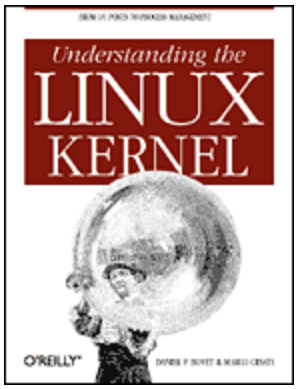

Understanding the Linux Kernel

By Daniel P. Bovet & Marco Cesati
October 2000
0-596-00002-2, Order Number: 0022
704 pages, $39.95

Chapter 10
Process Scheduling

Like any time-sharing system, Linux achieves the magical effect of an apparent simultaneous execution of multiple processes by switching from one process to another in a very short time frame. Process switch itself was discussed in *Chapter 3, Processes*; this chapter deals with *scheduling*, which is concerned with when to switch and which process to choose.

The chapter consists of three parts. The section "Scheduling Policy" introduces the choices made by Linux to schedule processes in the abstract. The section "The Scheduling Algorithm" discusses the data structures used to implement scheduling and the corresponding algorithm. Finally, the section "System Calls Related to Scheduling" describes the system calls that affect process scheduling.

Scheduling Policy

The scheduling algorithm of traditional Unix operating systems must fulfill several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low- and high-priority processes, and so on. The set of rules used to determine when and how selecting a new process to run is called *scheduling policy*.

Linux scheduling is based on the *time-sharing* technique already introduced in the section "CPU's Time Sharing" in *Chapter 5, Timing Measurements*: several processes are allowed to run "concurrently," which means that the CPU time is roughly divided into "slices," one

for each runnable process.[1] Of course, a single processor can run only one process at any given instant. If a currently running process is not terminated when its time slice or *quantum* expires, a process switch may take place. Time-sharing relies on timer interrupts and is thus transparent to processes. No additional code needs to be inserted in the programs in order to ensure CPU time-sharing.

The scheduling policy is also based on ranking processes according to their priority. Complicated algorithms are sometimes used to derive the current priority of a process, but the end result is the same: each process is associated with a value that denotes how appropriate it is to be assigned to the CPU.

In Linux, process priority is dynamic. The scheduler keeps track of what processes are doing and adjusts their priorities periodically; in this way, processes that have been denied the use of the CPU for a long time interval are boosted by dynamically increasing their priority. Correspondingly, processes running for a long time are penalized by decreasing their priority.

When speaking about scheduling, processes are traditionally classified as "I/O-bound" or "CPU-bound." The former make heavy use of I/O devices and spend much time waiting for I/O operations to complete; the latter are number-crunching applications that require a lot of CPU time.

An alternative classification distinguishes three classes of processes:

Interactive processes
: These interact constantly with their users, and therefore spend a lot of time waiting for keypresses and mouse operations. When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive. Typically, the average delay must fall between 50 and 150 ms. The variance of such delay must also be bounded, or the user will find the system to be erratic. Typical interactive programs are command shells, text editors, and graphical applications.

Batch processes
: These do not need user interaction, and hence they often run in the background. Since such processes do not need to be very responsive, they are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines, and scientific computations.

Real-time processes
: These have very strong scheduling requirements. Such processes should never be blocked by lower-priority processes, they should

have a short response time and, most important, such response time should have a minimum variance. Typical real-time programs are video and sound applications, robot controllers, and programs that collect data from physical sensors.

The two classifications we just offered are somewhat independent. For instance, a batch process can be either I/O-bound (e.g., a database server) or CPU-bound (e.g., an image-rendering program). While in Linux real-time programs are explicitly recognized as such by the scheduling algorithm, there is no way to distinguish between interactive and batch programs. In order to offer a good response time to interactive applications, Linux (like all Unix kernels) implicitly favors I/O-bound processes over CPU-bound ones.

Programmers may change the scheduling parameters by means of the system calls illustrated in Table 10-1. More details will be given in the section "System Calls Related to Scheduling."

Table 10-1: System Calls Related to Scheduling

System Call	Description
`nice( )`	Change the priority of a conventional process.
`getpriority( )`	Get the maximum priority of a group of conventional processes.
`setpriority( )`	Set the priority of a group of conventional processes.
`sched_getscheduler( )`	Get the scheduling policy of a process.
`sched_setscheduler( )`	Set the scheduling policy and priority of a process.
`sched_getparam( )`	Get the scheduling priority of a process.
`sched_setparam( )`	Set the priority of a process.
`sched_yield( )`	Relinquish the processor voluntarily without blocking.
`sched_get_ priority_min( )`	Get the minimum priority value for a policy.
`sched_get_ priority_max( )`	Get the maximum priority value for a policy.
`sched_rr_get_interval( )`	Get the time quantum value for the Round Robin policy.

Most system calls shown in the table apply to real-time processes, thus allowing users to develop real-time applications. However, Linux does not support the most demanding real-time applications because its kernel is nonpreemptive (see the later section "Performance of the

Scheduling Algorithm").

Process Preemption

As mentioned in the first chapter, Linux processes are *preemptive*. If a process enters the `TASK_RUNNING` state, the kernel checks whether its dynamic priority is greater than the priority of the currently running process. If it is, the execution of `current` is interrupted and the scheduler is invoked to select another process to run (usually the process that just became runnable). Of course, a process may also be preempted when its time quantum expires. As mentioned in the section "CPU's Time Sharing" in Chapter 5, when this occurs, the `need_resched` field of the current process is set, so the scheduler is invoked when the timer interrupt handler terminates.

For instance, let us consider a scenario in which only two programs--a text editor and a compiler--are being executed. The text editor is an interactive program, therefore it has a higher dynamic priority than the compiler. Nevertheless, it is often suspended, since the user alternates between pauses for think time and data entry; moreover, the average delay between two keypresses is relatively long. However, as soon as the user presses a key, an interrupt is raised, and the kernel wakes up the text editor process. The kernel also determines that the dynamic priority of the editor is higher than the priority of `current`, the currently running process (that is, the compiler), and hence it sets the `need_resched` field of this process, thus forcing the scheduler to be activated when the kernel finishes handling the interrupt. The scheduler selects the editor and performs a task switch; as a result, the execution of the editor is resumed very quickly and the character typed by the user is echoed to the screen. When the character has been processed, the text editor process suspends itself waiting for another keypress, and the compiler process can resume its execution.

Be aware that a preempted process is not suspended, since it remains in the `TASK_RUNNING` state; it simply no longer uses the CPU.

Some real-time operating systems feature preemptive kernels, which means that a process running in Kernel Mode can be interrupted after any instruction, just as it can in User Mode. The Linux kernel is not preemptive, which means that a process can be preempted only while running in User Mode; nonpreemptive kernel design is much simpler, since most synchronization problems involving the kernel data structures are easily avoided (see the section "Nonpreemptability of Processes in Kernel Mode" in *Chapter 11, Kernel Synchronization*).

How Long Must a Quantum Last?

The quantum duration is critical for system performances: it should be neither too long nor too short.

If the quantum duration is too short, the system overhead caused by task switches becomes excessively high. For instance, suppose that a task switch requires 10 milliseconds; if the quantum is also set to 10 milliseconds, then at least 50% of the CPU cycles will be dedicated to task switch.[2]

If the quantum duration is too long, processes no longer appear to be executed concurrently. For instance, let's suppose that the quantum is set to five seconds; each runnable process makes progress for about five seconds, but then it stops for a very long time (typically, five seconds times the number of runnable processes).

It is often believed that a long quantum duration degrades the response time of interactive applications. This is usually false. As described in the section "Process Preemption" earlier in this chapter, interactive processes have a relatively high priority, therefore they quickly preempt the batch processes, no matter how long the quantum duration is.

In some cases, a quantum duration that is too long degrades the responsiveness of the system. For instance, suppose that two users concurrently enter two commands at the respective shell prompts; one command is CPU-bound, while the other is an interactive application. Both shells fork a new process and delegate the execution of the user's command to it; moreover, suppose that such new processes have the same priority initially (Linux does not know in advance if an executed program is batch or interactive). Now, if the scheduler selects the CPU-bound process to run, the other process could wait for a whole time quantum before starting its execution. Therefore, if such duration is long, the system could appear to be unresponsive to the user that launched it.

The choice of quantum duration is always a compromise. The rule of thumb adopted by Linux is: choose a duration as long as possible, while keeping good system response time.

The Scheduling Algorithm

The Linux scheduling algorithm works by dividing the CPU time into *epochs*. In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins. In general, different processes have different time quantum durations. The time quantum value is the maximum CPU time portion assigned to the process in that epoch. When a process has exhausted its time

quantum, it is preempted and replaced by another runnable process. Of course, a process can be selected several times from the scheduler in the same epoch, as long as its quantum has not been exhausted--for instance, if it suspends itself to wait for I/O, it preserves some of its time quantum and can be selected again during the same epoch. The epoch ends when all runnable processes have exhausted their quantum; in this case, the scheduler algorithm recomputes the time-quantum durations of all processes and a new epoch begins.

Each process has a *base time quantum*: it is the time-quantum value assigned by the scheduler to the process if it has exhausted its quantum in the previous epoch. The users can change the base time quantum of their processes by using the `nice( )` and `setpriority( )` system calls (see the section "System Calls Related to Scheduling" later in this chapter). A new process always inherits the base time quantum of its parent.

The `INIT_TASK` macro sets the value of the base time quantum of process 0 (*swapper*) to `DEF_PRIORITY`; that macro is defined as follows:

```
#define DEF_PRIORITY (20*HZ/100)
```

Since `HZ`, which denotes the frequency of timer interrupts, is set to 100 for IBM PCs (see the section "Programmable Interval Timer" in Chapter 5), the value of `DEF_PRIORITY` is 20 ticks, that is, about 210 ms.

Users rarely change the base time quantum of their processes, so `DEF_PRIORITY` also denotes the base time quantum of most processes in the system.

In order to select a process to run, the Linux scheduler must consider the priority of each process. Actually, there are two kinds of priority:

Static priority
: This kind is assigned by the users to real-time processes and ranges from 1 to 99. It is never changed by the scheduler.

Dynamic priority
: This kind applies only to conventional processes; it is essentially the sum of the base time quantum (which is therefore also called the *base priority* of the process) and of the number of ticks of CPU time left to the process before its quantum expires in the current epoch.

Of course, the static priority of a real-time process is always higher than the dynamic priority of a conventional one: the scheduler will start running conventional processes only when there is no real-time process in a `TASK_RUNNING` state.

Data Structures Used by the Scheduler

We recall from the section "Process Descriptor" in Chapter 3 that the process list links together all process descriptors, while the runqueue list links together the process descriptors of all runnable processes--that is, of those in a `TASK_RUNNING` state. In both cases, the `init_task` process descriptor plays the role of list header.

Each process descriptor includes several fields related to scheduling:

`need_resched`
: A flag checked by `ret_from_intr( )` to decide whether to invoke the `schedule( )` function (see the section "The ret_ from_intr( ) Function" in *Chapter 4, Interrupts and Exceptions*).

`policy`
: The scheduling class. The values permitted are:

`SCHED_FIFO`
: A First-In, First-Out real-time process. When the scheduler assigns the CPU to the process, it leaves the process descriptor in its current position in the runqueue list. If no other higher-priority real-time process is runnable, the process will continue to use the CPU as long as it wishes, even if other real-time processes having the same priority are runnable.

`SCHED_RR`
: A Round Robin real-time process. When the scheduler assigns the CPU to the process, it puts the process descriptor at the end of the runqueue list. This policy ensures a fair assignment of CPU time to all `SCHED_RR` real-time processes that have the same priority.

`SCHED_OTHER`
: A conventional, time-shared process.

  The `policy` field also encodes a `SCHED_YIELD` binary flag. This flag is set when the process invokes the `sched_ yield( )` system call (a way of voluntarily relinquishing the processor without the need to start an I/O operation or go to sleep; see the section "System Calls Related to Real-Time Processes"). The scheduler puts the process descriptor at the bottom of the runqueue list (see the later section "System Calls Related to Scheduling").

`rt_priority`
: The static priority of a real-time process. Conventional processes do not make use of this field.

`priority`
: The base time quantum (or base priority) of the process.

`counter`
: The number of ticks of CPU time left to the process before its quantum expires; when a new epoch begins, this field contains the time-quantum duration of the process. Recall that the `update_process_times( )` function decrements the `counter` field of the current process by 1 at every tick.

When a new process is created, `do_fork( )` sets the `counter` field of both `current` (the parent) and `p` (the child) processes in the following way:

```
current->counter >>= 1;
p->counter = current->counter;
```

In other words, the number of ticks left to the parent is split in two halves, one for the parent and one for the child. This is done to prevent users from getting an unlimited amount of CPU time by using the following method: the parent process creates a child process that runs the same code and then kills itself; by properly adjusting the creation rate, the child process would always get a fresh quantum before the quantum of its parent expires. This programming trick does not work since the kernel does not reward forks. Similarly, a user cannot hog an unfair share of the processor by starting lots of background processes in a shell or by opening a lot of windows on a graphical desktop. More generally speaking, a process cannot hog resources (unless it has privileges to give itself a real-time policy) by forking multiple descendents.

Notice that the `priority` and `counter` fields play different roles for the various kinds of processes. For conventional processes, they are used both to implement time-sharing and to compute the process dynamic priority. For `SCHED_RR` real-time processes, they are used only to implement time-sharing. Finally, for `SCHED_FIFO` real-time processes, they are not used at all, because the scheduling algorithm regards the quantum duration as unlimited.

The schedule( ) Function

`schedule( )` implements the scheduler. Its objective is to find a process in the runqueue list and then assign the CPU to it. It is invoked, directly or in a lazy way, by several kernel routines.

Direct invocation

The scheduler is invoked directly when the `current` process must be

blocked right away because the resource it needs is not available. In this case, the kernel routine that wants to block it proceeds as follows:

1. Inserts `current` in the proper wait queue
2. Changes the state of `current` either to `TASK_INTERRUPTIBLE` or to `TASK_UNINTERRUPTIBLE`
3. Invokes `schedule( )`
4. Checks if the resource is available; if not, goes to step 2
5. Once the resource is available, removes `current` from the wait queue

As can be seen, the kernel routine checks repeatedly whether the resource needed by the process is available; if not, it yields the CPU to some other process by invoking `schedule( )`. Later, when the scheduler once again grants the CPU to the process, the availability of the resource is again checked.

You may have noticed that these steps are similar to those performed by the `sleep_on( )` and `interruptible_sleep_on( )` functions described in the section "Wait Queues" in Chapter 3. However, the functions we discuss here immediately remove the process from the wait queue as soon as it is woken up.

The scheduler is also directly invoked by many device drivers that execute long iterative tasks. At each iteration cycle, the driver checks the value of the `need_resched` field and, if necessary, invokes `schedule( )` to voluntarily relinquish the CPU.

Lazy invocation

The scheduler can also be invoked in a lazy way by setting the `need_resched` field of `current` to 1. Since a check on the value of this field is always made before resuming the execution of a User Mode process (see the section "Returning from Interrupts and Exceptions" in Chapter 4), `schedule( )` will definitely be invoked at some close future time.

Lazy invocation of the scheduler is performed in the following cases:

- When `current` has used up its quantum of CPU time; this is done by the `update_process_times( )` function.
- When a process is woken up and its priority is higher than that of the current process; this task is performed by the `reschedule_idle(`

) function, which is invoked by the wake_up_process( ) function (see the section "Identifying a Process" in Chapter 3):

```
if (goodness(current, p) > goodness(current, current))
current->need_resched = 1;
```

(The goodness( ) function will be described later in the section "How Good Is a Runnable Process?")

- When a sched_setscheduler( ) or sched_ yield( ) system call is issued (see the section "System Calls Related to Scheduling" later in this chapter).

Actions performed by schedule( )

Before actually scheduling a process, the schedule( ) function starts by running the functions left by other kernel control paths in various queues. The function invokes run_task_queue( ) on the tq _scheduler task queue. Linux puts a function in that task queue when it must defer its execution until the next schedule( ) invocation:

```
run_task_queue(&tq_scheduler);
```

The function then executes all active unmasked bottom halves. These are usually present to perform tasks requested by device drivers (see the section "Bottom Half" in Chapter 4):

```
if (bh_active & bh_mask)
    do_bottom_half(  );
```

Now comes the actual scheduling, and therefore a potential process switch.

The value of current is saved in the prev local variable and the need_resched field of prev is set to 0. The key outcome of the function is to set another local variable called next so that it points to the descriptor of the process selected to replace prev.

First, a check is made to determine whether prev is a Round Robin real-time process (policy field set to SCHED_RR) that has exhausted its quantum. If so, schedule( ) assigns a new quantum to prev and puts it at the bottom of the runqueue list:

```
if (!prev->counter && prev->policy == SCHED_RR) {
    prev->counter = prev->priority;
    move_last_runqueue(prev);
}
```

Now schedule( ) examines the state of prev. If it has nonblocked pending signals and its state is TASK_INTERRUPTIBLE, the function wakes

up the process as follows. This action is not the same as assigning the processor to `prev`; it just gives `prev` a chance to be selected for execution:

```
if (prev->state == TASK_INTERRUPTIBLE &&
    signal_pending(prev))
    prev->state = TASK_RUNNING;
```

If `prev` is not in the `TASK_RUNNING` state, `schedule( )` was directly invoked by the process itself because it had to wait on some external resource; therefore, `prev` must be removed from the runqueue list:

```
if (prev->state != TASK_RUNNING)
    del_from_runqueue(prev);
```

Next, `schedule( )` must select the process to be executed in the next time quantum. To that end, the function scans the runqueue list. It starts from the process referenced by the `next_run` field of `init_task`, which is the descriptor of process 0 (*swapper*). The objective is to store in `next` the process descriptor pointer of the highest priority process. In order to do this, `next` is initialized to the first runnable process to be checked, and `c` is initialized to its "goodness" (see the later section "How Good Is a Runnable Process?"):

```
if (prev->state == TASK_RUNNING) {
    next = prev;
    if (prev->policy & SCHED_YIELD) {
        prev->policy &= ~SCHED_YIELD;
        c = 0;
    } else
        c = goodness(prev, prev);
} else {
    c = -1000;
    next = &init_task;
}
```

If the `SCHED_YIELD` flag of `prev->policy` is set, `prev` has voluntarily relinquished the CPU by issuing a `sched_ yield( )` system call. In this case, the function assigns a zero goodness to it.

Now `schedule( )` repeatedly invokes the `goodness( )` function on the runnable processes to determine the best candidate:

```
p = init_task.next_run;
while (p != &init_task) {
    weight = goodness(prev, p);
    if (weight > c) {
        c = weight;
        next = p;
    }
    p = p->next_run;
}
```

The `while` loop selects the first process in the runqueue having maximum weight. If the previous process is runnable, it is preferred with respect to other runnable processes having the same weight.

Notice that if the runqueue list is empty (no runnable process exists except for *swapper*), the cycle is not entered and `next` points to `init_task`. Moreover, if all processes in the runqueue list have a priority lesser than or equal to the priority of `prev`, no process switch will take place and the old process will continue to be executed.

A further check must be made at the exit of the loop to determine whether `c` is 0. This occurs only when all the processes in the runqueue list have exhausted their quantum, that is, all of them have a zero `counter` field. When this happens, a new epoch begins, therefore `schedule( )` assigns to all existing processes (not only to the `TASK_RUNNING` ones) a fresh quantum, whose duration is the sum of the `priority` value plus half the `counter` value:

```
if (!c) {
    for_each_task(p)
        p->counter = (p->counter >> 1) + p->priority;
}
```

In this way, suspended or stopped processes have their dynamic priorities periodically increased. As stated earlier, the rationale for increasing the `counter` value of suspended or stopped processes is to give preference to I/O-bound processes. However, even after an infinite number of increases, the value of `counter` can never become larger than twice[3] the `priority` value.

Now comes the concluding part of `schedule( )`: if a process other than `prev` has been selected, a process switch must take place. Before performing it, however, the `context_swtch` field of `kstat` is increased by 1 to update the statistics maintained by the kernel:

```
if (prev != next) {
    kstat.context_swtch++;
    switch_to(prev,next);
}
return;
```

Notice that the `return` statement that exits from `schedule( )` will not be performed right away by the `next` process but at a later time by the `prev` one when the scheduler selects it again for execution.

How Good Is a Runnable Process?

The heart of the scheduling algorithm includes identifying the best candidate among all processes in the runqueue list. This is what the

`goodness( )` function does. It receives as input parameters `prev` (the descriptor pointer of the previously running process) and `p` (the descriptor pointer of the process to evaluate). The integer value `c` returned by `goodness( )` measures the "goodness" of `p` and has the following meanings:

c = -1000
: `p` must never be selected; this value is returned when the runqueue list contains only `init_task`.

c = 0
: `p` has exhausted its quantum. Unless `p` is the first process in the runqueue list and all runnable processes have also exhausted their quantum, it will not be selected for execution.

0 < *c* < 1000
: `p` is a conventional process that has not exhausted its quantum; a higher value of `c` denotes a higher level of goodness.

c >= 1000
: `p` is a real-time process; a higher value of `c` denotes a higher level of goodness.

The `goodness( )` function is equivalent to:

```
if (p->policy != SCHED_OTHER)
       return 1000 + p->rt_priority;
if (p->counter == 0)
       return 0;
if (p->mm == prev->mm)
       return p->counter + p->priority + 1;
return p->counter + p->priority;
```

If the process is real-time, its goodness is set to at least 1000. If it is a conventional process that has exhausted its quantum, its goodness is set to 0; otherwise, it is set to `p->counter + p->priority`.

A small bonus is given to `p` if it shares the address space with `prev` (i.e., if their process descriptors' `mm` fields point to the same memory descriptor). The rationale for this bonus is that if `p` runs right after `prev`, it will use the same page tables, hence the same memory; some of the valuable data may still be in the hardware cache.

The Linux/SMP Scheduler

The Linux scheduler must be slightly modified in order to support the symmetric multiprocessor (SMP) architecture. Actually, each processor runs the `schedule( )` function on its own, but processors must exchange information in order to boost system performance.

When the scheduler computes the goodness of a runnable process, it should consider whether that process was previously running on the same CPU or on another one. A process that was running on the same CPU is always preferred, since the hardware cache of the CPU could still include useful data. This rule helps in reducing the number of cache misses.

Let us suppose, however, that CPU 1 is running a process when a second, higher-priority process that was last running on CPU 2 becomes runnable. Now the kernel is faced with an interesting dilemma: should it immediately execute the higher-priority process on CPU 1, or should it defer that process's execution until CPU 2 becomes available? In the former case, hardware caches contents are discarded; in the latter case, parallelism of the SMP architecture may not be fully exploited when CPU 2 is running the idle process (*swapper*).

In order to achieve good system performance, Linux/SMP adopts an empirical rule to solve the dilemma. The adopted choice is always a compromise, and the trade-off mainly depends on the size of the hardware caches integrated into each CPU: the larger the CPU cache is, the more convenient it is to keep a process bound on that CPU.

Linux/SMP scheduler data structures

An `aligned_data` table includes one data structure for each processor, which is used mainly to obtain the descriptors of current processes quickly. Each element is filled by every invocation of the `schedule( )` function and has the following structure:

```
struct schedule_data {
    struct task_struct * curr;
    unsigned long last_schedule;
};
```

The `curr` field points to the descriptor of the process running on the corresponding CPU, while `last_schedule` specifies when `schedule( )` selected `curr` as the running process.

Several SMP-related fields are included in the process descriptor. In particular, the `avg_slice` field keeps track of the average quantum duration of the process, and the `processor` field stores the logical identifier of the last CPU that executed it.

The `cacheflush_time` variable contains a rough estimate of the minimal number of CPU cycles it takes to entirely overwrite the hardware cache content. It is initialized by the `smp_tune_scheduling( )` function to:

$$\left\lfloor \frac{\textit{cache size in KB}}{5000} \times \textit{cpu frequency in kHz} \right\rfloor$$

Intel Pentium processors have a hardware cache of 8 KB, so their `cacheflush_time` is initialized to a few hundred CPU cycles, that is, a few microseconds. Recent Intel processors have larger hardware caches, and therefore the minimal cache flush time could range from 50 to 100 microseconds.

As we shall see later, if `cacheflush_time` is greater than the average time slice of some currently running process, no process preemption is performed because it is convenient in this case to bind processes to the processors that last executed them.

The schedule( ) function

When the `schedule( )` function is executed on an SMP system, it carries out the following operations:

1. Performs the initial part of `schedule( )` as usual.
2. Stores the logical identifier of the executing processor in the `this_cpu` local variable; such value is read from the `processor` field of `prev` (that is, of the process to be replaced).
3. Initializes the `sched_data` local variable so that it points to the `schedule_data` structure of the `this_cpu` CPU.
4. Invokes `goodness( )` repeatedly to select the new process to be executed; this function also examines the `processor` field of the processes and gives a consistent bonus (`PROC_CHANGE_PENALTY`, usually 15) to the process that was last executed on the `this_cpu` CPU.
5. If needed, recomputes process dynamic priorities as usual.
6. Sets `sched_data->curr` to `next`.
7. Sets `next->has_cpu` to 1 and `next->processor` to `this_cpu`.
8. Stores the current Time Stamp Counter value in the `t` local variable.
9. Stores the last time slice duration of `prev` in the `this_slice` local variable; this value is the difference between `t` and `sched_data->last_schedule`.
10. Sets `sched_data->last_schedule` to `t`.

11. Sets the `avg_slice` field of `prev` to (`prev->avg_slice+this_slice`)/2; in other words, updates the average.

12. Performs the context switch.

13. When the kernel returns here, the original previous process has been selected again by the scheduler; the `prev` local variable now refers to the process that has just been replaced. If `prev` is still runnable and it is not the idle task of this CPU, invokes the `reschedule_idle( )` function on it (see the next section).

14. Sets the `has_cpu` field of `prev` to 0.

The reschedule_idle( ) function

The `reschedule_idle( )` function is invoked when a process `p` becomes runnable (see the earlier section "The schedule( ) Function"). On an SMP system, the function determines whether the process should preempt the current process of some CPU. It performs the following operations:

1. If `p` is a real-time process, always attempts to perform preemption: go to step 3.

2. Returns immediately (does not attempt to preempt) if there is a CPU whose current process satisfies both of the following conditions:[4]

   - `cacheflush_time` is greater than the average time slice of the current process. If this is true, the process is not dirtying the cache significantly.

   - Both `p` and the current process need the global kernel lock (see the section "Global and Local Kernel Locks" in Chapter 11) in order to access some critical kernel data structure. This check is performed because replacing a process holding the lock with another one that needs it is not fruitful.

3. If the `p->processor` CPU (the one on which `p` was last running) is idle, selects it.

4. Otherwise, computes the difference:

   ```
   goodness(tsk, p) - goodness(tsk, tsk)
   ```

   for each task `tsk` running on some CPU and selects the CPU for which the difference is greatest, provided it is a positive value.

5. If CPU has been selected, sets the `need_resched` field of the corresponding running process and sends a "reschedule" message to that processor (see the section "Interprocessor Interrupts" in Chapter 11).

Performance of the Scheduling Algorithm

The scheduling algorithm of Linux is both self-contained and relatively easy to follow. For that reason, many kernel hackers love to try to make improvements. However, the scheduler is a rather mysterious component of the kernel. While you can change its performance significantly by modifying just a few key parameters, there is usually no theoretical support to justify the results obtained. Furthermore, you can't be sure that the positive (or negative) results obtained will continue to hold when the mix of requests submitted by the users (real-time, interactive, I/O-bound, background, etc.) varies significantly. Actually, for almost every proposed scheduling strategy, it is possible to derive an artificial mix of requests that yields poor system performances.

Let us try to outline some pitfalls of the Linux scheduler. As it will turn out, some of these limitations become significant on large systems with many users. On a single workstation that is running a few tens of processes at a time, the Linux scheduler is quite efficient. Since Linux was born on an Intel 80386 and continues to be most popular in the PC world, we consider the current Linux scheduler quite appropriate.

The algorithm does not scale well

If the number of existing processes is very large, it is inefficient to recompute all dynamic priorities at once.

In old traditional Unix kernels, the dynamic priorities were recomputed every second, thus the problem was even worse. Linux tries instead to minimize the overhead of the scheduler. Priorities are recomputed only when all runnable processes have exhausted their time quantum. Therefore, when the number of processes is large, the recomputation phase is more expensive but is executed less frequently.

This simple approach has the disadvantage that when the number of runnable processes is very large, I/O-bound processes are seldom boosted, and therefore interactive applications have a longer response time.

The predefined quantum is too large for high system loads

The system responsiveness experienced by users depends heavily on the *system load*, which is the average number of processes that are runnable, and hence waiting for CPU time.[5]

As mentioned before, system responsiveness depends also on the average time-quantum duration of the runnable processes. In Linux, the predefined time quantum appears to be too large for high-end machines having a very high expected system load.

I/O-bound process boosting strategy is not optimal

The preference for I/O-bound processes is a good strategy to ensure a short response time for interactive programs, but it is not perfect. Indeed, some batch programs with almost no user interaction are I/O-bound. For instance, consider a database search engine that must typically read lots of data from the hard disk or a network application that must collect data from a remote host on a slow link. Even if these kinds of processes do not need a short response time, they are boosted by the scheduling algorithm.

On the other hand, interactive programs that are also CPU-bound may appear unresponsive to the users, since the increment of dynamic priority due to I/O blocking operations may not compensate for the decrement due to CPU usage.

Support for real-time applications is weak

As stated in the first chapter, nonpreemptive kernels are not well suited for real-time applications, since processes may spend several milliseconds in Kernel Mode while handling an interrupt or exception. During this time, a real-time process that becomes runnable cannot be resumed. This is unacceptable for real-time applications, which require predictable and low response times.[6]

Future versions of Linux will likely address this problem, either by implementing SVR4's "fixed preemption points" or by making the kernel fully preemptive.

However, kernel preemption is just one of several necessary conditions for implementing an effective real-time scheduler. Several other issues must be considered. For instance, real-time processes often must use resources also needed by conventional processes. A real-time process may thus end up waiting until a lower-priority process releases some resource. This phenomenon is called *priority inversion*. Moreover, a real-time process could require a kernel service that is granted on behalf of another lower-priority process (for example, a kernel thread). This phenomenon is called *hidden*

scheduling. An effective real-time scheduler should address and resolve such problems.

System Calls Related to Scheduling

Several system calls have been introduced to allow processes to change their priorities and scheduling policies. As a general rule, users are always allowed to lower the priorities of their processes. However, if they want to modify the priorities of processes belonging to some other user or if they want to increase the priorities of their own processes, they must have superuser privileges.

The nice( ) System Call

The `nice( )`[7] system call allows processes to change their base priority. The integer value contained in the `increment` parameter is used to modify the `priority` field of the process descriptor. The `nice` Unix command, which allows users to run programs with modified scheduling priority, is based on this system call.

The `sys_nice( )` service routine handles the `nice( )` system call. Although the `increment` parameter may have any value, absolute values larger than 40 are trimmed down to 40. Traditionally, negative values correspond to requests for priority increments and require superuser privileges, while positive ones correspond to requests for priority decrements.

The function starts by copying the value of `increment` into the `newprio` local variable. In the case of a negative increment, the function invokes the `capable( )` function to verify whether the process has a `CAP_SYS_NICE` capability. We shall discuss that function, together with the notion of capability, in *Chapter 19, Program Execution*. If the user turns out to have the capability required to change priorities, `sys_nice( )` changes the sign of `newprio` and it sets the `increase` local flag:

```
increase = 0
newprio = increment;
if (increment < 0) {
    if (!capable(CAP_SYS_NICE))
        return -EPERM;
    newprio = -increment;
    increase = 1;
}
```

If `newprio` has a value larger than 40, the function trims it down to 40. At this point, the `newprio` local variable may have any value included from 0 to 40, inclusive. The value is then converted according to the priority scale used by the scheduling algorithm. Since the highest

base priority allowed is 2 × DEF_PRIORITY, the new value is:

$$\lfloor (\text{newprio} \times 2 \times \text{DEF\_PRIORITY}) / 40 + 0.5 \rfloor$$

The resulting value is copied into increment with the proper sign:

```
if (newprio > 40)
    newprio = 40;
newprio = (newprio * DEF_PRIORITY + 10) / 20;
increment = newprio;
if (increase)
    increment = -increment;
```

Since newprio is an integer variable, the expression in the code is equivalent to the formula shown earlier.

The function then sets the final value of priority by subtracting the value of increment from it. However, the final base priority of the process cannot be smaller than 1 or larger than 2 × DEF_PRIORITY:

```
if (current->priority - increment < 1)
    current->priority = 1;
else if (current->priority > DEF_PRIORITY*2)
    current->priority = DEF_PRIORITY*2;
else
    current->priority -= increment;
return 0;
```

A niced process changes over time like any other process, getting extra priority if necessary or dropping back in deference to other processes.

The getpriority( ) and setpriority( ) System Calls

The nice( ) system call affects only the process that invokes it. Two other system calls, denoted as getpriority( ) and setpriority( ), act on the base priorities of all processes in a given group. getpriority( ) returns 20 plus the highest base priority among all processes in a given group; setpriority( ) sets the base priority of all processes in a given group to a given value.

The kernel implements these system calls by means of the sys_getpriority( ) and sys_setpriority( ) service routines. Both of them act essentially on the same group of parameters:

which
: Identifies the group of processes; it can assume one of the following values:

PRIO_PROCESS
: Select the processes according to their process ID (pid field of the

process descriptor).

PRIO_PGRP
: Select the processes according to their group ID (pgrp field of the process descriptor).

PRIO_USER
: Select the processes according to their user ID (uid field of the process descriptor).

who
: Value of the pid, pgrp, or uid field (depending on the value of which) to be used for selecting the processes. If who is 0, its value is set to that of the corresponding field of the current process.

niceval
: The new base priority value (needed only by sys_setpriority( )). It should range between -20 (highest priority) and +20 (minimum priority).

As stated before, only processes with a CAP_SYS_NICE capability are allowed to increase their own base priority or to modify that of other processes.

As we have seen in Chapter 8, system calls return a negative value only if some error occurred. For that reason, getpriority( ) does not return a normal nice value ranging between -20 and 20, but rather a nonnegative value ranging between 0 and 40.

System Calls Related to Real-Time Processes

We now introduce a group of system calls that allow processes to change their scheduling discipline and, in particular, to become real-time processes. As usual, a process must have a CAP_SYS_NICE capability in order to modify the values of the rt_priority and policy process descriptor fields of any process, including itself.

The sched_getscheduler( ) and sched_setscheduler( ) system calls

The sched_ getscheduler( ) system call queries the scheduling policy currently applied to the process identified by the pid parameter. If pid equals 0, the policy of the calling process will be retrieved. On success, the system call returns the policy for the process: SCHED_FIFO, SCHED_RR, or SCHED_OTHER. The corresponding sys_sched_getscheduler( ) service routine invokes find_task_by_pid( ), which locates the process descriptor corresponding to the given pid and returns the value of its

policy field.

The sched_setscheduler( ) system call sets both the scheduling policy and the associated parameters for the process identified by the parameter pid. If pid is equal to 0, the scheduler parameters of the calling process will be set.

The corresponding sys_sched_setscheduler( ) function checks whether the scheduling policy specified by the policy parameter and the new static priority specified by the param->sched_priority parameter are valid. It also checks whether the process has CAP_SYS_NICE capability or whether its owner has superuser rights. If everything is OK, it executes the following statements:

```
p->policy = policy;
p->rt_priority = param->sched_priority;
if (p->next_run)
    move_first_runqueue(p);
current->need_resched = 1;
```

The sched_ getparam( ) and sched_setparam( ) system calls

The sched_getparam( ) system call retrieves the scheduling parameters for the process identified by pid. If pid is 0, the parameters of the current process are retrieved. The corresponding sys_sched_getparam( ) service routine, as one would expect, finds the process descriptor pointer associated with pid, stores its rt_priority field in a local variable of type sched_param, and invokes copy_to_user( ) to copy it into the process address space at the address specified by the param parameter.

The sched_setparam( ) system call is similar to sched_setscheduler( ): it differs from the latter by not letting the caller set the policy field's value.[8] The corresponding sys_sched_setparam( ) service routine is almost identical to sys_sched_setscheduler( ), but the policy of the affected process is never changed.

The sched_ yield( ) system call

The sched_ yield( ) system call allows a process to relinquish the CPU voluntarily without being suspended; the process remains in a TASK_RUNNING state, but the scheduler puts it at the end of the runqueue list. In this way, other processes having the same dynamic priority will have a chance to run. The call is used mainly by SCHED_FIFO processes.

The corresponding sys_sched_ yield( ) service routine executes these statements:

```
if (current->policy == SCHED_OTHER)
    current->policy |= SCHED_YIELD;
```

```
current->need_resched = 1;
move_last_runqueue(current);
```

Notice that the SCHED_YIELD field is set in the policy field of the process descriptor only if the process is a conventional SCHED_OTHER process. As a result, the next invocation of schedule( ) will view this process as one that has exhausted its time quantum (see how schedule( ) handles the SCHED_YIELD field).

The sched_ get_priority_min( ) and sched_ get_priority_max( ) system calls

The sched_get_priority_min( ) and sched_get_priority_max( ) system calls return, respectively, the minimum and the maximum real-time static priority value that can be used with the scheduling policy identified by the policy parameter.

The sys_sched_get_priority_min( ) service routine returns 1 if current is a real-time process, 0 otherwise.

The sys_sched_get_priority_max( ) service routine returns 99 (the highest priority) if current is a real-time process, 0 otherwise.

The sched_rr_ get_interval( ) system call

The sched_rr_get_interval( ) system call should get the round robin time quantum for the named real-time process.

The corresponding sys_sched_rr_get_interval( ) service routine does not operate as expected, since it always returns a 150-millisecond value in the timespec structure pointed to by tp. This system call remains effectively unimplemented in Linux.

Anticipating Linux 2.4

Linux 2.4 introduces a subtle optimization concerning TLB flushing for kernel threads and zombie processes. As a result, the active Page Global Directory is set by the schedule( ) function rather than by the switch_to macro.

The Linux 2.4 scheduling algorithm for SMP machines has been improved and simplified. Whenever a new process becomes runnable, the kernel checks whether the preferred CPU of the process, that is, the CPU on which it was last running, is idle; in this case, the kernel assigns the process to that CPU. Otherwise, the kernel assigns the process to another idle CPU, if any. If all CPUs are busy, the kernel checks whether the process has enough priority to preempt the

process running on the preferred CPU. If not, the kernel tries to preempt some other CPU only if the new runnable process is real-time or if it has short average time slices compared to the hardware cache rewriting time. (Roughly, preemption occurs if the new runnable process is interactive and the preferred CPU will not reschedule shortly.)

---

1. Recall that stopped and suspended processes cannot be selected by the scheduling algorithm to run on the CPU.

2. Actually, things could be much worse than this; for example, if the time required for task switch is counted in the process quantum, all CPU time will be devoted to task switch and no process can progress toward its termination. Anyway, you got the point.

3. Assume both `priority` and `counter` equal to P; then the geometric series $P \times (1 + 1/2 + 1/4 + 1/8 + \ldots)$ converges to $2 \times P$.

4. These conditions look like voodoo magic; perhaps, they are empirical rules that make the SMP scheduler work better.

5. The `uptime` program returns the system load for the past 1, 5, and 15 minutes. The same information can be obtained by reading the */proc/loadavg* file.

6. The Linux kernel has been modified in several ways so it can handle a few hard real-time jobs if they remain short. Basically, hardware interrupts are trapped and kernel execution is monitored by a kind of "superkernel." These changes do not make Linux a true real-time system, though.

7. Since this system call is usually invoked to lower the priority of a process, users who invoke it for their processes are "nice" toward other users.

8. This anomaly is caused by a specific requirement of the POSIX standard.

Back to: Understanding the Linux Kernel
