CHAPTER 6

Advanced Process Management

Chapter 5 explained what a process is and what parts of the system it encompasses, along with system calls to create, control, and destroy it. This chapter builds on those concepts, beginning with a discussion of the Linux process scheduler and its scheduling algorithm, and then presenting advanced process management interfaces. These system calls manipulate the scheduling behavior of a process, influencing the scheduler's behavior in pursuit of an application or user-dictated goal.

Process Scheduling

The *process scheduler* is the kernel subsystem that divides the finite resource of processor time among a system's processes. In other words, the process scheduler (or simply the *scheduler*) is the component of a kernel that selects which process to run next. In deciding which processes can run and when, the scheduler is responsible for maximizing processor usage while simultaneously providing the illusion that multiple processes are executing concurrently and seamlessly.

In this chapter, we will talk a lot about *runnable* processes. A runnable process is one that is not blocked; a *blocked process* is one that is sleeping, waiting for I/O from the kernel. Processes that interact with users, read and write files heavily, or respond to network events tend to spend a lot of time blocked while they wait for resources to become available, and they are not runnable during those periods. Given only one runnable process, the job of a process scheduler is trivial: run that process! A scheduler proves its worth, however, when there are more runnable processes than processors. In such a situation, some processes will run while others must wait their turn. Deciding which processes run, when, and for how long is the process scheduler's fundamental responsibility.

An operating system on a single-processor machine is *multitasking* if it can interleave the execution of multiple processes, providing the illusion that more than one process

is running at the same time. On multiprocessor machines, a multitasking operating system allows processes to actually run in parallel, on different processors. A nonmultitasking operating system, such as DOS, can run only one application at a time.

Multitasking operating systems come in two variants: *cooperative* and *preemptive*. Linux implements the latter form of multitasking, where the scheduler decides when one process is to stop running and a different process is to resume. We call the act of suspending a running process in lieu of another one *preemption*. The length of time a process is allowed to run before the scheduler preempts it is known as the process's *timeslice*, so called because the scheduler allocates the process a "slice" of the processor's time.

In cooperative multitasking, conversely, a process does not stop running until it voluntarily decides to do so. Voluntarily suspending itself is called *yielding*. Ideally, processes yield often, but the operating system is unable to enforce this behavior. A rude or broken program can run long enough to break the illusion of multitasking, or even indefinitely so as to bring down the entire system. Due to the shortcomings of this approach, modern operating systems are almost universally preemptively multitasked; Linux is no exception.

Linux's process scheduler has changed over the years. The current process scheduler, available since Linux kernel version 2.6.23, is called the *Completely Fair Scheduler* (*CFS*). The name derives from the scheduler's adoption of *fair queuing*, a scheduling algorithm that attempts to enforce fair access to a resource among contending consumers. CFS is drastically unlike any other Unix process scheduler, including its predecessor, the *O(1) process scheduler*. We will further discuss CFS in "The Completely Fair Scheduler" on page 180.

Timeslices

The timeslice that the process scheduler allots to each process is an important variable in the overall behavior and performance of a system. If timeslices are too large, processes must wait a long time in between executions, minimizing the appearance of concurrent execution. The user may become frustrated at the perceptible delay. Conversely, if the timeslices are too small, a significant proportion of the system's time is spent switching from one application to another, and benefits such as temporal locality are lost.

Consequently, determining an ideal timeslice is not easy. Some operating systems give processes large timeslices, hoping to maximize system throughput and overall performance. Other operating systems give processes very small timeslices, hoping to provide a system with excellent interactive performance. As we will see, CFS answers the "what size timeslice" question in a very odd way: by abolishing timeslices.

I/O- Versus Processor-Bound Processes

Processes that continually consume all of their available timeslices are considered *processor-bound*. Such processes are hungry for CPU time and will consume all that the scheduler gives them. The simplest example is an infinite loop:

```
// 100% processor-bound
while (1)
  ;
```

Other, less extreme examples include scientific computations, mathematical calculations, and image processing.

On the other hand, processes that spend more time blocked waiting for some resource than executing are considered *I/O-bound*. I/O-bound processes are often issuing and waiting for file or network I/O, blocking on keyboard input, or waiting for the user to move the mouse. Examples of I/O-bound applications include file utilities that do very little except issue system calls asking the kernel to perform I/O, such as *cp* or *mv*, and many GUI applications, which spend a great deal of time waiting for user input.

Processor- and I/O-bound applications differ in the type of scheduler behavior from which they most benefit. Processor-bound applications crave the largest timeslices possible, allowing them to maximize cache hit rates (via temporal locality) and get their jobs done as quickly as possible. In contrast, I/O-bound processes do not necessarily need large timeslices because they typically run for only very short periods before issuing I/O requests and blocking on some kernel resource. I/O-bound processes, however, do benefit from prioritized attention from the scheduler. The quicker such an application can restart after blocking and dispatch more I/O requests, the better it can utilize the system's hardware. Further, if the application was waiting for user input, the faster it is scheduled, the greater the user's perception of seamless execution will be.

Juggling the needs of processor- and I/O-bound processes is not easy. In reality, most applications are some mix of I/O- and processor-bound. Audio/video encoding/decoding is a good example of a type of application that defies categorization. Many games are also quite mixed. It is not always possible to identify the proclivity of a given application, and, at any point in time, a given process may behave in one way only to behave in another way later.

Preemptive Scheduling

In traditional Unix process scheduling, all runnable processes are assigned a timeslice. When a process exhausts its timeslice, the kernel suspends it and begins running a different process. If there are no runnable processes on the system, the kernel takes the set of processes with exhausted timeslices, replenishes their timeslices, and begins running them again. And so the process repeats, with processes continually entering and exiting the runnable process list as they are created or terminate, block on I/O, or wake

up from sleep. In this fashion, all processes eventually get to run, even if there are higher-priority processes on the system—the lower-priority processes just have to wait for the higher-priority processes to exhaust their timeslices or block. This behavior formulates an important but tacit rule of Unix scheduling: all processes must make forward progress.

The Completely Fair Scheduler

The Completely Fair Scheduler (CFS) is a significant departure from traditional Unix process schedulers. In most Unix systems, including Linux before CFS's introduction, there were two fundamental per-process variables in process scheduling: priority and timeslice. As discussed in the previous section, in traditional process schedulers, processes are assigned a timeslice that represents the "slice" of the processor allotted to that process. Processes may run until they exhaust that timeslice. Similarly, processes are assigned a priority. The process scheduler runs higher-priority processes before lower-priority ones. The algorithm is very simple and worked very well for early time-sharing Unix systems. It performs less admirably on systems requiring good interactive performance and fairness, such as today's modern desktops and mobile devices.

CFS introduces a quite different algorithm called *fair scheduling* that eliminates timeslices as the unit of allotting access to the processor. Instead of timeslices, CFS assigns each process a *proportion* of the processor's time. The algorithm is simple: CFS starts by assigning *N* processes each *1/N* of the processor's time. CFS then adjusts this allotment by weighting each process's proportion by its nice value. Processes with the default nice value of zero have a weight of one, so their proportion is unchanged. Processes with a smaller nice value (higher priority) receive a larger weight, increasing their fraction of the processor, while process's with a larger nice value (lower priority) receive a smaller weight, decreasing their fraction of the processor.

CFS now has a weighted proportion of processor time assigned to each process. To determine the actual length of time each process runs, CFS needs to divide the proportions into a fixed period. That period is called the *target latency*, as it represents the scheduling latency of the system. To understand the target latency, let's assume it is set to 20 milliseconds and that there are two runnable processes of the same priority. Thus, each process has the same weight and is assigned the same proportion of the processor, 10 milliseconds. Thus, CFS will run one process for 10 milliseconds, then the other for 10 milliseconds, and then repeat. If there are 5 runnable processes on the system, CFS will run each for 4 milliseconds.

So far, so good. But what if we have, say, 200 processes? With a target latency of 20 milliseconds, CFS would run each of those processes for only 100 microseconds. Due to the cost of context switching from one process to another, known as *switching costs*, and the reduced temporal locality, the system's overall throughput would suffer. To deal with this situation, CFS introduces a second key variable, the minimum granularity.

The *minimum granularity* is a floor on the length of time any process is run. All processes, regardless of their allotted proportion of the processor, will run for at least the minimum granularity (or until they block). This helps ensure that switching costs do not consume an unacceptably large amount of the system's total time at the expense of honoring the target latency. That is, when the minimum granularity kicks in, fairness is violated. With typical values for the target latency and minimum granularity, in the common case of a reasonable number of runnable processes, the minimum granularity is not applied, the target latency is met, and fairness is maintained.

By assigning proportions of the processor and not fixed timeslices, CFS is able to enforce fairness: each process gets its *fair share* of the processor. Moreover, CFS is able to enforce a configurable scheduling latency, as the *target latency* is user-settable. On traditional Unix schedulers, processes run for fixed timeslices known *a priori*, but the scheduling latency (how often they run) is unknown. On CFS, processes run for proportions and with a latency that is known *a priori*, but the timeslice is dynamic, a function of the number of runnable processes on the system. It is a markedly different way of handling process scheduling, solving many of the problems around interactive and I/O-bound processes that has plagued traditional process schedulers.

Yielding the Processor

Although Linux is a preemptively multitasked operating system, it also provides a system call that allows processes to explicitly yield execution and instruct the scheduler to select a new process for execution:

```
#include <sched.h>

int sched_yield (void);
```

A call to `sched_yield()` results in suspension of the currently running process, after which the process scheduler selects a new process to run, in the same manner as if the kernel had itself preempted the currently running process in favor of executing a new process. Note that if no other runnable process exists, which is often the case, the yielding process will immediately resume execution. Because of this uncertainty, coupled with the general belief that there are generally better choices, use of this system call is not common.

On success, the call returns `0`; on failure, it returns `-1` and sets `errno` to the appropriate error code. On Linux—and, more than likely, most other Unix systems—`sched_yield()` cannot fail and thus always returns `0`. A thorough programmer may still check the return value, however:

```
if (sched_yield ())
        perror ("sched_yield");
```

Legitimate Uses

In practice, there are few legitimate uses of `sched_yield()` on a proper preemptive multitasking system such as Linux. The kernel is fully capable of making the optimal and most efficient scheduling decisions—certainly, the kernel is better equipped than an individual application to decide what to preempt and when. This is precisely why operating systems ditched cooperative multitasking in favor of preemptive multitasking.

Why, then, does POSIX dictate a "reschedule me" system call at all? The answer lies in applications having to wait for external events, which may be caused by the user, a hardware component, or another process. For instance, if one process needs to wait for another, "just yield the processor until the other process is done" is a first-pass solution. As an example, the implementation of a naïve consumer in a consumer/producer pair might be similar to the following:

```
/* the consumer... */
do {
        while (producer_not_ready ())
                sched_yield ());
        process_data ();
} while (!time_to_quit ());
```

Thankfully, Unix programmers do not tend to write code such as this. Unix programs are normally event-driven and tend to utilize some sort of blockable mechanism (such as a pipe) between the consumer and the producer in lieu of `sched_yield()`. In this case, the consumer reads from the pipe, blocking as necessary until data is available. The producer, in turn, writes to the pipe as fresh data becomes available. This removes the responsibility for coordination from the user-space process, who just busy-loops, to the kernel, who can optimally manage the situation by putting the processes to sleep and waking them up only as needed. In general, Unix programs should aim toward event-driven solutions that rely on blockable file descriptors.

Until recently, one situation vexingly required `sched_yield()`: user-space thread locking. When a thread attempted to acquire a lock that another thread already held, the new thread would yield the processor until the lock became available. Without kernel support for user-space locks, this approach was the simplest and most efficient. Thankfully, the modern Linux thread implementation (the Native POSIX Threading Library, or NPTL) ushered in an optimal solution using *futexes*, which provide kernel support for efficient user-space locks.

One other use for `sched_yield()` is "playing nicely": a processor-intensive program can call `sched_yield()` periodically, attempting to minimize its impact on the system. While noble in pursuit, this strategy has two flaws. First, the kernel is able to make global scheduling decisions much better than an individual process and consequently the responsibility for ensuring smooth system operation should lie with the process scheduler, not the processes. Second, mitigating the overhead of a processor-intensive application

with respect to other applications is the responsibility of the user, not of individual applications. The user can convey her relative preferences for application performance via the *nice* shell command, which we will discuss later in this chapter.

Process Priorities

The discussion in this section pertains to normal, non-real-time processes. Real-time processes require different scheduling criteria and a separate priority system. We will discuss real-time computing later in this chapter.

Linux does not schedule processes willy-nilly. Instead, processes are assigned *priorities* that affect how long they run: recall that the proportion of processor allotted to a process is weighted by its nice value. Unix has historically called these priorities *nice values* because the idea behind them was to "be nice" to other processes on the system by lowering a process's priority, allowing other processes to consume more of the system's processor time.

Legal nice values range from −20 to 19 inclusive, with a default value of 0. Somewhat confusingly, the lower a process's nice value, the higher its priority and the larger its timeslice; conversely, the higher the value, the lower the process's priority and the smaller its timeslice. Increasing a process's nice value is therefore "nice" to the rest of the system. The numerical inversion is rather confusing. When we say a process has a "high priority" we mean that it can run for longer than lower-priority processes, but such a process will have a lower nice value.

nice()

Linux provides several system calls for retrieving and setting a process's nice value. The simplest is `nice()`:

```
#include <unistd.h>

int nice (int inc);
```

A successful call to `nice()` increments a process's nice value by `inc` and returns the newly updated value. Only a process with the `CAP_SYS_NICE` capability (effectively, processes owned by root) may provide a negative value for `inc`, decreasing its nice value and thereby increasing its priority. Consequently, nonroot processes may only lower their priorities (by increasing their nice values).

On error, `nice()` returns `-1`. However, because `nice()` returns the new nice value, `-1` is also a successful return value. To differentiate between success and failure, you can zero out `errno` before invocation and subsequently check its value. For example:

```
int ret;

errno = 0;
ret = nice (10);    /* increase our nice by 10 */
if (ret == -1 && errno != 0)
        perror ("nice");
else
        printf ("nice value is now %d\n", ret);
```

Linux returns only a single error code: `EPERM`, signifying that the invoking process attempted to increase its priority (via a negative `inc` value), but it does not possess the `CAP_SYS_NICE` capability. Other systems also return `EINVAL` when `inc` would place the nice value out of the range of valid values, but Linux does not. Instead, Linux silently rounds invalid `inc` values up or down to the value at the limit of the allowable range, as needed.

Passing `0` for `inc` is an easy way to obtain the current nice value:

```
printf ("nice value is currently %d\n", nice (0));
```

Often, a process wants to set an absolute nice value rather than a relative increment. This can be done with code like the following:

```
int ret, val;

/* get current nice value */
val = nice (0);

/* we want a nice value of 10 */
val = 10 - val;
errno = 0;
ret = nice (val);
if (ret == -1 && errno != 0)
        perror ("nice");
else
        printf ("nice value is now %d\n", ret);
```

getpriority() and setpriority()

A preferable solution is to use the `getpriority()` and `setpriority()` system calls, which allow more control but are more complex in operation:

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority (int which, int who);
int setpriority (int which, int who, int prio);
```

These calls operate on the process, process group, or user, as specified by `which` and `who`. The value of `which` must be one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, in which case `who` specifies a process ID, process group ID, or user ID, respectively. If `who` is `0`, the call operates on the current process ID, process group ID, or user ID, respectively.

A call to `getpriority()` returns the highest priority (lowest numerical nice value) of any of the specified processes. A call to `setpriority()` sets the priority of all specified processes to `prio`. As with `nice()`, only a process possessing `CAP_SYS_NICE` may raise a process's priority (lower the numerical nice value). Further, only a process with this capability can raise or lower the priority of a process not owned by the invoking user.

Like `nice()`, `getpriority()` returns `-1` on error. As this is also a successful return value, programmers should clear `errno` before invocation if they want to handle error conditions. Calls to `setpriority()` have no such problem; `setpriority()` always returns `0` on success and `-1` on error.

The following code returns the current process's priority:

```
int ret;

ret = getpriority (PRIO_PROCESS, 0);
printf ("nice value is %d\n", ret);
```

The following code sets the priority of all processes in the current process group to 10:

```
int ret;

ret = setpriority (PRIO_PGRP, 0, 10);
if (ret == -1)
        perror ("setpriority");
```

On error, both functions set `errno` to one of the following values:

`EACCES`
: The process attempted to raise the specified process's priority but does not possess `CAP_SYS_NICE` (`setpriority()` only).

`EINVAL`
: The value specified by `which` was not one of `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`.

`EPERM`
: The effective user ID of the matched process does not match the effective user ID of the running process, and the running process does not possess `CAP_SYS_NICE` (`setpriority()` only).

`ESRCH`
: No process was found matching the criteria provided by `which` and `who`.

I/O Priorities

In addition to a scheduling priority, Linux allows processes to specify an *I/O priority*. This value affects the relative priority of the processes' I/O requests. The kernel's I/O scheduler (discussed in Chapter 4) services requests originating from processes with higher I/O priorities before requests from processes with lower I/O priorities.

By default, I/O schedulers use a process's nice value to determine the I/O priority. Ergo, setting the nice value automatically changes the I/O priority. However, the Linux kernel additionally provides two system calls for explicitly setting and retrieving the I/O priority independently of the nice value:

```
int ioprio_get (int which, int who)
int ioprio_set (int which, int who, int ioprio)
```

Unfortunately, *glibc* does not provide a user-space interface to these system calls. Without *glibc* support, usage is cumbersome at best. Further, when and if *glibc* support arrives, the interfaces may differ from the system calls. Until such support is available, there are two portable ways to manipulate a process's I/O priority: via the nice value or a utility such as *ionice*, part of the *util-linux* package.[1]

Not all I/O schedulers support I/O priorities. Specifically, the Completely Fair Queuing (CFQ) I/O Scheduler supports them; currently, the other standard schedulers do not. If the current I/O scheduler does not support I/O priorities, they are silently ignored.

Processor Affinity

Linux supports multiple processors in a single system. Aside from the boot process, the bulk of the work of supporting multiple processors rests on the process scheduler. On a multiprocessing machine, the process scheduler must decide which processes run on each CPU.

Two challenges derive from this responsibility: the scheduler must work toward fully utilizing all of the system's processors because it is inefficient for one CPU to sit idle while a process is waiting to run. However, once a process has been scheduled on one CPU, the process scheduler should aim to schedule it on the same CPU in the future. This is beneficial because *migrating* a process from one processor to another has costs.

The largest of these costs are related to the *cache effects* of migration. Due to the design of modern SMP systems, most of the caches associated with each processor are separate and distinct. That is, the data in one processor's cache is not in another's. Therefore, if a process moves to a new CPU and writes new data into memory, the data in the old CPU's cache can become stale. Relying on that cache would now cause corruption. To

1. The *util-linux* package is located at kernel.org. It is licensed under the GNU General Public License v2.

prevent this, caches *invalidate* each other's data whenever they cache a new chunk of memory. Consequently, a given piece of data is strictly in only one processor's cache at any given moment (assuming the data is cached at all). When a process moves from one processor to another, there are thus two associated costs: cached data is no longer accessible to the process that moved, and data in the original processor's cache must be invalidated. Because of these costs, process schedulers attempt to keep a process on a specific CPU for as long as possible.

The process scheduler's two goals, of course, are potentially conflicting. If one processor has a significantly larger process load than another—or, worse, if one processor is busy while another is idle—it makes sense to reschedule some processes on the less-busy CPU. Deciding when to move processes in response to such imbalances, called *load balancing*, is of great importance to the performance of SMP machines.

Processor affinity refers to the likelihood of a process to be scheduled consistently on the same processor. The term *soft affinity* refers to the scheduler's natural propensity to continue scheduling a process on the same processor. As we've discussed, this is a worthwhile trait. The Linux scheduler attempts to schedule the same processes on the same processors for as long as possible, migrating a process from one CPU to another only in situations of extreme load imbalance. This allows the scheduler to minimize the cache effects of migration but still ensure that all processors in a system are evenly loaded.

Sometimes, however, the user or an application wants to enforce a process-to-processor bond. This is often because the process is strongly cache-sensitive and desires to remain on the same processor. Bonding a process to a particular processor and having the kernel enforce the relationship is called setting a *hard affinity*.

sched_getaffinity() and sched_setaffinity()

Processes inherit the CPU affinities of their parents and, by default, processes may run on any CPU. Linux provides two system calls for retrieving and setting a process's hard affinity:

```
#define _GNU_SOURCE

#include <sched.h>

typedef struct cpu_set_t;

size_t CPU_SETSIZE;

void CPU_SET (unsigned long cpu, cpu_set_t *set);
void CPU_CLR (unsigned long cpu, cpu_set_t *set);
int CPU_ISSET (unsigned long cpu, cpu_set_t *set);
void CPU_ZERO (cpu_set_t *set);
```

```
int sched_setaffinity (pid_t pid, size_t setsize,
                       const cpu_set_t *set);

int sched_getaffinity (pid_t pid, size_t setsize,
                       cpu_set_t *set);
```

A call to `sched_getaffinity()` retrieves the CPU affinity of the process `pid` and stores it in the special `cpu_set_t` type, which is accessed via special macros. If `pid` is `0`, the call retrieves the current process's affinity. The `setsize` parameter is the size of the `cpu_set_t` type, which may be used by *glibc* for compatibility with future changes in the size of this type. On success, `sched_getaffinity()` returns `0`; on failure, it returns `-1`, and `errno` is set. Here's an example:

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
        perror ("sched_getaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
        int cpu;

        cpu = CPU_ISSET (i, &set);
        printf ("cpu=%i is %s\n", i,
                cpu ? "set" : "unset");
}
```

Before invocation, we use `CPU_ZERO` to "zero out" all of the bits in the set. We then iterate from `0` to `CPU_SETSIZE` over the set. Note that `CPU_SETSIZE` is, confusingly, not the size of the set: you should *never* pass it for `setsize`. Instead, it represents the number of processors that could potentially be represented by a set. Because the current implementation represents each processor with a single bit, `CPU_SETSIZE` is much larger than `sizeof(cpu_set_t)`. We use `CPU_ISSET` to check whether a given processor in the system, `i`, is bound or unbound to this process. The macro returns `0` if the processor is unbound and a nonzero value if bound.

Only processors physically on the system are set. Thus, running this snippet on a system with two processors will yield:

```
cpu=0 is set
cpu=1 is set
cpu=2 is unset
cpu=3 is unset
...
cpu=1023 is unset
```

As the output shows, `CPU_SETSIZE` (which is zero-based) is currently 1,024.

We are concerned only with CPUs #0 and #1 because they are the only physical processors on this system. Perhaps we want to ensure that our process runs only on CPU #0, and never on #1. This code does just that:

```
cpu_set_t set;
int ret, i;

CPU_ZERO (&set);        /* clear all CPUs */
CPU_SET (0, &set);      /* allow CPU #0 */
CPU_CLR (1, &set);      /* disallow CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1)
        perror ("sched_setaffinity");

for (i = 0; i < CPU_SETSIZE; i++) {
        int cpu;

        cpu = CPU_ISSET (i, &set);
        printf ("cpu=%i is %s\n", i,
                cpu ? "set" : "unset");
}
```

We start, as always, by zeroing out the set with `CPU_ZERO`. We then set CPU #0 with `CPU_SET` and unset (clear) CPU #1 with `CPU_CLR`. The `CPU_CLR` operation is redundant as we just zeroed out the whole set, but it is provided for completeness.

Running this on the same two-processor system will result in slightly different output than before:

```
cpu=0 is set
cpu=1 is unset
cpu=2 is unset
...
cpu=1023 is unset
```

Now, CPU #1 is unset. This process will run only on CPU #0, no matter what!

Four `errno` values are possible:

`EFAULT`
: The provided pointer was outside of the process's address space or otherwise invalid.

`EINVAL`
: In this case, there were no processors physically on the system enabled in `set` (`sched_setaffinity()` only), or `setsize` is smaller than the size of the kernel's internal data structure that represents sets of processors.

`EPERM`
: The process associated with `pid` is not owned by the current effective user ID of the calling process, and the process does not possess `CAP_SYS_NICE`.

`ESRCH`
: No process associated with `pid` was found.

Real-Time Systems

In computing, the term *real-time* is often the source of some confusion and misunderstanding. A system is "real-time" if it is subject to *operational deadlines*: minimum and mandatory times between stimuli and responses. A familiar real-time system is the *anti-lock braking system* (ABS) found on nearly all modern automobiles. In this system, when the brake is pressed, a computer regulates the brake pressure, often applying and releasing maximum brake pressure many times a second. This prevents the wheels from "locking up," which can reduce stopping power or even send the car into an uncontrolled skid. In such a system, the operational deadlines are how fast the system must respond to a "locked" wheel condition and how quickly the system can apply brake pressure.

Most modern operating systems, Linux included, provide some level of real-time support.

Hard Versus Soft Real-Time Systems

Real-time systems come in two varieties: hard and soft. A *hard real-time system* requires absolute adherence to operational deadlines. Exceeding the deadlines constitutes failure and is a major bug. A *soft real-time system*, on the other hand, does not consider overrunning a deadline to be a critical failure.

Hard real-time applications are easy to identify: some examples are anti-lock braking systems, military weapons systems, medical devices, and signal processing. Soft real-time applications are not always so easy to identify. One obvious member of that group is video-processing applications: users notice a drop in quality if their deadlines are missed, but a few lost frames can be tolerated.

Many other applications have timing constraints that, if not met, result in a detriment to the user experience. Multimedia applications, games, and networking programs come to mind. What about a text editor, however? If the program cannot respond quickly enough to keypresses, the experience is poor, and the user may grow angry or frustrated. Is this a soft real-time application? Certainly, when the developers were writing the application, they realized that they needed to respond to keypresses in a timely manner. But does this count as an operational deadline? The line defining soft real-time applications is anything but clear.

Contrary to common belief, a real-time system is not necessarily fast. Indeed, given comparable hardware, a real-time system is probably slower than a non-real-time system—due to, if nothing else, the increase in overhead required to support real-time processes. Likewise, the division between hard and soft real-time systems is independent of the size of the operational deadlines. A nuclear reactor will overheat if the SCRAM system does not lower the control rods within several seconds of detecting excessive neutron flux. This is a hard real-time system with a lengthy (as far as computers are concerned) operational deadline. Conversely, a video player might skip a frame or stutter the sound if the application cannot refill the playback buffer within 100 ms. This is a soft real-time system with a demanding operational deadline.

Latency, Jitter, and Deadlines

Latency refers to the period from the occurrence of the stimulus until the execution of the response. If latency is less than or equal to the operational deadline, the system is operating correctly. In many hard real-time systems, the operational deadline and the latency are equal—the system handles stimuli in fixed intervals, at exact times. In soft real-time systems, the required response is less exact, and latency exhibits some amount of variance—the aim is simply for the response to occur within the deadline.

It is often hard to measure latency because its calculation requires knowing the time when the stimulus occurred. The ability to timestamp the stimulus, however, begs the ability to respond to it. Therefore, many attempts at instrumenting latency do no such thing; instead, they measure the variation in timing between responses. The variation in timing between successive events is *jitter*, not latency.

For example, consider a stimulus that occurs every 10 milliseconds. To measure the performance of our system, we might timestamp our responses to ensure that they occur every 10 milliseconds. The deviation from this target is not latency, however—it is jitter. What we are measuring is the variance in successive responses. Without knowing when the stimulus occurred, we do not know the actual difference in time between stimulus and response. Even knowing that the stimulus occurs every 10 ms, we do not know when the *first* occurrence was. Perhaps surprisingly, many attempts at measuring latency make this mistake and report jitter, not latency. To be sure, jitter is a useful metric, and such instrumentation is probably quite useful. Nevertheless, we must call a duck a duck!

Hard real-time systems often exhibit very low jitter because they respond to stimuli after—not *within*—an exact amount of time. Such systems aim for a jitter of zero and a latency equal to the operational delay. If the latency exceeds the delay, the system fails.

Soft real-time systems are more susceptible to jitter. In these systems, the response time is ideally within the operational delay—often much sooner, sometimes not. Jitter, therefore, is often an excellent surrogate for latency as a performance metric.

Linux's Real-Time Support

Linux provides applications with soft real-time support via a family of system calls defined by IEEE Std 1003.1b-1993 (often shortened to POSIX 1993 or POSIX.1b).

Technically speaking, the POSIX standard does not dictate whether the provided real-time support is soft or hard. In fact, all the POSIX standard really does is describe several scheduling policies that respect priorities. What sorts of timing constraints the operating system enforces on these policies is up to the OS designers.

Over the years, the Linux kernel has gained better and better real-time support, providing lower and lower latency and more consistent jitter, without compromising system performance. Much of this is because improving latency helps many classes of application, such as desktop and I/O-bound processes, and not just real-time applications. The improvements are also attributable to the success of Linux in embedded and real-time systems.

Unfortunately, many of the embedded and real-time modifications that have been made to the Linux kernel exist only in custom Linux solutions outside of the mainstream official kernel. Some of these modifications provide further reductions in latency and even hard real-time behavior. The following sections discuss only the official kernel interfaces and the behavior of the mainstream kernel. Luckily, most real-time modifications continue to utilize the POSIX interfaces. Ergo, the subsequent discussion is also relevant on modified systems.

Linux Scheduling Policies and Priorities

The behavior of the Linux scheduler with respect to a process depends on the process's *scheduling policy*, also called the *scheduling class*. In addition to the normal default policy, Linux provides two real-time scheduling policies. A preprocessor macro from the header `<sched.h>` represents each policy: the macros are `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`.

Every process possesses a *static priority*, unrelated to the nice value. For normal applications, this priority is always 0. For the real-time processes, it ranges from 1 to 99, inclusive. The Linux scheduler always selects the highest-priority process to run (i.e., the one with the largest numerical static priority value). If a process is running with a static priority of 50, and a process with a priority of 51 becomes runnable, the scheduler will immediately preempt the running process and switch to the newly runnable process. Conversely, if a process is running with a priority of 50, and a process with a priority of 49 becomes runnable, the scheduler will not run it until the priority-50 process blocks, becoming unrunnable. Because normal processes have a priority of 0, any real-time process that is runnable will always preempt a normal process and run.

The first in, first out policy

The *first in, first out (FIFO) class* is a very simple real-time policy without timeslices. A FIFO-classed process will continue running so long as no higher-priority process becomes runnable. The FIFO class is represented by the macro `SCHED_FIFO`.

As the policy lacks timeslices, its rules of operation are rather simple:

- A runnable FIFO-classed process will always run if it is the highest-priority process on the system. Particularly, once a FIFO-classed process becomes runnable, it will immediately preempt a normal process.
- A FIFO-classed process will continue running until it blocks or calls `sched_yield()`, or until a higher-priority process becomes runnable.
- When a FIFO-classed process blocks, the scheduler removes it from the list of runnable processes. When it again becomes runnable, it is inserted at the end of the list of processes at its priority. Thus, it will not run until any other processes of higher *or equal* priority cease execution.
- When a FIFO-classed process calls `sched_yield()`, the scheduler moves it to the end of the list of processes at its priority. Thus, it will not run until any other equal-priority processes cease execution. If the invoking process is the only process at its priority, `sched_yield()` will have no effect.
- When a higher-priority process preempts a FIFO-classed process, the FIFO-classed process remains at the same location in the list of processes for its given priority. Thus, once the higher-priority process ceases execution, the preempted FIFO-classed process will continue executing.
- When a process joins the FIFO class, or when a process's static priority changes, it is put at the head of the list of processes for its given priority. Consequently, a newly prioritized FIFO-classed process can preempt an executing process of the same priority.

Essentially, we can say that FIFO-classed processes always run for as long as they want, so long as they are the highest-priority processes on the system. The interesting rules pertain to what happens among FIFO-classed processes with the same priority.

The round-robin policy

The *round-robin (RR) class* is identical to the FIFO class, except that it imposes additional rules in the case of processes with the same priority. The macro `SCHED_RR` represents this class.

The scheduler assigns each RR-classed process a timeslice. When an RR-classed process exhausts its timeslice, the scheduler moves it to the end of the list of processes at its priority. In this manner, RR-classed processes of a given priority are scheduled

round-robin amongst themselves. If there is only one process at a given priority, the RR class is identical to the FIFO class. In such a case, when its timeslice expires, the process simply resumes execution.

We can think of an RR-classed process as identical to a FIFO-classed process, except that it additionally ceases execution when it exhausts its timeslice, at which time it moves to the end of the list of runnable processes at its priority.

Deciding whether to use `SCHED_FIFO` or `SCHED_RR` is entirely a question of intra-priority behavior. The RR class's timeslices are relevant only among processes of the same priority. FIFO-classed processes will continue running unabated; RR-classed processes will schedule amongst themselves at a given priority. In neither case will a lower-priority process ever run if a higher-priority process exists.

The normal policy

`SCHED_OTHER` represents the standard scheduling policy, the default nonreal-time class. All normal-classed processes have a static priority of 0. Consequently, any runnable FIFO- or RR-classed process will preempt a running normal-classed process.

The scheduler uses the nice value, discussed earlier, to prioritize processes within the normal class. The nice value has no bearing on the static priority, which remains 0.

The batch scheduling policy

`SCHED_BATCH` is the *batch* or *idle scheduling policy*. Its behavior is somewhat the antithesis of the real-time policies: processes in this class run only when there are no other runnable processes on the system, even if the other processes have exhausted their timeslices. This is different from the behavior of processes with the largest nice values (i.e., the lowest-priority processes) in that eventually such processes will run, as the higher-priority processes exhaust their timeslices.

Setting the Linux scheduling policy

Processes can manipulate the Linux scheduling policy via `sched_getscheduler()` and `sched_setscheduler()`:

```
#include <sched.h>

struct sched_param {
        /* ... */
        int sched_priority;
        /* ... */
};

int sched_getscheduler (pid_t pid);

int sched_setscheduler (pid_t pid,
```

```
                int policy,
                const struct sched_param *sp);
```

A successful call to `sched_getscheduler()` returns the scheduling policy of the process represented by `pid`. If `pid` is `0`, the call returns the invoking process's scheduling policy. An integer defined in `<sched.h>` represents the scheduling policy: the first in, first out policy is `SCHED_FIFO`; the round-robin policy is `SCHED_RR`; and the normal policy is `SCHED_OTHER`. On error, the call returns `-1` (which is never a valid scheduling policy), and `errno` is set as appropriate.

Usage is simple:

```
int policy;

/* get our scheduling policy */
policy = sched_getscheduler (0);

switch (policy) {
case SCHED_OTHER:
        printf ("Policy is normal\n");
        break;
case SCHED_RR:
        printf ("Policy is round-robin\n");
        break;
case SCHED_FIFO:
        printf ("Policy is first-in, first-out\n");
        break;
case -1:
        perror ("sched_getscheduler");
        break;
default:
        fprintf (stderr, "Unknown policy!\n");
}
```

A call to `sched_setscheduler()` sets the scheduling policy of the process represented by `pid` to `policy`. Any parameters associated with the policy are set via `sp`. If `pid` is `0`, the invoking process's policy and parameters are set. On success, the call returns `0`. On failure, the call returns `-1`, and `errno` is set as appropriate.

The valid fields inside the `sched_param` structure depend on the scheduling policies supported by the operating system. The `SCHED_RR` and `SCHED_FIFO` policies require one field, `sched_priority`, which represents the static priority. `SCHED_OTHER` does not use any field, while scheduling policies supported in the future may use new fields. Portable and legal programs must therefore not make assumptions about the layout of the structure.

Setting a process's scheduling policy and parameters is easy:

```
struct sched_param sp = { .sched_priority = 1 };
int ret;

ret = sched_setscheduler (0, SCHED_RR, &sp);
if (ret == -1) {
        perror ("sched_setscheduler");
        return 1;
}
```

This snippet sets the invoking process's scheduling policy to round-robin with a static priority of 1. We presume that 1 is a valid priority—technically, it need not be. We will discuss how to find the valid priority range for a given policy in an upcoming section.

Setting a scheduling policy other than `SCHED_OTHER` requires the `CAP_SYS_NICE` capability. Consequently, the root user typically runs real-time processes. Since the 2.6.12 kernel, the `RLIMIT_RTPRIO` resource limit allows nonroot users to set real-time policies up to a certain priority ceiling.

On error, four `errno` values are possible:

`EFAULT`
: The pointer `sp` points to an invalid or inaccessible region of memory.

`EINVAL`
: The scheduling policy denoted by `policy` is invalid, or a value set in `sp` does not make sense for the given policy (`sched_setscheduler()` only).

`EPERM`
: The invoking process does not have the necessary capabilities.

`ESRCH`
: The value `pid` does not denote a running process.

Setting Scheduling Parameters

The POSIX-defined `sched_getparam()` and `sched_setparam()` interfaces retrieve and set the parameters associated with a scheduling policy that has already been set:

```
#include <sched.h>

struct sched_param {
        /* ... */
        int sched_priority;
        /* ... */
};

int sched_getparam (pid_t pid, struct sched_param *sp);

int sched_setparam (pid_t pid, const struct sched_param *sp);
```

The `sched_getscheduler()` interface returns only the scheduling policy, not any associated parameters. A call to `sched_getparam()` returns via `sp` the scheduling parameters associated with `pid`:

```
struct sched_param sp;
int ret;

ret = sched_getparam (0, &sp);
if (ret == -1) {
        perror ("sched_getparam");
        return 1;
}

printf ("Our priority is %d\n", sp.sched_priority);
```

If `pid` is `0`, the call returns the parameters of the invoking process. On success, the call returns `0`. On failure, it returns `-1` and sets `errno` as appropriate.

Because `sched_setscheduler()` also sets any associated scheduling parameters, `sched_setparam()` is useful only to later modify the parameters:

```
struct sched_param sp;
int ret;

sp.sched_priority = 1;
ret = sched_setparam (0, &sp);
if (ret == -1) {
        perror ("sched_setparam");
        return 1;
}
```

On success, the scheduling parameters of `pid` are set according to `sp`, and the call returns `0`. On failure, the call returns `-1`, and `errno` is set as appropriate.

If we ran the two preceding snippets in reverse order, we would see the following output:

```
Our priority is 1
```

This example again assumes that 1 is a valid priority. It is, but portable applications should make sure. We'll look at how to check the range of valid priorities momentarily.

Error codes

On error, four `errno` values are possible:

`EFAULT`
: The pointer `sp` points to an invalid or inaccessible region of memory.

`EINVAL`
: A value set in `sp` does not make sense for the given policy (`sched_getparam()` only).

EPERM
: The invoking process does not have the necessary capabilities.

ESRCH
: The value `pid` does not denote a running process.

Determining the range of valid priorities

Our previous examples have passed hardcoded priority values into the scheduling system calls. POSIX makes no guarantees about what scheduling priorities exist on a given system, except to say that there must be at least 32 priorities between the minimum and maximum values. As mentioned earlier in "Linux Scheduling Policies and Priorities" on page 192, Linux implements a range of 1 to 99 inclusive for the two real-time scheduling policies. A clean, portable program normally implements its own range of priority values and maps them onto the operating system's range. For instance, if you want to run processes at four different real-time priority levels, you dynamically determine the range of priorities and choose four values.

Linux provides two system calls for retrieving the range of valid priority values. One returns the minimum value and the other returns the maximum:

```
#include <sched.h>

int sched_get_priority_min (int policy);

int sched_get_priority_max (int policy);
```

On success, the call `sched_get_priority_min()` returns the minimum, and the call `sched_get_priority_max()` returns the maximum valid priority associated with the scheduling policy denoted by `policy`. Upon failure, the calls both return `-1`. The only possible error is if `policy` is invalid, in which case `errno` is set to `EINVAL`.

Usage is simple:

```
int min, max;

min = sched_get_priority_min (SCHED_RR);
if (min == -1) {
        perror ("sched_get_priority_min");
        return 1;
}

max = sched_get_priority_max (SCHED_RR);
if (max == -1) {
        perror ("sched_get_priority_max");
        return 1;
}

printf ("SCHED_RR priority range is %d - %d\n", min, max);
```

On a standard Linux system, this snippet yields the following:

```
SCHED_RR priority range is 1 - 99
```

As discussed previously, numerically larger priority values denote higher priorities. To set a process to the highest priority for its scheduling policy, you can do the following:

```
/*
 * set_highest_priority - set the associated pid's scheduling
 * priority to the highest value allowed by its current
 * scheduling policy. If pid is zero, sets the current
 * process's priority.
 *
 * Returns zero on success.
 */
int set_highest_priority (pid_t pid)
{
        struct sched_param sp;
        int policy, max, ret;

        policy = sched_getscheduler (pid);
        if (policy == -1)
                return -1;

        max = sched_get_priority_max (policy);
        if (max == -1)
                return -1;

        memset (&sp, 0, sizeof (struct sched_param));
        sp.sched_priority = max;
        ret = sched_setparam (pid, &sp);

        return ret;
}
```

Programs typically retrieve the system's minimum or maximum value and then use increments of 1 (such as `max-1`, `max-2`, etc.) to assign priorities as desired.

sched_rr_get_interval()

As discussed earlier, `SCHED_RR` processes behave the same as `SCHED_FIFO` processes, except that the scheduler assigns these processes timeslices. When a `SCHED_RR` process exhausts its timeslice, the scheduler moves the process to the end of the run list for its current priority. In this manner, all `SCHED_RR` processes of the same priority are executed in a round-robin rotation. Higher-priority processes (and `SCHED_FIFO` processes of the same or higher priority) will always preempt a running `SCHED_RR` process, regardless of whether it has any of its timeslice remaining.

POSIX defines an interface for retrieving the length of a given process's timeslice:

```
#include <sched.h>

struct timespec {
        time_t  tv_sec;     /* seconds */
        long    tv_nsec;    /* nanoseconds */
};

int sched_rr_get_interval (pid_t pid, struct timespec *tp);
```

A successful call to the awfully named `sched_rr_get_interval()` saves in the `timespec` structure pointed at by `tp` the duration of the timeslice allotted to `pid` and returns `0`. On failure, the call returns `-1`, and `errno` is set as appropriate.

According to POSIX, this function is required to work only with `SCHED_RR` processes. On Linux, however, it can retrieve the length of any process's timeslice. Portable applications should assume that the function works only with round-robin processes; Linux-specific programs may exploit the call as needed. Here's an example:

```
struct timespec tp;
int ret;

/* get the current task's timeslice length */
ret = sched_rr_get_interval (0, &tp);
if (ret == -1) {
        perror ("sched_rr_get_interval");
        return 1;
}

/* convert the seconds and nanoseconds to milliseconds */
printf ("Our time quantum is %.2lf milliseconds\n",
        (tp.tv_sec * 1000.0f) + (tp.tv_nsec / 1000000.0f));
```

If the process is running in the FIFO class, `tv_sec` and `tv_nsec` are both `0`, denoting infinity.

Error codes

On error, three `errno` values are possible:

`EFAULT`
: The memory pointed at by the pointer `tp` is invalid or inaccessible.

`EINVAL`
: The value `pid` is invalid (for example, it is negative).

`ESRCH`
: The value `pid` is valid but refers to a nonexistent process.

Precautions with Real-Time Processes

Because of the nature of real-time processes, developers should exercise caution when developing and debugging such programs. If a real-time program goes off the deep end, the system can become unresponsive. Any CPU-bound loop in a real-time program—that is, any chunk of code that does not block—will continue running ad infinitum so long as no higher-priority real-time processes become runnable.

Consequently, designing real-time programs requires care and attention. Such programs reign supreme and can easily bring down the entire system. Some tips and precautions:

- Keep in mind that any CPU-bound loop will run until completion, without interruption, if there is no higher-priority real-time process on the system. If the loop is infinite, the system will become unresponsive.
- Because real-time processes run at the expense of everything else on the system, special attention must be paid to their design. Take care not to starve the rest of the system of processor time.
- Be very careful with busy waiting. If a real-time process busy-waits for a resource held by a lower-priority process, the real-time process will busy-wait forever.
- While developing a real-time process, keep a terminal open, running as a real-time process with a higher priority than the process in development. In an emergency, the terminal will remain responsive and allow you to kill the runaway real-time process. (As the terminal remains idle, waiting for keyboard input, it will not interfere with the other real-time process until you need it.)
- The *chrt* utility, part of the *util-linux* package of tools, makes it easy to retrieve and set the real-time attributes of other processes. This tool makes it easy to launch arbitrary programs in a real-time scheduling class, such as the aforementioned terminal, or change the real-time priorities of existing applications.

Determinism

Real-time processes are big on determinism. In real-time computing, an action is *deterministic* if, given the same input, it always produces the same result in the same amount of time. Modern computers are the very definition of something that is not deterministic: multiple levels of caches (which incur hits or misses without predictability), multiple processors, paging, swapping, and multitasking wreak havoc on any estimate of how long a given action will take. Sure, we have reached a point where just about every action (modulo hard drive access) is "incredibly fast," but simultaneously, modern systems have also made it hard to pinpoint exactly how long a given operation will take.

Real-time applications often try to limit unpredictability in general and worst-case delays specifically. The following sections discuss two methods that are used to this end.

Prefaulting data and locking memory

Picture this: the hardware interrupt from the custom incoming intercontinental ballistic missile (ICBM) monitor hits, and the device's driver quickly copies data from the hardware into the kernel. The driver notes that a process is asleep, blocked on the hardware's device node, waiting for data. The driver tells the kernel to wake up the process. The kernel, noting that this process is running with a real-time scheduling policy and a high priority, immediately preempts the currently running process and shifts into overdrive, determined to schedule the real-time process immediately. The scheduler switches to running the real-time process, and context-switches into its address space. The process is now running. The whole ordeal took 0.3 ms, well within the 1 ms worst-case acceptable latency period.

Now, in user space, the real-time process notes the incoming ICBM and begins processing its trajectory. With the ballistics calculated, the real-time process initiates the deployment of an anti-ballistic missile system. Only another 0.1 ms have passed—quick enough to deploy the anti-ballistic missile (ABM) response and save lives. But—oh no! —the ABM code has been swapped to disk. A page fault occurs, the processor switches back to kernel mode, and the kernel initiates hard disk I/O to retrieve the swapped-out data. The scheduler puts the process to sleep until the page fault is serviced. Several seconds elapse. It is too late.

Clearly, paging and swapping introduce quite undeterministic behavior that can wreak havoc on a real-time process. To prevent this catastrophe, a real-time application will often "lock" or "hardwire" all of the pages in its address space into physical memory, prefaulting them into memory and preventing them from being swapped out. Once the pages are locked into memory, the kernel will never swap them out to disk. Any accesses of the pages will not cause page faults. Most real-time applications lock some or all of their pages into physical memory.

Linux provides interfaces for both prefaulting and locking data. Chapter 4 discussed interfaces for prefaulting data into physical memory. Chapter 9 will discuss interfaces for locking data into physical memory.

CPU affinity and real-time processes

A second concern of real-time applications is multitasking. Although the Linux kernel is preemptive, its scheduler is not always able to immediately reschedule one process in favor of another. Sometimes, the currently running process is executing inside of a critical region in the kernel, and the scheduler cannot preempt it until it exits that region. If the process that is waiting to run is real-time, this delay may be unacceptable, quickly overrunning the operational deadline.

Ergo, multitasking introduces indeterminism similar in nature to the unpredictability associated with paging. The solution with respect to multitasking is the same: eliminate it. Of course, chances are you cannot simply abolish all other processes. If that were possible in your environment, you probably would not need Linux to begin with—a simple custom operating system would suffice. If, however, your system has multiple processors, you can dedicate one or more of those processors to your real-time process or processes. In effect, you can shield the real-time processes from multitasking.

We discussed system calls for manipulating a process's CPU affinity earlier in this chapter. A potential optimization for real-time applications is to reserve one processor for each real-time process and let all other processes time-share on the remaining processor(s).

The simplest way to achieve this is to modify Linux's *init* program, *SysVinit*,[2] to do something similar to the following before it begins the boot process:

```
cpu_set_t set;
int ret;

CPU_ZERO (&set);        /* clear all CPUs */
ret = sched_getaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
        perror ("sched_getaffinity");
        return 1;
}

CPU_CLR (1, &set);      /* forbid CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
        perror ("sched_setaffinity");
        return 1;
}
```

This snippet grabs init's current set of allowed processors, which we expect is all of them. It then removes one processor, CPU #1, from the set and updates the list of allowed processors.

Because the set of allowed processors is inherited from parent to child, and init is the super-parent of all processes, all of the system's processes will run with this set of allowed processors. Consequently, no processes will ever run on CPU #1.

Next, modify your real-time process to run only on CPU #1:

```
cpu_set_t set;
int ret;
```

2. The *SysVinit* source is located at *http://freecode.com/projects/sysvinit*. It is licensed under the GNU General Public License v2.

```
CPU_ZERO (&set);        /* clear all CPUs */
CPU_SET (1, &set);      /* allow CPU #1 */
ret = sched_setaffinity (0, sizeof (cpu_set_t), &set);
if (ret == -1) {
        perror ("sched_setaffinity");
        return 1;
}
```

The result is that your real-time process runs only on CPU #1 and all other processes run on the other processors.

Resource Limits

The Linux kernel imposes several *resource limits* on processes. These resource limits place hard ceilings on the amount of kernel resources that a process can consume—that is, the number of open files, pages of memory, pending signals, and so on. The limits are strictly enforced; the kernel will not allow an action that places a process's resource consumption over a hard limit. For example, if opening a file would cause a process to have more open files than allowed by the applicable resource limit, the `open()` invocation will fail.[3]

Linux provides two system calls for manipulating resource limits. POSIX standardized both interfaces, but Linux supports several limits in addition to those dictated by the standard. Limits can be checked with `getrlimit()` and set with `setrlimit()`:

```
#include <sys/time.h>
#include <sys/resource.h>

struct rlimit {
        rlim_t rlim_cur;  /* soft limit */
        rlim_t rlim_max;  /* hard limit */
};

int getrlimit (int resource, struct rlimit *rlim);
int setrlimit (int resource, const struct rlimit *rlim);
```

Integer constants, such as `RLIMIT_CPU`, represent the resources. The `rlimit` structure represents the actual limits. The structure defines two ceilings: a *soft limit* and a *hard limit*. The kernel enforces soft resource limits on processes, but a process may freely change its soft limit to any value from `0` up to and including the hard limit. A process without the `CAP_SYS_RESOURCE` capability (i.e., any nonroot process) can only lower its hard limit. An unprivileged process can never raise its hard limit, not even to a previously higher value; lowering the hard limit is irreversible. A privileged process can set the hard limit to any valid value.

3. In which case the call will set `errno` to `EMFILE`, indicating that the process hit the resource limit on the maximum number of open files. Chapter 2 discusses the `open()` system call.

What the limits actually represent depends on the resource in question. If `resource` is `RLIMIT_FSIZE`, for example, the limit represents the maximum size of a file that a process can create, in bytes. In this case, if `rlim_cur` is 1,024, a process cannot create or extend a file to a size greater than one kilobyte.

All of the resource limits have two special values: `0` and infinity. The former disables use of the resource altogether. For example, if `RLIMIT_CORE` is `0`, the kernel will never create a core file. Conversely, the latter removes any limit on the resource. The kernel denotes infinity by the special value `RLIM_INFINITY`, which happens to be `-1` (this can cause some confusion, as `-1` is also the return value indicating error). If `RLIMIT_CORE` is infinity, the kernel will create core files of any size.

The function `getrlimit()` places the current hard and soft limits on the resource denoted by `resource` in the structure pointed at by `rlim`. On success, the call returns `0`. On failure, the call returns `-1` and sets `errno` as appropriate.

Correspondingly, the function `setrlimit()` sets the hard and soft limits associated with `resource` to the values pointed at by `rlim`. On success, the call returns `0`, and the kernel updates the resource limits as requested. On failure, the call returns `-1` and sets `errno` as appropriate.

The Limits

Linux currently provides 16 resource limits:

`RLIMIT_AS`
: Limits the maximum size of a process's address space, in bytes. Attempts to increase the size of the address space past this limit (via calls such as `mmap()` and `brk()`) will fail and return `ENOMEM`. If the process's stack, which automatically grows as needed, expands beyond this limit, the kernel sends the process the `SIGSEGV` signal. This limit is usually `RLIM_INFINITY`.

`RLIMIT_CORE`
: Dictates the maximum size of core files, in bytes. If nonzero, core files larger than this limit are truncated to the maximum size. If `0`, core files are never created.

`RLIMIT_CPU`
: Dictates the maximum CPU time that a process can consume, in seconds. If a process runs for longer than this limit, the kernel sends it a `SIGXCPU` signal, which processes may catch and handle. Portable programs should terminate on receipt of this signal, as POSIX leaves undefined what action the kernel may take next. Some systems may terminate the process if it continues to run. Linux, however, allows the process to continue executing and continues sending `SIGXCPU` signals at one second intervals. Once the process reaches the hard limit, it is sent a `SIGKILL` and terminated.

`RLIMIT_DATA`
: Controls the maximum size of a process's data segment and heap, in bytes. Attempts to enlarge the data segment beyond this limit via `brk()` will fail and return `ENOMEM`.

`RLIMIT_FSIZE`
: Specifies the maximum file size that a process may create, in bytes. If a process expands a file beyond this size, the kernel sends the process a `SIGXFSZ` signal. By default, this signal terminates the process. A process may, however, elect to catch and handle this signal, in which case the offending system call fails and returns `EFBIG`.

`RLIMIT_LOCKS`
: Controls the maximum number of file locks that a process may hold (see Chapter 8 for a discussion of file locks). Once this limit is reached, further attempts to acquire additional file locks should fail and return `ENOLCK`. Linux kernel 2.4.25, however, removed this functionality. In current kernels, this limit is settable, but has no effect.

`RLIMIT_MEMLOCK`
: Specifies the maximum number of bytes of memory that a process without the `CAP_SYS_IPC` capability (effectively, a nonroot process) can lock into memory via `mlock()`, `mlockall()`, or `shmctl()`. If this limit is exceeded, these calls fail, and return `EPERM`. In practice, the effective limit is rounded down to an integer multiple of pages. Processes possessing `CAP_SYS_IPC` can lock any number of pages into memory, and this limit has no effect. Before kernel 2.6.9, this limit was the maximum that a process with `CAP_SYS_IPC` could lock into memory, and unprivileged processes could not lock any pages whatsoever. This limit is not part of POSIX; BSD introduced it.

`RLIMIT_MSGQUEUE`
: Specifies the maximum number of bytes that a user may allocate for POSIX message queues. If a newly created message queue would exceed this limit, `mq_open()` fails and returns `ENOMEM`. This limit is not part of POSIX; it was added in kernel 2.6.8 and is Linux-specific.

`RLIMIT_NICE`
: Specifies the maximum value to which a process can lower its nice value (raise its priority). As discussed earlier in this chapter, normally processes can only raise their nice values (lower their priorities). This limit allows the administrator to impose a maximum level (nice value floor) to which processes may legally raise their priorities. Because nice values may be negative, the kernel interprets the value as `20 - rlim_cur`. Thus, if this limit is set to 40, a process can lower its nice value to the minimum value of –20 (the highest priority). Kernel 2.6.12 introduced this limit.

`RLIMIT_NOFILE`
: Specifies one greater than the maximum number of file descriptors that a process may hold open. Attempts to surpass this limit result in failure and the applicable system call returning `EMFILE`. This limit is also specifiable as `RLIMIT_OFILE`, which is BSD's name for it.

`RLIMIT_NPROC`
: Specifies the maximum number of processes that the user may have running on the system at any given moment. Attempts to surpass this limit result in failure and `fork()` will return `EAGAIN`. This limit is not part of POSIX; it was introduced by BSD.

`RLIMIT_RSS`
: Specifies the maximum number of pages that a process may have resident in memory (known as the resident set size, or RSS). Only early 2.4 kernels enforced this limit. Current kernels allow the setting of this limit, but it is not enforced. This limit is not part of POSIX; BSD introduced it.

`RLIMIT_RTTIME`
: Specifies a limit (in microseconds) on CPU time that a real-time process may consume without issuing a blocking system call. Once the process makes a blocking system call, the CPU time is reset to zero. This prevents a runaway real-time process from taking down the system. It was added in Linux kernel version 2.6.25 and is Linux-specific.

`RLIMIT_RTPRIO`
: Specifies the maximum real-time priority level a process without the `CAP_SYS_NICE` capability (effectively, nonroot processes) may request. Normally, unprivileged processes may not request any real-time scheduling class. It was added in kernel 2.6.12 and is Linux-specific.

`RLIMIT_SIGPENDING`
: Specifies the maximum number of signals (standard and real-time) that may be queued for this user. Attempts to queue additional signals fail, and system calls such as `sigqueue()` return `EAGAIN`. Note that it is always possible, regardless of this limit, to queue one instance of a not-yet-queued signal. Therefore, it is always possible to deliver to the process a `SIGKILL` or `SIGTERM`. This limit is not part of POSIX; it is Linux-specific.

`RLIMIT_STACK`
: Denotes the maximum size of a process's stack, in bytes. Surpassing this limit results in the delivery of a `SIGSEGV`.

The kernel manages the resource limits on a per-process basis. A child process inherits its limits from its parent during fork; limits are maintained across exec.

Default limits

The default limits available to your process depend on three variables: the initial soft limit, the initial hard limit, and your system administrator. The kernel dictates the initial hard and soft limits; Table 6-1 lists them. The kernel sets these limits on the init process, and because children inherit the limits of their parents, all subsequent processes inherit the soft and hard limits of init.

Table 6-1. Default soft and hard resource limits

Resource limit	Soft limit	Hard limit
RLIMIT_AS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_CORE	0	RLIM_INFINITY
RLIMIT_CPU	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_DATA	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_FSIZE	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_LOCKS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_MEMLOCK	8 pages	8 pages
RLIMIT_MSGQUEUE	800 KB	800 KB
RLIMIT_NICE	0	0
RLIMIT_NOFILE	1024	1024
RLIMIT_NPROC	0 (implies no limit)	0 (implies no limit)
RLIMIT_RSS	RLIM_INFINITY	RLIM_INFINITY
RLIMIT_RTPRIO	0	0
RLIMIT_SIGPENDING	0	0
RLIMIT_STACK	8 MB	RLIM_INFINITY

Two things can change these default limits:

- Any process is free to increase a soft limit to any value from 0 to the hard limit, or to decrease a hard limit. Children will inherit these updated limits during a fork.
- A privileged process is free to set a hard limit to any value. Children will inherit these updated limits during a fork.

It is unlikely that a root process in a regular process's lineage will change any hard limits. Consequently, the first item is a much more likely source of limit changes than the second. Indeed, the actual limits presented to a process are generally set by the user's shell, which the system administrator can set up to provide various limits. In the Bourne-again shell (*bash*), for example, the administrator accomplishes this via the *ulimit* command. Note that the administrator need not lower values; he can also raise soft limits to the hard limits, providing users with saner defaults. This is often done with RLIMIT_STACK, which is set to RLIM_INFINITY on many systems.

Setting and Retrieving Limits

With the explanations of the various resource limits behind us, let's look at retrieving and setting limits. Retrieving a resource limit is quite simple:

```
struct rlimit rlim;
int ret;

/* get the limit on core sizes */
ret = getrlimit (RLIMIT_CORE, &rlim);
if (ret == -1) {
        perror ("getrlimit");
        return 1;
}

printf ("RLIMIT_CORE limits: soft=%ld hard=%ld\n",
        rlim.rlim_cur, rlim.rlim_max);
```

Compiling this snippet in a larger program and running it yields the following:

```
RLIMIT_CORE limits: soft=0 hard=-1
```

We have a soft limit of `0` and a hard limit of infinity (`-1` denotes `RLIM_INFINITY`). Therefore, we can set a new soft limit of any size. This example sets the maximum core size to 32 MB:

```
struct rlimit rlim;
int ret;

rlim.rlim_cur = 32 * 1024 * 1024; /* 32 MB */
rlim.rlim_max = RLIM_INFINITY;    /* leave it alone */
ret = setrlimit (RLIMIT_CORE, &rlim);
if (ret == -1) {
        perror ("setrlimit");
        return 1;
}
```

Error codes

On error, three `errno` codes are possible:

`EFAULT`
: The memory pointed at by `rlim` is invalid or inaccessible.

`EINVAL`
: The value denoted by `resource` is invalid, or `rlim.rlim_cur` is greater than `rlim.rlim_max` (`setrlimit()` only).

`EPERM`
: The caller did not possess `CAP_SYS_RESOURCE` but tried to raise the hard limit.