Chapter 11 Kernel Mechanisms

This chapter describes some of the general tasks and mechanisms that the Linux kernel needs to supply so that other parts of the kernel work effectively together.

11.1 Bottom Half Handling

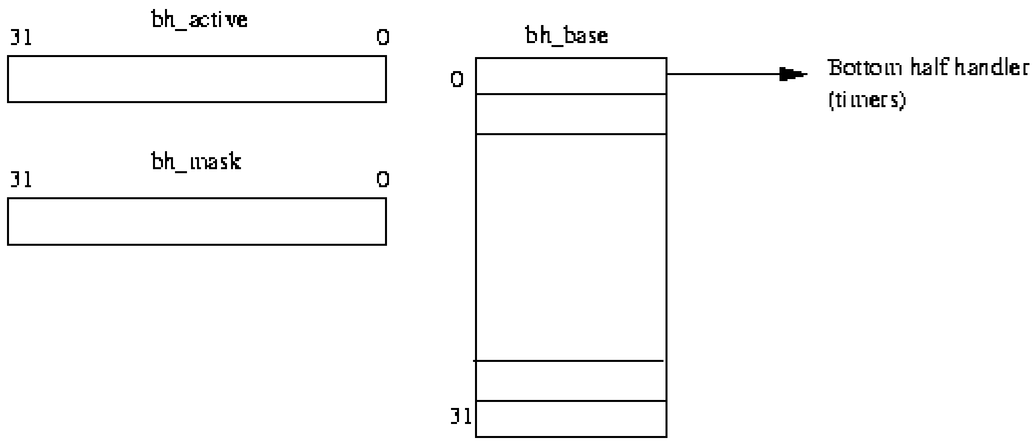

Figure 11.1: Bottom Half Handling Data Structures

There are often times in a kernel when you do not want to do work at this moment. A good example of this is during interrupt processing. When the interrupt was asserted, the processor stopped what it was doing and the operating system delivered the interrupt to the appropriate device driver. Device drivers should not spend too much time handling interrupts as, during this time, nothing else in the system can run. There is often some work that could just as well be done later on. Linux's bottom half handlers were invented so that device drivers and other parts of the Linux kernel could queue work to be done later on. Figure 11.1 shows the kernel data structures associated with bottom half handling.

There can be up to 32 different bottom half handlers; `bh_base` is a vector of pointers to each of the kernel's bottom half handling routines. `bh_active` and `bh_mask` have their bits set according to what handlers have been installed and are active. If bit N of `bh_mask` is set then the Nth element of `bh_base` contains the address of a bottom half routine. If bit N of `bh_active` is set then the N'th bottom half handler routine should be called as soon as the scheduler deems reasonable. These indices are statically defined; the timer bottom half handler is the highest priority (index 0), the console bottom half handler is next in priority (index 1) and so on. Typically the bottom half handling routines have lists of tasks associated with them. For example, the *immediate* bottom half handler works its way through the immediate tasks queue (`tq_immediate`) which contains tasks that need to be performed immediately.

Some of the kernel's bottom half handers are device specific, but others are more generic:

TIMER
: This handler is marked as active each time the system's periodic timer interrupts and is used to drive the kernel's timer queue mechanisms,

CONSOLE
: This handler is used to process console messages,

TQUEUE
: This handler is used to process `tty` messages,

NET
: This handler handles general network processing,

IMMEDIATE
: This is a generic handler used by several device drivers to queue work to be done later.

Whenever a device driver, or some other part of the kernel, needs to schedule work to be done later, it adds work to the appropriate system queue, for example the timer queue, and then signals the kernel that some bottom half handling needs to be done. It does this by setting the appropriate bit in `bh_active`. Bit 8 is set if the driver has queued something on the immediate queue and wishes the immediate bottom half handler to run and process it. The `bh_active` bitmask is checked at the end of each system call, just before control is returned to the calling process. If it has any bits set, the bottom half handler routines that are active are called. Bit 0 is checked first, then 1 and so on until bit 31.

The bit in `bh_active` is cleared as each bottom half handling routine is called. `bh_active` is transient; it only has meaning between calls to the scheduler and is a way of not calling bottom half handling routines when there is no work for them to do.

11.2 Task Queues

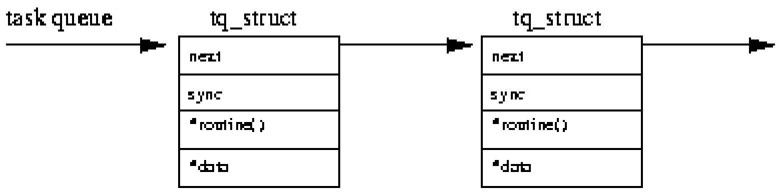

Figure 11.2: A Task Queue

Task queues are the kernel's way of deferring work until later. Linux has a generic mechanism for queuing work on queues and for processing them later.

Task queues are often used in conjunction with bottom half handlers; the timer task queue is processed when the timer queue bottom half handler runs. A task queue is a simple data structure, see figure 11.2 which consists of a singly linked list of `tq_struct` data structures each of which contains the address of a routine and a pointer to some data.

The routine will be called when the element on the task queue is processed and it will be passed a pointer to the data.

Anything in the kernel, for example a device driver, can create and use task queues but there are three task queues created and managed by the kernel:

timer
: This queue is used to queue work that will be done as soon after the next system clock tick as is possible. Each clock tick, this queue is checked to see if it contains any entries and, if it does, the timer queue bottom half handler is made active. The timer queue bottom half handler is processed, along with all the other bottom half handlers, when the scheduler next runs. This queue should not be confused with system timers, which are a much more sophisticated mechanism.

immediate
: This queue is also processed when the scheduler processes the active bottom half handlers. The immediate bottom half handler is not as high in priority as the timer queue bottom half handler and so these tasks will be run later.

scheduler

This task queue is processed directly by the scheduler. It is used to support other task queues in the system and, in this case, the task to be run will be a routine that processes a task queue, say for a device driver.

When task queues are processed, the pointer to the first element in the queue is removed from the queue and replaced with a null pointer. In fact, this removal is an atomic operation, one that cannot be interrupted. Then each element in the queue has its handling routine called in turn. The elements in the queue are often statically allocated data. However there is no inherent mechanism for discarding allocated memory. The task queue processing routine simply moves onto the next element in the list. It is the job of the task itself to ensure that it properly cleans up any allocated kernel memory.

11.3 Timers

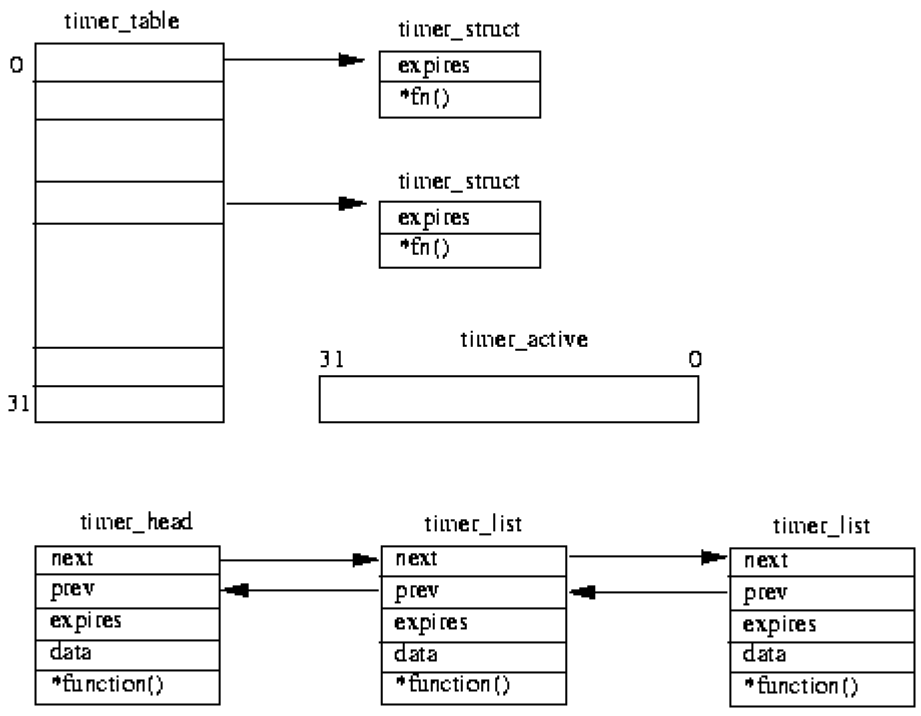

Figure 11.3: System Timers

An operating system needs to be able to schedule an activity sometime in the future. A mechanism is needed whereby activities can be scheduled to run at some relatively precise time. Any microprocessor that wishes to support an operating system must have a programmable interval timer that periodically interrupts the processor. This periodic interrupt is known as a system clock tick and it acts like a metronome, orchestrating the system's activities.

Linux has a very simple view of what time it is; it measures time in clock ticks since the system booted. All system times are based on this measurement, which is known as `jiffies` after the globally available variable of the same name.

Linux has two types of system timers, both queue routines to be called at some system time but they are slightly different in their implementations. Figure 11.3 shows both mechanisms.

The first, the old timer mechanism, has a static array of 32 pointers to `timer_struct` data structures and a mask of active timers, `timer_active`.

Where the timers go in the timer table is statically defined (rather like the bottom half handler table `bh_base`). Entries are added into this table mostly at system initialization time. The second, newer, mechanism uses a linked list of `timer_list` data structures held in ascending expiry time order.

Both methods use the time in `jiffies` as an expiry time so that a timer that wished to run in 5s would have to convert 5s to units of `jiffies` and add that to the current system time to get the system time in `jiffies` when the timer should expire. Every system clock tick the timer bottom half handler is marked as active so that the when the scheduler next runs, the timer queues will be processed. The timer bottom half handler processes both types of system timer. For the old system timers the `timer_active` bit mask is check for bits that are set.

If the expiry time for an active timer has expired (expiry time is less than the current system `jiffies`), its timer routine is called and its active bit is cleared. For new system timers, the entries in the linked list of `timer_list` data structures are checked.

Every expired timer is removed from the list and its routine is called. The new timer mechanism has the advantage of being able to pass an argument to the timer routine.

11.4 Wait Queues

There are many times when a process must wait for a system resource. For example a process may need the VFS inode describing a directory in the file system and that inode may not be in the buffer cache. In this case the process must wait for that inode to be fetched from the physical media containing the file system before it can carry on.

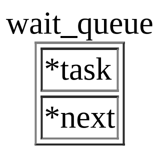

Figure 11.4: Wait Queue

The Linux kernel uses a simple data structure, a wait queue (see figure 11.4),

which consists of a pointer to the processes `task_struct` and a pointer to the next element in the wait queue.

When processes are added to the end of a wait queue they can either be interruptible or uninterruptible. Interruptible processes may be interrupted by events such as timers expiring or signals being delivered whilst they are waiting on a wait queue. The waiting processes state will reflect this and either be `INTERRUPTIBLE` or `UNINTERRUPTIBLE`. As this process can not now continue to run, the scheduler is run and, when it selects a new process to run, the waiting process will be suspended. [1]

When the wait queue is processed, the state of every process in the wait queue is set to `RUNNING`. If the process has been removed from the run queue, it is put back onto the run queue. The next time the scheduler runs, the processes that are on the wait queue are now candidates to be run as they are now no longer waiting. When a process on the wait queue is scheduled the first thing that it will do is remove itself from the wait queue. Wait queues can be used to synchronize access to system resources and they are used by Linux in its implementation of semaphores (see below).

11.5 Buzz Locks

These are better known as spin locks and they are a primitive way of protecting a data structure or piece of code. They only allow one process at a time to be within a critical region of code. They are used in Linux to restrict access to fields in data structures, using a single integer field as a lock. Each process wishing to enter the region attempts to change the lock's initial value from 0 to 1. If its current value is 1, the process tries again, spinning in a tight loop of code. The access to the memory location holding the lock must be atomic, the action of reading its value, checking that it is 0 and then changing it to 1 cannot be interrupted by any other process. Most CPU architectures provide support for this via special instructions but you can also implement buzz locks using uncached main memory.

When the owning process leaves the critical region of code it decrements the buzz lock, returning its value to 0. Any processes spinning on the lock will now read it as 0, the first one to do this will increment it to 1 and enter the critical region.

11.6 Semaphores

Semaphores are used to protect critical regions of code or data structures. Remember that each access of a critical piece of data such as a VFS inode describing a directory is made by kernel code running on behalf of a process. It would be very dangerous to allow one process to alter a critical data structure that is being used by another process. One way to achieve this would be to use a buzz lock around the critical piece of data is being accessed but this is a simplistic approach that would not give very good system performance. Instead Linux uses semaphores to allow just one process at a time to access critical regions of code and data; all other processes wishing to access this resource will be made to wait until it becomes free. The waiting processes are suspended, other processes in the system can continue to run as normal.

A Linux `semaphore` data structure contains the following information:

count
: This field keeps track of the count of processes wishing to use this resource. A positive value means that the resource is available. A negative or zero value means that processes are waiting for it. An initial value of 1 means that one and only one process at a time can use this resource. When processes want this resource they decrement the count and when they have finished with this resource they increment the count,

waking
: This is the count of processes waiting for this resource which is also the number of process waiting to be woken up when this resource becomes free,

wait queue
: When processes are waiting for this resource they are put onto this wait queue,

lock
: A buzz lock used when accessing the `waking` field.

Suppose the initial count for a semaphore is 1, the first process to come along will see that the count is positive and decrement it by 1, making it 0. The process now ``owns'' the critical piece of code or resource that is being protected by the semaphore. When the process leaves the critical region it increments the semphore's count. The most optimal case is where there are no other processes contending for ownership of the critical region. Linux has implemented semaphores to work efficiently for this, the most common, case.

If another process wishes to enter the critical region whilst it is owned by a process it too will decrement the count. As the count is now negative (-1) the process cannot enter the critical region. Instead it must wait until the owning process exits it. Linux makes the waiting process sleep until the owning process wakes it on exiting the critical region. The waiting process adds itself to the semaphore's wait queue and sits in a loop checking the value of the `waking` field and calling the scheduler until `waking` is non-zero.

The owner of the critical region increments the semaphore's count and if it is less than or equal to zero then there are processes sleeping, waiting for this resource. In the optimal case the semaphore's count would have been returned to its initial value of 1 and no further work would be neccessary. The owning process increments the waking counter and wakes up the process sleeping on the semaphore's wait queue. When the waiting process wakes up, the waking counter is now 1 and it knows that it it may now enter the critical region. It decrements the waking counter, returning it to a value of zero, and continues. All access to the waking field of semaphore are protected by a buzz lock using the semaphore's lock.

Footnotes:

[1] REVIEW NOTE: *What is to stop a task in state `INTERRUPTIBLE` being made to run the next time the scheduler runs? Processes in a wait queue should never run until they are woken up.*

Top of Chapter, Table of Contents, Show Frames, No Frames