4

Process Scheduling

The prior chapter discussed *processes*, the operating system abstraction of active program code. This chapter discusses the *process scheduler*, the kernel subsystem that puts those processes to work.

The process scheduler decides which process runs, when, and for how long. The process scheduler (or simply the *scheduler*, to which it is often shortened) divides the finite resource of processor time between the runnable processes on a system. The scheduler is the basis of a multitasking operating system such as Linux. By deciding which process runs next, the scheduler is responsible for best utilizing the system and giving users the impression that multiple processes are executing simultaneously.

The idea behind the scheduler is simple. To best utilize processor time, assuming there are *runnable* processes, a process should always be running. If there are more runnable processes than processors in a system, some processes will not be running at a given moment. These processes are *waiting to run*. Deciding which process runs next, given a set of runnable processes, is the fundamental decision that the scheduler must make.

Multitasking

A *multitasking* operating system is one that can simultaneously interleave execution of more than one process. On single processor machines, this gives the illusion of multiple processes running concurrently. On multiprocessor machines, such functionality enables processes to actually run concurrently, in parallel, on different processors. On either type of machine, it also enables many processes to *block* or *sleep*, not actually executing until work is available. These processes, although in memory, are not *runnable*. Instead, such processes utilize the kernel to wait until some event (keyboard input, network data, passage of time, and so on) occurs. Consequently, a modern Linux system can have many processes in memory but, say, only one in a runnable state.

Multitasking operating systems come in two flavors: *cooperative multitasking* and *preemptive multitasking*. Linux, like all Unix variants and most modern operating systems, implements preemptive multitasking. In preemptive multitasking, the scheduler decides when a process is to cease running and a new process is to begin running. The act of

involuntarily suspending a running process is called *preemption*. The time a process runs before it is preempted is usually predetermined, and it is called the *timeslice* of the process. The timeslice, in effect, gives each runnable process a *slice* of the processor's time. Managing the timeslice enables the scheduler to make global scheduling decisions for the system. It also prevents any one process from monopolizing the processor. On many modern operating systems, the timeslice is dynamically calculated as a function of process behavior and configurable system policy. As we shall see, Linux's unique "fair" scheduler does not employ timeslices *per se*, to interesting effect.

Conversely, in *cooperative multitasking,* a process does not stop running until it voluntary decides to do so. The act of a process voluntarily suspending itself is called *yielding*. Ideally, processes yield often, giving each runnable process a decent chunk of the processor, but the operating system cannot enforce this. The shortcomings of this approach are manifest: The scheduler cannot make global decisions regarding how long processes run; processes can monopolize the processor for longer than the user desires; and a hung process that never yields can potentially bring down the entire system. Thankfully, most operating systems designed in the last two decades employ preemptive multitasking, with Mac OS 9 (and earlier) and Windows 3.1 (and earlier) being the most notable (and embarrassing) exceptions. Of course, Unix has sported preemptive multitasking since its inception.

Linux's Process Scheduler

From Linux's first version in 1991 through the 2.4 kernel series, the Linux scheduler was simple, almost pedestrian, in design. It was easy to understand, but scaled poorly in light of many runnable processes or many processors.

In response, during the 2.5 kernel development series, the Linux kernel received a scheduler overhaul. A new scheduler, commonly called the *`O(1)` scheduler* because of its algorithmic behavior,[1] solved the shortcomings of the previous Linux scheduler and introduced powerful new features and performance characteristics. By introducing a constant-time algorithm for timeslice calculation and per-processor runqueues, it rectified the design limitations of the earlier scheduler.

The `O(1)` scheduler performed admirably and scaled effortlessly as Linux supported large "iron" with tens if not hundreds of processors. Over time, however, it became evident that the `O(1)` scheduler had several pathological failures related to scheduling latency-sensitive applications. These applications, called *interactive processes*, include any application with which the user interacts. Thus, although the `O(1)` scheduler was ideal for large server workloads—which lack interactive processes—it performed below par on desktop systems, where interactive applications are the *raison d'être*. Beginning early in the

[1] *`O(1)` is an example of big-o notation. In short, it means the scheduler can perform its work in constant time, regardless of the size of any inputs. A full explanation of big-o notation is in Chapter 6, "Kernel Data Structures."*

2.6 kernel series, developers introduced new process schedulers aimed at improving the interactive performance of the `O(1)` scheduler. The most notable of these was the *Rotating Staircase Deadline* scheduler, which introduced the concept of *fair scheduling*, borrowed from queuing theory, to Linux's process scheduler. This concept was the inspiration for the `O(1)` scheduler's eventual replacement in kernel version 2.6.23, the *Completely Fair Scheduler*, or *CFS*.

This chapter discusses the fundamentals of scheduler design and how they apply to the Completely Fair Scheduler and its goals, design, implementation, algorithms, and related system calls. We also discuss the `O(1)` scheduler because its implementation is a more "classic" Unix process scheduler model.

Policy

Policy is the behavior of the scheduler that determines what runs when. A scheduler's policy often determines the overall feel of a system and is responsible for optimally utilizing processor time. Therefore, it is very important.

I/O-Bound Versus Processor-Bound Processes

Processes can be classified as either *I/O-bound* or *processor-bound*. The former is characterized as a process that spends much of its time submitting and waiting on I/O requests. Consequently, such a process is runnable for only short durations, because it eventually blocks waiting on more I/O. (Here, by *I/O*, we mean any type of blockable resource, such as keyboard input or network I/O, and not just disk I/O.) Most graphical user interface (GUI) applications, for example, are I/O-bound, even if they never read from or write to the disk, because they spend most of their time waiting on user interaction via the keyboard and mouse.

Conversely, processor-bound processes spend much of their time executing code. They tend to run until they are preempted because they do not block on I/O requests very often. Because they are not I/O-driven, however, system response does not dictate that the scheduler run them often. A scheduler policy for processor-bound processes, therefore, tends to run such processes less frequently but for longer durations. The ultimate example of a processor-bound process is one executing an infinite loop. More palatable examples include programs that perform a lot of mathematical calculations, such as *ssh-keygen* or *MATLAB*.

Of course, these classifications are not mutually exclusive. Processes can exhibit both behaviors simultaneously: The X Window server, for example, is both processor and I/O-intense. Other processes can be I/O-bound but dive into periods of intense processor action. A good example of this is a word processor, which normally sits waiting for key presses but at any moment might peg the processor in a rabid fit of spell checking or macro calculation.

The scheduling policy in a system must attempt to satisfy two conflicting goals: fast process response time (low latency) and maximal system utilization (high throughput). To

satisfy these at-odds requirements, schedulers often employ complex algorithms to determine the most worthwhile process to run while not compromising fairness to other, lower priority, processes. The scheduler policy in Unix systems tends to explicitly favor I/O-bound processes, thus providing good process response time. Linux, aiming to provide good interactive response and desktop performance, optimizes for process response (low latency), thus favoring I/O-bound processes over processor-bound processors. As we will see, this is done in a creative manner that does not neglect processor-bound processes.

Process Priority

A common type of scheduling algorithm is *priority-based* scheduling. The goal is to rank processes based on their worth and need for processor time. The general idea, which isn't exactly implemented on Linux, is that processes with a higher priority run before those with a lower priority, whereas processes with the same priority are scheduled *round-robin* (one after the next, repeating). On some systems, processes with a higher priority also receive a longer timeslice. The runnable process with timeslice remaining and the highest priority always runs. Both the user and the system can set a process's priority to influence the scheduling behavior of the system.

The Linux kernel implements two separate priority ranges. The first is the *nice* value, a number from –20 to +19 with a default of 0. Larger nice values correspond to a lower priority—you are being "nice" to the other processes on the system. Processes with a lower nice value (higher priority) receive a larger proportion of the system's processor compared to processes with a higher nice value (lower priority). Nice values are the standard priority range used in all Unix systems, although different Unix systems apply them in different ways, reflective of their individual scheduling algorithms. In other Unix-based systems, such as Mac OS X, the nice value is a control over the *absolute* timeslice allotted to a process; in Linux, it is a control over the *proportion* of timeslice. You can see a list of the processes on your system and their respective nice values (under the column marked *NI*) with the command `ps -el`.

The second range is the *real-time priority*. The values are configurable, but by default range from 0 to 99, inclusive. Opposite from nice values, higher real-time priority values correspond to a greater priority. All real-time processes are at a higher priority than normal processes; that is, the real-time priority and nice value are in disjoint value spaces. Linux implements real-time priorities in accordance with the relevant Unix standards, specifically POSIX.1b. All modern Unix systems implement a similar scheme. You can see a list of the processes on your system and their respective real-time priority (under the column marked *RTPRIO*) with the command

```
ps -eo state,uid,pid,ppid,rtprio,time,comm.
```

A value of "-" means the process is not real-time.

Timeslice

The timeslice[2] is the numeric value that represents how long a task can run until it is preempted. The scheduler policy must dictate a default timeslice, which is not a trivial exercise. Too long a timeslice causes the system to have poor interactive performance; the system will no longer feel as if applications are concurrently executed. Too short a timeslice causes significant amounts of processor time to be wasted on the overhead of switching processes because a significant percentage of the system's time is spent switching from one process with a short timeslice to the next. Furthermore, the conflicting goals of I/O-bound versus processor-bound processes again arise: I/O-bound processes do not need longer timeslices (although they do like to run often), whereas processor-bound processes crave long timeslices (to keep their caches hot).

With this argument, it would seem that *any* long timeslice would result in poor interactive performance. In many operating systems, this observation is taken to heart, and the default timeslice is rather low—for example, 10 milliseconds. Linux's CFS scheduler, however, does not directly assign timeslices to processes. Instead, in a novel approach, CFS assigns processes a *proportion* of the processor. On Linux, therefore, the amount of processor time that a process receives is a function of the load of the system. This assigned proportion is further affected by each process's nice value. The nice value acts as a weight, changing the proportion of the processor time each process receives. Processes with higher nice values (a lower priority) receive a deflationary weight, yielding them a smaller proportion of the processor; processes with smaller nice values (a higher priority) receive an inflationary weight, netting them a larger proportion of the processor.

As mentioned, the Linux operating system is *preemptive*. When a process enters the runnable state, it becomes eligible to run. In most operating systems, whether the process runs immediately, preempting the currently running process, is a function of the process's priority and available timeslice. In Linux, under the new CFS scheduler, the decision is a function of how much of a proportion of the processor the newly runnable processor has consumed. If it has consumed a smaller proportion of the processor than the currently executing process, it runs immediately, preempting the current process. If not, it is scheduled to run at a later time.

The Scheduling Policy in Action

Consider a system with two runnable tasks: a text editor and a video encoder. The text editor is I/O-bound because it spends nearly all its time waiting for user key presses. (No matter how fast the user types, it is not *that* fast.) Despite this, when the text editor does receive a key press, the user expects the editor to respond immediately. Conversely, the video encoder is processor-bound. Aside from reading the raw data stream from the disk

---

[2] *Timeslice is sometimes called* quantum *or* processor slice *in other systems. Linux calls it timeslice, thus so should you.*

and later writing the resulting video, the encoder spends all its time applying the video codec to the raw data, easily consuming 100% of the processor. The video encoder does not have any strong time constraints on when it runs—if it started running now or in half a second, the user could not tell and would not care. Of course, the sooner it finishes the better, but latency is not a primary concern.

In this scenario, ideally the scheduler gives the text editor a larger proportion of the available processor than the video encoder, because the text editor is interactive. We have two goals for the text editor. First, we want it to have a large amount of processor time available to it; not because it needs a lot of processor (it does not) but because we want it to always have processor time available the moment it needs it. Second, we want the text editor to preempt the video encoder the moment it wakes up (say, when the user presses a key). This can ensure the text editor has good *interactive performance* and is responsive to user input. On most operating systems, these goals are accomplished (if at all) by giving the text editor a higher priority and larger timeslice than the video encoder. Advanced operating systems do this automatically, by detecting that the text editor is interactive. Linux achieves these goals too, but by different means. Instead of assigning the text editor a specific priority and timeslice, it guarantees the text editor a specific proportion of the processor. If the video encoder and text editor are the only running processes and both are at the same nice level, this proportion would be 50%—each would be guaranteed half of the processor's time. Because the text editor spends most of its time blocked, waiting for user key presses, it does not use anywhere near 50% of the processor. Conversely, the video encoder is free to use *more* than its allotted 50%, enabling it to finish the encoding quickly.

The crucial concept is what happens when the text editor wakes up. Our primary goal is to ensure it runs immediately upon user input. In this case, when the editor wakes up, CFS notes that it is allotted 50% of the processor but has used considerably less. Specifically, CFS determines that the text editor has run for *less time* than the video encoder. Attempting to give all processes a *fair share* of the processor, it then preempts the video encoder and enables the text editor to run. The text editor runs, quickly processes the user's key press, and again sleeps, waiting for more input. As the text editor has not consumed its allotted 50%, we continue in this manner, with CFS always enabling the text editor to run when it wants and the video encoder to run the rest of the time.

The Linux Scheduling Algorithm

In the previous sections, we discussed process scheduling in the abstract, with only occasional mention of how Linux applies a given concept to reality. With the foundation of scheduling now built, we can dive into Linux's process scheduler.

Scheduler Classes

The Linux scheduler is modular, enabling different algorithms to schedule different types of processes. This modularity is called *scheduler classes*. Scheduler classes enable different, pluggable algorithms to coexist, scheduling their own types of processes. Each scheduler

class has a priority. The base scheduler code, which is defined in `kernel/sched.c`, iterates over each scheduler class in order of priority. The highest priority scheduler class that has a runnable process wins, selecting who runs next.

The Completely Fair Scheduler (CFS) is the registered scheduler class for normal processes, called `SCHED_NORMAL` in Linux (and `SCHED_OTHER` in POSIX). CFS is defined in `kernel/sched_fair.c`. The rest of this section discusses the CFS algorithm and is germane to any Linux kernel since 2.6.23. We discuss the scheduler class for real-time processes in a later section.

Process Scheduling in Unix Systems

To discuss fair scheduling, we must first describe how traditional Unix systems schedule processes. As mentioned in the previous section, modern process schedulers have two common concepts: process priority and timeslice. Timeslice is how long a process runs; processes start with some default timeslice. Processes with a higher priority run more frequently and (on many systems) receive a higher timeslice. On Unix, the priority is exported to user-space in the form of nice values. This sounds simple, but in practice it leads to several pathological problems, which we now discuss.

First, mapping nice values onto timeslices requires a decision about what absolute timeslice to allot each nice value. This leads to suboptimal switching behavior. For example, let's assume we assign processes of the default nice value (zero) a timeslice of 100 milliseconds and processes at the highest nice value (+20, the lowest priority) a timeslice of 5 milliseconds. Further, let's assume one of each of these processes is runnable. Our default-priority process thus receives 20/21 (100 out of 105 milliseconds) of the processor, whereas our low priority process receives 1/21 (5 out of 105 milliseconds) of the processor. We could have used any numbers for this example, but we assume this allotment is optimal since we chose it. Now, what happens if we run exactly two low priority processes? We'd expect they each receive 50% of the processor, which they do. But they each enjoy the processor for only 5 milliseconds at a time (5 out of 10 milliseconds each)! That is, instead of context switching twice every 105 milliseconds, we now context switch twice every 10 milliseconds. Conversely, if we have two normal priority processes, each again receives the correct 50% of the processor, but in 100 millisecond increments. Neither of these timeslice allotments are necessarily ideal; each is simply a byproduct of a given nice value to timeslice mapping coupled with a specific runnable process priority mix. Indeed, given that high nice value (low priority) processes tend to be background, processor-intensive tasks, while normal priority processes tend to be foreground user tasks, this timeslice allotment is exactly *backward* from ideal!

A second problem concerns relative nice values and again the nice value to timeslice mapping. Say we have two processes, each a single nice value apart. First, let's assume they are at nice values 0 and 1. This might map (and indeed did in the `O(1)` scheduler) to timeslices of 100 and 95 milliseconds, respectively. These two values are nearly identical, and thus the difference here between a single nice value is small. Now, instead, let's assume our two processes are at nice values of 18 and 19. This now maps to timeslices of

10 and 5 milliseconds, respectively—the former receiving twice the processor time as the latter! Because nice values are most commonly used in relative terms (as the system call accepts an increment, not an absolute value), this behavior means that "nicing down a process by one" has wildly different effects depending on the starting nice value.

Third, if performing a nice value to timeslice mapping, we need the ability to assign an absolute timeslice. This absolute value must be measured in terms the kernel can measure. In most operating systems, this means the timeslice must be some integer multiple of the timer tick. (See Chapter 11, "Timers and Time Management," for a discussion on time.) This introduces several problems. First, the minimum timeslice has a floor of the period of the timer tick, which might be as high as 10 milliseconds or as low as 1 millisecond. Second, the system timer limits the difference between two timeslices; successive nice values might map to timeslices as much as 10 milliseconds or as little as 1 millisecond apart. Finally, timeslices change with different timer ticks. (If this paragraph's discussion of timer ticks is foreign, reread it after reading Chapter 11. This is only one motivation behind CFS.)

The fourth and final problem concerns handling process wake up in a priority-based scheduler that wants to optimize for interactive tasks. In such a system, you might want to give freshly woken-up tasks a priority boost by allowing them to run immediately, even if their timeslice was expired. Although this improves interactive performance in many, if not most, situations, it also opens the door to pathological cases where certain sleep/wake up use cases can game the scheduler into providing one process an unfair amount of processor time, at the expense of the rest of the system.

Most of these problems are solvable by making substantial but not paradigm-shifting changes to the old-school Unix scheduler. For example, making nice values geometric instead of additive solves the second problem. And mapping nice values to timeslices using a measurement decoupled from the timer tick solves the third problem. But such solutions belie the true problem, which is that assigning absolute timeslices yields a constant switching rate but variable fairness. The approach taken by CFS is a radical (for process schedulers) rethinking of timeslice allotment: Do away with timeslices completely and assign each process a proportion of the processor. CFS thus yields constant fairness but a variable switching rate.

Fair Scheduling

CFS is based on a simple concept: Model process scheduling as if the system had an ideal, perfectly multitasking processor. In such a system, each process would receive 1/n of the processor's time, where *n* is the number of runnable processes, and we'd schedule them for infinitely small durations, so that in any measurable period we'd have run all *n* processes for the same amount of time. As an example, assume we have two processes. In the standard Unix model, we might run one process for 5 milliseconds and then another process for 5 milliseconds. While running, each process would receive 100% of the processor. In an ideal, perfectly multitasking processor, we would run both processes *simultaneously* for 10 milliseconds, each at 50% power. This latter model is called *perfect multitasking*.

Of course, such a model is also impractical, because it is not possible on a single processor to *literally* run multiple processes simultaneously. Moreover, it is not efficient to run processes for infinitely small durations. That is, there is a *switching cost* to preempting one process for another: the overhead of swapping one process for another and the effects on caches, for example. Thus, although we'd like to run processes for very small durations, CFS is mindful of the overhead and performance hit in doing so. Instead, CFS will run each process for some amount of time, round-robin, selecting next the process that has run the least. Rather than assign each process a timeslice, CFS calculates how long a process should run as a function of the total number of runnable processes. Instead of using the nice value to calculate a timeslice, CFS uses the nice value to *weight* the proportion of processor a process is to receive: Higher valued (lower priority) processes receive a fractional weight relative to the default nice value, whereas lower valued (higher priority) processes receive a larger weight.

Each process then runs for a "timeslice" proportional to its weight divided by the total weight of all runnable threads. To calculate the actual timeslice, CFS sets a target for its approximation of the "infinitely small" scheduling duration in perfect multitasking. This target is called the *targeted latency*. Smaller targets yield better interactivity and a closer approximation to perfect multitasking, at the expense of higher switching costs and thus worse overall throughput. Let's assume the targeted latency is 20 milliseconds and we have two runnable tasks at the same priority. *Regardless of those task's priority*, each will run for 10 milliseconds before preempting in favor of the other. If we have four tasks at the same priority, each will run for 5 milliseconds. If there are 20 tasks, each will run for 1 millisecond.

Note as the number of runnable tasks approaches infinity, the proportion of allotted processor and the assigned timeslice approaches zero. As this will eventually result in unacceptable switching costs, CFS imposes a floor on the timeslice assigned to each process. This floor is called the *minimum granularity*. By default it is 1 millisecond. Thus, even as the number of runnable processes approaches infinity, each will run for at least 1 millisecond, to ensure there is a ceiling on the incurred switching costs. (Astute readers will note that CFS is thus not perfectly fair when the number of processes grows so large that the calculated proportion is floored by the minimum granularity. This is true. Although modifications to fair queuing exist to improve upon this fairness, CFS was explicitly designed to make this trade-off. In the common case of only a handful of runnable processes, CFS is perfectly fair.)

Now, let's again consider the case of two runnable processes, except with dissimilar nice values—say, one with the default nice value (zero) and one with a nice value of 5. These nice values have dissimilar weights and thus our two processes receive different proportions of the processor's time. In this case, the weights work out to about a 1⁄3 penalty for the nice-5 process. If our target latency is again 20 milliseconds, our two processes will receive 15 milliseconds and 5 milliseconds each of processor time, respectively. Say our two runnable processes instead had nice values of 10 and 15. What would be the allotted timeslices? Again 15 and 5 milliseconds each! Absolute nice values no

longer affect scheduling decisions; only relative values affect the proportion of processor time allotted.

Put generally, the proportion of processor time that any process receives is determined only by the relative difference in niceness between it and the other runnable processes. The nice values, instead of yielding additive increases to timeslices, yield geometric differences. The absolute timeslice allotted any nice value is not an absolute number, but a given proportion of the processor. CFS is called a *fair scheduler* because it gives each process a fair share—a proportion—of the processor's time. As mentioned, note that CFS isn't perfectly fair, because it only approximates perfect multitasking, but it *can* place a lower bound on latency of *n* for *n* runnable processes on the unfairness.

The Linux Scheduling Implementation

With the discussion of the motivation for and logic of CFS behind us, we can now explore CFS's actual implementation, which lives in `kernel/sched_fair.c`. Specifically, we discuss four components of CFS:

- Time Accounting
- Process Selection
- The Scheduler Entry Point
- Sleeping and Waking Up

Time Accounting

All process schedulers must account for the time that a process runs. Most Unix systems do so, as discussed earlier, by assigning each process a timeslice. On each tick of the system clock, the timeslice is decremented by the tick period. When the timeslice reaches zero, the process is preempted in favor of another runnable process with a nonzero timeslice.

The Scheduler Entity Structure

CFS does not have the notion of a timeslice, but it must still keep account for the time that each process runs, because it needs to ensure that each process runs only for its fair share of the processor. CFS uses the *scheduler entity structure*, `struct sched_entity`, defined in `<linux/sched.h>`, to keep track of process accounting:

```
struct sched_entity {
        struct load_weight      load;
        struct rb_node          run_node;
        struct list_head        group_node;
        unsigned int            on_rq;
        u64                     exec_start;
        u64                     sum_exec_runtime;
        u64                     vruntime;
        u64                     prev_sum_exec_runtime;
```