35

PROCESS PRIORITIES AND SCHEDULING

This chapter discusses various system calls and process attributes that determine when and which processes obtain access to the CPU(s). We begin by describing the *nice* value, a process characteristic that influences the amount of CPU time that a process is allocated by the kernel scheduler. We follow this with a description of the POSIX realtime scheduling API. This API allows us to define the policy and priority used for scheduling processes, giving us much tighter control over how processes are allocated to the CPU. We conclude with a discussion of the system calls for setting a process's CPU affinity mask, which determines the set of CPUs on which a process running on a multiprocessor system will run.

35.1 Process Priorities (Nice Values)

On Linux, as with most other UNIX implementations, the default model for scheduling processes for use of the CPU is *round-robin time-sharing*. Under this model, each process in turn is permitted to use the CPU for a brief period of time, known as a *time slice* or *quantum*. Round-robin time-sharing satisfies two important requirements of an interactive multitasking system:

- *Fairness*: Each process gets a share of the CPU.
- *Responsiveness*: A process doesn't need to wait for long periods before it receives use of the CPU.

Under the round-robin time-sharing algorithm, processes can't exercise direct control over when and for how long they will be able to use the CPU. By default, each process in turn receives use of the CPU until its time slice runs out or it voluntarily gives up the CPU (for example, by putting itself to sleep or performing a disk read). If all processes attempt to use the CPU as much as possible (i.e., no process ever sleeps or blocks on an I/O operation), then they will receive a roughly equal share of the CPU.

However, one process attribute, the *nice value*, allows a process to indirectly influence the kernel's scheduling algorithm. Each process has a nice value in the range −20 (high priority) to +19 (low priority); the default is 0 (refer to Figure 35-1). In traditional UNIX implementations, only privileged processes can assign themselves (or other processes) a negative (high) priority. (We'll explain some Linux differences in Section 35.3.2.) Unprivileged processes can only lower their priority, by assuming a nice value greater than the default of 0. By doing this, they are being "nice" to other processes, and this fact gives the attribute its name.

The nice value is inherited by a child created via *fork()* and preserved across an *exec()*.

> Rather than returning the actual nice value, the *getpriority()* system call service routine returns a number in the range 1 (low priority) to 40 (high priority), calculated according to the formula *unice = 20 − knice*. This is done to avoid having a negative return value from a system call service routine, which is used to indicate an error. (See the description of system call service routines in Section 3.1.) Applications are unaware of the manipulated value returned by the system call service routine, since the C library *getpriority()* wrapper function reverses the calculation, returning the value *20 − unice* to the calling program.

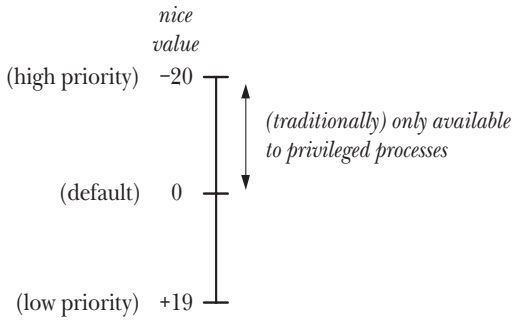

Figure 35-1: Range and interpretation of the process nice value

Effect of the nice value

Processes are not scheduled in a strict hierarchy by nice value; rather, the nice value acts as weighting factor that causes the kernel scheduler to favor processes with higher priorities. Giving a process a low priority (i.e., high nice value) won't cause it to be completely starved of the CPU, but causes it to receive relatively less CPU time. The extent to which the nice value influences the scheduling of a process has varied across Linux kernel versions, as well as across UNIX systems.

> Starting in kernel 2.6.23, a new kernel scheduling algorithm means that relative differences in nice values have a much stronger effect than in previous kernels. As a result, processes with low nice values receive less CPU than before, and processes with high nice values obtain a greater proportion of the CPU.

Retrieving and modifying priorities

The *getpriority()* and *setpriority()* system calls allow a process to retrieve and change its own nice value or that of another process.

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

int getpriority(int which, id_t who);
```

Returns (possibly negative) nice value of specified process on success, or −1 on error

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

Returns 0 on success, or −1 on error

Both system calls take the arguments *which* and *who*, identifying the process(es) whose priority is to be retrieved or modified. The *which* argument determines how *who* is interpreted. This argument takes one of the following values:

`PRIO_PROCESS`
: Operate on the process whose process ID equals *who*. If *who* is 0, use the caller's process ID.

`PRIO_PGRP`
: Operate on all of the members of the process group whose process group ID equals *who*. If *who* is 0, use the caller's process group.

`PRIO_USER`
: Operate on all processes whose real user ID equals *who*. If *who* is 0, use the caller's real user ID.

The *id_t* data type, used for the *who* argument, is an integer type of sufficient size to accommodate a process ID or a user ID.

The *getpriority()* system call returns the nice value of the process specified by *which* and *who*. If multiple processes match the criteria specified (which may occur if *which* is `PRIO_PGRP` or `PRIO_USER`), then the nice value of the process with the highest priority (i.e., lowest numerical value) is returned. Since *getpriority()* may legitimately return a value of −1 on a successful call, we must test for an error by setting *errno* to 0 prior to the call, and then checking for a −1 return status and a nonzero *errno* value after the call.

The *setpriority()* system call sets the nice value of the process(es) specified by *which* and *who* to the value specified in *prio*. Attempts to set a nice value to a number outside the permitted range (−20 to +19) are silently bounded to this range.

> Historically, the nice value was changed using the call *nice(incr)*, which added *incr* to the calling process's nice value. This function is still available, but it is superseded by the more general *setpriority()* system call.
>
> The command-line analogs of *setpriority()* are *nice(1)*, which can be used by unprivileged users to run a command with a lower priority or by privileged users to run a command with a raised priority, and *renice(8)*, which can be used by the superuser to change the nice value of an existing process.

A privileged (CAP_SYS_NICE) process can change the priority of any process. An unprivileged process may change its own priority (by specifying *which* as PRIO_PROCESS, and *who* as 0) or the priority of another (target) process, if its effective user ID matches the real or effective user ID of the target process. The Linux permission rules for *setpriority()* differ from SUSv3, which specifies that an unprivileged process can change the priority of another process if its real or effective user ID matches the effective user ID of the target process. UNIX implementations show some variation on this point. Some follow the SUSv3 rules, but others—notably the BSDs—behave in the same way as Linux.

> In Linux kernels before 2.6.12, the permission rules for calls to *setpriority()* by unprivileged processes are different from later kernels (and also deviate from SUSv3). An unprivileged process can change the priority of another process if its real or effective user ID matches the real user ID of the target process. From Linux 2.6.12 onward, the permissions checks were changed to be consistent with other similar APIs available on Linux, such as *sched_setscheduler()* and *sched_setaffinity()*.

In Linux kernels before 2.6.12, an unprivileged process may use *setpriority()* only to (irreversibly) lower its own or another process's nice value. A privileged (CAP_SYS_NICE) process can use *setpriority()* to raise nice values.

Since kernel 2.6.12, Linux provides the RLIMIT_NICE resource limit, which permits unprivileged processes to increase nice values. An unprivileged process can raise its own nice value to the maximum specified by the formula *20 − rlim_cur*, where *rlim_cur* is the current RLIMIT_NICE soft resource limit. As an example, if a process's RLIMIT_NICE soft limit is 25, then its nice value can be raised to −5. From this formula, and the knowledge that the nice value has the range +19 (low) to −20 (high), we can deduce that the effectively useful range of the RLIMIT_NICE limit is 1 (low) to 40 (high). (RLIMIT_NICE doesn't use the number range +19 to −20 because some negative resource-limit values have special meanings—for example, RLIM_INFINITY has the same representation as −1.)

An unprivileged process can make a *setpriority()* call to change the nice value of another (target) process, if the effective user ID of the process calling *setpriority()* matches the real or effective user ID of the target process, and the change to the nice value is consistent with the target process's RLIMIT_NICE limit.

The program in Listing 35-1 uses *setpriority()* to change the nice value of the process(es) specified by its command-line arguments (which correspond to the arguments of *setpriority()*), and then calls *getpriority()* to verify the change.

Listing 35-1: Modifying and retrieving a process's nice value

procpri/t_setpriority.c

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

int
main(int argc, char *argv[])
{
    int which, prio;
    id_t who;
```

```
    if (argc != 4 || strchr("pgu", argv[1][0]) == NULL)
        usageErr("%s {p|g|u} who priority\n"
                 "    set priority of: p=process; g=process group; "
                 "u=processes for user\n", argv[0]);

    /* Set nice value according to command-line arguments */

    which = (argv[1][0] == 'p') ? PRIO_PROCESS :
                (argv[1][0] == 'g') ? PRIO_PGRP : PRIO_USER;
    who = getLong(argv[2], 0, "who");
    prio = getInt(argv[3], 0, "prio");

    if (setpriority(which, who, prio) == -1)
        errExit("getpriority");

    /* Retrieve nice value to check the change */

    errno = 0;                  /* Because successful call may return -1 */
    prio = getpriority(which, who);
    if (prio == -1 && errno != 0)
        errExit("getpriority");

    printf("Nice value = %d\n", prio);

    exit(EXIT_SUCCESS);
}
```

procpri/t_setpriority.c

35.2 Overview of Realtime Process Scheduling

The standard kernel scheduling algorithm usually provides adequate performance and responsiveness for the mixture of interactive and background processes typically run on a system. However, realtime applications have more stringent requirements of a scheduler, including the following:

- A realtime application must provide a guaranteed maximum response time for external inputs. In many cases, these guaranteed maximum response times must be quite small (e.g., of the order of a fraction of a second). For example, a slow response by a vehicle navigation system could be catastrophic. To satisfy this requirement, the kernel must provide the facility for a high-priority process to obtain control of the CPU in a timely fashion, preempting any process that may currently be running.

  > A time-critical application may need to take other steps to avoid unacceptable delays. For example, to avoid being delayed by a page fault, an application can lock all of its virtual memory into RAM using *mlock()* or *mlockall()* (described in Section 50.2).

- A high-priority process should be able to maintain exclusive access to the CPU until it completes or voluntarily relinquishes the CPU.

- A realtime application should be able to control the precise order in which its component processes are scheduled.

SUSv3 specifies a realtime process scheduling API (originally defined in POSIX.1b) that partly addresses these requirements. This API provides two realtime scheduling policies: SCHED_RR and SCHED_FIFO. Processes operating under either of these policies always have priority over processes scheduled using the standard round-robin time-sharing policy described in Section 35.1, which the realtime scheduling API identifies using the constant SCHED_OTHER.

Each of the realtime policies allows for a range of priority levels. SUSv3 requires that an implementation provide at least 32 discrete priorities for the realtime policies. In each scheduling policy, runnable processes with higher priority always have precedence over lower-priority processes when seeking access to the CPU.

> The statement that runnable processes with higher priority always have precedence over lower-priority processes needs to be qualified for multiprocessor Linux systems (including hyperthreaded systems). On a multiprocessor system, each CPU has a separate run queue (this provides better performance than a single system-wide run queue), and processes are prioritized only per CPU run queue. For example, on a dual-processor system with three processes, process A with realtime priority 20 could be queued waiting for CPU 0, which is currently running process B with priority 30, even though CPU 1 is running process C with a priority of 10.
>
> Realtime applications that employ multiple processes (or threads) can use the CPU affinity API described in Section 35.4 to avoid any problems that might result from this scheduling behavior. For example, on a four-processor system, all noncritical processes could be isolated onto a single CPU, leaving the other three CPUs available for use by the application.

Linux provides 99 realtime priority levels, numbered 1 (lowest) to 99 (highest), and this range applies in both realtime scheduling policies. The priorities in each policy are equivalent. This means that, given two processes with the same priority, one operating under the SCHED_RR policy and the other under SCHED_FIFO, either may be the next one eligible for execution, depending on the order in which they were scheduled. In effect, each priority level maintains a queue of runnable processes, and the next process to run is selected from the front of the highest-priority nonempty queue.

POSIX realtime versus hard realtime

Applications with all of the requirements listed at the start of this section are sometimes referred to as *hard* realtime applications. However, the POSIX realtime process scheduling API doesn't satisfy all of these requirements. In particular, it provides no way for an application to guarantee response times for handling input. To make such guarantees requires operating system features that are not part of the mainline Linux kernel (nor most other standard operating systems). The POSIX API merely provides us with so-called *soft* realtime, allowing us to control which processes are scheduled for use of the CPU.

Adding support for hard realtime applications is difficult to achieve without imposing an overhead on the system that conflicts with the performance requirements of the time-sharing applications that form the majority of applications on typical desktop and server systems. This is why most UNIX kernels—including, historically, Linux—have not natively supported realtime applications. Nevertheless, starting from around version 2.6.18, various features have been added to the Linux kernel with the eventual aim of allowing Linux to natively provide full support for hard realtime applications, without imposing the aforementioned overhead for time-sharing operation.

35.2.1 The SCHED_RR Policy

Under the SCHED_RR (round-robin) policy, processes of equal priority are executed in a round-robin time-sharing fashion. A process receives a fixed-length time slice each time it uses the CPU. Once scheduled, a process employing the SCHED_RR policy maintains control of the CPU until either:

- it reaches the end of its time slice;
- it voluntarily relinquishes the CPU, either by performing a blocking system call or by calling the *sched_yield()* system call (described in Section 35.3.3);
- it terminates; or
- it is preempted by a higher-priority process.

For the first two events above, when a process running under the SCHED_RR policy loses access to the CPU, it is placed at the back of the queue for its priority level. In the final case, when the higher-priority process has ceased execution, the preempted process continues execution, consuming the remainder of its time slice (i.e., the preempted process remains at the head of the queue for its priority level).

In both the SCHED_RR and the SCHED_FIFO policies, the currently running process may be preempted for one of the following reasons:

- a higher-priority process that was blocked became unblocked (e.g., an I/O operation on which it was waiting completed);
- the priority of another process was raised to a higher level than the currently running process; or
- the priority of the currently running process was decreased to a lower value than that of some other runnable process.

The SCHED_RR policy is similar to the standard round-robin time-sharing scheduling algorithm (SCHED_OTHER), in that it allows a group of processes with the same priority to share access to the CPU. The most notable difference is the existence of strictly distinct priority levels, with higher-priority processes always taking precedence over lower-priority processes. By contrast, a low nice value (i.e., high priority) doesn't give a process exclusive access to the CPU; it merely gives the process a favorable weighting in scheduling decisions. As noted in Section 35.1, a process with a low priority (i.e., high nice value) always receives at least some CPU time. The other important difference is that the SCHED_RR policy allows us to precisely control the order in which processes are scheduled.

35.2.2 The SCHED_FIFO Policy

The SCHED_FIFO (first-in, first-out) policy is similar to the SCHED_RR policy. The major difference is that there is no time slice. Once a SCHED_FIFO process gains access to the CPU, it executes until either:

- it voluntarily relinquishes the CPU (in the same manner as described for the SCHED_FIFO policy above);
- it terminates; or
- it is preempted by a higher-priority process (in the same circumstances as described for the SCHED_FIFO policy above).

In the first case, the process is placed at the back of the queue for its priority level. In the last case, when the higher-priority process has ceased execution (by blocking or terminating), the preempted process continues execution (i.e., the preempted process remains at the head of the queue for its priority level).

35.2.3 The SCHED_BATCH and SCHED_IDLE Policies

The Linux 2.6 kernel series added two nonstandard scheduling policies: SCHED_BATCH and SCHED_IDLE. Although these policies are set via the POSIX realtime scheduling API, they are not actually realtime policies.

The SCHED_BATCH policy, added in kernel 2.6.16, is similar to the default SCHED_OTHER policy. The difference is that the SCHED_BATCH policy causes jobs that frequently wake up to be scheduled less often. This policy is intended for batch-style execution of processes.

The SCHED_IDLE policy, added in kernel 2.6.23, is also similar to SCHED_OTHER, but provides functionality equivalent to a very low nice value (i.e., lower than +19). The process nice value has no meaning for this policy. It is intended for running low-priority jobs that will receive a significant proportion of the CPU only if no other job on the system requires the CPU.

35.3 Realtime Process Scheduling API

We now look at the various system calls constituting the realtime process scheduling API. These system calls allow us to control process scheduling policies and priorities.

> Although realtime scheduling has been a part of Linux since version 2.0 of the kernel, several problems persisted for a long time in the implementation. A number of features of the implementation remained broken in the 2.2 kernel, and even in early 2.4 kernels. Most of these problems were rectified by about kernel 2.4.20.

35.3.1 Realtime Priority Ranges

The *sched_get_priority_min()* and *sched_get_priority_max()* system calls return the available priority range for a scheduling policy.

```
#include <sched.h>

int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

Both return nonnegative integer priority on success, or –1 on error

For both system calls, *policy* specifies the scheduling policy about which we wish to obtain information. For this argument, we specify either SCHED_RR or SCHED_FIFO. The *sched_get_priority_min()* system call returns the minimum priority for the specified policy, and *sched_get_priority_max()* returns the maximum priority. On Linux, these system calls return the numbers 1 and 99, respectively, for both the SCHED_RR and SCHED_FIFO policies. In other words, the priority ranges of the two realtime policies completely coincide, and SCHED_RR and SCHED_FIFO processes with the same priority are equally eligible for scheduling. (Which one is scheduled first depends on their order in the queue for that priority level.)

The range of realtime priorities differs from one UNIX implementation to another. Therefore, instead of hard-coding priority values into an application, we should specify priorities relative to the return value from one of these functions. Thus, the lowest SCHED_RR priority would be specified as *sched_get_priority_min(SCHED_FIFO)*, the next higher priority as *sched_get_priority_min(SCHED_FIFO) + 1*, and so on.

> SUSv3 doesn't require that the SCHED_RR and SCHED_FIFO policies use the same priority ranges, but they do so on most UNIX implementations. For example, on Solaris 8, the priority range for both policies is 0 to 59, and on FreeBSD 6.1, it is 0 to 31.

35.3.2 Modifying and Retrieving Policies and Priorities

In this section, we look at the system calls that modify and retrieve scheduling policies and priorities.

Modifying scheduling policies and priorities

The *sched_setscheduler()* system call changes both the scheduling policy and the priority of the process whose process ID is specified in *pid*. If *pid* is specified as 0, the attributes of the calling process are changed.

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);
```

Returns 0 on success, or –1 on error

The *param* argument is a pointer to a structure of the following form:

```
struct sched_param {
    int sched_priority;        /* Scheduling priority */
};
```

SUSv3 defines the *param* argument as a structure to allow an implementation to include additional implementation-specific fields, which may be useful if an implementation provides additional scheduling policies. However, like most UNIX implementations, Linux provides just the *sched_priority* field, which specifies the scheduling priority. For the SCHED_RR and SCHED_FIFO policies, this must be a value in the range indicated by *sched_get_priority_min()* and *sched_get_priority_max()*; for other policies, the priority must be 0.

The *policy* argument determines the scheduling policy for the process. It is specified as one of the policies shown in Table 35-1.

Table 35-1: Linux realtime and nonrealtime scheduling policies

Policy	Description	SUSv3
SCHED_FIFO	Realtime first-in first-out	•
SCHED_RR	Realtime round-robin	•
SCHED_OTHER	Standard round-robin time-sharing	•
SCHED_BATCH	Similar to SCHED_OTHER, but intended for batch execution (since Linux 2.6.16)	
SCHED_IDLE	Similar to SCHED_OTHER, but with priority even lower than nice value +19 (since Linux 2.6.23)	

A successful *sched_setscheduler()* call moves the process specified by *pid* to the back of the queue for its priority level.

SUSv3 specifies that the return value of a successful *sched_setscheduler()* call should be the previous scheduling policy. However, Linux deviates from the standard in that a successful call returns 0. A portable application should test for success by checking that the return status is not −1.

The scheduling policy and priority are inherited by a child created via *fork()*, and they are preserved across an *exec()*.

The *sched_setparam()* system call provides a subset of the functionality of *sched_setscheduler()*. It modifies the scheduling priority of a process while leaving the policy unchanged.

```
#include <sched.h>

int sched_setparam(pid_t pid, const struct sched_param *param);
                                                Returns 0 on success, or −1 on error
```

The *pid* and *param* arguments are the same as for *sched_setscheduler()*.

A successful *sched_setparam()* call moves the process specified by *pid* to the back of the queue for its priority level.

The program in Listing 35-2 uses *sched_setscheduler()* to set the policy and priority of the processes specified by its command-line arguments. The first argument is a letter specifying a scheduling policy, the second is an integer priority, and the remaining arguments are the process IDs of the processes whose scheduling attributes are to be changed.

Listing 35-2: Modifying process scheduling policies and priorities

procpri/sched_set.c

```
#include <sched.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int j, pol;
    struct sched_param sp;

    if (argc < 3 || strchr("rfo", argv[1][0]) == NULL)
        usageErr("%s policy priority [pid...]\n"
                 "    policy is 'r' (RR), 'f' (FIFO), "
#ifdef SCHED_BATCH              /* Linux-specific */
                 "'b' (BATCH), "
#endif
#ifdef SCHED_IDLE               /* Linux-specific */
                 "'i' (IDLE), "
#endif
                 "or 'o' (OTHER)\n",
                 argv[0]);

    pol = (argv[1][0] == 'r') ? SCHED_RR :
                 (argv[1][0] == 'f') ? SCHED_FIFO :
#ifdef SCHED_BATCH
                 (argv[1][0] == 'b') ? SCHED_BATCH :
#endif
#ifdef SCHED_IDLE
                 (argv[1][0] == 'i') ? SCHED_IDLE :
#endif
                 SCHED_OTHER;
    sp.sched_priority = getInt(argv[2], 0, "priority");

    for (j = 3; j < argc; j++)
        if (sched_setscheduler(getLong(argv[j], 0, "pid"), pol, &sp) == -1)
            errExit("sched_setscheduler");

    exit(EXIT_SUCCESS);
}
```

procpri/sched_set.c

Privileges and resource limits affecting changes to scheduling parameters

In kernels before 2.6.12, a process generally must be privileged (CAP_SYS_NICE) to make changes to scheduling policies and priorities. The one exception to this requirement is that an unprivileged process can change the scheduling policy of a process to SCHED_OTHER if the effective user ID of the caller matches either the real or effective user ID of the target process.

Since kernel 2.6.12, the rules about setting realtime scheduling policies and priorities have changed with the introduction of a new, nonstandard resource limit, RLIMIT_RTPRIO. As with older kernels, privileged (CAP_SYS_NICE) processes can

make arbitrary changes to the scheduling policy and priority of any process. However, an unprivileged process can also change scheduling policies and priorities, according to the following rules:

- If the process has a nonzero RLIMIT_RTPRIO soft limit, then it can make arbitrary changes to its scheduling policy and priority, subject to the constraint that the upper limit on the realtime priority that it may set is the maximum of its current realtime priority (if the process is currently operating under a realtime policy) and the value of its RLIMIT_RTPRIO soft limit.
- If the value of a process's RLIMIT_RTPRIO soft limit is 0, then the only change that it can make is to lower its realtime scheduling priority or to switch from a realtime policy to a nonrealtime policy.
- The SCHED_IDLE policy is special. A process that is operating under this policy can't make any changes to its policy, regardless of the value of the RLIMIT_RTPRIO resource limit.
- Policy and priority changes can also be performed from another unprivileged process, as long as the effective user ID of that process matches either the real or effective user ID of the target process.
- A process's soft RLIMIT_RTPRIO limit determines only what changes can be made to its own scheduling policy and priority, either by the process itself or by another unprivileged process. A nonzero limit doesn't give an unprivileged process the ability to change the scheduling policy and priority of other processes.

> Starting with kernel 2.6.25, Linux adds the concept of realtime scheduling groups, configurable via the CONFIG_RT_GROUP_SCHED kernel option, which also affect the changes that can be made when setting realtime scheduling policies. See the kernel source file Documentation/scheduler/sched-rt-group.txt for details.

Retrieving scheduling policies and priorities

The *sched_getscheduler()* and *sched_getparam()* system calls retrieve the scheduling policy and priority of a process.

```
#include <sched.h>

int sched_getscheduler(pid_t pid);
```

Returns scheduling policy, or −1 on error

```
int sched_getparam(pid_t pid, struct sched_param *param);
```

Returns 0 on success, or −1 on error

For both of these system calls, *pid* specifies the ID of the process about which information is to be retrieved. If *pid* is 0, information is retrieved about the calling process. Both system calls can be used by an unprivileged process to retrieve information about any process, regardless of credentials.

The *sched_getparam()* system call returns the realtime priority of the specified process in the *sched_priority* field of the *sched_param* structure pointed to by *param*.

Upon successful execution, *sched_getscheduler()* returns one of the policies shown earlier in Table 35-1.

The program in Listing 35-3 uses *sched_getscheduler()* and *sched_getparam()* to retrieve the policy and priority of all of the processes whose process IDs are given as command-line arguments. The following shell session demonstrates the use of this program, as well as the program in Listing 35-2:

```
$ su                          Assume privilege so we can set realtime policies
Password:
# sleep 100 &                 Create a process
[1] 2006
# ./sched_view 2006           View initial policy and priority of sleep process
2006: OTHER  0
# ./sched_set f 25 2006       Switch process to SCHED_FIFO policy, priority 25
# ./sched_view 2006           Verify change
2006: FIFO  25
```

Listing 35-3: Retrieving process scheduling policies and priorities

procpri/sched_view.c

```
#include <sched.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int j, pol;
    struct sched_param sp;

    for (j = 1; j < argc; j++) {
        pol = sched_getscheduler(getLong(argv[j], 0, "pid"));
        if (pol == -1)
            errExit("sched_getscheduler");

        if (sched_getparam(getLong(argv[j], 0, "pid"), &sp) == -1)
            errExit("sched_getparam");

        printf("%s: %-5s %2d\n", argv[j],
                (pol == SCHED_OTHER) ? "OTHER" :
                (pol == SCHED_RR) ? "RR" :
                (pol == SCHED_FIFO) ? "FIFO" :
#ifdef SCHED_BATCH              /* Linux-specific */
                (pol == SCHED_BATCH) ? "BATCH" :
#endif
#ifdef SCHED_IDLE               /* Linux-specific */
                (pol == SCHED_IDLE) ? "IDLE" :
#endif
                "???", sp.sched_priority);
    }

    exit(EXIT_SUCCESS);
}
```

procpri/sched_view.c

Preventing realtime processes from locking up the system

Since SCHED_RR and SCHED_FIFO processes preempt any lower-priority processes (e.g., the shell under which the program is run), when developing applications that use these policies, we need to be aware of the possibility that a runaway realtime process could lock up the system by hogging the CPU. Programmatically, there are a few of ways to avoid this possibility:

- Establish a suitably low soft CPU time resource limit (RLIMIT_CPU, described in Section 36.3) using *setrlimit()*. If the process consumes too much CPU time, it will be sent a SIGXCPU signal, which kills the process by default.
- Set an alarm timer using *alarm()*. If the process continues running for a wall clock time that exceeds the number of seconds specified in the *alarm()* call, then it will be killed by a SIGALRM signal.
- Create a watchdog process that runs with a high realtime priority. This process can loop repeatedly, sleeping for a specified interval, and then waking and monitoring the status of other processes. Such monitoring could include measuring the value of the CPU time clock for each process (see the discussion of the *clock_getcpuclockid()* function in Section 23.5.3) and checking its scheduling policy and priority using *sched_getscheduler()* and *sched_getparam()*. If a process is deemed to be misbehaving, the watchdog thread could lower the process's priority, or stop or terminate it by sending an appropriate signal.
- Since kernel 2.6.25, Linux provides a nonstandard resource limit, RLIMIT_RTTIME, for controlling the amount of CPU time that can be consumed in a single burst by a process running under a realtime scheduling policy. Specified in microseconds, RLIMIT_RTTIME limits the amount of CPU time that the process may consume without performing a system call that blocks. When the process does perform such a call, the count of consumed CPU time is reset to 0. The count of consumed CPU time is not reset if the process is preempted by a higher-priority process, is scheduled off the CPU because its time slice expired (for a SCHED_RR process), or calls *sched_yield()* (Section 35.3.3). If the process reaches its limit of CPU time, then, as with RLIMIT_CPU, it will be sent a SIGXCPU signal, which kills the process by default.

  > The changes in kernel 2.6.25 can also help prevent runaway realtime processes from locking up the system. For details, see the kernel source file Documentation/scheduler/sched-rt-group.txt.

Preventing child processes from inheriting privileged scheduling policies

Linux 2.6.32 added SCHED_RESET_ON_FORK as a value that can be specified in *policy* when calling *sched_setscheduler()*. This is a flag value that is ORed with one of the policies in Table 35-1. If this flag is set, then children that are created by this process using *fork()* do not inherit privileged scheduling policies and priorities. The rules are as follows:

- If the calling process has a realtime scheduling policy (SCHED_RR or SCHED_FIFO), then the policy in child processes is reset to the standard round-robin time-sharing policy, SCHED_OTHER.
- If the process has a negative (i.e., high) nice value, then the nice value in child processes is reset to 0.

The SCHED_RESET_ON_FORK flag was designed to be used in media-playback applications. It permits the creation of single processes that have realtime scheduling policies that can't be passed to child processes. Using the SCHED_RESET_ON_FORK flag prevents the creation of fork bombs that try to evade the ceiling set by the RLIMIT_RTTIME resource limit by creating multiple children running under realtime scheduling policies.

Once the SCHED_RESET_ON_FORK flag has been enabled for a process, only a privileged process (CAP_SYS_NICE) can disable it. When a child process is created, its reset-on-fork flag is disabled.

35.3.3 Relinquishing the CPU

A realtime process may voluntarily relinquish the CPU in two ways: by invoking a system call that blocks the process (e.g., a *read()* from a terminal) or by calling *sched_yield()*.

```
#include <sched.h>

int sched_yield(void);
```

Returns 0 on success, or −1 on error

The operation of *sched_yield()* is simple. If there are any other queued runnable processes at the same priority level as the calling process, then the calling process is placed at the back of the queue, and the process at the head of the queue is scheduled to use the CPU. If no other runnable processes are queued at this priority, then *sched_yield()* does nothing; the calling process simply continues using the CPU.

Although SUSv3 permits a possible error return from *sched_yield()*, this system call always succeeds on Linux, as well as on many other UNIX implementations. Portable applications should nevertheless always check for an error return.

The use of *sched_yield()* for nonrealtime processes is undefined.

35.3.4 The SCHED_RR Time Slice

The *sched_rr_get_interval()* system call enables us to find out the length of the time slice allocated to a SCHED_RR process each time it is granted use of the CPU.

```
#include <sched.h>

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

Returns 0 on success, or −1 on error

As with the other process scheduling system calls, *pid* identifies the process about which we want to obtain information, and specifying *pid* as 0 means the calling process. The time slice is returned in the *timespec* structure pointed to by *tp*:

```
struct timespec {
    time_t tv_sec;          /* Seconds */
    long   tv_nsec;         /* Nanoseconds */
};
```

On recent 2.6 kernels, the realtime round-robin time slice is 0.1 seconds.

35.4 CPU Affinity

When a process is rescheduled to run on a multiprocessor system, it doesn't necessarily run on the same CPU on which it last executed. The usual reason it may run on another CPU is that the original CPU is already busy.

When a process changes CPUs, there is a performance impact: in order for a line of the process's data to be loaded into the cache of the new CPU, it must first be invalidated (i.e., either discarded if it is unmodified, or flushed to main memory if it was modified), if present in the cache of the old CPU. (To prevent cache inconsistencies, multiprocessor architectures allow data to be kept in only one CPU cache at a time.) This invalidation costs execution time. Because of this performance impact, the Linux (2.6) kernel tries to ensure *soft* CPU affinity for a process—wherever possible, the process is rescheduled to run on the same CPU.

> A *cache line* is the cache analog of a page in a virtual memory management system. It is the size of the unit used for transfers between the CPU cache and main memory. Typical line sizes range from 32 to 128 bytes. For further information, see [Schimmel, 1994] and [Drepper, 2007].
>
> One of the fields in the Linux-specific `/proc/PID/stat` file displays the number of the CPU on which a process is currently executing or last executed. See the *proc(5)* manual page for details.

Sometimes, it is desirable to set *hard* CPU affinity for a process, so that it is explicitly restricted to always running on one, or a subset, of the available CPUs. Among the reasons we may want to do this are the following:

- We can avoid the performance impacts caused by invalidation of cached data.
- If multiple threads (or processes) are accessing the same data, then we may obtain performance benefits by confining them all to the same CPU, so that they don't contend for the data and thus cause cache misses.
- For a time-critical application, it may be desirable to confine most processes on the system to other CPUs, while reserving one or more CPUs for the time-critical application.

> The *isolcpus* kernel boot option can be used to isolate one or more CPUs from the normal kernel scheduling algorithms. The only way to move a process on or off a CPU that has been isolated is via the CPU affinity system calls described in this section. The *isolcpus* boot option is the preferred method of implementing the last of the scenarios listed above. For details, see the kernel source file `Documentation/kernel-parameters.txt`.
>
> Linux also provides a *cpuset* kernel option, which can be used on systems containing large numbers of CPUs to achieve more sophisticated control over how the CPUs and memory are allocated to processes. For details, see the kernel source file `Documentation/cpusets.txt`.

Linux 2.6 provides a pair of nonstandard system calls to modify and retrieve the hard CPU affinity of a process: *sched_setaffinity()* and *sched_getaffinity()*.

> Many other UNIX implementations provide interfaces for controlling CPU affinity. For example, HP-UX and Solaris provide a *pset_bind()* system call.

The *sched_setaffinity()* system call sets the CPU affinity of the process specified by *pid*. If *pid* is 0, the CPU affinity of the calling process is changed.

```
#define _GNU_SOURCE
#include <sched.h>

int sched_setaffinity(pid_t pid, size_t len, cpu_set_t *set);
                                                    Returns 0 on success, or –1 on error
```

The CPU affinity to be assigned to the process is specified in the *cpu_set_t* structure pointed to by *set*.

> CPU affinity is actually a per-thread attribute that can be adjusted independently for each of the threads in a thread group. If we want to change the CPU affinity of a specific thread in a multithreaded process, we can specify *pid* as the value returned by a call to *gettid()* in that thread. Specifying *pid* as 0 means the calling thread.

Although the *cpu_set_t* data type is implemented as a bit mask, we should treat it as an opaque structure. All manipulations of the structure should be done using the macros CPU_ZERO(), CPU_SET(), CPU_CLR(), and CPU_ISSET().

```
#define _GNU_SOURCE
#include <sched.h>

void CPU_ZERO(cpu_set_t *set);
void CPU_SET(int cpu, cpu_set_t *set);
void CPU_CLR(int cpu, cpu_set_t *set);

int CPU_ISSET(int cpu, cpu_set_t *set);
                                Returns true (1) if cpu is in set, or false (0) otherwise
```

These macros operate on the CPU set pointed to by *set* as follows:

- CPU_ZERO() initializes *set* to be empty.
- CPU_SET() adds the CPU *cpu* to *set*.
- CPU_CLR() removes the CPU *cpu* from *set*.
- CPU_ISSET() returns true if the CPU *cpu* is a member of *set*.

> The GNU C library also provides a number of other macros for working with CPU sets. See the *CPU_SET(3)* manual page for details.

The CPUs in a CPU set are numbered starting at 0. The <sched.h> header file defines the constant CPU_SETSIZE to be one greater than the maximum CPU number that can be represented in a *cpu_set_t* variable. CPU_SETSIZE has the value 1024.

The *len* argument given to *sched_setaffinity()* should specify the number of bytes in the *set* argument (i.e., *sizeof(cpu_set_t)*).

The following code confines the process identified by *pid* to running on any CPU other than the first CPU of a four-processor system:

```
cpu_set_t set;

CPU_ZERO(&set);
CPU_SET(1, &set);
CPU_SET(2, &set);
CPU_SET(3, &set);

sched_setaffinity(pid, CPU_SETSIZE, &set);
```

If the CPUs specified in *set* don't correspond to any CPUs on the system, then *sched_setaffinity()* fails with the error `EINVAL`.

If *set* doesn't include the CPU on which the calling process is currently running, then the process is migrated to one of the CPUs in *set*.

An unprivileged process may set the CPU affinity of another process only if its effective user ID matches the real or effective user ID of the target process. A privileged (`CAP_SYS_NICE`) process may set the CPU affinity of any process.

The *sched_getaffinity()* system call retrieves the CPU affinity mask of the process specified by *pid*. If *pid* is 0, the CPU affinity mask of the calling process is returned.

```
#define _GNU_SOURCE
#include <sched.h>

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

Returns 0 on success, or −1 on error

The CPU affinity mask is returned in the *cpu_set_t* structure pointed to by *set*. The *len* argument should be set to indicate the number of bytes in this structure (i.e., *sizeof(cpu_set_t)*). We can use the `CPU_ISSET()` macro to determine which CPUs are in the returned *set*.

If the CPU affinity mask of the target process has not otherwise been modified, *sched_getaffinity()* returns a set containing all of the CPUs on the system.

No permission checking is performed by *sched_getaffinity()*; an unprivileged process can retrieve the CPU affinity mask of any process on the system.

A child process created by *fork()* inherits its parent's CPU affinity mask, and this mask is preserved across an *exec()*.

The *sched_setaffinity()* and *sched_getaffinity()* system calls are Linux-specific.

> The `t_sched_setaffinity.c` and `t_sched_getaffinity.c` programs in the `procpri` subdirectory in the source code distribution for this book demonstrate the use of *sched_setaffinity()* and *sched_getaffinity()*.

35.5 Summary

The default kernel scheduling algorithm employs a round-robin time-sharing policy. By default, all processes have equal access to the CPU under this policy, but we can set a process's nice value to a number in the range −20 (high priority) to +19 (low priority) to cause the scheduler to favor or disfavor that process. However, even if we give a process the lowest priority, it is not completely starved of the CPU.

Linux also implements the POSIX realtime scheduling extensions. These allow an application to precisely control the allocation of the CPU to processes. Processes operating under the two realtime scheduling policies, `SCHED_RR` (round-robin) and `SCHED_FIFO` (first-in, first-out), always have priority over processes operating under nonrealtime policies. Realtime processes have priorities in the range 1 (low) to 99 (high). As long as it is runnable, a higher-priority process completely excludes lower-priority processes from the CPU. A process operating under the `SCHED_FIFO` policy maintains exclusive access to the CPU until either it terminates, it voluntarily relinquishes the CPU, or it is preempted because a higher-priority process became runnable. Similar rules apply to the `SCHED_RR` policy, with the addition that if multiple processes are running at the same priority, then the CPU is shared among these processes in a round-robin fashion.

A process's CPU affinity mask can be used to restrict the process to running on a subset of the CPUs available on a multiprocessor system. This can improve the performance of certain types of applications.

Further information

[Love, 2010] provides background detail on process priorities and scheduling on Linux. [Gallmeister, 1995] provides further information about the POSIX realtime scheduling API. Although targeted at POSIX threads, much of the discussion of the realtime scheduling API in [Butenhof, 1996] is useful background to the realtime scheduling discussion in this chapter.

For further information about CPU affinity and controlling the allocation of threads to CPUs and memory nodes on multiprocessor systems, see the kernel source file `Documentation/cpusets.txt`, and the *mbind(2)*, *set_mempolicy(2)*, and *cpuset(7)* manual pages.

35.6 Exercises

35-1. Implement the *nice(1)* command.

35-2. Write a set-user-ID-*root* program that is the realtime scheduling analog of *nice(1)*. The command-line interface of this program should be as follows:

```
# ./rtsched policy priority command arg...
```

In the above command, *policy* is *r* for `SCHED_RR` or *f* for `SCHED_FIFO`. This program should drop its privileged ID before execing the command, for the reasons described in Sections 9.7.1 and 38.3.

35-3. Write a program that places itself under the `SCHED_FIFO` scheduling policy and then creates a child process. Both processes should execute a function that causes the process to consume a maximum of 3 seconds of CPU time. (This can be done by using a loop in which the *times()* system call is repeatedly called to determine the amount of CPU time so far consumed.) After each quarter of a second of consumed CPU time, the function should print a message that displays the process ID and the amount of CPU time so far consumed. After each second of consumed CPU time, the function should call *sched_yield()* to yield the CPU to the other process. (Alternatively, the processes could raise each other's scheduling priority using *sched_setparam()*.) The program's output should demonstrate that the two processes alternately execute for 1 second of CPU time. (Note carefully the advice given in Section 35.3.2 about preventing a runaway realtime process from hogging the CPU.)

35-4. If two processes use a pipe to exchange a large amount of data on a multiprocessor system, the communication should be faster if the processes run on the same CPU than if they run on different CPUs. The reason is that when the two processes run on the same CPU, the pipe data will be more quickly accessed because it can remain in that CPU's cache. By contrast, when the processes run on separate CPUs, the benefits of the CPU cache are lost. If you have access to a multiprocessor system, write a program that uses *sched_setaffinity()* to demonstrate this effect, by forcing the processes either onto the same CPUs or onto different CPUs. (Chapter 44 describes the use of pipes.)

> The advantage in favor of processes running on the same CPU won't hold true on hyperthreaded systems and on some modern multiprocessor architectures where the CPUs do share the cache. In these cases, the advantage will be in favor of processes running on different CPUs. Information about the CPU topology of a multiprocessor system can be obtained by inspecting the contents of the Linux-specific `/proc/cpuinfo` file.