Process Scheduler 3

The UNIX system scheduler determines when processes run. It maintains process priorities based on configuration parameters, process behavior, and user requests; it uses these priorities to assign processes to the CPU.

This chapter describes the process scheduler for the process model. See the *Multithreaded Programming Guide* for scheduler information under the multithreading model.This chapter is addressed to programmers who need more control over order of process execution than they get using default scheduler parameters.

The SunOS 5.x system gives users absolute control over the order in which certain processes run and the amount of time each process can use the CPU before another process gets a chance.

By default, the scheduler uses a time-sharing policy. A time-sharing policy adjusts process priorities dynamically to provide good response time to interactive processes and good throughput to processes that use a lot of CPU time.

The SunOS 5.x system scheduler offers a real-time scheduling policy as well as a time-sharing policy. Real-time scheduling allows users to set fixed priorities on a per-process basis. The highest-priority real-time user process always gets the CPU as soon as the process is runnable, even if system processes are runnable. A program can therefore specify the order in which processes run.

A program can also be written so that its real-time processes have a guaranteed response time from the system. See Chapter 7, "Realtime Programming and Administration" for detailed information.

For most UNIX environments, the default scheduler configuration works well and no real-time processes are needed. Administrators should not change configuration parameters and users should not change scheduler properties of their processes. However, when the requirements for a program include strict timing constraints, real-time processes sometimes provide the only way to satisfy those constraints.

Note – Real-time processes used carelessly can have a dramatically negative effect on the performance of time-sharing processes.

Because changes in scheduler administration can affect scheduler behavior, programmers might also need to know something about scheduler administration.

There are a few reference manual entries with information on scheduler administration:

- `dispadmin`(1M) tells how to change scheduler configuration in a running system.
- `ts_dptbl`(4) and `rt_dptbl`(4) describe the time-sharing and real-time parameter tables that are used to configure the scheduler.

The rest of this chapter is organized as follows.

- The *"Overview of the Process Scheduler"* tells what the scheduler does and how it does it. It also introduces scheduler classes.
- The *"Commands and Functions"* section describes and gives examples of the `priocntl`(1) command and the `priocntl`(2) and `priocntlset`(2) functions, which are the user interfaces to scheduler services. The `priocntl` functions allow you to retrieve scheduler parameters for a process or for a set of processes.
- *"Interaction with Other Functions"* describes the interactions between the scheduler and related functions.
- The *"Performance"* section discusses scheduler latencies about which some programs must be aware.

Overview of the Process Scheduler

Figure 3-1 shows how the SunOS 5.x process scheduler works:

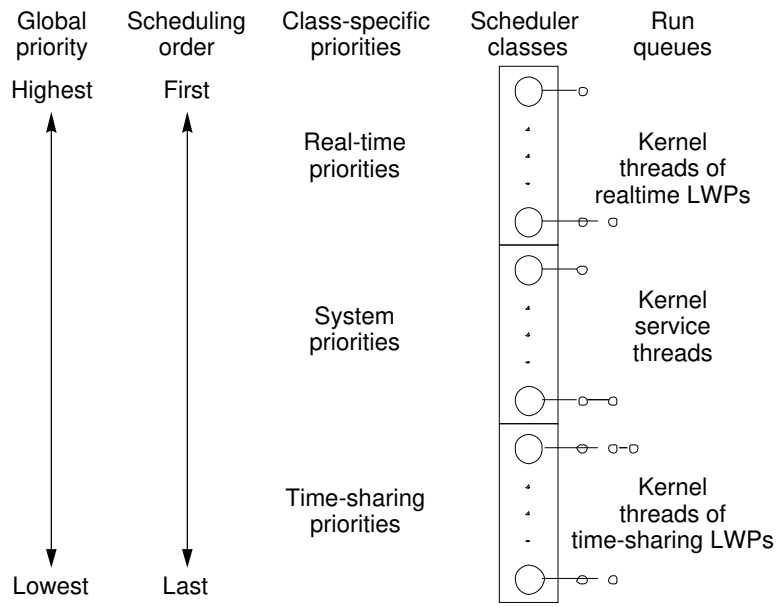


Figure 3-1 SunOS 5.x Process Scheduler

When a process is created, it inherits its scheduler parameters, including scheduler class and a priority within that class. A process changes class only as a result of a user request. The system manages the priority of a process based on user requests and a policy associated with the scheduler class of the process.

In the default configuration, the initialization process belongs to the time-sharing class. Because processes inherit their scheduler parameters, all user login shells begin as time-sharing processes in the default configuration.

The scheduler converts class-specific priorities into global priorities. The global priority of a process determines when it runs—the scheduler always runs the runnable process with the highest global priority. Numerically higher priorities

run first. Once the scheduler assigns a process to the CPU, the process runs until it uses up its time slice, sleeps, or is preempted by a higher-priority process. Processes with the same priority run round-robin.

Administrators specify default time slices in the configuration tables, but users can assign per-process time slices to real-time processes.

You can display the global priority of a process with the `-cl` options of the `ps`(1) command. You can display configuration information about class-specific priorities with the `priocntl`(1) command and the `dispadmin`(1M) command.

By default, all real-time processes have higher priorities than any kernel process, and all kernel processes have higher priorities than any time-sharing process.

Note – As long as there is a runnable real-time process, no kernel process and no time-sharing process run.

The following sections describe the scheduling policies of the three default classes.

Time-Sharing Class

The goal of the time-sharing policy is to provide good response time to interactive processes and good throughput to CPU-bound processes. The scheduler switches CPU allocation frequently enough to provide good response time, but not so frequently that it spends too much time doing the switching. Time slices are typically on the order of a few hundred milliseconds.

The time-sharing policy changes priorities dynamically and assigns time slices of different lengths. The scheduler raises the priority of a process that sleeps after only a little CPU use (a process sleeps, for example, when it starts an I/O operation such as a terminal read or a disk read); frequent sleeps are characteristic of interactive tasks such as editing and running simple shell commands. On the other hand, the time-sharing policy lowers the priority of a process that uses the CPU for long periods without sleeping.

The default time-sharing policy gives larger time slices to processes with lower priorities. A process with a low priority is likely to be CPU-bound. Other processes get the CPU first, but when a low-priority process finally gets the CPU, it gets a bigger chunk of time. If a higher-priority process becomes runnable during a time slice, however, it preempts the running process.

The scheduler manages time-sharing processes using configurable parameters in the time-sharing parameter table `ts_dptbl`. This table contains information specific to the time-sharing class.

System Class

The system class uses a fixed-priority policy to run kernel processes such as servers and housekeeping processes like the paging daemon. The system class is reserved for use by the kernel; users can neither add nor remove a process from the system class. Priorities for system class processes are set up in the kernel code for those processes; once established, the priorities of system processes do not change. (User processes running in kernel mode are not in the system class.)

Real-time Class

The real-time class uses a fixed-priority scheduling policy so that critical processes can run in predetermined order. Real-time priorities never change except when a user requests a change. Contrast this fixed-priority policy with the time-sharing policy, in which the system changes priorities to provide good interactive response time.

Privileged users can use the `priocntl` command or the `priocntl` function to assign real-time priorities.

The scheduler manages real-time processes using configurable parameters in the real-time parameter table `rt_dptbl`. This table contains information specific to the real-time class.

Commands and Functions

Here is a programmer's view of default process priorities.

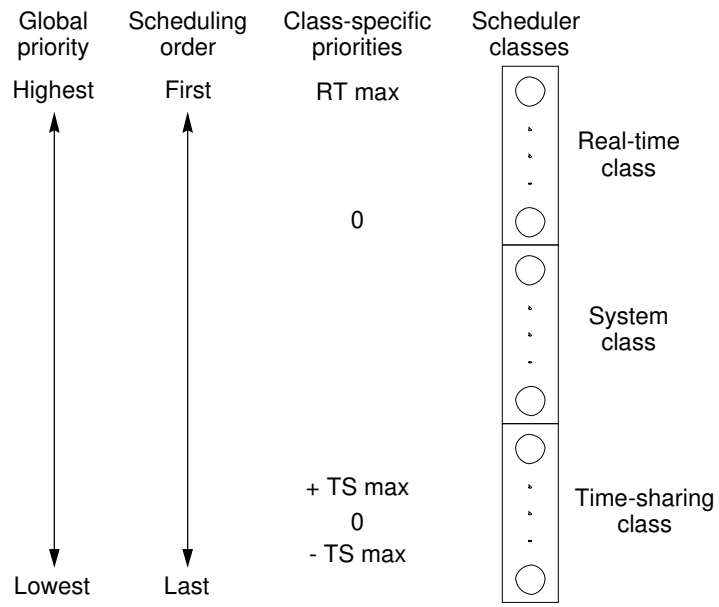


Figure 3-2 Process Priorities (Programmer's View)

From a user's or programmer's point of view, a process priority has meaning only in the context of a scheduler class. You specify a process priority by specifying a class and a class-specific priority value. The class and class-specific value are mapped by the system into a global priority that the system uses to schedule processes.

- Real-time priorities run from zero to a configuration-dependent maximum. The system maps them directly into global priorities. They never change except when a user changes them.
- System priorities are controlled entirely in the kernel. Users cannot affect them.
- Time-sharing priorities have a user-controlled component (the "user priority") and a component controlled by the system. The system does not change the user priority except as the result of a user request. The system changes the system-controlled component dynamically on a per-process

basis to provide good overall system performance; users cannot affect the system-controlled component. The scheduler combines these two components to get the process global priority.

The user priority runs from the negative of a configuration-dependent maximum to the positive of that maximum. A process inherits its user priority. Zero is the default initial user priority.

The "user priority limit" is the configuration-dependent maximum value of the user priority. You can set a user priority to any value below the user priority limit. With appropriate permission, you can raise the user priority limit. Zero is the default user priority limit.

You can lower the user priority of a process to give the process reduced access to the CPU or, with the appropriate permission, raise the user priority to get better service. Because you cannot set the user priority above the user priority limit, you must raise the user priority limit before you raise the user priority if both have their default values of zero.

An administrator configures the maximum user priority independent of global time-sharing priorities. In the default configuration, for example, a user can set a user priority only in the range from –20 to +20, but 60 time-sharing global priorities are configured.

A system administrator's view of priorities is different from that of a user or programmer. When configuring scheduler classes, an administrator deals directly with global priorities. The system maps priorities supplied by users into these global priorities. See *System Administration Guide, Volume I* for more information about priorities.

The `ps -cel` command reports global priorities for all active processes. The `priocntl` command reports the class-specific priorities that users and programmers use.

Note – Global process priorities and user-supplied priorities are in ascending order: numerically higher priorities run first.

The `priocntl`(1) command and the `priocntl`(2) and `priocntlset`(2) functions set or retrieve scheduler parameters for processes. The basic idea for setting priorities is the same for all three functions:

- Specify the target processes.

- Specify the scheduler parameters you want for those processes.
- Do the command or function to set the parameters for the processes.

You specify the target processes using an ID type and an ID. The ID type tells how to interpret the ID. [This concept of a set of processes applies to signals as well as to the scheduler; see `sigsend`(2).] The following table lists the valid ID types that you can specify.

Table 3-1 Valid `priocntl` ID Types

`priocntl` **ID types**
process ID
parent process ID
process group ID
session ID
class ID
effective user ID
effective group ID
all processes

These IDs are basic properties of UNIX processes. [See `intro`(2).] The class ID refers to the scheduler class of the process. `priocntl` works only for the time-sharing and the real-time classes, not for the system class. Processes in the system class have fixed priorities assigned when they are started by the kernel.

The `priocntl` *Command*

The `priocntl` command comes in four forms:

- `priocntl -l` displays configuration information.
- `priocntl -d` displays the scheduler parameters of processes.
- `priocntl -s` sets the scheduler parameters of processes.
- `priocntl -e` executes a command with the specified scheduler parameters.

Here is the output of the `-l` option for the default configuration.

```
$ priocntl -l
CONFIGURED CLASSES
==================

SYS (System Class)

TS (Time Sharing)
Configured TS User Priority Range: -20 through 20

RT (Real Time)
Maximum Configured RT Priority: 59
```

The `-d` option displays the scheduler parameters of a process or a set of processes. The syntax for this option is

`priocntl -d -i` *`idtype idlist`*

idtype tells what kind of IDs are in *idlist*. *idlist* is a list of IDs separated by white space. Here are the valid values for *idtype* and their corresponding ID types in *idlist*:

Table 3-2 Valid *idtype* Values

idtype	***idlist***
`pid`	process IDs
`ppid`	parent process IDs
`pgid`	process group IDs
`sid`	session IDs
`class`	class names (TS or RT)
`uid`	effective user IDs
`gid`	effective group IDs
`all`	

Here are some examples of the `-d` option of `priocntl`.

Display information on all processes

```
$ priocntl -d -i all
          .
          .
          .
```

Display information on all time-sharing processes

```
$ priocntl -d -i class TS
          .
          .
          .
```

Display information on all processes with user ID 103 or 6626

```
$ priocntl -d -i uid 103 6626
          .
          .
          .
```

The `-s` option sets scheduler parameters for a process or a set of processes. The syntax for this option is

```
priocntl -s -c class class_options -i idtype idlist
```

idtype and *idlist* are the same as for the `-d` option described above.

class is `TS` for time-sharing or `RT` for real-time. You must be superuser to create a real-time process, to raise a time-sharing user priority above a per-process limit, or to raise the per-process limit above zero. Class options are class-specific:

Table 3-3 Class-Specific Options for `priocntl`

Class-specific options for `priocntl`			
class	-c *class*	Options	Meaning
real-time	RT	-p *pri*	priority
		-t *tslc*	time slice
		-r *res*	resolution
time-sharing	TS	-p *upri*	user priority
		-m *uprilim*	user priority limit

For a real-time process you can assign a priority and a time slice.

- The priority is a number from 0 to the real-time maximum as reported by `priocntl -l`; the default maximum value is 59.
- You specify the time slice as a number of clock intervals and the resolution of the interval. Resolution is specified in intervals per second. The time slice, therefore, is *tslc/res* seconds. To specify a time slice of one-tenth of a second, for example, you could specify a *tslc* of 1 and a *res* of 10. If you specify a time slice without specifying a resolution, millisecond resolution (a *res* of 1000) is assumed.

If you change a time-sharing process into a real-time process, it gets a default priority and time slice if you don't specify one. To change only the priority of a real-time process and leave its time slice unchanged, omit the `-t` option. To change only the time slice of a real-time process and leave its priority unchanged, omit the `-p` option.

For a time-sharing process you can assign a user priority and a user priority limit.

- The user priority is the user-controlled component of a time-sharing priority. The scheduler calculates the global priority of a time-sharing process by combining this user priority with a system-controlled component that depends on process behavior. The user priority has the same effect as a value set by `nice` (except that `nice` uses higher numbers for lower priority).
- The user priority limit is the maximum user priority a process can set for itself without being superuser. By default, the user priority limit is 0; you must be superuser to set a user priority limit above 0.

Both the user priority and the user priority limit must be within the user priority range reported by the `priocntl -l` command. The default range is -20 to +20.

You can lower and raise a process user priority as often as you like, as long as the value is below the process user priority limit. It is a courtesy to other users to lower your user priority for big chunks of low-priority work. On the other hand, if you lower your user priority limit, you must be superuser to raise it. A typical use of the user priority limit is to reduce permanently the priority of child processes or of some other set of low-priority processes.

The user priority can never be greater than the user priority limit. If you set the user priority limit below the user priority, the user priority is lowered to the new user priority limit. If you attempt to set the user priority above the user priority limit, the user priority is set to the user priority limit.

Here are some examples of the `-s` option of `priocntl`:

Make the process with ID 24668 a real-time process with default parameters

```
$ priocntl -s -c RT -i pid 24668
```

Make 3608 RT with priority 55 and a one-fifth second time slice.

```
$ priocntl -s -c RT -p 55 -t 1 -r 5 -i pid 3608
```

Change all processes into time-sharing processes

```
$ priocntl -s -c TS -i all
```

For uid 1122, reduce TS user priority and user priority limit to -10

```
$ priocntl -s -c TS -p -10 -m -10 -i uid 1122
```

The –e option sets scheduler parameters for a specified command and executes the command. The syntax for this option is

```
priocntl -e -c class class_options command [command arguments]
```

The class and class options are the same as for the `-s` option described above.

Start a real-time shell with default real-time priority

```
$ priocntl -e -c RT /bin/sh
```

Run make with a time-sharing user priority of -10.

```
$ priocntl -e -c TS -p -10 make bigprog
```

The `priocntl` command subsumes the function of `nice`. `nice` works only on time-sharing processes and uses higher numbers to assign lower priorities. The example above is equivalent to using `nice` to set an “increment” of 10

```
$ nice -10 make bigprog
```

The `priocntl` *Function*

```
#include   <sys/types.h>
#include   <sys/procset.h>
#include   <sys/priocntl.h>
#include   <sys/rtpriocntl.h>
#include   <sys/tspriocntl.h>
```

```
long priocntl(idtype_t idtype, id_t id, int cmd,
    cmd_struct  arg);
```

The `priocntl` function gets or sets the scheduler parameters of a set of processes. The input arguments follow.

- `idtype` is the type of ID you are specifying.
- `id` is the ID.
- `cmd` specifies which `priocntl` function to perform. The functions are listed in Table 3-4.
- `arg` is a pointer to a structure that depends on `cmd`.

Here are the valid values for *idtype*, which are defined in `priocntl.h`, and their corresponding ID types in *id*:

Table 3-4 Valid `priocntl.h` *idtypes*

idtype	Interpretation of id
P_PID	process ID (of a single process)
P_PPID	parent process ID
P_PGID	process group ID
P_SID	session ID
P_CID	class ID
P_UID	effective user ID
P_GID	effective group ID
P_ALL	all processes

Here are the valid values for `cmd`, their meanings, and the type of `arg`:

Table 3-5 Valid `cmd` Values

priocntl Commands		
cmd	**arg Type**	**Function**
PC_GETCID	pcinfo_t	get class ID and attributes

Table 3-5 Valid `cmd` Values

priocntl Commands		
cmd	**arg Type**	**Function**
`PC_GETCLINFO`	`pcinfo_t`	get class name and attributes
`PC_SETPARMS`	`pcparms_t`	set class and scheduling parameters
`PC_GETPARMS`	`pcparms_t`	get class and scheduling parameters

Here are the values `priocntl` returns on success:

- The `GETCID` and `GETCLINFO` commands return the number of configured scheduler classes.
- `PC_SETPARMS` returns 0.
- `PC_GETPARMS` returns the process ID of the process whose scheduler properties it is returning.

On failure, `priocntl` returns –1 and sets `errno` to indicate the reason for the failure. See `priocntl`(2) for the complete list of error conditions.

PC_GETCID, PC_GETCLINFO

The `PC_GETCID` and `PC_GETCLINFO` commands retrieve scheduler parameters for a class based on the class ID or class name. Both commands use the `pcinfo` structure to send arguments and receive return values:

```
typedef struct pcinfo {
    id_t    pc_cid;/* class id */
    char    pc_clname[PC_CLNMSZ];/* class name */
    long    pc_clinfo[PC_CLINFOSZ];/* class information */
} pcinfo_t;
```

The `PC_GETCID` command gets scheduler class ID and parameters given the class name. The class ID is used in some of the other `priocntl` commands to specify a scheduler class. The valid class names are `TS` for time-sharing and `RT` for real-time.

For the real-time class, `pc_clinfo` contains an `rtinfo` structure, which holds `rt_maxpri`, the maximum valid real-time priority. In the default configuration, this is the highest priority any process can have. The minimum valid real-time priority is zero. `rt_maxpri` is a configurable value

```
typedef struct rtinfo {
    short rt_maxpri;/* maximum real-time priority */
} rtinfo_t;
```

For the time-sharing class, `pc_clinfo` contains a `tsinfo` structure, which holds `ts_maxupri`, the maximum time-sharing user priority. The minimum time-sharing user priority is `-ts_maxupri`. `ts_maxupri` is also a configurable value.

```
typedef struct tsinfo {
    short ts_maxupri;/* limits of user priority range */
} tsinfo_t;
```

The following program is a substitute for `priocntl -l`; it gets and prints the range of valid priorities for the time-sharing and real-time scheduler classes

```
/*
 * Get scheduler class IDs and priority ranges.
 */

#include <sys/types.h>
#include <sys/priocntl.h>
#include <sys/rtpriocntl.h>
#include <sys/tspriocntl.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
main ()
{
    pcinfo_t    pcinfo;
    tsinfo_t    *tsinfop;
    rtinfo_t*   rtinfop;
    short       maxtsupri, maxrtpri;

    /* time sharing */
        (void) strcpy (pcinfo.pc_clname, "TS");
        if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
            perror ("PC_GETCID failed for time-sharing class");
            exit (1);
        }
        tsinfop = (struct tsinfo *) pcinfo.pc_clinfo;
        maxtsupri = tsinfop->ts_maxupri;
        (void) printf("Time sharing: ID %ld, priority range -%d
through %d\n",
            pcinfo.pc_cid, maxtsupri, maxtsupri);
```

The following screen shows the output of this program, called `getcid` in this example.

```
$ getcid
Time sharing: ID 1, priority range -20 through 20
Real time: ID 2, priority range 0 through 59
```

The following function is useful in the examples below. Given a class name, it uses `PC_GETCID` to return the class ID and maximum priority in the class.

Note – The following examples omit the lines that include header files. The examples compile with the same header files used in the previous code example.

```
/*
 * Return class ID and maximum priority.
 * Input argument name is class name.
 * Maximum priority is returned in *maxpri.
 */

id_t
schedinfo (name, maxpri)
    char *name;
    short *maxpri;
{
    pcinfo_t    info;
    tsinfo_t    *tsinfop;
    rtinfo_     *rtinfop;

    (void) strcpy(info.pc_clname, name);
    if (priocntl (0L, 0L, PC_GETCID, &info) == -1L) {
        return (-1);
    }
    if (strcmp(name, "TS") == 0) {
        tsinfop = (struct tsinfo *) info.pc_clinfo;
        *maxpri = tsinfop->ts_maxupri;
    } else if (strcmp(name, "RT") == 0) {
        rtinfop = (struct rtinfo *) info.pc_clinfo;
        *maxpri = rtinfop->rt_maxpri;
    } else {
        return (-1);
```

```
    }
    return (info.pc_cid);
}
```

The PC_GETCLINFO command gets a scheduler class name and parameters given the class ID. This command makes it easy to write programs that make no assumptions about what classes are configured.

The following program uses PC_GETCLINFO to get the class name of a process based on the process ID. This program assumes the existence of a function getclassID, which retrieves the class ID of a process given the process ID; this function is given in the following section

```
/* Get scheduler class name given process ID. */

main (argc, argv)
    int argc;
    char *argv[];
{
    pcinfo_t    pcinfo;
    id_t        pid, classID;
    id_t        getclassID();

    if ((pid = atoi(argv[1])) <= 0) {
        perror ("bad pid");
        exit (1);
    }
    if ((classID = getclassID(pid)) == -1) {
        perror ("unknown class ID");
        exit (2);
    }
    pcinfo.pc_cid = classID;
    if (priocntl (0L, 0L, PC_GETCLINFO, &pcinfo) == -1L) {
        perror ("PC_GETCLINFO failed");
        exit (3);
    }
    (void) printf("process ID %d, class %s\n", pid,
     pcinfo.pc_clname);
}
```

PC_GETPARMS, PC_SETPARMS

The PC_GETPARMS command gets and the PC_SETPARMS command sets scheduler parameters for processes. Both commands use the pcparms structure to send arguments or receive return values:

```
typedef struct pcparms {
    id_t pc_cid;/* process class */
    long pc_clparms[PC_CLPARMSZ];/* class specific */
} pcparms_t;
```

Ignoring class-specific information for the moment, here is a simple function for returning the scheduler class ID of a process, as promised in the previous section.

```
/*
 * Return scheduler class ID of process with ID pid.
 */

getclassID (pid)
    id_t pid;
{
    pcparms_t  pcparms;

    pcparms.pc_cid = PC_CLNULL;
    if (priocntl(P_PID, pid, PC_GETPARMS, &pcparms) == -1) {
        return (-1);
    }
    return (pcparms.pc_cid);
}
```

For the real-time class, pc_clparms contains an rtparms structure. rtparms holds scheduler parameters specific to the real-time class.

```
typedef struct rtparms {
    short   rt_pri;    /* realtime priority */
    ulong   rt_tqsecs; /* seconds in time quantum */
    long    rt_tqnsecs;/* additional nsecs in quantum */
} rtparms_t;
```

`rt_pri` is the real-time priority; `rt_tqsecs` is the number of seconds and `rt_tqnsecs` is the number of additional nanoseconds in a time slice. That is, `rt_tqsecs` seconds plus `rt_tqnsecs` nanoseconds is the interval a process can use the CPU without sleeping before the scheduler gives another process a chance at the CPU.

For the time-sharing class, `pc_clparms` contains a `tsparms` structure. `tsparms` holds the scheduler parameter specific to the time-sharing class.

```
typedef struct tsparms {
    short ts_uprilim;      /* user priority limit */
    short ts_upri;         /* user priority */
} tsparms_t;
```

ts_upri is the user priority, the user-controlled component of a time-sharing priority. ts_uprilim is the user priority limit, the maximum user priority a process can set for itself without being superuser. These values are described above in the discussion of the -s option of the priocntl command. Both the user priority and the user priority limit must be within the range reported by the priocntl -l command; this range is also reported by the PC_GETCID and PC_GETCLINFO commands to the priocntl function.

The PC_GETPARMS command gets the scheduler class and parameters of a single process. The return value of the priocntl is the process ID of the process whose parameters are returned in the pcparms structure. The process chosen depends on the idtype and id arguments to priocntl and on the value of pcparms.pc_cid, which contains PC_CLNULL or a class ID returned by PC_GETCID:

Table 3-6 What PC_GETPARMS Returns

Number of Processes Selected by idtype and id	pc_cid		
	RT class ID	**TS class ID**	**PC_CLNULL**
1	RT parameters of process selected	TS parameters of process selected	class and parameters of process selected
More than 1	RT parameters of highest-priority RT process	TS parameters of process with highest user priority	(error)

If idtype and id select a single process and pc_cid does not conflict with the class of that process, priocntl returns the scheduler parameters of the process. If they select more than one process of a single scheduler class, priocntl returns parameters using class-specific criteria as shown in the table. priocntl returns an error in the following cases:

- idtype and id select one or more processes and none is in the class specified by pc_cid.
- idtype and id select more than one process and pc_cid is PC_CLNULL.
- idtype and id select no processes.

The following program takes a process ID as its input and prints the scheduler class and class-specific parameters of that process.

```
/*
 * Get scheduler class and parameters of
 * process whose pid is input argument.
 */

main (argc, argv)
        int argc;
        char *argv[];
{
        pcparms_t   pcparms;
        rtparms_t   *rtparmsp;
        tsparms_t   *tsparmsp;
        id_t        pid, rtID, tsID;
        id_t        schedinfo();
        short       priority, tsmaxpri, rtmaxpri;
        ulong       secs;
        long        nsecs;

        pcparms.pc_cid = PC_CLNULL;
        rtparmsp = (rtparms_t *) pcparms.pc_clparms;
        tsparmsp = (tsparms_t *) pcparms.pc_clparms;
        if ((pid = atoi(argv[1])) <= 0) {
            perror ("bad pid");
            exit (1);
        }
/* get scheduler properties for this pid */
...
}
```

The `PC_SETPARMS` command sets the scheduler class and parameters of a set of processes. The `idtype` and `id` input arguments specify the processes to be changed.

The `pcparms` structure contains the new parameters: `pc_cid` contains the ID of the scheduler class to which the processes are to be assigned, as returned by `PC_GETCID`; `pc_clparms` contains the class-specific parameters:

- If `pc_cid` is the real-time class ID, `pc_clparms` contains an `rtparms` structure in which `rt_pri` contains the real-time priority and `rt_tqsecs` plus `rt_tqnsecs` contains the time slice to be assigned to the processes.
- If `pc_cid` is the time-sharing class ID, `pc_clparms` contains a `tsparms` structure in which `ts_uprilim` contains the user priority limit and `ts_upri` contains the user priority to be assigned to the processes.

The following program takes a process ID as input, makes the process a real-time process with the highest valid priority minus 1, and gives it the default time slice for that priority. The program calls the `schedinfo` function listed above to get the real-time class ID and maximum priority.

```
/*
 * Input arg is proc ID. Make process a realtime
 * process with highest priority minus 1.
 */

main (argc, argv)
    int argc;
    char *argv[];
{
    pcparms_tpcparms;
    rtparms_t*rtparmsp;
    id_t    pid, rtID;
    id_t    schedinfo();
    short   maxrtpri;
    if ((pid = atoi(argv[1])) <= 0) {
        perror ("bad pid");
        exit (1);
    }

    /* Get highest valid RT priority. */
    if ((rtID = schedinfo ("RT", &maxrtpri)) == -1) {
        perror ("schedinfo failed for RT");
        exit (2);
    }

    /* Change proc to RT, highest prio - 1, default time slice */
    pcparms.pc_cid = rtID;
    rtparmsp = (struct rtparms *) pcparms.pc_clparms;
    rtparmsp->rt_pri = maxrtpri - 1;
    rtparmsp->rt_tqnsecs = RT_TQDEF;

    if (priocntl(P_PID, pid, PC_SETPARMS, &pcparms) == -1) {
        perror ("PC_SETPARMS failed");
        exit (3);
    }
}
```

The following table lists the special values `rt_tqnsecs` can take when `PC_SETPARMS` is used on real-time processes. When any of these is used, `rt_tqsecs` is ignored. These values are defined in the header file `rtpriocntl.h`.

Table 3-7 Special Values for `rt_tqnsecs`

rt_tqnsecs	**Time Slice**
RT_TQINF	infinite
RT_TQDEF	default
RT_NOCHANGE	unchanged

`RT_TQINF` specifies an infinite time slice. `RT_TQDEF` specifies the default time slice configured for the real-time priority being set with the `SETPARMS` call. `RT_NOCHANGE` specifies no change from the current time slice; this value is useful, for example, when you change process priority but do not want to change the time slice. (You can also use `RT_NOCHANGE` in the `rt_pri` field to change a time slice without changing the priority.)

The `priocntlset` *Function*

```
#include<sys/types.h>
#include<sys/signal.h>
#include<sys/procset.h>
#include<sys/priocntl.h>
#include<sys/rtpriocntl.h>
#include<sys/tspriocntl.h>

long priocntlset(procset_t *psp, int cmd, cmd_struct arg);
```

The `pri ocntlset` function changes scheduler parameters of a set of processes, just like `priocntl`. `priocntlset` has the same command set as `priocntl`; the `cmd` and `arg` input arguments are the same. But while `priocntl` applies to a set of processes specified by a single `idtype/id` pair, `priocntlset` applies to a set of processes that results from a logical combination of two `idtype/id` pairs.

The input argument `psp` points to a `procset` structure that specifies the two `idtype/id` pairs and the logical operation to perform. This structure is defined in `procset.h`.

```
typedef struct procset {
        idop_t      p_op        /* operator connecting */
                                /* left and right sets */

    /* left set: */
        idtype_t    p_lidtype;  /* left ID type */
        id_t        p_lid;      /* left ID */

    /* right set: */
        idtype_t    p_ridtype;  /* right ID type */
        id_t    p_rid;          /* right ID */
} procset_t;
```

`p_lidtype` and `p_lid` specify the ID type and ID of one ("left") set of processes; `p_ridtype` and `p_rid` specify the ID type and ID of a second ("right") set of processes. `p_op` specifies the operation to perform on the two sets of processes to get the set of processes to operate on.

The valid values for `p_op` and the processes they specify are:

- `POP_DIFF`: set difference—processes in left set and not in right set
- `POP_AND`: set intersection—processes in both left and right sets
- `POP_OR`: set union—processes in either left or right sets or both
- `POP_XOR`: set exclusive-or—processes in left or right set but not in both

The following macro, also defined in `procset.h`, offers a convenient way to initialize a `procset` structure.

```
#define setprocset(psp, op, ltype, lid, rtype, rid) \
    (psp)->p_op= (op); \
    (psp)->p_lidtype= (ltype); \
    (psp)->p_lid= (lid); \
    (psp)->p_ridtype= (rtype); \
    (psp)->p_rid= (rid);
```

Here is a situation where `priocntlset` would be useful: suppose a program had both real-time and time-sharing processes that ran under a single user ID. If the program wanted to change the priority of only its real-time processes without changing the time-sharing processes to real-time processes, it could do so as follows. (This example uses the function `schedinfo`, which is defined above in the section on `PC_GETCID`.

```
/*
 * Change real-time priorities of this uid
 * to highest realtime priority minus 1.
```

```
 */

main (argc, argv)
    int argc;
    char *argv[];
{
    procset_t      procset;
    pcparms_t      pcparms;
    struct rtparms *rtparmsp;
    id_t           rtclassID;
    id_t           schedinfo();
    short          maxrtpri;

    /* left set: select processes with same uid as this process */
    procset.p_lidtype = P_UID;
    procset.p_lid = getuid();

    /* get info on realtime class */
    if ((rtclassID = schedinfo ("RT", &maxrtpri)) == -1) {
        perror ("schedinfo failed");
        exit (1);
    }

...
}
```

`priocntl` offers a simple scheduler interface that is adequate for many applications. When a process needs a more powerful way to specify sets, use `priocntlset`.

Interaction with Other Functions

Kernel Processes

The kernel assigns its daemon and housekeeping processes to the system scheduler class. Users can neither add processes to nor remove processes from this class, nor can they change the priorities of these processes. The command `ps -cel` lists the scheduler class of all processes. Processes in the system class are identified by a `SYS` entry in the `CLS` column.

If the work load on a machine contains real-time processes that use too much CPU, they can lock out system processes, which can lead to trouble. Real-time applications must ensure that they leave some CPU time for system and other processes.

`fork` *and* `exec`

Scheduler class, priority, and other scheduler parameters are inherited across the `fork`(2) and `exec`(2) functions.

`nice`

The `nice`(1) command and the `nice`(2) function work as in previous versions of the UNIX system. They allow you to change the priority of a time-sharing process. You still use lower numeric values to assign higher time-sharing priorities with these functions.

To change the scheduler class of a process or to specify a real-time priority, you must use one of the `priocntl` functions. You use higher numeric values to assign higher priorities with the `priocntl` functions.

`init`

The `init` process is treated as a special case by the scheduler. To change the scheduler properties of `init`, `init` must be the only process specified by `idtype` and `id` or by the `procset` structure.

Performance

Because the scheduler determines when and for how long processes run, it has an overriding importance in the performance and perceived performance of a system.

By default, all processes are time-sharing processes. A process changes class only as a result of one of the `priocntl` functions.

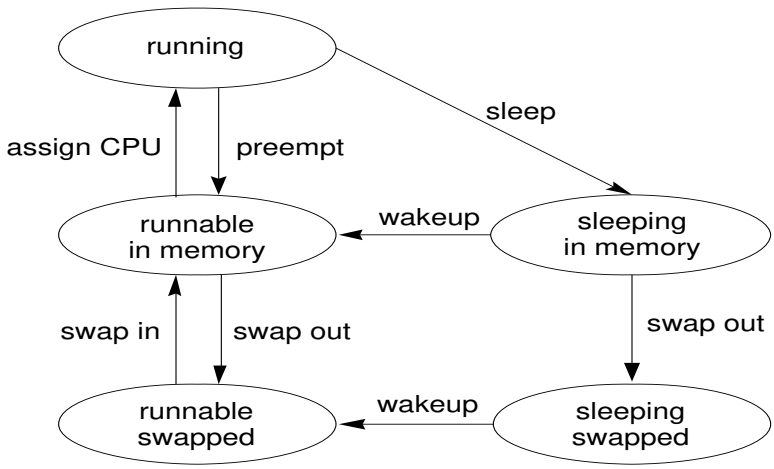
In the default configuration, all real-time process priorities are above any time-sharing process priority. This implies that as long as any real-time process is runnable, no time-sharing process or system process ever runs. So if a real-time application is not written carefully, it can completely lock out users and essential kernel housekeeping.

Besides controlling process class and priorities, a real-time application must also control several other factors that influence its performance. The most important factors in performance are CPU power, amount of primary memory, and I/O throughput. These factors interact in complex ways. In particular, the `sar`(1) command has options for reporting on all the factors discussed in this section.

Process State Transition

Applications that have strict real-time constraints might need to prevent processes from being swapped or paged out to secondary memory. Here's a simplified overview of UNIX process states and the transitions between states:

Figure 3-3 Process State Transition Diagram



An active process is normally in one of the five states in the diagram. The arrows show how it changes states.

- A process is running if it is assigned to a CPU. A process is preempted—that is, removed from the running state—by the scheduler if a process with a higher priority becomes runnable. A process is also preempted if it consumes its entire time slice and a process of equal priority is runnable.
- A process is runnable in memory if it is in primary memory and ready to run, but is not assigned to a CPU.
- A process is sleeping in memory if it is in primary memory but is waiting for a specific event before it can continue execution. For example, a process is sleeping if it is waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, the process is sent a wake up; if the reason for its sleep is gone, the process becomes runnable.
- A process is runnable and swapped if it is not waiting for a specific event but has had its whole address space written to secondary memory to make room in primary memory for other processes.
- A process is sleeping and swapped if it is both waiting for a specific event and has had its whole address space written to secondary memory to make room in primary memory for other processes.

If a machine does not have enough primary memory to hold all its active processes, it must page or swap some address space to secondary memory:

- When the system is short of primary memory, it writes individual pages of some processes to secondary memory but still leaves those processes runnable. When a process runs, if it accesses those pages, it must sleep while the pages are read back into primary memory.
- When the system gets into a more serious shortage of primary memory, it writes all the pages of some processes to secondary memory and marks those processes as swapped. Such processes get back into a state where they can be scheduled only by being chosen by the system scheduler daemon process, then read back into memory.

Both paging and swapping, and especially swapping, introduce delay when a process is ready to run again. For processes that have strict timing requirements, this delay can be unacceptable.

To avoid swapping delays, real-time processes are never swapped, though parts of them can be paged. A program can prevent paging and swapping by locking its text and data into primary memory.

For more information see `memcntl`(2). Of course, how much can be locked is limited by how much memory is configured. Also, locking too much can cause intolerable delays to processes that do not have their text and data locked into memory.

Trade-offs between performance of real-time processes and performance of other processes depend on local needs. On some systems, process locking might be required to guarantee the necessary real-time response.

Software Latencies

See “Dispatch Latency” on page 112 for information about latencies in real-time applications.