CHAPTER 5

Concurrency and Race Conditions

Thus far, we have paid little attention to the problem of concurrency—i.e., what happens when the system tries to do more than one thing at once. The management of concurrency is, however, one of the core problems in operating systems programming. Concurrency-related bugs are some of the easiest to create and some of the hardest to find. Even expert Linux kernel programmers end up creating concurrency-related bugs on occasion.

In early Linux kernels, there were relatively few sources of concurrency. Symmetric multiprocessing (SMP) systems were not supported by the kernel, and the only cause of concurrent execution was the servicing of hardware interrupts. That approach offers simplicity, but it no longer works in a world that prizes performance on systems with more and more processors, and that insists that the system respond to events quickly. In response to the demands of modern hardware and applications, the Linux kernel has evolved to a point where many more things are going on simultaneously. This evolution has resulted in far greater performance and scalability. It has also, however, significantly complicated the task of kernel programming. Device driver programmers must now factor concurrency into their designs from the beginning, and they must have a strong understanding of the facilities provided by the kernel for concurrency management.

The purpose of this chapter is to begin the process of creating that understanding. To that end, we introduce facilities that are immediately applied to the *scull* driver from Chapter 3. Other facilities presented here are not put to use for some time yet. But first, we take a look at what could go wrong with our simple *scull* driver and how to avoid these potential problems.

Pitfalls in scull

Let us take a quick look at a fragment of the *scull* memory management code. Deep down inside the *write* logic, *scull* must decide whether the memory it requires has been allocated yet or not. One piece of the code that handles this task is:

```
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto out;
}
```

Suppose for a moment that two processes (we'll call them "A" and "B") are independently attempting to write to the same offset within the same *scull* device. Each process reaches the `if` test in the first line of the fragment above at the same time. If the pointer in question is `NULL`, each process will decide to allocate memory, and each will assign the resulting pointer to `dptr->data[s_pos]`. Since both processes are assigning to the same location, clearly only one of the assignments will prevail.

What will happen, of course, is that the process that completes the assignment second will "win." If process A assigns first, its assignment will be overwritten by process B. At that point, *scull* will forget entirely about the memory that A allocated; it only has a pointer to B's memory. The memory allocated by A, thus, will be dropped and never returned to the system.

This sequence of events is a demonstration of a *race condition*. Race conditions are a result of uncontrolled access to shared data. When the wrong access pattern happens, something unexpected results. For the race condition discussed here, the result is a memory leak. That is bad enough, but race conditions can often lead to system crashes, corrupted data, or security problems as well. Programmers can be tempted to disregard race conditions as extremely low probability events. But, in the computing world, one-in-a-million events can happen every few seconds, and the consequences can be grave.

We will eliminate race conditions from *scull* shortly, but first we need to take a more general view of concurrency.

Concurrency and Its Management

In a modern Linux system, there are numerous sources of concurrency and, therefore, possible race conditions. Multiple user-space processes are running, and they can access your code in surprising combinations of ways. SMP systems can be executing your code simultaneously on different processors. Kernel code is preemptible; your driver's code can lose the processor at any time, and the process that replaces it could also be running in your driver. Device interrupts are asynchronous events that can cause concurrent execution of your code. The kernel also provides various mechanisms for delayed code execution, such as workqueues, tasklets, and timers, which

can cause your code to run at any time in ways unrelated to what the current process is doing. In the modern, hot-pluggable world, your device could simply disappear while you are in the middle of working with it.

Avoidance of race conditions can be an intimidating task. In a world where anything can happen at any time, how does a driver programmer avoid the creation of absolute chaos? As it turns out, most race conditions can be avoided through some thought, the kernel's concurrency control primitives, and the application of a few basic principles. We'll start with the principles first, then get into the specifics of how to apply them.

Race conditions come about as a result of shared access to resources. When two threads of execution* have a reason to work with the same data structures (or hardware resources), the potential for mixups always exists. So the first rule of thumb to keep in mind as you design your driver is to avoid shared resources whenever possible. If there is no concurrent access, there can be no race conditions. So carefully-written kernel code should have a minimum of sharing. The most obvious application of this idea is to avoid the use of global variables. If you put a resource in a place where more than one thread of execution can find it, there should be a strong reason for doing so.

The fact of the matter is, however, that such sharing is often required. Hardware resources are, by their nature, shared, and software resources also must often be available to more than one thread. Bear in mind as well that global variables are far from the only way to share data; any time your code passes a pointer to some other part of the kernel, it is potentially creating a new sharing situation. Sharing is a fact of life.

Here is the hard rule of resource sharing: any time that a hardware or software resource is shared beyond a single thread of execution, and the possibility exists that one thread could encounter an inconsistent view of that resource, you must explicitly manage access to that resource. In the *scull* example above, process B's view of the situation is inconsistent; unaware that process A has already allocated memory for the (shared) device, it performs its own allocation and overwrites A's work. In this case, we must control access to the *scull* data structure. We need to arrange things so that the code either sees memory that has been allocated or knows that no memory has been *or will be* allocated by anybody else. The usual technique for access management is called *locking* or *mutual exclusion*—making sure that only one thread of execution can manipulate a shared resource at any time. Much of the rest of this chapter will be devoted to locking.

* For the purposes of this chapter, a "thread" of execution is any context that is running code. Each process is clearly a thread of execution, but so is an interrupt handler or other code running in response to an asynchronous kernel event.

First, however, we must briefly consider one other important rule. When kernel code creates an object that will be shared with any other part of the kernel, that object must continue to exist (and function properly) until it is known that no outside references to it exist. The instant that *scull* makes its devices available, it must be prepared to handle requests on those devices. And *scull* must continue to be able to handle requests on its devices until it knows that no reference (such as open user-space files) to those devices exists. Two requirements come out of this rule: no object can be made available to the kernel until it is in a state where it can function properly, and references to such objects must be tracked. In most cases, you'll find that the kernel handles reference counting for you, but there are always exceptions.

Following the above rules requires planning and careful attention to detail. It is easy to be surprised by concurrent access to resources you hadn't realized were shared. With some effort, however, most race conditions can be headed off before they bite you—or your users.

Semaphores and Mutexes

So let us look at how we can add locking to *scull*. Our goal is to make our operations on the *scull* data structure *atomic*, meaning that the entire operation happens at once as far as other threads of execution are concerned. For our memory leak example, we need to ensure that if one thread finds that a particular chunk of memory must be allocated, it has the opportunity to perform that allocation before any other thread can make that test. To this end, we must set up *critical sections*: code that can be executed by only one thread at any given time.

Not all critical sections are the same, so the kernel provides different primitives for different needs. In this case, every access to the *scull* data structure happens in process context as a result of a direct user request; no accesses will be made from interrupt handlers or other asynchronous contexts. There are no particular latency (response time) requirements; application programmers understand that I/O requests are not usually satisfied immediately. Furthermore, the *scull* is not holding any other critical system resource while it is accessing its own data structures. What all this means is that if the *scull* driver goes to sleep while waiting for its turn to access the data structure, nobody is going to mind.

"Go to sleep" is a well-defined term in this context. When a Linux process reaches a point where it cannot make any further processes, it goes to sleep (or "blocks"), yielding the processor to somebody else until some future time when it can get work done again. Processes often sleep when waiting for I/O to complete. As we get deeper into the kernel, we will encounter a number of situations where we cannot sleep. The *write* method in *scull* is not one of those situations, however. So we can use a locking mechanism that might cause the process to sleep while waiting for access to the critical section.

Just as importantly, we will be performing an operation (memory allocation with *kmalloc*) that could sleep—so sleeps are a possibility in any case. If our critical sections are to work properly, we must use a locking primitive that works when a thread that owns the lock sleeps. Not all locking mechanisms can be used where sleeping is a possibility (we'll see some that don't later in this chapter). For our present needs, however, the mechanism that fits best is a *semaphore*.

Semaphores are a well-understood concept in computer science. At its core, a semaphore is a single integer value combined with a pair of functions that are typically called *P* and *V*. A process wishing to enter a critical section will call *P* on the relevant semaphore; if the semaphore's value is greater than zero, that value is decremented by one and the process continues. If, instead, the semaphore's value is `0` (or less), the process must wait until somebody else releases the semaphore. Unlocking a semaphore is accomplished by calling *V*; this function increments the value of the semaphore and, if necessary, wakes up processes that are waiting.

When semaphores are used for *mutual exclusion*—keeping multiple processes from running within a critical section simultaneously—their value will be initially set to `1`. Such a semaphore can be held only by a single process or thread at any given time. A semaphore used in this mode is sometimes called a *mutex*, which is, of course, an abbreviation for "mutual exclusion." Almost all semaphores found in the Linux kernel are used for mutual exclusion.

The Linux Semaphore Implementation

The Linux kernel provides an implementation of semaphores that conforms to the above semantics, although the terminology is a little different. To use semaphores, kernel code must include *<asm/semaphore.h>*. The relevant type is `struct semaphore`; actual semaphores can be declared and initialized in a few ways. One is to create a semaphore directly, then set it up with *sema_init*:

```
void sema_init(struct semaphore *sem, int val);
```

where `val` is the initial value to assign to a semaphore.

Usually, however, semaphores are used in a mutex mode. To make this common case a little easier, the kernel has provided a set of helper functions and macros. Thus, a mutex can be declared and initialized with one of the following:

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

Here, the result is a semaphore variable (called `name`) that is initialized to `1` (with `DECLARE_MUTEX`) or `0` (with `DECLARE_MUTEX_LOCKED`). In the latter case, the mutex starts out in a locked state; it will have to be explicitly unlocked before any thread will be allowed access.

If the mutex must be initialized at runtime (which is the case if it is allocated dynamically, for example), use one of the following:

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

In the Linux world, the *P* function is called *down*—or some variation of that name. Here, "down" refers to the fact that the function decrements the value of the semaphore and, perhaps after putting the caller to sleep for a while to wait for the semaphore to become available, grants access to the protected resources. There are three versions of *down*:

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
```

down decrements the value of the semaphore and waits as long as need be. *down_interruptible* does the same, but the operation is interruptible. The interruptible version is almost always the one you will want; it allows a user-space process that is waiting on a semaphore to be interrupted by the user. You do not, as a general rule, want to use noninterruptible operations unless there truly is no alternative. Noninterruptible operations are a good way to create unkillable processes (the dreaded "D state" seen in *ps*), and annoy your users. Using *down_interruptible* requires some extra care, however, if the operation is interrupted, the function returns a nonzero value, and the caller does *not* hold the semaphore. Proper use of *down_interruptible* requires always checking the return value and responding accordingly.

The final version (*down_trylock*) never sleeps; if the semaphore is not available at the time of the call, *down_trylock* returns immediately with a nonzero return value.

Once a thread has successfully called one of the versions of *down*, it is said to be "holding" the semaphore (or to have "taken out" or "acquired" the semaphore). That thread is now entitled to access the critical section protected by the semaphore. When the operations requiring mutual exclusion are complete, the semaphore must be returned. The Linux equivalent to *V* is *up*:

```
void up(struct semaphore *sem);
```

Once *up* has been called, the caller no longer holds the semaphore.

As you would expect, any thread that takes out a semaphore is required to release it with one (and only one) call to *up*. Special care is often required in error paths; if an error is encountered while a semaphore is held, that semaphore must be released before returning the error status to the caller. Failure to free a semaphore is an easy error to make; the result (processes hanging in seemingly unrelated places) can be hard to reproduce and track down.

Using Semaphores in scull

The semaphore mechanism gives *scull* a tool that can be used to avoid race conditions while accessing the `scull_dev` data structure. But it is up to us to use that tool correctly. The keys to proper use of locking primitives are to specify exactly which resources are to be protected and to make sure that every access to those resources uses the proper locking. In our example driver, everything of interest is contained within the `scull_dev` structure, so that is the logical scope for our locking regime.

Let's look again at that structure:

```
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */
    int qset;                 /* the current array size */
    unsigned long size;       /* amount of data stored here */
    unsigned int access_key;  /* used by sculluid and scullpriv */
    struct semaphore sem;     /* mutual exclusion semaphore     */
    struct cdev cdev;     /* Char device structure      */
};
```

Toward the bottom of the structure is a member called `sem` which is, of course, our semaphore. We have chosen to use a separate semaphore for each virtual *scull* device. It would have been equally correct to use a single, global semaphore. The various *scull* devices share no resources in common, however, and there is no reason to make one process wait while another process is working with a different *scull* device. Using a separate semaphore for each device allows operations on different devices to proceed in parallel and, therefore, improves performance.

Semaphores must be initialized before use. *scull* performs this initialization at load time in this loop:

```
for (i = 0; i < scull_nr_devs; i++) {
    scull_devices[i].quantum = scull_quantum;
    scull_devices[i].qset = scull_qset;
    init_MUTEX(&scull_devices[i].sem);
    scull_setup_cdev(&scull_devices[i], i);
}
```

Note that the semaphore must be initialized *before* the *scull* device is made available to the rest of the system. Therefore, *init_MUTEX* is called before *scull_setup_cdev*. Performing these operations in the opposite order would create a race condition where the semaphore could be accessed before it is ready.

Next, we must go through the code and make sure that no accesses to the `scull_dev` data structure are made without holding the semaphore. Thus, for example, *scull_write* begins with this code:

```
if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
```

Note the check on the return value of *down_interruptible*; if it returns nonzero, the operation was interrupted. The usual thing to do in this situation is to return `-ERESTARTSYS`. Upon seeing this return code, the higher layers of the kernel will either restart the call from the beginning or return the error to the user. If you return `-ERESTARTSYS`, you must first undo any user-visible changes that might have been made, so that the right thing happens when the system call is retried. If you cannot undo things in this manner, you should return `-EINTR` instead.

scull_write must release the semaphore whether or not it was able to carry out its other tasks successfully. If all goes well, execution falls into the final few lines of the function:

```
out:
  up(&dev->sem);
  return retval;
```

This code frees the semaphore and returns whatever status is called for. There are several places in *scull_write* where things can go wrong; these include memory allocation failures or a fault while trying to copy data from user space. In those cases, the code performs a `goto out`, ensuring that the proper cleanup is done.

Reader/Writer Semaphores

Semaphores perform mutual exclusion for all callers, regardless of what each thread may want to do. Many tasks break down into two distinct types of work, however: tasks that only need to read the protected data structures and those that must make changes. It is often possible to allow multiple concurrent readers, as long as nobody is trying to make any changes. Doing so can optimize performance significantly; read-only tasks can get their work done in parallel without having to wait for other readers to exit the critical section.

The Linux kernel provides a special type of semaphore called a *rwsem* (or "reader/writer semaphore") for this situation. The use of rwsems in drivers is relatively rare, but they are occasionally useful.

Code using rwsems must include *<linux/rwsem.h>*. The relevant data type for reader/writer semaphores is `struct rw_semaphore`; an rwsem must be explicitly initialized at runtime with:

```
void init_rwsem(struct rw_semaphore *sem);
```

A newly initialized rwsem is available for the next task (reader or writer) that comes along. The interface for code needing read-only access is:

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

A call to *down_read* provides read-only access to the protected resources, possibly concurrently with other readers. Note that *down_read* may put the calling process

into an uninterruptible sleep. *down_read_trylock* will not wait if read access is unavailable; it returns nonzero if access was granted, `0` otherwise. Note that the convention for *down_read_trylock* differs from that of most kernel functions, where success is indicated by a return value of `0`. A rwsem obtained with *down_read* must eventually be freed with *up_read*.

The interface for writers is similar:

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

down_write, *down_write_trylock*, and *up_write* all behave just like their reader counterparts, except, of course, that they provide write access. If you have a situation where a writer lock is needed for a quick change, followed by a longer period of read-only access, you can use *downgrade_write* to allow other readers in once you have finished making changes.

An rwsem allows either one writer or an unlimited number of readers to hold the semaphore. Writers get priority; as soon as a writer tries to enter the critical section, no readers will be allowed in until all writers have completed their work. This implementation can lead to reader *starvation*—where readers are denied access for a long time—if you have a large number of writers contending for the semaphore. For this reason, rwsems are best used when write access is required only rarely, and writer access is held for short periods of time.

Completions

A common pattern in kernel programming involves initiating some activity outside of the current thread, then waiting for that activity to complete. This activity can be the creation of a new kernel thread or user-space process, a request to an existing process, or some sort of hardware-based action. It such cases, it can be tempting to use a semaphore for synchronization of the two tasks, with code such as:

```
struct semaphore sem;

init_MUTEX_LOCKED(&sem);
start_external_task(&sem);
down(&sem);
```

The external task can then call `up(&sem)` when its work is done.

As is turns out, semaphores are not the best tool to use in this situation. In normal use, code attempting to lock a semaphore finds that semaphore available almost all the time; if there is significant contention for the semaphore, performance suffers and the locking scheme needs to be reviewed. So semaphores have been heavily optimized for the "available" case. When used to communicate task completion in the way shown above, however, the thread calling *down* will almost always have to wait; performance

will suffer accordingly. Semaphores can also be subject to a (difficult) race condition when used in this way if they are declared as automatic variables. In some cases, the semaphore could vanish before the process calling *up* is finished with it.

These concerns inspired the addition of the "completion" interface in the 2.4.7 kernel. Completions are a lightweight mechanism with one task: allowing one thread to tell another that the job is done. To use completions, your code must include *<linux/completion.h>*. A completion can be created with:

```
DECLARE_COMPLETION(my_completion);
```

Or, if the completion must be created and initialized dynamically:

```
struct completion my_completion;
/* ... */
init_completion(&my_completion);
```

Waiting for the completion is a simple matter of calling:

```
void wait_for_completion(struct completion *c);
```

Note that this function performs an uninterruptible wait. If your code calls *wait_for_completion* and nobody ever completes the task, the result will be an unkillable process.*

On the other side, the actual completion event may be signalled by calling one of the following:

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

The two functions behave differently if more than one thread is waiting for the same completion event. *complete* wakes up only one of the waiting threads while *complete_all* allows all of them to proceed. In most cases, there is only one waiter, and the two functions will produce an identical result.

A completion is normally a one-shot device; it is used once then discarded. It is possible, however, to reuse completion structures if proper care is taken. If *complete_all* is not used, a completion structure can be reused without any problems as long as there is no ambiguity about what event is being signalled. If you use *complete_all*, however, you must reinitialize the completion structure before reusing it. The macro:

```
INIT_COMPLETION(struct completion c);
```

can be used to quickly perform this reinitialization.

As an example of how completions may be used, consider the *complete* module, which is included in the example source. This module defines a device with simple semantics: any process trying to read from the device will wait (using *wait_for_completion*)

* As of this writing, patches adding interruptible versions were in circulation but had not been merged into the mainline.

until some other process writes to the device. The code which implements this behavior is:

```
DECLARE_COMPLETION(comp);

ssize_t complete_read (struct file *filp, char __user *buf, size_t count, loff_t
*pos)
{
    printk(KERN_DEBUG "process %i (%s) going to sleep\n",
            current->pid, current->comm);
    wait_for_completion(&comp);
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
    return 0; /* EOF */
}

ssize_t complete_write (struct file *filp, const char __user *buf, size_t count,
        loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
            current->pid, current->comm);
    complete(&comp);
    return count; /* succeed, to avoid retrial */
}
```

It is possible to have multiple processes "reading" from this device at the same time. Each write to the device will cause exactly one read operation to complete, but there is no way to know which one it will be.

A typical use of the completion mechanism is with kernel thread termination at module exit time. In the prototypical case, some of the driver internal workings is performed by a kernel thread in a `while (1)` loop. When the module is ready to be cleaned up, the exit function tells the thread to exit and then waits for completion. To this aim, the kernel includes a specific function to be used by the thread:

```
void complete_and_exit(struct completion *c, long retval);
```

Spinlocks

Semaphores are a useful tool for mutual exclusion, but they are not the only such tool provided by the kernel. Instead, most locking is implemented with a mechanism called a *spinlock*. Unlike semaphores, spinlocks may be used in code that cannot sleep, such as interrupt handlers. When properly used, spinlocks offer higher performance than semaphores in general. They do, however, bring a different set of constraints on their use.

Spinlocks are simple in concept. A spinlock is a mutual exclusion device that can have only two values: "locked" and "unlocked." It is usually implemented as a single bit in an integer value. Code wishing to take out a particular lock tests the relevant bit. If the lock is available, the "locked" bit is set and the code continues into the critical section. If, instead, the lock has been taken by somebody else, the code goes into

a tight loop where it repeatedly checks the lock until it becomes available. This loop is the "spin" part of a spinlock.

Of course, the real implementation of a spinlock is a bit more complex than the description above. The "test and set" operation must be done in an atomic manner so that only one thread can obtain the lock, even if several are spinning at any given time. Care must also be taken to avoid deadlocks on *hyperthreaded* processors—chips that implement multiple, virtual CPUs sharing a single processor core and cache. So the actual spinlock implementation is different for every architecture that Linux supports. The core concept is the same on all systems, however, when there is contention for a spinlock, the processors that are waiting execute a tight loop and accomplish no useful work.

Spinlocks are, by their nature, intended for use on multiprocessor systems, although a uniprocessor workstation running a preemptive kernel behaves like SMP, as far as concurrency is concerned. If a nonpreemptive uniprocessor system ever went into a spin on a lock, it would spin forever; no other thread would ever be able to obtain the CPU to release the lock. For this reason, spinlock operations on uniprocessor systems without preemption enabled are optimized to do nothing, with the exception of the ones that change the IRQ masking status. Because of preemption, even if you never expect your code to run on an SMP system, you still need to implement proper locking.

Introduction to the Spinlock API

The required include file for the spinlock primitives is *<linux/spinlock.h>*. An actual lock has the type `spinlock_t`. Like any other data structure, a spinlock must be initialized. This initialization may be done at compile time as follows:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

or at runtime with:

```
void spin_lock_init(spinlock_t *lock);
```

Before entering a critical section, your code must obtain the requisite lock with:

```
void spin_lock(spinlock_t *lock);
```

Note that all spinlock waits are, by their nature, uninterruptible. Once you call *spin_lock*, you will spin until the lock becomes available.

To release a lock that you have obtained, pass it to:

```
void spin_unlock(spinlock_t *lock);
```

There are many other spinlock functions, and we will look at them all shortly. But none of them depart from the core idea shown by the functions listed above. There is very little that one can do with a lock, other than lock and release it. However, there

are a few rules about how you must work with spinlocks. We will take a moment to look at those before getting into the full spinlock interface.

Spinlocks and Atomic Context

Imagine for a moment that your driver acquires a spinlock and goes about its business within its critical section. Somewhere in the middle, your driver loses the processor. Perhaps it has called a function (*copy_from_user*, say) that puts the process to sleep. Or, perhaps, kernel preemption kicks in, and a higher-priority process pushes your code aside. Your code is now holding a lock that it will not release any time in the foreseeable future. If some other thread tries to obtain the same lock, it will, in the best case, wait (spinning in the processor) for a very long time. In the worst case, the system could deadlock entirely.

Most readers would agree that this scenario is best avoided. Therefore, the core rule that applies to spinlocks is that any code must, while holding a spinlock, be atomic. It cannot sleep; in fact, it cannot relinquish the processor for any reason except to service interrupts (and sometimes not even then).

The kernel preemption case is handled by the spinlock code itself. Any time kernel code holds a spinlock, preemption is disabled on the relevant processor. Even uniprocessor systems must disable preemption in this way to avoid race conditions. That is why proper locking is required even if you never expect your code to run on a multiprocessor machine.

Avoiding sleep while holding a lock can be more difficult; many kernel functions can sleep, and this behavior is not always well documented. Copying data to or from user space is an obvious example: the required user-space page may need to be swapped in from the disk before the copy can proceed, and that operation clearly requires a sleep. Just about any operation that must allocate memory can sleep; *kmalloc* can decide to give up the processor, and wait for more memory to become available unless it is explicitly told not to. Sleeps can happen in surprising places; writing code that will execute under a spinlock requires paying attention to every function that you call.

Here's another scenario: your driver is executing and has just taken out a lock that controls access to its device. While the lock is held, the device issues an interrupt, which causes your interrupt handler to run. The interrupt handler, before accessing the device, must also obtain the lock. Taking out a spinlock in an interrupt handler is a legitimate thing to do; that is one of the reasons that spinlock operations do not sleep. But what happens if the interrupt routine executes in the same processor as the code that took out the lock originally? While the interrupt handler is spinning, the noninterrupt code will not be able to run to release the lock. That processor will spin forever.

Avoiding this trap requires disabling interrupts (on the local CPU only) while the spinlock is held. There are variants of the spinlock functions that will disable interrupts for you (we'll see them in the next section). However, a complete discussion of interrupts must wait until Chapter 10.

The last important rule for spinlock usage is that spinlocks must always be held for the minimum time possible. The longer you hold a lock, the longer another processor may have to spin waiting for you to release it, and the chance of it having to spin at all is greater. Long lock hold times also keep the current processor from scheduling, meaning that a higher priority process—which really should be able to get the CPU—may have to wait. The kernel developers put a great deal of effort into reducing kernel latency (the time a process may have to wait to be scheduled) in the 2.5 development series. A poorly written driver can wipe out all that progress just by holding a lock for too long. To avoid creating this sort of problem, make a point of keeping your lock-hold times short.

The Spinlock Functions

We have already seen two functions, *spin_lock* and *spin_unlock*, that manipulate spinlocks. There are several other functions, however, with similar names and purposes. We will now present the full set. This discussion will take us into ground we will not be able to cover properly for a few chapters yet; a complete understanding of the spinlock API requires an understanding of interrupt handling and related concepts.

There are actually four functions that can lock a spinlock:

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock)
```

We have already seen how *spin_lock* works. *spin_lock_irqsave* disables interrupts (on the local processor only) before taking the spinlock; the previous interrupt state is stored in `flags`. If you are absolutely sure nothing else might have already disabled interrupts on your processor (or, in other words, you are sure that you should enable interrupts when you release your spinlock), you can use *spin_lock_irq* instead and not have to keep track of the flags. Finally, *spin_lock_bh* disables software interrupts before taking the lock, but leaves hardware interrupts enabled.

If you have a spinlock that can be taken by code that runs in (hardware or software) interrupt context, you must use one of the forms of *spin_lock* that disables interrupts. Doing otherwise can deadlock the system, sooner or later. If you do not access your lock in a hardware interrupt handler, but you do via software interrupts (in code that runs out of a tasklet, for example, a topic covered in Chapter 7), you can use *spin_lock_bh* to safely avoid deadlocks while still allowing hardware interrupts to be serviced.

There are also four ways to release a spinlock; the one you use must correspond to the function you used to take the lock:

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

Each *spin_unlock* variant undoes the work performed by the corresponding *spin_lock* function. The `flags` argument passed to *spin_unlock_irqrestore* must be the same variable passed to *spin_lock_irqsave*. You must also call *spin_lock_irqsave* and *spin_unlock_irqrestore* in the same function; otherwise, your code may break on some architectures.

There is also a set of nonblocking spinlock operations:

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

These functions return nonzero on success (the lock was obtained), `0` otherwise. There is no "try" version that disables interrupts.

Reader/Writer Spinlocks

The kernel provides a reader/writer form of spinlocks that is directly analogous to the reader/writer semaphores we saw earlier in this chapter. These locks allow any number of readers into a critical section simultaneously, but writers must have exclusive access. Reader/writer locks have a type of `rwlock_t`, defined in *<linux/spinlock.h>*. They can be declared and initialized in two ways:

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Static way */

rwlock_t my_rwlock;
rwlock_init(&my_rwlock);  /* Dynamic way */
```

The list of functions available should look reasonably familiar by now. For readers, the following functions are available:

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);

void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

Interestingly, there is no *read_trylock*.

The functions for write access are similar:

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
```

```
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);

void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

Reader/writer locks can starve readers just as rwsems can. This behavior is rarely a problem; however, if there is enough lock contention to bring about starvation, performance is poor anyway.

Locking Traps

Many years of experience with locks—experience that predates Linux—have shown that locking can be very hard to get right. Managing concurrency is an inherently tricky undertaking, and there are many ways of making mistakes. In this section, we take a quick look at things that can go wrong.

Ambiguous Rules

As has already been said above, a proper locking scheme requires clear and explicit rules. When you create a resource that can be accessed concurrently, you should define which lock will control that access. Locking should really be laid out at the beginning; it can be a hard thing to retrofit in afterward. Time taken at the outset usually is paid back generously at debugging time.

As you write your code, you will doubtless encounter several functions that all require access to structures protected by a specific lock. At this point, you must be careful: if one function acquires a lock and then calls another function that also attempts to acquire the lock, your code deadlocks. Neither semaphores nor spinlocks allow a lock holder to acquire the lock a second time; should you attempt to do so, things simply hang.

To make your locking work properly, you have to write some functions with the assumption that their caller has already acquired the relevant lock(s). Usually, only your internal, static functions can be written in this way; functions called from outside must handle locking explicitly. When you write internal functions that make assumptions about locking, do yourself (and anybody else who works with your code) a favor and document those assumptions explicitly. It can be very hard to come back months later and figure out whether you need to hold a lock to call a particular function or not.

In the case of *scull*, the design decision taken was to require all functions invoked directly from system calls to acquire the semaphore applying to the device structure

that is accessed. All internal functions, which are only called from other *scull* functions, can then assume that the semaphore has been properly acquired.

Lock Ordering Rules

In systems with a large number of locks (and the kernel is becoming such a system), it is not unusual for code to need to hold more than one lock at once. If some sort of computation must be performed using two different resources, each of which has its own lock, there is often no alternative to acquiring both locks.

Taking multiple locks can be dangerous, however. If you have two locks, called *Lock1* and *Lock2*, and code needs to acquire both at the same time, you have a potential deadlock. Just imagine one thread locking *Lock1* while another simultaneously takes *Lock2*. Then each thread tries to get the one it doesn't have. Both threads will deadlock.

The solution to this problem is usually simple: when multiple locks must be acquired, they should always be acquired in the same order. As long as this convention is followed, simple deadlocks like the one described above can be avoided. However, following lock ordering rules can be easier said than done. It is very rare that such rules are actually written down anywhere. Often the best you can do is to see what other code does.

A couple of rules of thumb can help. If you must obtain a lock that is local to your code (a device lock, say) along with a lock belonging to a more central part of the kernel, take your lock first. If you have a combination of semaphores and spinlocks, you must, of course, obtain the semaphore(s) first; calling *down* (which can sleep) while holding a spinlock is a serious error. But most of all, try to avoid situations where you need more than one lock.

Fine- Versus Coarse-Grained Locking

The first Linux kernel that supported multiprocessor systems was 2.0; it contained exactly one spinlock. The *big kernel lock* turned the entire kernel into one large critical section; only one CPU could be executing kernel code at any given time. This lock solved the concurrency problem well enough to allow the kernel developers to address all of the other issues involved in supporting SMP. But it did not scale very well. Even a two-processor system could spend a significant amount of time simply waiting for the big kernel lock. The performance of a four-processor system was not even close to that of four independent machines.

So, subsequent kernel releases have included finer-grained locking. In 2.2, one spinlock controlled access to the block I/O subsystem; another worked for networking, and so on. A modern kernel can contain thousands of locks, each protecting one small resource. This sort of fine-grained locking can be good for scalability; it allows

each processor to work on its specific task without contending for locks used by other processors. Very few people miss the big kernel lock.*

Fine-grained locking comes at a cost, however. In a kernel with thousands of locks, it can be very hard to know which locks you need—and in which order you should acquire them—to perform a specific operation. Remember that locking bugs can be very difficult to find; more locks provide more opportunities for truly nasty locking bugs to creep into the kernel. Fine-grained locking can bring a level of complexity that, over the long term, can have a large, adverse effect on the maintainability of the kernel.

Locking in a device driver is usually relatively straightforward; you can have a single lock that covers everything you do, or you can create one lock for every device you manage. As a general rule, you should start with relatively coarse locking unless you have a real reason to believe that contention could be a problem. Resist the urge to optimize prematurely; the real performance constraints often show up in unexpected places.

If you do suspect that lock contention is hurting performance, you may find the *lockmeter* tool useful. This patch (available at *http://oss.sgi.com/projects/lockmeter/*) instruments the kernel to measure time spent waiting in locks. By looking at the report, you are able to determine quickly whether lock contention is truly the problem or not.

Alternatives to Locking

The Linux kernel provides a number of powerful locking primitives that can be used to keep the kernel from tripping over its own feet. But, as we have seen, the design and implementation of a locking scheme is not without its pitfalls. Often there is no alternative to semaphores and spinlocks; they may be the only way to get the job done properly. There are situations, however, where atomic access can be set up without the need for full locking. This section looks at other ways of doing things.

Lock-Free Algorithms

Sometimes, you can recast your algorithms to avoid the need for locking altogether. A number of reader/writer situations—if there is only one writer—can often work in this manner. If the writer takes care that the view of the data structure, as seen by the reader, is always consistent, it may be possible to create a lock-free data structure.

A data structure that can often be useful for lockless producer/consumer tasks is the *circular buffer*. This algorithm involves a producer placing data into one end of an

* This lock still exists in 2.6, though it covers very little of the kernel now. If you stumble across a *lock_kernel* call, you have found the big kernel lock. Do not even think about using it in any new code, however.

array, while the consumer removes data from the other. When the end of the array is reached, the producer wraps back around to the beginning. So a circular buffer requires an array and two index values to track where the next new value goes and which value should be removed from the buffer next.

When carefully implemented, a circular buffer requires no locking in the absence of multiple producers or consumers. The producer is the only thread that is allowed to modify the write index and the array location it points to. As long as the writer stores a new value into the buffer before updating the write index, the reader will always see a consistent view. The reader, in turn, is the only thread that can access the read index and the value it points to. With a bit of care to ensure that the two pointers do not overrun each other, the producer and the consumer can access the buffer concurrently with no race conditions.

Figure 5-1 shows circular buffer in several states of fill. This buffer has been defined such that an empty condition is indicated by the read and write pointers being equal, while a full condition happens whenever the write pointer is immediately behind the read pointer (being careful to account for a wrap!). When carefully programmed, this buffer can be used without locks.

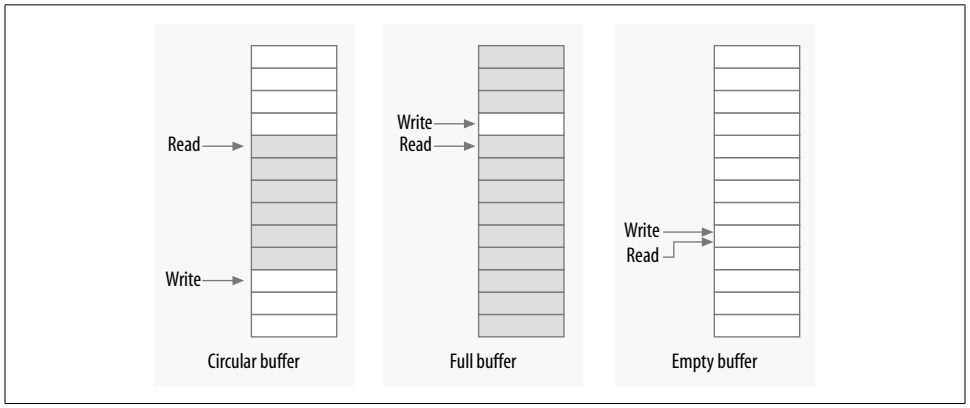

Figure 5-1. A circular buffer

Circular buffers show up reasonably often in device drivers. Networking adaptors, in particular, often use circular buffers to exchange data (packets) with the processor. Note that, as of 2.6.10, there is a generic circular buffer implementation available in the kernel; see *<linux/kfifo.h>* for information on how to use it.

Atomic Variables

Sometimes, a shared resource is a simple integer value. Suppose your driver maintains a shared variable `n_op` that tells how many device operations are currently outstanding. Normally, even a simple operation such as:

```
n_op++;
```

would require locking. Some processors might perform that sort of increment in an atomic manner, but you can't count on it. But a full locking regime seems like overhead for a simple integer value. For cases like this, the kernel provides an atomic integer type called `atomic_t`, defined in *<asm/atomic.h>*.

An `atomic_t` holds an `int` value on all supported architectures. Because of the way this type works on some processors, however, the full integer range may not be available; thus, you should not count on an `atomic_t` holding more than 24 bits. The following operations are defined for the type and are guaranteed to be atomic with respect to all processors of an SMP computer. The operations are very fast, because they compile to a single machine instruction whenever possible.

```
void atomic_set(atomic_t *v, int i);
atomic_t v = ATOMIC_INIT(0);
```
Set the atomic variable `v` to the integer value `i`. You can also initialize atomic values at compile time with the `ATOMIC_INIT` macro.

```
int atomic_read(atomic_t *v);
```
Return the current value of `v`.

```
void atomic_add(int i, atomic_t *v);
```
Add `i` to the atomic variable pointed to by `v`. The return value is `void`, because there is an extra cost to returning the new value, and most of the time there's no need to know it.

```
void atomic_sub(int i, atomic_t *v);
```
Subtract `i` from `*v`.

```
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
```
Increment or decrement an atomic variable.

```
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
```
Perform the specified operation and test the result; if, after the operation, the atomic value is `0`, then the return value is true; otherwise, it is false. Note that there is no *atomic_add_and_test*.

```
int atomic_add_negative(int i, atomic_t *v);
```
Add the integer variable `i` to `v`. The return value is true if the result is negative, false otherwise.

```
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```
Behave just like *atomic_add* and friends, with the exception that they return the new value of the atomic variable to the caller.

As stated earlier, `atomic_t` data items must be accessed only through these functions. If you pass an atomic item to a function that expects an integer argument, you'll get a compiler error.

You should also bear in mind that `atomic_t` values work only when the quantity in question is truly atomic. Operations requiring multiple `atomic_t` variables still require some other sort of locking. Consider the following code:

```
atomic_sub(amount, &first_atomic);
atomic_add(amount, &second_atomic);
```

There is a period of time where the `amount` has been subtracted from the first atomic value but not yet added to the second. If that state of affairs could create trouble for code that might run between the two operations, some form of locking must be employed.

Bit Operations

The `atomic_t` type is good for performing integer arithmetic. It doesn't work as well, however, when you need to manipulate individual bits in an atomic manner. For that purpose, instead, the kernel offers a set of functions that modify or test single bits atomically. Because the whole operation happens in a single step, no interrupt (or other processor) can interfere.

Atomic bit operations are very fast, since they perform the operation using a single machine instruction without disabling interrupts whenever the underlying platform can do that. The functions are architecture dependent and are declared in *<asm/bitops.h>*. They are guaranteed to be atomic even on SMP computers and are useful to keep coherence across processors.

Unfortunately, data typing in these functions is architecture dependent as well. The `nr` argument (describing which bit to manipulate) is usually defined as `int` but is `unsigned long` for a few architectures. The address to be modified is usually a pointer to `unsigned long`, but a few architectures use `void *` instead.

The available bit operations are:

`void set_bit(nr, void *addr);`
: Sets bit number `nr` in the data item pointed to by `addr`.

`void clear_bit(nr, void *addr);`
: Clears the specified bit in the `unsigned long` datum that lives at `addr`. Its semantics are otherwise the same as *set_bit*.

`void change_bit(nr, void *addr);`
: Toggles the bit.

`test_bit(nr, void *addr);`
: This function is the only bit operation that doesn't need to be atomic; it simply returns the current value of the bit.

`int test_and_set_bit(nr, void *addr);`
`int test_and_clear_bit(nr, void *addr);`
`int test_and_change_bit(nr, void *addr);`
: Behave atomically like those listed previously, except that they also return the previous value of the bit.

When these functions are used to access and modify a shared flag, you don't have to do anything except call them; they perform their operations in an atomic manner. Using bit operations to manage a lock variable that controls access to a shared variable, on the other hand, is a little more complicated and deserves an example. Most modern code does not use bit operations in this way, but code like the following still exists in the kernel.

A code segment that needs to access a shared data item tries to atomically acquire a lock using either *test_and_set_bit* or *test_and_clear_bit*. The usual implementation is shown here; it assumes that the lock lives at bit `nr` of address `addr`. It also assumes that the bit is `0` when the lock is free or nonzero when the lock is busy.

```
/* try to set lock */
while (test_and_set_bit(nr, addr) != 0)
    wait_for_a_while( );

/* do your work */

/* release lock, and check... */
if (test_and_clear_bit(nr, addr) == 0)
    something_went_wrong( ); /* already released: error */
```

If you read through the kernel source, you find code that works like this example. It is, however, far better to use spinlocks in new code; spinlocks are well debugged, they handle issues like interrupts and kernel preemption, and others reading your code do not have to work to understand what you are doing.

seqlocks

The 2.6 kernel contains a couple of new mechanisms that are intended to provide fast, lockless access to a shared resource. Seqlocks work in situations where the resource to be protected is small, simple, and frequently accessed, and where write access is rare but must be fast. Essentially, they work by allowing readers free access to the resource but requiring those readers to check for collisions with writers and, when such a collision happens, retry their access. Seqlocks generally cannot be used to protect data structures involving pointers, because the reader may be following a pointer that is invalid while the writer is changing the data structure.

Seqlocks are defined in *<linux/seqlock.h>*. There are the two usual methods for initializing a seqlock (which has type `seqlock_t`):

```
seqlock_t lock1 = SEQLOCK_UNLOCKED;

seqlock_t lock2;
seqlock_init(&lock2);
```

Read access works by obtaining an (unsigned) integer sequence value on entry into the critical section. On exit, that sequence value is compared with the current value; if there is a mismatch, the read access must be retried. As a result, reader code has a form like the following:

```
unsigned int seq;

do {
    seq = read_seqbegin(&the_lock);
    /* Do what you need to do */
} while read_seqretry(&the_lock, seq);
```

This sort of lock is usually used to protect some sort of simple computation that requires multiple, consistent values. If the test at the end of the computation shows that a concurrent write occurred, the results can be simply discarded and recomputed.

If your seqlock might be accessed from an interrupt handler, you should use the IRQ-safe versions instead:

```
unsigned int read_seqbegin_irqsave(seqlock_t *lock,
                                   unsigned long flags);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq,
                             unsigned long flags);
```

Writers must obtain an exclusive lock to enter the critical section protected by a seqlock. To do so, call:

```
void write_seqlock(seqlock_t *lock);
```

The write lock is implemented with a spinlock, so all the usual constraints apply. Make a call to:

```
void write_sequnlock(seqlock_t *lock);
```

to release the lock. Since spinlocks are used to control write access, all of the usual variants are available:

```
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);

void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

There is also a *write_tryseqlock* that returns nonzero if it was able to obtain the lock.

Read-Copy-Update

Read-copy-update (RCU) is an advanced mutual exclusion scheme that can yield high performance in the right conditions. Its use in drivers is rare but not unknown, so it is worth a quick overview here. Those who are interested in the full details of the RCU algorithm can find them in the white paper published by its creator (*http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html*).

RCU places a number of constraints on the sort of data structure that it can protect. It is optimized for situations where reads are common and writes are rare. The resources being protected should be accessed via pointers, and all references to those resources must be held only by atomic code. When the data structure needs to be changed, the writing thread makes a copy, changes the copy, then aims the relevant pointer at the new version—thus, the name of the algorithm. When the kernel is sure that no references to the old version remain, it can be freed.

As an example of real-world use of RCU, consider the network routing tables. Every outgoing packet requires a check of the routing tables to determine which interface should be used. The check is fast, and, once the kernel has found the target interface, it no longer needs the routing table entry. RCU allows route lookups to be performed without locking, with significant performance benefits. The Starmode radio IP driver in the kernel also uses RCU to keep track of its list of devices.

Code using RCU should include <*linux/rcupdate.h*>.

On the read side, code using an RCU-protected data structure should bracket its references with calls to *rcu_read_lock* and *rcu_read_unlock*. As a result, RCU code tends to look like:

```
struct my_stuff *stuff;

rcu_read_lock( );
stuff = find_the_stuff(args...);
do_something_with(stuff);
rcu_read_unlock( );
```

The *rcu_read_lock* call is fast; it disables kernel preemption but does not wait for anything. The code that executes while the read "lock" is held must be atomic. No reference to the protected resource may be used after the call to *rcu_read_unlock*.

Code that needs to change the protected structure has to carry out a few steps. The first part is easy; it allocates a new structure, copies data from the old one if need be, then replaces the pointer that is seen by the read code. At this point, for the purposes of the read side, the change is complete; any code entering the critical section sees the new version of the data.

All that remains is to free the old version. The problem, of course, is that code running on other processors may still have a reference to the older data, so it cannot be freed immediately. Instead, the write code must wait until it knows that no such reference

can exist. Since all code holding references to this data structure must (by the rules) be atomic, we know that once every processor on the system has been scheduled at least once, all references must be gone. So that is what RCU does; it sets aside a callback that waits until all processors have scheduled; that callback is then run to perform the cleanup work.

Code that changes an RCU-protected data structure must get its cleanup callback by allocating a `struct rcu_head`, although it doesn't need to initialize that structure in any way. Often, that structure is simply embedded within the larger resource that is protected by RCU. After the change to that resource is complete, a call should be made to:

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

The given `func` is called when it is safe to free the resource; it is passed to the same `arg` that was passed to *call_rcu*. Usually, the only thing `func` needs to do is to call *kfree*.

The full RCU interface is more complex than we have seen here; it includes, for example, utility functions for working with protected linked lists. See the relevant header files for the full story.

Quick Reference

This chapter has introduced a substantial set of symbols for the management of concurrency. The most important of these are summarized here:

`#include <asm/semaphore.h>`
: The include file that defines semaphores and the operations on them.

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```
: Two macros for declaring and initializing a semaphore used in mutual exclusion mode.

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```
: These two functions can be used to initialize a semaphore at runtime.

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
void up(struct semaphore *sem);
```
: Lock and unlock a semaphore. *down* puts the calling process into an uninterruptible sleep if need be; *down_interruptible*, instead, can be interrupted by a signal. *down_trylock* does not sleep; instead, it returns immediately if the semaphore is unavailable. Code that locks a semaphore must eventually unlock it with *up*.

```
struct rw_semaphore;
init_rwsem(struct rw_semaphore *sem);
```
The reader/writer version of semaphores and the function that initializes it.

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```
Functions for obtaining and releasing read access to a reader/writer semaphore.

```
void down_write(struct rw_semaphore *sem)
int down_write_trylock(struct rw_semaphore *sem)
void up_write(struct rw_semaphore *sem)
void downgrade_write(struct rw_semaphore *sem)
```
Functions for managing write access to a reader/writer semaphore.

```
#include <linux/completion.h>
DECLARE_COMPLETION(name);
init_completion(struct completion *c);
INIT_COMPLETION(struct completion c);
```
The include file describing the Linux completion mechanism, and the normal methods for initializing completions. `INIT_COMPLETION` should be used only to reinitialize a completion that has been previously used.

```
void wait_for_completion(struct completion *c);
```
Wait for a completion event to be signalled.

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```
Signal a completion event. *complete* wakes, at most, one waiting thread, while *complete_all* wakes all waiters.

```
void complete_and_exit(struct completion *c, long retval);
```
Signals a completion event by calling *complete* and calls *exit* for the current thread.

```
#include <linux/spinlock.h>
spinlock_t lock = SPIN_LOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);
```
The include file defining the spinlock interface and the two ways of initializing locks.

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);
```
The various ways of locking a spinlock and, possibly, disabling interrupts.

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

Nonspinning versions of the above functions; these return `0` in case of failure to obtain the lock, nonzero otherwise.

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

The corresponding ways of releasing a spinlock.

```
rwlock_t lock = RW_LOCK_UNLOCKED
rwlock_init(rwlock_t *lock);
```

The two ways of initializing reader/writer locks.

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```

Functions for obtaining read access to a reader/writer lock.

```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

Functions for releasing read access to a reader/writer spinlock.

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
```

Functions for obtaining write access to a reader/writer lock.

```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

Functions for releasing write access to a reader/writer spinlock.

```
#include <asm/atomic.h>
atomic_t v = ATOMIC_INIT(value);
void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_add_negative(int i, atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```

Atomically access integer variables. The `atomic_t` variables must be accessed only through these functions.

```
#include <asm/bitops.h>
void set_bit(nr, void *addr);
void clear_bit(nr, void *addr);
void change_bit(nr, void *addr);
test_bit(nr, void *addr);
int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
```

Atomically access bit values; they can be used for flags or lock variables. Using these functions prevents any race condition related to concurrent access to the bit.

```
#include <linux/seqlock.h>
seqlock_t lock = SEQLOCK_UNLOCKED;
seqlock_init(seqlock_t *lock);
```

The include file defining seqlocks and the two ways of initializing them.

```
unsigned int read_seqbegin(seqlock_t *lock);
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry(seqlock_t *lock, unsigned int seq);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long
  flags);
```

Functions for obtaining read access to a seqlock-protected resources.

```
void write_seqlock(seqlock_t *lock);
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);
int write_tryseqlock(seqlock_t *lock);
```

Functions for obtaining write access to a seqlock-protected resource.

```
void write_sequnlock(seqlock_t *lock);
void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

Functions for releasing write access to a seqlock-protected resource.

```
#include <linux/rcupdate.h>
```

The include file required to use the read-copy-update (RCU) mechanism.

```
void rcu_read_lock;
void rcu_read_unlock;
```

Macros for obtaining atomic read access to a resource protected by RCU.

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

Arranges for a callback to run after all processors have been scheduled and an RCU-protected resource can be safely freed.