing systems—master-slave, functionally asymmetric, and symmetric. A *master-slave system* [Gobl 81] is asymmetric: one processor plays the role of a *master processor*, and the rest are *slaves*. The master processor may be the only one allowed to do I/O and receive device interrupts. In some cases, only the master processor runs kernel code, and the slaves run only user-level code. Such constraints may simplify the system design, but reduce the advantage of multiple processors. Benchmark results [Bach 84] have shown that a UNIX system typically spends more than 40% of its time running in kernel mode, and it is desirable to spread this kernel activity among all processors.

Functionally asymmetric multiprocessors run different subsystems on different processors. For instance, one processor may run the networking layer, while another manages I/O. Such an approach is more suitable for a special-purpose system rather than a general-purpose operating system like UNIX. The Auspex NS5000 file server [Hitz 90] is a successful implementation of this model.

Symmetric multiprocessing (SMP) is by far the more popular approach. In an SMP system, all CPUs are equal, share a single copy of the kernel text and data, and compete for system resources such as devices and memory. Each CPU may run the kernel code, and any user process may be scheduled on any processor. This chapter describes SMP systems only, except where explicitly stated otherwise.

The rest of this chapter describes modern synchronization mechanisms, used for uniprocessor and multiprocessor systems.

7.4 Multiprocessor Synchronization Issues

One of the basic assumptions in the traditional synchronization model is that a thread retains exclusive use of the kernel (except for interrupts) until it is ready to leave the kernel or block on a resource. This is no longer valid on a multiprocessor, since each processor could be executing kernel code at the same time. *We now need to protect all kinds of data that did not need protection on a uniprocessor.* Consider for example, access to an *IPC resource table* (see Section 6.3.1). This data structure is not accessed by interrupt handlers and does not support any operations that might block the process. Hence on a uniprocessor, the kernel can manipulate the table without locking it. In the multiprocessor case, two threads on different processors can access the table simultaneously, and hence must lock it in some manner before use.

The locking primitives must be changed as well. In a traditional system, the kernel simply checks the *locked* flag and sets it to lock the object. On a multiprocessor, two threads on different processors can concurrently examine the *locked* flag for the same resource. Both will find it clear and assume that the resource is available. Both threads will then set the flag and proceed to access the resource, with unpredictable results. The system must therefore provide some kind of an atomic *test-and-set* operation to ensure that only one thread can lock the resource.

Another example involves blocking of interrupts. On a multiprocessor, a thread can typically block interrupts only on the processor on which it is running. It is usually not possible to block interrupts across all processors—in fact, some other processor may have already received a conflicting interrupt. The handler running on another processor may corrupt a data structure that the thread is

accessing. This is compounded by the fact that the handler cannot use the sleep/wakeup synchronization model, since most implementations do not permit interrupt handlers to block. The system should provide some mechanism for blocking interrupts on other processors. One possible solution is a global *ipl* managed in software.

7.4.1 The Lost Wakeup Problem

The sleep/wakeup mechanism does not function correctly on a multiprocessor. Figure 7-6 illustrates a potential race condition. Thread **T1** has locked a resource **R1**. Thread **T2**, running on another processor, tries to acquire the resource, and finds it locked. **T2** calls `sleep()` to wait for the resource. Between the time **T2** finds the resource locked and the time it calls `sleep()`, **T1** frees the resource and proceeds to wake up all threads blocked on it. Since **T2** has not yet been put on the sleep queue, it will miss the wakeup. The end result is that the resource is not locked, but **T2** is blocked waiting for it to be unlocked. If no one else tries to access the resource, **T2** could block indefinitely. This is known as the lost wakeup problem, and requires some mechanism to combine the test for the resource and the call to `sleep()` into a single atomic operation.

It is clear then, that we need a whole new set of primitives that will work correctly on a multiprocessor. This gives us a good opportunity to examine other problems with the traditional model and devise better solutions. Most of these issues are performance related.

7.4.2 The Thundering Herd Problem

When a thread releases a resource, it wakes up all threads waiting for it. One of them may now be able to lock the resource; the others will find the resource still locked and will have to go back to sleep. This may lead to extra overhead in wakeups and context switches.

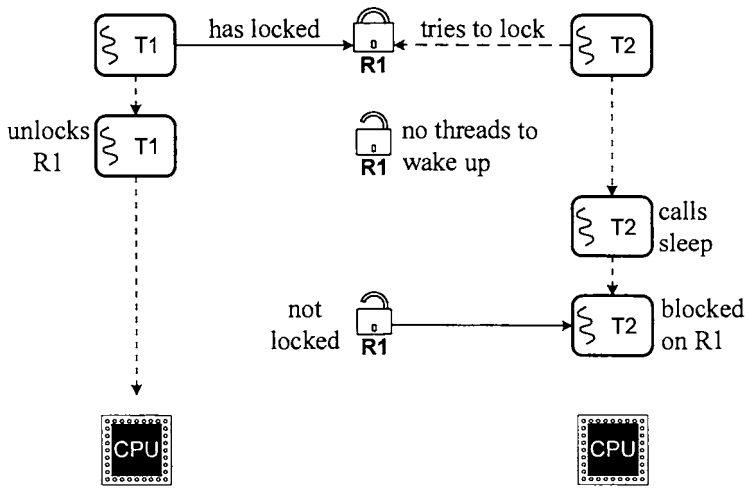

Figure 7-6. The lost wakeup problem.

This problem is not as acute on a uniprocessor, since by the time a thread runs, whoever had locked the resource is likely to have released it. On a multiprocessor, however, if several threads were blocked on a resource, waking them all may cause them to be simultaneously scheduled on different processors, and they would all fight for the same resource again. This is frequently referred to as the *thundering herd* problem.

Even if only one thread was blocked on the resource, there is still a time delay between its waking up and actually running. In this interval, an unrelated thread may grab the resource, causing the awakened thread to block again. If this happens frequently, it could lead to starvation of this thread.

We have examined several problems with the traditional synchronization model that affect correct operation and performance. The rest of this chapter describes several synchronization mechanisms that function well on both uniprocessors and multiprocessors.

7.5 Semaphores

The early implementations of UNIX on multiprocessors relied almost exclusively on *Dijkstra's semaphores* [Dijk 65] (also called *counted semaphores*) for synchronization. A semaphore is an integer-valued variable that supports two basic operations—`P()` and `V()`. `P()` decrements the semaphore and blocks if its new value is less than zero. `V()` increments the semaphore; if the resulting value is less than or equal to zero, it wakes up a thread blocked on it (if any). Example 7-1 describes these functions, plus an initialization function `initsem()` and a `CP()` function, which is a nonblocking version of `P()`:

```
void initsem (semaphore *sem, int val)
{
    *sem = val;
}

void P(semaphore *sem)  /* acquire the semaphore */
{
    *sem -= 1;
    while (*sem < 0)
        sleep;
}

void V(semaphore *sem)  /* release the semaphore */
{
    *sem += 1;
    if (*sem <= 0)
        wakeup a thread blocked on sem;
}
```

```
boolean_t CP(semaphore *sem)    /*  try to acquire semaphore without blocking */
{
    if (*sem > 0)   {
        *sem -= 1;
        return TRUE;
    } else
        return FALSE;
}
```

Example 7-1. Semaphore operations.

The kernel guarantees that the semaphore operations will be atomic, even on a multiprocessor system. Thus if two threads try to operate on the same semaphore, one operation will complete or block before the other starts. The `P()` and `V()` operations are comparable to sleep and wakeup, but with somewhat different semantics. The `CP()` operation allows a way to poll the semaphore without blocking and is used in interrupt handlers and other functions that cannot afford to block. It is also used in deadlock avoidance cases, where a `P()` operation risks a deadlock.

7.5.1 Semaphores to Provide Mutual Exclusion

Example 7-2 shows how a semaphore can provide mutual exclusion on a resource. A semaphore can be associated with a shared resource such as a linked list, and initialized to one. Each thread does a `P()` operation to lock a resource and a `V()` operations to release it. The first `P()` sets the value to zero, causing subsequent `P()` operations to block. When a `V()` is done, the value is incremented and one of the blocked threads is awakened.

```
/* During initialization */
semaphore sem;
initsem (&sem, 1);

/* On each use */
P (&sem);
Use resource;
V (&sem);
```

Example 7-2. Semaphore used to lock resource for exclusive use.

7.5.2 Event-Wait Using Semaphores

Example 7-3 shows how a semaphore can be used to wait for an event by initializing it to zero. Threads doing a `P()` will block. When the event occurs, a `V()` needs to be done for each blocked thread. This can be achieved by calling a single `V()` when the event occurs and having each thread do another `V()` upon waking up, as is shown in Example 7-3.

```
/* During initialization */
semaphore event;
initsem (&event, 0);      /* probably at boot time */

/* Code executed by thread that must wait on event */
P (&event);               /* Blocks if event has not occurred */
/* Event has occurred */
V (&event);               /* So that another thread may wake up */
/* Continue processing */

/* Code executed when event occurs */
V (&event);               /* Wake up one thread */
```

Example 7-3. Semaphores used to wait for an event.

7.5.3 Semaphores to Control Countable Resources

Semaphores are also useful for allocating countable resources, such as message block headers in a STREAMS implementation. As shown in Example 7-4, the semaphore is initialized to the number of available instances of that resource. Threads call `P()` while acquiring an instance of the resource and `V()` while releasing it. Thus the value of the semaphore indicates the number of instances currently available. If the value is negative, then its absolute value is the number of pending requests (blocked threads) for that resource. This is a natural solution to the classic producers-consumers problem.

```
/* During initialization */
semaphore counter;
initsem (&counter, resourceCount);

/* Code executed to use the resource */
P (&counter);     /* Blocks until resource is available */
Use resource;     /* Guaranteed to be available now */
V (&counter);     /* Release the resource */
```

Example 7-4. Semaphore used to count available instances of a resource.

7.5.4 Drawbacks of Semaphores

Although semaphores provide a single abstraction flexible enough to handle several different types of synchronization problems, they suffer from a number of drawbacks that make them unsuitable in several situations. To begin with, a semaphore is a high-level abstraction based on lower-level primitives that provide atomicity and a blocking mechanism. For the `P()` and `V()` operations to be atomic on a multiprocessor system, there must be a lower-level atomic operation to guarantee exclusive access to the semaphore variable itself. Blocking and unblocking require context switches

and manipulation of sleep and scheduler queues, all of which make the operations slow. This expense may be tolerable for some resources that need to be held for a long time, but is unacceptable for locks held for a short time.

The semaphore abstraction also hides information about whether the thread actually had to block in the `P()` operation. This is often unimportant, but in some cases it may be crucial. The UNIX buffer cache, for instance, uses a function called `getblk()` to look for a particular disk block in the buffer cache. If the desired block is found in the cache, `getblk()` attempts to lock it by calling `P()`. If `P()` were to sleep because the buffer was locked, there is no guarantee that, when awakened, the buffer would contain the same block that it originally had. The thread that had locked the buffer may have reassigned it to some other block. Thus after `P()` returns, the thread may have locked the wrong buffer. This problem can be solved within the framework of semaphores, but the solution is cumbersome and inefficient, and indicates that other abstractions might be more suitable [Ruan 90].

7.5.5 Convoys

Compared to the traditional sleep/wakeup mechanism, semaphores offer the advantage that processes do not wake up unnecessarily. When a thread wakes up within a `P()`, it is guaranteed to have the resource. The semantics ensure that the ownership of the semaphore is transferred to the woken up thread before that thread actually runs. If another thread tries to acquire the semaphore in the meantime, it will not be able to do so. This very fact, however, leads to a performance problem called semaphore *convoys* [Lee 87]. A convoy is created when there is frequent contention on a semaphore. Although this can degrade the performance of any locking mechanism, the peculiar semantics of semaphores compound the problem.

Figure 7-7 shows the formation of a convoy. **R1** is a critical region protected by a semaphore. At instant **(a)**, thread **T2** holds the semaphore, while **T3** is waiting to acquire it. **T1** is run-

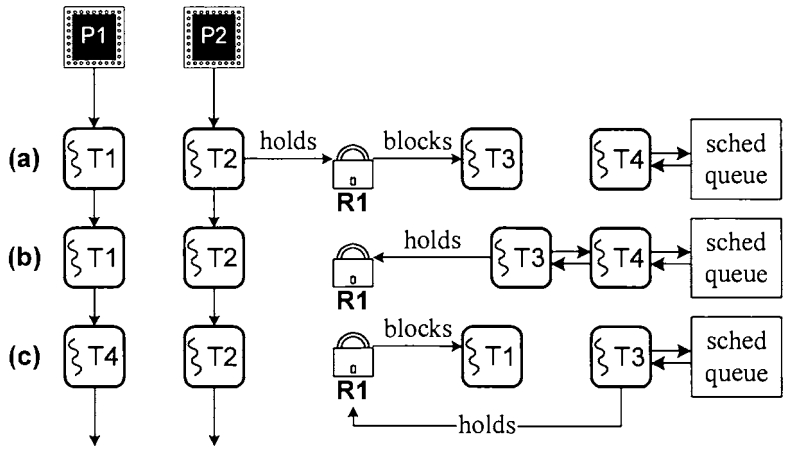

Figure 7-7. Convoy formation.

ning on another processor, and **T4** is waiting to be scheduled. Now suppose **T2** exits the critical region and releases the semaphore. It wakes up **T3** and puts it on the scheduler queue. **T3** now holds the semaphore, as shown in **(b)**.

Now suppose **T1** needs to enter the critical region. Since the semaphore is held by **T3**, **T1** will block, freeing up processor **P1**. The system will schedule thread **T4** to run on **P1**. Hence in **(c)**, **T3** holds the semaphore and **T1** is blocked on it; neither thread can run until **T2** or **T4** yields its processor.

The problem lies in step **(c)**. Although the semaphore has been assigned to **T3**, **T3** is not running and hence is not in the critical region. As a result, **T1** must block on the semaphore even though no thread is in the critical region. The semaphore semantics force allocation in a first-come, first-served order.[2] This forces a number of unnecessary context switches. Suppose the semaphore was replaced by an *exclusive lock*, or *mutex*. Then, in step **(b)**, **T2** would release the lock and wake up **T3**, but **T3** would not own the lock at this point. Consequently, in step **(c)**, **T1** would acquire the lock, eliminating the context switch.

> Note: *Mutex, short for mutual exclusion lock, is a general term that refers to any primitive that enforces exclusive access semantics.*

In general, it is desirable to have a set of inexpensive lower-level primitives instead of a single monolithic higher-level abstraction. This is the trend in the modern multiprocessing kernels, and the following sections examine the low-level mechanisms that together provide a versatile synchronization facility.

7.6 Spin Locks

The simplest locking primitive is a *spin lock*, also called a *simple lock* or a *simple mutex*. If a resource is protected by a spin lock,a thread trying to acquire the resource will *busy-wait* (loop, or spin) until the resource is unlocked. It is usually a scalar variable that is zero if available and one if locked. The variable is manipulated using a busy-wait loop around an atomic test-and-set or similar instruction available on the machine. Example 7-5 shows an implementation of a spin lock. It assumes that `test_and_set()` returns the old value of the object.

```
void spin_lock (spinlock_t *s)  {
    while (test_and_set (s) != 0)    /* already locked */
        ;                            /* loop until successful */
}

void spin_unlock (spinlock_t *s)     { *s = 0; }
```

Example 7-5. Spin lock implementation.

---

[2] Some implementation may choose the thread to wake up based on priority. The effect in this example would be the same.

Even this simple algorithm is flawed. On many processors, `test_and_set()` works by locking the memory bus, so this loop could monopolize the bus and severely degrade system performance. A better approach is to use two loops—if the test fails, the inner loop simply waits for the variable to become zero. The simple test in the inner loop does not require locking the bus. Example 7-6 shows the improved implementation:

```
void spin_lock (spinlock_t *s)
{
    while (test_and_set (s) != 0)    /* already locked */
        while (*s != 0)
            ;                        /* wait until unlocked */
}

void spin_unlock (spinlock_t *s)     { *s = 0; }
```

Example 7-6. Revised spin lock implementation.

7.6.1 Use of Spin Locks

The most important characteristic of spin locks is that a thread ties up a CPU while waiting for the lock to be released. It is essential, then, to hold spin locks only for extremely short durations. In particular, they must not be held across blocking operations. It may also be desirable to block interrupts on the current processor prior to acquiring a spin lock, so as to guarantee low holding time on the lock.

The basic premise of a spin lock is that a thread busy-waits on a resource on one processor while another thread is using the resource on a different processor. This is only possible on a multiprocessor. On a uniprocessor, if a thread tries to acquire a spin lock that is already held, it will loop forever. Multiprocessor algorithms, however, must operate correctly regardless of the number of processors, which means that they should handle the uniprocessor case as well. This requires strict adherence to the rule that threads not relinquish control of the CPU while holding a spin lock. On a uniprocessor, this ensures that a thread will never have to busy-wait on a spin lock.

The major advantage of spin locks is that they are inexpensive. When there is no contention on the lock, both the *lock* and the *unlock* operations typically require only a single instruction each. They are ideal for locking data structures that need to be accessed briefly, such as while removing an item from a doubly linked list or while performing a *load-modify-store* type of operation on a variable. Hence they are used to protect those data structures that do not need protection in a uniprocessor system. They are also used extensively to protect more complex locks, as shown in the following sections. Semaphores, for instance, use a spin lock to guarantee atomicity of their operations, as shown in Example 7-7.

```
spinlock_t list;
spin_lock (&list);
item->forw->back = item->back;
item->back->forw = item->forw;
spin_unlock (&list);
```

Example 7-7. Using a spin lock to access a doubly linked list.

7.7 Condition Variables

A *condition variable* is a more complex mechanism associated with a *predicate* (a logical expression that evaluates to TRUE or FALSE) based on some shared data. It allows threads to block on it and provides facilities to wakeup one or all blocked threads when the result of the predicate changes. It is more useful for waiting on events than for resource locking.

Consider, for example, one or more server threads waiting for client requests. Incoming requests are to be passed to waiting threads or put on a queue if no one is ready to service them. When a server thread is ready to process the next request, it first checks the queue. If there is a pending message, the thread removes it from the queue and services it. If the queue is empty, the thread blocks until a request arrives. This can be implemented by associating a condition variable with this queue. The shared data is the message queue itself, and the predicate is that the queue be nonempty.

The condition variable is similar to a sleep channel in that server threads block on the condition and incoming messages awaken them. On a multiprocessor, however, we need to guard against some race conditions, such as the lost wakeup problem. Suppose a message arrives after a thread checks the queue but before the thread blocks. The thread will block even though a message is available. We therefore need an atomic operation to test the predicate and block the thread if necessary.

Condition variables provide this atomicity by using an additional mutex, usually a spin lock. The mutex protects the shared data, and avoids the lost wakeup problem. The server thread acquires the mutex on the message queue, then checks if the queue is empty. If so, it calls the `wait()` function of the condition with the spin lock held. The `wait()` function takes the mutex as an argument and atomically blocks the thread and releases the mutex. When the message arrives on the queue and the thread is woken up, the `wait()` call reacquires the spin lock before returning. Example 7-8 provides a sample implementation of condition variables:

```
struct condition  {
    proc *next;          /* doubly linked list */
    proc *prev;          /* of blocked threads */
    spinlock_t listLock; /* protects this list */
};
```

```
void wait (condition *c, spinlock_t *s)
{
    spin_lock (&c->listLock);
    add self to the linked list;
    spin_unlock (&c->listLock);
    spin_unlock (s);        /* release spinlock before blocking */
    swtch();                /* perform context switch */
    /* When we return from swtch, the event has occurred */
    spin_lock (s);          /* acquire the spin lock again */
    return;
}

void do_signal (condition *c)
/* Wake up one thread waiting on this condition */
{
    spin_lock (&c->listLock);
    remove one thread from linked list, if it is nonempty;
    spin_unlock (&c->listLock);
    if a thread was removed from the list, make it runnable;
    return;
}

void do_broadcast (condition *c)
/* Wake up all threads waiting on this condition */
{
    spin_lock (&c->listLock);
    while (linked list is nonempty)  {
        remove a thread from linked list;
        make it runnable;
    }
    spin_unlock (&c->listLock);
}
```

Example 7-8. Implementation of condition variables.

7.7.1 Implementation Issues

There are a few important points to note. The predicate itself is not part of the condition variable. It must be tested by the calling routine before calling `wait()`. Further, note that the implementation uses two separate mutexes. One is `listLock`, which protects the doubly linked list of threads blocked on the condition. The second mutex protects the tested data itself. It is not a part of the condition variable, but is passed as an argument to the `wait()` function. The `swtch()` function and the code to make blocked threads runnable may use a third mutex to protect the scheduler queues.

We thus have a situation where a thread tries to acquire one spin lock while holding another. This is not disastrous since the restriction on spin locks is only that threads are not allowed to block while holding one. Deadlocks are avoided by maintaining a strict locking order—the lock on the predicate must be acquired before `listLock`.

It is not necessary for the queue of blocked threads to be a part of the condition structure itself. Instead, we may have a global set of sleep queues as in traditional UNIX. In that case, the `listLock` in the condition is replaced by a mutex protecting the appropriate sleep queue. Both methods have their own advantages, as discussed earlier.

One of the major advantages of a condition variable is that it provides two ways to handle event completion. When an event occurs, there is the option of waking up just one thread with `do_signal()` or all threads with `do_broadcast()`. Each may be appropriate in different circumstances. In the case of the server application, waking one thread is sufficient, since each request is handled by a single thread. However, consider several threads running the same program, thus sharing a single copy of the program text. More than one of these threads may try to access the same nonresident page of the text, resulting in page faults in each of them. The first thread to fault initiates a disk access for that page. The other threads notice that the read has already been issued and block waiting for the I/O to complete. When the page is read into memory, it is desirable to call `do_broadcast()` and wake up all the blocked threads, since they can all access the page without conflict.

7.7.2 Events

Frequently, the predicate of the condition is simple. Threads need to wait for a particular task to complete. The completion may be flagged by setting a global variable. This situation may be better expressed by a higher-level abstraction called an *event* that combines a *done* flag, the spin lock protecting it, and the condition variable into a single object. The event object presents a simple interface, allowing two basic operations—`awaitDone()` and `setDone()`. `awaitDone()` blocks until the event occurs, while `setDone()` marks the `event` as having occurred and wakes up all threads blocked on it. In addition, the interface may support a nonblocking `testDone()` function and a `reset()` function, which once again marks the `event` as not done. In some cases, the boolean *done* flag may be replaced by a variable that returns more descriptive completion information when the event occurs.

7.7.3 Blocking Locks

Often, a resource must be locked for a long period of time and the thread holding this lock must be permitted to block on other events. Thus a thread that needs the resource cannot afford to spin until the resource becomes available, and must block instead. This requires a *blocking lock* primitive that offers two basic operations—`lock()` and `unlock()`—and optionally, a `tryLock()`. Again there are two objects to synchronize—the *locked* flag on the resource and the sleep queue—which means that we need a spin lock to guarantee atomicity of the operations. Such locks may be trivially implemented using condition variables, with the predicate being the clearing of the locked flag. For per-

formance reasons, blocking locks might be provided as fundamental primitives. In particular, if each resource has its own sleep queue, a single spin lock might protect both the flag and the queue.

7.8 Read-Write Locks

Although modification of a resource requires exclusive access, it is usually acceptable to allow several threads to simultaneously read the shared data, as long as no one is trying to write to it at that time. This requires a complex lock that permits both shared and exclusive modes of access. Such a facility may be built on top of simple locks and conditions [Birr 89]. Before we look at an implementation, let us examine the desired semantics. A read-write lock may permit either a single writer or multiple readers. The basic operations are `lockShared()`, `lockExclusive()`, `unlockShared()` and `unlockExclusive()`. In addition, there might be `tryLockShared()` and `tryLockExclusive()`, which return `FALSE` instead of blocking, and also `upgrade()` and `downgrade()`, which convert a shared lock to exclusive and vice versa. A `lockShared()` operation must block if there is an exclusive lock present, whereas `lockExclusive()` must block if there is either an exclusive or shared lock on the resource.

7.8.1 Design Considerations

What should a thread do when releasing a lock? The traditional UNIX solution is to wake up all threads waiting for the resource. This is clearly inefficient—if a writer acquires the lock next, other readers and writers will have to go back to sleep; if a reader acquires the lock, other writers will have to go back to sleep. It is preferable to find a protocol that avoids needless wakeups.

If a reader releases a resource, it takes no action if other readers are still active. When the last active reader releases its shared lock, it must wake up a single waiting writer.

When a writer releases its lock, it must choose whether to wake up another writer or the other readers (assuming both readers and writers are waiting). If writers are given preference, the readers could starve indefinitely under heavy contention. The preferred solution is to wake up all waiting readers when releasing an exclusive lock. If there are no waiting readers, we wake up a single waiting writer.

This scheme can lead to writer starvation. If there is a constant stream of readers, they will keep the resource read-locked, and the writer will never acquire the lock. To avoid this situation, a `lockShared()` request must block if there is any waiting writer, even though the resource is currently only read-locked. Such a solution, under heavy contention, will alternate access between individual writers and batches of readers.

The `upgrade()` function must be careful to avoid deadlocks. A deadlock can occur unless the implementation takes care to give preference to upgrade requests over waiting writers. If two threads try to upgrade a lock, each would block since the other holds a shared lock. One way to avoid that is for `upgrade()` to release the shared lock before blocking if it cannot get the exclusive lock immediately. This results in additional problems for the user, since another thread could have modified the object before `upgrade()` returns. Another solution is for `upgrade()` to fail and release the shared lock if there is another pending upgrade.

7.8.2 Implementation

Example 7-9 implements a read-write lock facility:

```
struct rwlock   {
    int nActive;     /* num of active readers,  or -1 if a writer is active */
    int nPendingReads;
    int nPendingWrites;
    spinlock_t sl;
    condition canRead;
    condition canWrite;
};

void lockShared (struct rwlock *r)
{
    spin_lock (&r->sl);
    r->nPendingReads++;
    if (r->nPendingWrites > 0)
        wait (&r->canRead, &r->sl); /* don't starve writers */
    while (r->nActive < 0)          /* someone has exclusive lock */
        wait (&r->canRead, &r->sl);
    r->nActive++;
    r->nPendingReads--;
    spin_unlock (&r->sl);
}

void unlockShared (struct rwlock *r)
{
    spin_lock (&r->sl);
    r->nActive--;
    if (r->nActive == 0)    {       /* no other readers */
        spin_unlock (&r->sl);
        do_signal (&r->canWrite);
    } else
        spin_unlock (&r->sl);
}

void lockExclusive (struct rwlock *r)
{
    spin_lock (&r->sl);
    r->nPendingWrites++;
    while (r->nActive)
        wait (&r->canWrite, &r->sl);
    r->nPendingWrites--;
```

```
    r->nActive = -1;
    spin_unlock (&r->sl);
}

void unlockExclusive (struct rwlock *r)
{
    boolean_t wakeReaders;
    spin_lock (&r->sl);
    r->nActive = 0;
    wakeReaders = (r->nPendingReads != 0);
    spin_unlock (&r->sl);
    if (wakeReaders)
        do_broadcast (&r->canRead); /* wake all readers */
    else
        do_signal (&r->canWrite);   /* wake a single writer */
}

void downgrade (struct rwlock *r)
{
    boolean_t wakeReaders;
    spin_lock (&r->sl);
    r->nActive = 1;
    wakeReaders = (r->nPendingReads != 0);
    spin_unlock (&r->sl);
    if (wakeReaders)
        do_broadcast (&r->canRead); /* wake all readers */
}

void upgrade (struct rwlock *r)
{
    spin_lock (&r->sl);
    if (r->nActive == 1) {          /* no other reader */
        r->nActive = -1;
    } else {
        r->nPendingWrites++;
        r->nActive--;               /* release shared lock */
        while (r->nActive)
            wait (&r->canWrite, &r->sl);
        r->nPendingWrites--;
        r->nActive = -1;
    }
    spin_unlock (&r->sl);
}
```

Example 7-9. Implementation of read-write locks.

7.9 Reference Counts

Although a lock may protect the data inside an object, we frequently need another mechanism to protect the object itself. Many kernel objects are dynamically allocated and deallocated. If a thread deallocates such an object, other threads have no way of knowing it and may try to access the object using a direct pointer (which they earlier acquired) to it. In the meantime, the kernel could have reallocated the memory to a different object, leading to severe system corruption.

If a thread has a pointer to an object, it expects the pointer to be valid until the thread relinquishes it. The kernel can guarantee it by associating a reference count with each such object. The kernel sets the count to one when it first allocates the object (thus generating the first pointer). It increments the count each time it generates a new pointer to the object.

This way, when a thread gets a pointer to an object, it really acquires a reference to it. It is the thread's responsibility to release the reference when no longer needed, at which time the kernel decrements the object's reference count. When the count reaches zero, no thread has a valid reference to the object, and the kernel may deallocate the object.

For example, the file system maintains reference counts for *vnodes*, which hold information about active files (see Section 8.7). When a user opens a file, the kernel returns a file descriptor, which constitutes a reference to the vnode. The user passes the descriptor to subsequent *read* and *write* system calls, allowing the kernel to access the file quickly without repeated name translations. When the user closes the file, the reference is released. If several users have the same file open, they reference the same vnode. When the last user closes the file, the kernel can deallocate the vnode.

The previous example shows that reference counts are useful in a uniprocessor system as well. They are even more essential for multiprocessors, since without proper reference counting, a thread may deallocate an object while a thread on another processor is actively accessing it.

7.10 Other Considerations

There are several other factors to consider in the design of a complex locking facility and the way in which the locks are used. This section examines some important issues.

7.10.1 Deadlock Avoidance

It is often necessary for a thread to hold locks on multiple resources. For instance, the implementation of condition variables described in Section 7.7 uses two mutexes: one protects the data and predicate of the condition, while the other protects the linked list of threads blocked on the condition. Trying to acquire multiple locks lead to a deadlock, as illustrated in Figure 7-8. Thread **T1** holds resource **R1** and tries to acquire resource **R2**. At the same time, thread **T2** may be holding **R2** and trying to acquire **R1**. Neither thread can make progress.

The two common deadlock avoidance techniques are *hierarchical locking* and *stochastic locking*. Hierarchical locking imposes an order on related locks and requires that all threads take locks in the same order. In the case of condition variables, for instance, a thread must lock the condition's predicate before locking the linked list. As long as the ordering is strictly followed, deadlock cannot occur.

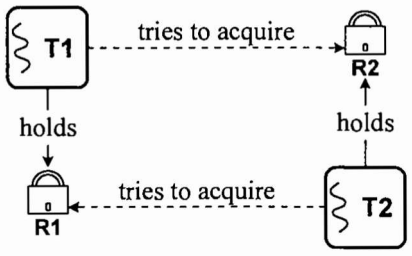

Figure 7-8. Possible deadlock when using spin locks.

There are situations in which the ordering must be violated. Consider a buffer cache implementation that maintains disk block buffers on a doubly linked list, sorted in *least recently used (LRU)* order. All buffers that are not actively in use are on the LRU list. A single spin lock protects both the queue header and the forward and backward pointers in the buffers on that queue. Each buffer also has a spin lock to protect the other information in the buffer. This lock must be held while the buffer is actively in use.

When a thread wants a particular disk block, it locates the buffer (using hash queues or other pointers not relevant to this discussion) and locks it. It then locks the LRU list in order to remove the buffer from it. Thus the normal locking order is *"first the buffer, then the list."*

Sometimes a thread simply wants any free buffer and tries to get it from the head of the LRU list. It first locks the list, then locks the first buffer on the list and removes it from the list. This, however, reverses the locking order, since it locks the list before the buffer.

It is easy to see how a deadlock can occur. One thread locks the buffer at the head of the list and tries to lock the list. At the same time, another thread that has locked the list tries to lock the buffer at the head. Each will block waiting for the other to release the lock.

The kernel uses stochastic locking to handle this situation. When a thread attempts to acquire a lock that would violate the hierarchy, it uses a `try_lock()` operation instead of `lock()`. This function attempts to acquire the lock, but returns failure instead of blocking if the lock is already held. In this example, the thread that wants to get any free buffer will lock the list and then go down the list, using `try_lock()` until it finds a buffer it can lock. Example 7-10 describes an implementation of `try_lock()` for spin locks:

```
int try_lock (spinlock_t *s)
{
    if (test_and_set (s) != 0)  /* already locked */
        return FAILURE;
    else
        return SUCCESS;
}
```

Example 7-10. Implementation of `try_lock()`.

7.10.2 Recursive Locks

A lock is recursive if an attempt by a thread to acquire a lock it already owns would succeed without blocking. Why is this a desirable feature? Why should a thread attempt to lock something it has already locked? The typical scenario has a thread locking a resource, then calling a lower-level routine to perform some operation on it. This lower-level routine may also be used by other higher-level routines that do not lock the resource prior to calling it. Thus the lower-level routine does not know if the resource is already locked. If it tries to lock the resource, a single process deadlock may occur.

Such a situation can, of course, be avoided by explicitly informing the lower-level routine about the lock via an extra argument. This, however, breaks many existing interfaces and is awkward, since sometimes the lower routine may be several function calls down. The resulting interfaces would be extremely nonmodular. An alternative is to allow the locks to be recursive. This adds some overhead, since the lock must now store some sort of owner ID and check it any time it would normally block or deny a request. More important, it allows functions to deal only with their own locking requirements, without worrying about which locks its callers hold, resulting in clean, modular interfaces.

One example when such a lock is used is directory writes in the BSD file system (ufs). The routine `ufs_write()` handles writes to both files and directories. Requests for file writes usually access the file through the file table entry, which directly gives a pointer to the file's *vnode*. Thus the vnode is passed on directly to `ufs_write()`, which is responsible for locking it. For a directory write, however, the directory vnode is acquired by the pathname traversal routine, which returns the vnode in a locked state. When `ufs_write()` is called for this node, it will deadlock if the lock is not recursive.

7.10.3 To Block or to Spin

Most complex locks can be implemented as blocking locks or as complex spin locks, without impacting their functionality or interface. Consider an object that is protected by a complex lock (such as a semaphore or a read-write lock). In most of the implementations described in this chapter, if a thread tries to acquire the object and finds it locked, the thread blocks until the object is released. The thread could just as easily busy-wait and still preserve the semantics of the lock.

The choice between blocking and busy-waiting is often dictated by performance considerations. Since busy-waiting ties up a processor, it is generally frowned upon. However, certain situations mandate busy-waiting. If the thread already holds a simple mutex, it is not allowed to block. If the thread tries to acquire another simple mutex, it will busy-wait; if it tries to acquire a complex lock, it will release the mutex it already holds (such as with conditions).

Sleep and wakeup, however, are expensive operations themselves, involving one context switch at each end and manipulating sleep and scheduler queues. Just as it is preposterous to do a busy-wait on a lock for an extended period of time, it is inefficient to sleep on a resource that is likely to be available soon.

Moreover, some resources may be subject to short-term or long-term locking, depending on the situation. For instance, the kernel may keep a partial list of free disk blocks in memory. When this list becomes empty, it must be replenished from disk. In most instances, the list needs to be

locked briefly while adding or removing entries to it in memory. When disk I/O is required, the list must be locked for a long time. Thus neither a spin lock nor a blocking lock is by itself a good solution. One alternative is to provide two locks, with the blocking lock being used only when the list is being replenished. It is preferable, however, to have a more flexible locking primitive.

These issues can be effectively addressed by storing a hint in the lock that suggests whether contending threads should spin or block. The hint is set by the owner of the lock and examined whenever an attempt to acquire the lock does not immediately succeed. The hint may be either advisory or mandatory.

An alternative solution is provided by the *adaptive locks* of Solaris 2.*x* [Eykh 92]. When a thread **T1** tries to acquire an adaptive lock held by another thread **T2**, it checks to see if **T2** is currently active on any processor. As long as **T2** is active, **T1** executes a busy-wait. If **T2** is blocked, **T1** blocks as well.

7.10.4 What to Lock

A lock can protect several things—data, predicates, invariants, or operations. Reader-writer locks, for example, protect data. Condition variables are associated with predicates. An invariant is similar to a predicate but has slightly different semantics. When a lock protects an invariant, it means that the invariant is TRUE except when the lock is held. For instance, a linked list might use a single lock while adding or removing elements. The invariant protected by this lock is that the list is in a consistent state.

Finally, a lock can control access to an operation or function. This restricts the execution of that code to at most one processor at a time, even if different data structures are involved. The *monitors* model of synchronization [Hoar 74] is based on this approach. Many UNIX variants use a master processor for unparallelized (not multiprocessor-safe) portions of the kernel, thus serializing access to such code. This approach usually leads to severe bottlenecks and should be avoided if possible.

7.10.5 Granularity and Duration

System performance depends greatly on the locking granularity. At one extreme, some asymmetric multiprocessing systems run all kernel code on the master processor, thus having a single lock for the whole kernel.[3] At the other extreme, a system could use extremely fine-grained locking with a separate lock for each data variable. Clearly, that is not the ideal solution either. The locks would consume a large amount of memory, performance would suffer due to the overhead of constantly acquiring and releasing locks, and the chances of deadlock would increase because it is difficult to enforce a locking order for such a large number of objects.

The ideal solution, as usual, lies somewhere in between, and there is no consensus on what it is. Proponents of coarse-granularity locking [Sink 88] suggest starting with a small number of locks to protect major subsystems and adding finer granularity locks only where the system exhibits a

[3] Sometimes, even SMP systems use a master processor to run code that is not multiprocessor-safe. This is known as *funneling*.

bottleneck. Systems such as Mach, however, use a fine-grained locking structure and associate locks with individual data objects.

Locking duration, too, must be carefully examined. It is best to hold the lock for as short a time as possible, so as to minimize contention on it. Sometimes, however, this may result in extra locking and unlocking. Suppose a thread needs to perform two operations on an object, both requiring a lock on it. In between the two operations, the thread needs to do some unrelated work. It could unlock the object after the first operation and lock it again for the second one. It might be better, instead, to keep the object locked the whole time, provided that the unrelated work is fairly short. Such decisions must be made on a case-by-case basis.

7.11 Case Studies

The primitives described in Sections 7.5–7.8 constitute a kind of grab bag from which an operating system can mix and match to provide a comprehensive synchronization interface. This section examines the synchronization facilities in the major multiprocessing variants of UNIX.

7.11.1 SVR4.2/MP

SVR4.2/MP is the multiprocessor version of SVR4.2. It provides four types of locks—basic locks, sleep locks, read-write locks, and synchronization variables [UNIX 92]. Each lock must be explicitly allocated and deallocated through *xxx*_ALLOC and *xxx*_DEALLOC operations, where *xxx*_ is the type-specific prefix. The allocation operation takes arguments that are used for debugging.

Basic Locks

The basic lock is a nonrecursive mutex lock that allows short-term locking of resources. It may not be held across a blocking operation. It is implemented as a variable of type `lock_t`. It is locked and unlocked by the following operations:

```
pl_t LOCK (lock_t *lockp, pl_t new_ipl);
UNLOCK (lock_t *lockp, pl_t old_ipl);
```

The `LOCK` call raises the interrupt priority level to `new_ipl` before acquiring the lock and returns the previous priority level. This value must be passed to the `UNLOCK` operation, so that it may restore the *ipl* to the old level.

Read-Write Locks

A *read-write lock* is a nonrecursive lock that allows short-term locking with single-writer, multiple-reader semantics. It may not be held across a blocking operation. It is implemented as a variable of type `rwlock_t` and provides the following operations:

```
pl_t RW_RDLOCK (rwlock_t *lockp, pl_t new_ipl);
pl_t RW_WRLOCK (rwlock_t *lockp, pl_t new_ipl);
void RW_UNLOCK (rwlock_t *lockp, pl_t old_ipl);
```

The treatment of interrupt priorities is identical to that for basic locks. The locking operations raise the *ipl* to the specified level and return the previous *ipl*. RW_UNLOCK restores the *ipl* to the old level. The lock also provides nonblocking operations RW_TRYRDLOCK and RW_TRYWRLOCK.

Sleep Locks

A *sleep lock* is a nonrecursive mutex lock that permits long-term locking of resources. It may be held across a blocking operation. It is implemented as a variable of type sleep_t, and provides the following operations:

```
void SLEEP_LOCK (sleep_t *lockp, int pri);
bool_t SLEEP_LOCK_SIG (sleep_t *lockp, int pri);
void SLEEP_UNLOCK (sleep_t *lockp);
```

The pri parameter specifies the scheduling priority to assign to the process after it awakens. If a process blocks on a call to SLEEP_LOCK, it will not be interrupted by a signal. If it blocks on a call to SLEEP_LOCK_SIG, a signal will interrupt the process; the call returns TRUE if the lock is acquired, and FALSE if the sleep was interrupted. The lock also provides other operations, such as SLEEP_LOCK_AVAIL (checks if lock is available), SLEEP_LOCKOWNED (checks if caller owns the lock), and SLEEP_TRYLOCK (returns failure instead of blocking if lock cannot be acquired).

Synchronization Variables

A *synchronization variable* is identical to the *condition variables* discussed in Section 7.7. It is implemented as a variable of type sv_t, and its predicate, which is managed separately by users of the lock, must be protected by a basic lock. It supports the following operations:

```
void SV_WAIT (sv_t *svp, int pri, lock_t *lockp);
bool_t SV_WAIT_SIG (sv_t *svp, int pri, lock_t *lockp);
void SV_SIGNAL (sv_t *svp, int flags);
void SV_BROADCAST (sv_t *svp, int flags);
```

As in sleep locks, the pri argument specifies the scheduling priority to assign to the process after it wakes up, and SV_WAIT_SIG allows interruption by a signal. The lockp argument is used to pass a pointer to the basic lock protecting the predicate of the condition. The caller must hold lockp before calling SV_WAIT or SV_WAIT_SIG. The kernel atomically blocks the caller and releases lockp. When the caller returns from SV_WAIT or SV_WAIT_SIG, lockp is not held. The predicate is not guaranteed to be true when the caller runs after blocking. Hence the call to SV_WAIT or SV_WAIT_SIG should be enclosed in a while loop that checks the predicate each time.

7.11.2 Digital UNIX

The Digital UNIX synchronization primitives are derived from those of Mach. There are two types of locks—simple and complex [Denh 94]. A simple lock is a basic spin lock, implemented using the atomic test-and-set instruction of the machine. It must be declared and initialized before being used.

It is initialized to the unlocked state and cannot be held across blocking operations or context switches.

The complex lock is a single high-level abstraction supporting a number of features, such as shared and exclusive access, blocking, and recursive locking. It is a reader-writer lock and provides two options—*sleep* and *recursive*. The sleep option can be enabled or disabled while initializing the lock or at any later time. If set, the kernel will block requesters if the lock cannot be granted immediately. Further, the sleep option must be set if a thread wishes to block while holding the lock. The recursive option can be set only by a thread that has acquired the lock for exclusive use; it can be cleared only by the same thread that set the option.

The interface provides nonblocking versions of various routines, which return failure if the lock cannot be acquired immediately. There are also functions to upgrade (shared to exclusive) or downgrade (exclusive to shared). The upgrade routine will release the shared lock and return failure if there is another pending upgrade request. The nonblocking version of upgrade returns failure but does not drop the shared lock in this situation.

Sleep and Wakeup

Since a lot of code was ported from 4BSD, it was desirable to retain the basic functions of `sleep()` and `wakeup()`. Thus blocked threads were put on global sleep queues rather than per-lock queues. The algorithms needed modification to work correctly on multiprocessors, and the chief issue here was the lost wakeup problem. To fix this while retaining the framework of thread states, the `sleep()` function was rewritten using two lower-level primitives—`assert_wait()` and `thread_block()`, as shown in Figure 7-9.

Suppose a thread needs to wait for an event that is described by a predicate and a spin lock that protects it. The thread acquires the spin lock and then tests the predicate. If the thread needs to

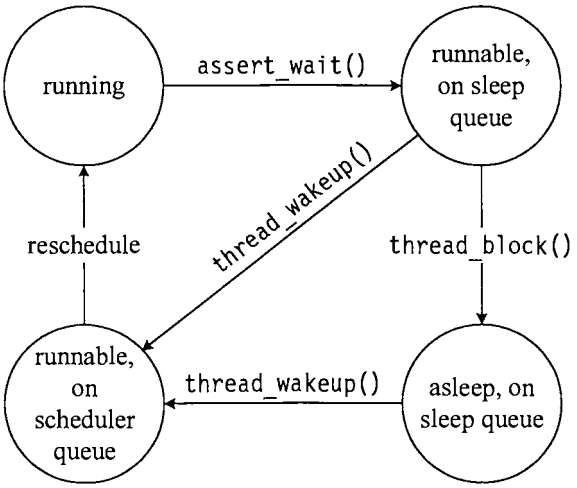

Figure 7-9. Sleep implementation in Digital UNIX.

block, it calls `assert_wait()`, which puts it on the appropriate sleep queue. It then releases the spin lock and calls `thread_block()` to initiate a context switch. If the event occurs between the release of the spin lock and the context switch, the kernel will remove the thread from the sleep queue and put it on the scheduler queue. Thus the thread does not lose the wakeup.

7.11.3 Other Implementations

The initial multiprocessing version of SVR4 was developed at NCR [Camp 91]. This version introduced the concept of *advisory processor locks* (APLs), which are recursive locks containing a *hint* for contending threads. The hint specifies if contending threads should spin or sleep, and whether the hint is advisory or mandatory. The thread owning the lock may change the hint from sleep to spin or vice versa. Such APLs may not be held across calls to sleep and are used mainly to single-thread the access to longer-term locks. The distinctive feature of APLs is that they are automatically released and reacquired across a context switch. This means that the traditional sleep/wakeup interface can be used without change. Further, locks of the same class can avoid deadlock by sleeping, since the sleep releases all previously held locks. The implementation also provided nonrecursive spin locks and read-write APLs.

The NCR version was modified by the Intel Multiprocessor Consortium, formed by a group of companies to develop the official multiprocessing release of SVR4 [Peac 92]. One important change involved the function that acquires an APL lock. This function now also takes an interrupt priority level as an argument. This allows raising of the processor priority around the holding of the lock. If the lock cannot be granted immediately, the busy-wait occurs at the original (lower) priority. The function returns the original priority, which can be passed to the unlocking function.

The lowest-level primitives are a set of atomic arithmetic and logical operations. The arithmetic operations allow atomic incrementing and decrementing of reference counts. The logical functions are used for fine-grained multithreading of bit manipulations of flag fields. They all return the original value of the variable on which they operate. At the next higher level are simple spin locks that are not released automatically on context switches. These are used for simple operations like queue insertion or removal. The highest-level locks are *resource locks*. Resource locks are long-term locks with single-writer, multiple-reader semantics, which may be held across blocking operations. The implementation also provides synchronous and asynchronous *cross-processor interrupts*, which are used for operations such as *clock tick distribution* and *address translation cache coherency* (see Section 15.9).

Solaris 2.*x* uses adaptive locks (see Section 7.10.3) and turnstiles (see Section 7.2.3) for better performance. It provides semaphores, reader-writer locks, and condition variables as high-level synchronization objects. It also uses kernel threads to handle interrupts, so that interrupt handlers use the same synchronization primitives as the rest of the kernel and block if necessary. Section 3.6.5 discusses this feature in greater detail.

Every known multiprocessor implementation uses some form of spin locks for low-level short-term synchronization. The sleep/wakeup mechanism is usually retained, perhaps with some changes, to avoid rewriting a lot of code. The major differences are in the choice of the higher-level abstractions. The early implementations on the IBM/370 and the AT&T 3B20A [Bach 84] relied exclusively on semaphores. Ultrix [Sink 88] uses blocking exclusive locks. Amdahl's UTS kernel

[Ruan 90] is based on conditions. DG/UX [Kell 89] uses *indivisible event counters* to implement *sequenced locks*, which provide a somewhat different way of waking one process at a time.

7.12 Summary

Synchronization problems on a multiprocessor are intrinsically different from and more complex than those on a uniprocessor. There are a number of different solutions, such as sleep/wakeup, conditions, events, read-write locks, and semaphores. These primitives are more similar than different, and it is possible, for example, to implement semaphores on top of conditions and vice-versa. Many of these solutions are not limited to multiprocessors and may also be applied to synchronization problems on uniprocessors or on loosely coupled distributed systems. Many multiprocessing UNIX systems are based on existing uniprocessing variants and, for these, porting considerations strongly influence the decision of which abstraction to use. Mach and Mach-based systems are mainly free of these considerations, which is reflected in their choice of primitives.

7.13 Exercises

1. Many systems have a swap-atomic instruction that swaps the value of a register with that of a memory location. Show how such an instruction may be used to implement an atomic test-and-set.
2. How can an atomic test-and-set be implemented on a machine using load-linked and store-conditional?
3. Suppose a convoy forms due to heavy contention on a critical region that is protected by a semaphore. If the region could be divided into two critical regions, each protected by a separate semaphore, would it reduce the convoy problem?
4. One way to eliminate a convoy is to replace the semaphore with another locking mechanism. Could this risk starvation of threads?
5. How is a reference count different from a shared lock?
6. Implement a blocking lock on a resource, using a spin lock and a condition variable, with a locked flag as the predicate (see Section 7.7.3).
7. In exercise 6, is it necessary to hold the spin lock protecting the predicate while clearing the flag? [Ruan 90] discusses a `waitlock()` operation that can improve this algorithm.
8. How do condition variables avoid the lost wakeup problem?
9. Implement an *event* abstraction that returns a status value to waiting threads upon event completion.
10. Suppose an object is accessed frequently for reading or writing. In what situations is it better to protect it with a simple mutex, rather than with a read-write lock?
11. Does a read-write lock have to be blocking? Implement a read-write lock that causes threads to busy-wait if the resoure is locked.
12. Describe a situation in which a deadlock may be avoided by making the locking granularity *finer*.

13. Describe a situation in which a deadlock may be avoided by making the locking granularity *coarser*.
14. Is it necessary for a multiprocessor kernel to lock each variable or resource before accessing it? Enumerate the kinds of situations where a thread may access or modify an object without locking it.
15. Monitors [Hoar 74] are language-supported constructs providing mutual exclusion to a region of code. For what sort of situations do they form a natural solution?
16. Implement `upgrade()` and `downgrade()` functions to the read-write lock implementation in Section 7.8.2.

7.14 References

[Bach 84] Bach, M., and Buroff, S., "Multiprocessor UNIX Operating Systems," *AT&T Bell Laboratories Technical Journal,* Vol. 63, Oct. 1984, pp. 1733–1749.

[Birr 89] Birrell, A.D., "An Introduction to Programming with Threads," Digital Equipment Corporation Systems Research Center, 1989.

[Camp 91] Campbell, M., Barton, R., Browning, J., Cervenka, D., Curry, B., Davis, T., Edmonds, T., Holt, R., Slice, R., Smith, T., and Wescott, R., "The Parallelization of UNIX System V Release 4.0," *Proceedings of the Winter 1991 USENIX Conference,* Jan. 1991, pp. 307–323.

[Denh 94] Denham, J.M., Long, P., and Woodward, J.A., "DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation," *Digital Technical Journal,* Vol. 6, No. 3, Summer 1994, pp. 29–54.

[Digi 87] Digital Equipment Corporation, *VAX Architecture Reference Manual,* 1984.

[Dijk 65] Dijkstra, E.W., "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM,* Vol. 8, Sep. 1965, pp. 569–578.

[Eykh 92] Eykholt, J.R., Kleinman, S.R., Barton, S., Faulkner, R., Shivalingiah, A., Smith, M., Stein, D., Voll, J., Weeks, M., and Williams, D., "Beyond Multiprocessing: Multithreading the SunOS Kernel," *Proceedings of the Summer 1992 USENIX Conference,* Jun. 1992, pp. 11–18.

[Gobl 81] Goble, G.H., "A Dual-Processor VAX 11/780," *USENIX Association Conference Proceedings,* Sep. 1981.

[Hoar 74] Hoare, C. A.R., "Monitors: An Operating System Structuring Concept," *Communications of the ACM,* Vol. 17, Oct. 1974, pp.549–557.

[Hitz 90] Hitz, D., Harris, G., Lau, J.K., and Schwartz, A.M., "Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers," *Proceedings of the Winter 1990 USENIX Technical Conference,* Jan. 1990, pp. 285–295.

[Kell 89] Kelley, M.H., "Multiprocessor Aspects of the DG/UX Kernel," *Proceedings of the Winter 1989 USENIX Conference,* Jan. 1989, pp. 85–99.

[Lee 87] Lee, T.P., and Luppi, M.W., "Solving Performance Problems on a Multiprocessor UNIX System," *Proceedings of the Summer 1987 USENIX Conference,* Jun. 1987, pp. 399–405.

[Nati 84] National Semiconductor Corporation, *Series 32000 Instruction Set Reference Manual,* 1984.

[Peac 92] Peacock, J.K., Saxena, S., Thomas, D., Yang, F., and Yu, F., "Experiences from Multithreading System V Release 4," *The Third USENIX Symposium of Experiences with Distributed and Multiprocessor Systems (SEDMS III),* Mar. 1992, pp. 77–91.

[Ruan 90] Ruane, L.M., "Process Synchronization in the UTS Kernel," *Computing Systems,* Vol. 3, Summer 1990, pp. 387–421.

[Sink 88] Sinkewicz, U., "A Strategy for SMP ULTRIX," *Proceedings of the Summer 1988 USENIX Technical Conference,* Jun. 1988, pp. 203–212.

[UNIX 92] UNIX System Laboratories, *Device Driver Reference—UNIX SVR4.2,* UNIX Press, Prentice-Hall, Englewood Cliffs, NJ, 1992.