Realtime Processing 11

Introduction

The management of data acquisition or process control requires a realtime environment that can react to external events within strict timing constraints. This chapter describes the realtime environment that is provided by SunOS 5.x. The various components of this environment are described along with sample programs that are important to writing and porting realtime applications.

What is Realtime in SunOS 5.x?

A *realtime* environment may be defined as one that enables an application to respond to internal or external events in a bounded amount of time. This environment is provided in SunOS 5.x through the following features:

- static priority and fixed quantum scheduling for realtime threads
- fully preemptive kernel, providing bounded dispatch latency
- memory locking
- asynchronous I/O and networking
- interprocess communication
- high resolution timers

Using these features, a realtime process under SunOS 5.x:

- runs in the `RT` (Realtime) scheduling class and has locked down all of the memory in its process address space.
- is from a statically-linked program or from a program in which all dynamic linking has been completed before the program's `main()` is called.

Realtime operations are described in this chapter in terms of single-threaded processes, but the description can also apply to multithreaded processes. To guarantee realtime scheduling of a thread, it must be created as a bound thread, and the thread's LWP must be run in the `RT` scheduling class. The locking of memory and early dynamic binding is effective for all threads in a process. For more information on threads, see Chapter 10 in this book.

When a process is the highest priority realtime process, it:

- acquires the processor within the guaranteed dispatch latency period of becoming runnable
- continues to run for as long as it remains the highest priority runnable process.

A realtime process can lose control of the processor or can be unable to gain control of the processor because of other events on the system. These events include external events (such as interrupts), resource starvation, waiting on external events (synchronous I/O), and preemption by a higher priority process.

Realtime scheduling generally does not apply to system initialization and termination services such as `open` and `close`.

Process Scheduling

Realtime scheduling constraints can exceed the capabilities of a standard UNIX scheduler, which is designed for a time-sharing environment that treats every process equally and has a limited notion of priority. The new SunOS 5.x scheduler is one of the features that enables realtime processing. This section describes the scheduler and how to use the system calls and utilities that control scheduling.

Overview of the Process Scheduler

The *process scheduler*, or dispatcher, is the portion of the kernel that controls allocation of the CPU to processes. The scheduler supports the concept of scheduling classes where each class defines a scheduling policy, used to schedule processes within that class.

Figure 11-1 illustrates the scheduling elements of SunOS 5.x.

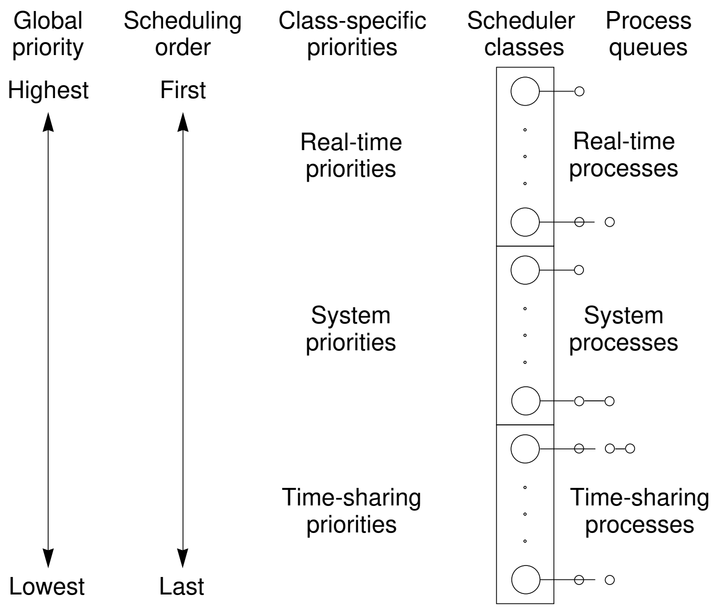

Figure 11-1 SunOS 5.x Process Scheduler

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 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.

Scheduling Classes

SunOS 5.x supports three types of scheduling classes for processes:

- Realtime (`RT`)
- System (`sys`)
- Time-Sharing (`TS`).

Each class has a unique scheduling policy for dispatching processes within its class.

The kernel dispatches highest priority processes first. By default, `RT` processes have precedence over `sys` and `TS` processes, but administrators can configure systems so that `TS` and `RT` processes have overlapping priorities.

A child process inherits the scheduling class and attributes of the parent process through `fork`; a process' scheduling class and attributes are unchanged by `exec`.

Figure 11-2 illustrates the concept of classes as viewed by the SunOS 5.x kernel.

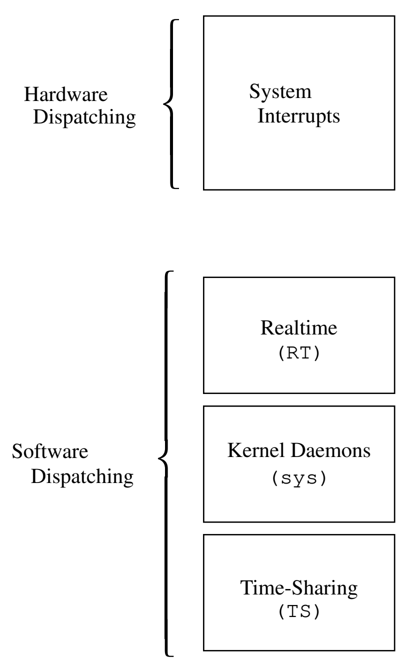

Figure 11-2 Dispatch Priorities for Scheduling Classes

Hardware interrupts have the highest priority and are not controllable from software. The interrupt processing routines are dispatched directly and immediately from interrupts, without regard to the priority of the current process.

Realtime processes have the highest default software priority. Processes in the `RT` class have a priority and *time quantum* value. `RT` processes are scheduled strictly on the basis of these parameters. As long as an `RT` process is ready to run, no `sys` or `TS` process can run. Fixed priority scheduling allows critical processes to run in a predetermined order until completion. These priorities never change, unless an application changes them.

An `RT` class process inherits the parent's time quantum, whether finite or infinite. A process with a finite time quantum runs until the time quantum expires or the process terminates, blocks (while waiting for an I/O event), or is preempted by a higher priority runnable realtime process, or A process with an infinite time quantum ceases execution only when it terminates, blocks, or is preempted.

The `sys` class exists for scheduling execution of special system processes, such as paging, STREAMS and the swapper,. It is not possible to change the class of a process of the `sys` class; when started these processes have fixed priorities established by the kernel.

At lowest priority are the timesharing (`TS`) processes. `TS` class processes are scheduled dynamically with a few hundred milliseconds per time slice. The `TS` scheduler switches context in round-robin fashion often enough to give every process an equal opportunity to run, depending upon its time slice value, its process history (when the process was last put to sleep), and considerations for CPU utilization. The default timesharing policy gives larger time slices to processes with lower priority. A timesharing policy adjusts process priorities dynamically in an attempt to provide good response time to interactive processes and good throughput to processes that use a lot of CPU time.

Different algorithms are used to dispatch each scheduling class. Class dependent routines are called by the kernel to make decisions about CPU process scheduling. The kernel is class-independent, and takes the highest priority process off its queue. Each class is responsible for calculating a process' priority value for its class. This value is placed into the dispatch priority variable of that process.

As Figure 11-3 illustrates, each class algorithm has its own method of nominating the highest priority process to place on the global run queue:

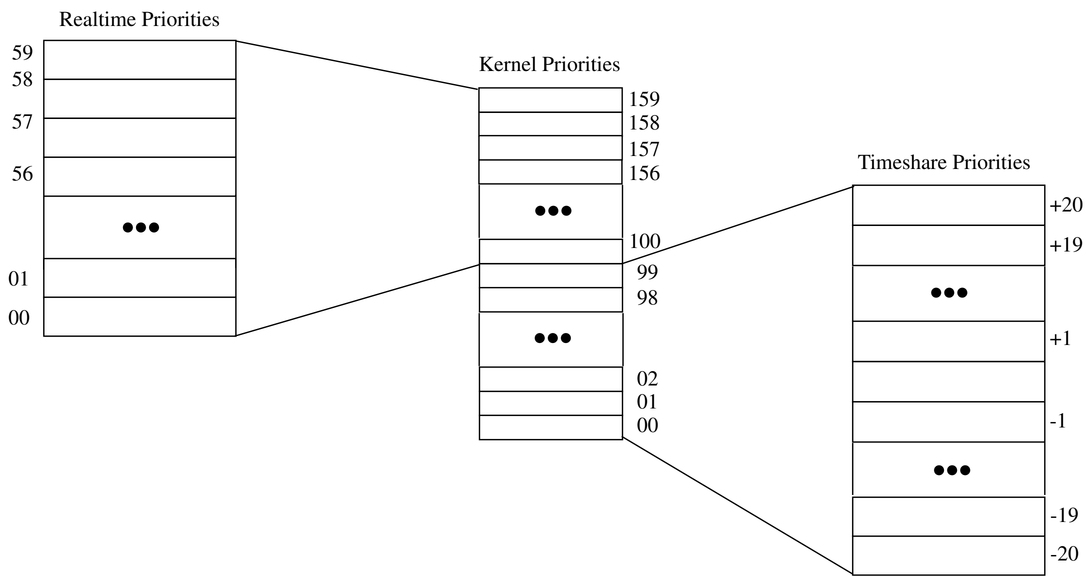

Figure 11-3 Scheduler Algorithms

Each class has a set of priority levels which apply to processes in that class. A class-specific mapping maps these priorities into a set of global priorities. It is not required that a set of global scheduling priority maps start with zero, nor is it required that they are contiguous.

By default, the global priority values for time-sharing processes range from -20 to +20, mapped into the kernel from 0-40, with temporary assignments as high as 99. The default priorities for realtime processes range from 0-59 and are mapped into the kernel from 100 to 159. The kernel's class-independent code runs the process with the highest global priority on the queue.

Dispatch Queue

The dispatch queue is a linear linked list of processes with the same dispatch priority. Each process is invoked with class-specific information saved to its `proc` structure. A process is dispatched from the kernel dispatch table based upon its dispatch priority class and time quantum.

Dispatching

When a process is dispatched, its context is mapped into memory, along with its memory management information, its registers, and its stack; then execution begins. Memory management information is in the form of hardware registers containing data needed to perform virtual memory translations for the current running process.

Preemption

When a higher priority process becomes disputable, the kernel interrupts its computation and forces the context switch, thus preempting the currently running process. A process can be preempted at any time if the kernel finds a higher priority process is now dispatchable.

Kernel Priority Inversion

Priority inversion is a condition which occurs when a higher priority process is blocked by one or more lower priority processes for a prolonged period of time. The use of synchronization primitives, such as mutual-exclusion locks in the SunOS 5.x kernel, can lead to priority inversion.

The term *blocking* describes the situation in which a process must wait for one or more processes to relinquish resources. If this blocking continues, it may lead to deadlines being missed, even for low usage levels.

The problem of priority inversion has been addressed for mutual-exclusion locks in the SunOS 5.x kernel by implementing a basic priority inheritance policy. This policy states that a lower priority process inherits the priority of a higher priority process when the lower priority process blocks the execution of the higher priority process. This places an upper bound on the amount of time a process can remain blocked. The policy is a property of the kernel's behavior, not a solution that a programmer institutes through system calls or function execution. However, user-level processes can still exhibit priority inversion.

User Priority Inversion

There is no mechanism by which processes synchronizing with other processes will automatically inherit the priority of waiting processes. An application can bound its priority inversion by using priority ceiling emulation.

Under this model, the application associates a priority with each synchronization object, which is typically the highest priority of any process that can block on that object.

Each process then uses the following sequence when manipulating the shared resources:

Code Example 11-1

```
/*
 * raise process priority to maximum of current level
 * and synchronization object level
 */
    ...

/*
 * acquire synchronization object
 */
    ...

/*
 * perform the critical sequence
 */
    ...

/*
 * release synchronized object
 */
    ...

/*
 * return to previous process priority level
 */
    ...
```

Performance

Scheduling performance from the realtime perspective can be defined in terms of the *dispatch latency*. The term dispatch latency describes the amount of time it takes for a system to respond to a request for a process to begin operation. With a scheduler written specifically to honor application priorities, it is possible for realtime applications to be developed with a bounded dispatch latency. Figure 11-4 illustrates the amount of time it takes an application to respond to a request from an external event.

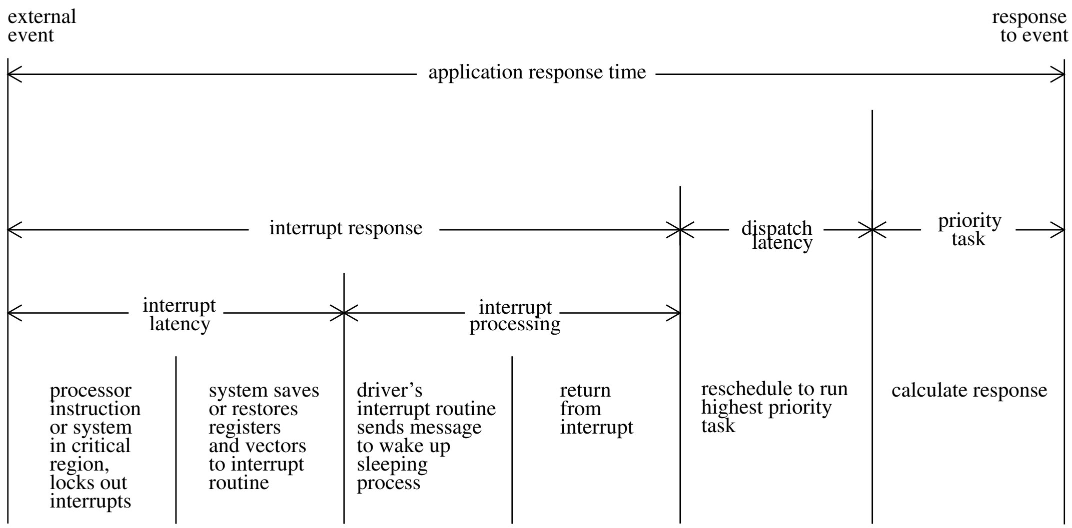

Figure 11-4 Application Response Time

The overall application response time is composed of the interrupt response time, the dispatch latency, and the time it takes the application to determine its response.

The interrupt response time for an application includes both the interrupt latency of the system and the device driver's own interrupt processing time. The interrupt latency is determined by the longest interval that the system must run with interrupts disabled; this is minimized in SunOS 5.x using synchronization primitives that do not commonly require a raised processor interrupt level.

During interrupt processing, the driver's interrupt routine wakes up the high priority process and returns when finished. The system detects that a process with higher priority than the interrupted process in now dispatchable and arranges to dispatch that process. The time to switch context from a lower priority process to a higher priority process is included in the dispatch latency time.

Figure 11-5 illustrates the ratio of internal dispatch latency to application response time of a system defined in terms of the amount of time it takes for a system to respond to an internal event. The dispatch latency of an internal event represents the amount of time required for one process to wake up another higher priority process, and for the system to dispatch the higher priority process.

The application response time is the amount of time it takes for a driver to wake up a higher priority process, have a low priority process release resources, reschedule the higher priority task, calculate the response, and dispatch the task.

Note – Interrupts can arrive and be processed during the dispatch latency interval. This processing increases the application response time, but is not attributed to the dispatch latency measurement, and so is not bounded by the dispatch latency guarantee.

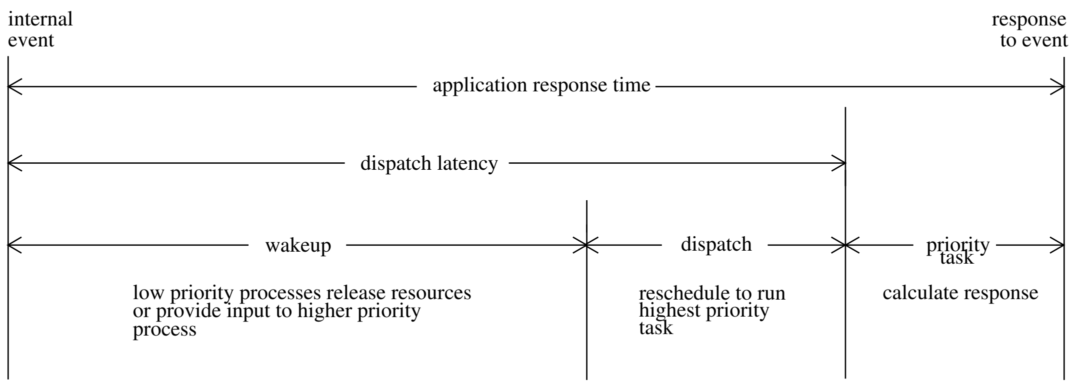

Figure 11-5 Internal Dispatch Latency

With the new scheduling techniques provided with realtime SunOS 5.x, the system dispatch latency time is within specified bounds.

As you can see in Table 11-1, dispatch latency improves with a bounded number of processes.

Table 11-1 Realtime System Dispatch Latency with SunOS 5.x

Workstation	Dispatch Latency	
	Bounded Number of Processes	Arbitrary Number of Processes
SPARCstation 1	< 2.0 milliseconds in a system with fewer than 8 active processes	4.5 milliseconds
SPARCstation 1+	< 2.0 milliseconds in a system with fewer than 8 active processes	4.0 milliseconds
SPARCstation IPX	< 1.0 milliseconds in a system with fewer than 8 active processes	2.2 milliseconds
SPARCstation 2	< 1.0 milliseconds in a system with fewer than 16 active processes	2.0 milliseconds

Tests for dispatch latency and experience with such critical environments as manufacturing and data acquisition have proven that the SPARC workstation is an able platform for the development of realtime applications.

Controlling Scheduling

You can control the scheduling of processes through system calls, administrative utilities, and configuration files. From within an application, you can use the system calls `priocntl` and `priocntlset` to set and retrieve scheduler parameters for these processes. The `priocntl` function provides an interface for specifying a realtime process, a set of processes, or a class to which the system call will apply. The `priocntlset` system call performs the same functions as `priocntl` except it provides a more general interface for specifying processes.

At the system administrative level, you can display and set scheduling parameters for specified process(es) with the `priocntl` and `dispadmin` utilities. The usage of the `priocntl` utility is the same for the `priocntl` and `priocntlset` system calls:

- specify the target process(es)
- specify the scheduler parameters you want for those process(es)
- execute the command or function to set the parameters for the process(es)

You specify the target processes using an ID type and an ID. The ID type tells how to interpret the ID; IDs are basic properties of UNIX processes.

In addition to the administrative utilities, there are dispatcher parameter tables which can be configured to control scheduling. These files can be reconfigured with the `dispadmin` utility at runtime or with a loadable module at boot-time

The following sections describe in detail the system calls, utilities and configuration files that allow you to control realtime scheduling.

System Calls that Control Scheduling

System calls implemented for realtime scheduling are described in this section along with sample programs. For more information on using these functions, see the *SunOS 5.2 Reference Manual*.

Using priocntl

The `priocntl` function gets or sets scheduler parameters of a set of processes. Its format is:

Code Example 11-2

```
#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);
```

where:

- `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 11-2)
- `arg` is a pointer to a structure that depends on `cmd`

Table 11-2 contains the valid values for `idtype` as defined in `priocntl.h`, along with their corresponding ID types in `id`.

Table 11-2 Valid priocntl.h idtypes

idtype	Interpretation of id
`P_PID`	process ID (of a single process)
`P_PPID`	parent process ID
`P_LWP`	LWP ID (LWP within the calling process)
`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

Table 11-3 lists the valid values for `cmd`, their meanings, and the type of `arg`:

Table 11-3 Valid priocntl.h cmd Values

cmd	arg Type	Function
PC_GETCID	pcinfo_t	get class ID and attributes
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

On success, `priocntl` returns the following values:

- `PC_GETCID` and `PC_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 the man page for `priocntl` for the complete list of error conditions.

PC_GETCID and PC_GETCLINFO Commands

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 following `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 realtime.

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

```
typedef struct rtinfo {
    short rt_maxpri;       /* maximum realtime priority */
} rtinfo_t;
```

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

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

The following program, `getcid.c`, gets and prints the range of valid priorities for the time-sharing and realtime scheduler classes.

Code Example 11-3   getcid.c Program

```
/*
 * 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;

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

    /* realtime */
        (void) strcpy(pcinfo.pc_clname, "RT");
        if (priocntl (0L, 0L, PC_GETCID, &pcinfo) == -1L) {
```

Code Example 11-3 getcid.c Program (Continued)

```
            perror ("PC_GETCID failed for realtime class");
            exit (2);
        }
    rtinfop = (struct rtinfo *) pcinfo.pc_clinfo;
    maxrtpri = rtinfop->rt_maxpri;
    (void) printf("Realtime: ID %ld, priority range 0 through %d\n",
            pcinfo.pc_cid, maxrtpri);
    return (0);
}
```

The following figure shows the output of this program:

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

Figure 11-6 Output from getcid program

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

Note – All the following examples omit the lines that include header files. The examples compile with the same header files as in the Code Example 11-3.

Code Example 11-4 Function to Return Class ID and Priority

```
/*
* 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);
```

Code Example 11-4 Function to Return Class ID and Priority (Continued)

```
    }
    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.

Code Example 11-5 Program to Get Class Name Given Process ID

```
/* 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);
```

Code Example 11-5 Program to Get Class Name Given Process ID (Continued)

```
    }
    (void) printf("process ID %d, class %s\n", pid,
     pcinfo.pc_clname);
}
```

PC_GETPARMS and PC_SETPARMS Commands

The PC_GETPARMS command gets and the PC_SETPARMS command sets scheduler parameters for processes. Both commands use the following 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;
```

The following function, invoked in an earlier program example, uses PC_GETPARMS to obtain the scheduler class ID of a process.

```
/*
 * 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 realtime class, pc_clparms contains an rtparms structure. The rtparms structure holds scheduler parameters specific to the realtime 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;
```

In this structure, rt_pri is the realtime priority, rt_tqsecs is the number of seconds, and rt_tqnsecs is the number of additional nanoseconds, rounded up to the next multiple of the system clock's resolution, in a time slice. The sum of rt_tqsecs seconds and 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 timesharing class.

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

In this structure, ts_upri is the user priority, the user-controlled component of a time-sharing priority, and ts_uprilim is the user priority limit, the maximum user priority a process can set for itself without being super-user.

The PC_GETPARMS command gets the scheduler class and parameters of a single process. The return value of 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 11-4 Return Values for PC_GETPARMS

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.

Code Example 11-6 Program to Print the Scheduler Class and Class-Specific Parameters of a 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 */
        if (priocntl(P_PID, pid, PC_GETPARMS, &pcparms) == -1) {
            perror ("GETPARMS failed");
            exit (2);
        }

    /* get class IDs and maximum priorities for TS and RT */
        if ((tsID = schedinfo ("TS", &tsmaxpri)) == -1) {
            perror ("schedinfo failed for TS");
            exit (3);
        }
        if ((rtID = schedinfo ("RT", &rtmaxpri)) == -1) {
            perror ("schedinfo failed for RT");
            exit (4);
        }

    /* print results */
```

Code Example 11-6 Program to Print the Scheduler Class and Class-Specific Parameters of a Process (Continued)

```
        if (pcparms.pc_cid == rtID) {
            priority = rtparmsp->rt_pri;
            secs = rtparmsp->rt_tqsecs;
            nsecs = rtparmsp->rt_tqnsecs;
            (void) printf ("process %d: RT priority %d\n",
                pid, priority);
            (void) printf ("time slice %ld secs, %ld nsecs\n",
                secs, nsecs);
        } else if (pcparms.pc_cid == tsID) {
            priority = tsparmsp->ts_upri;
            (void) printf ("process %d: TS priority %d\n",
                pid, priority);
        } else {
            printf ("Unknown scheduler class %d\n",
                pcparms.pc_cid);
            exit (5);
        }
        return (0);
}
```

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 realtime class ID, `pc_clparms` contains an `rtparms` structure in which `rt_pri` contains the realtime 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 realtime process with the highest valid priority minus 1, and gives it the default time slice for that priority. The program calls the `schedinfo` function to get the realtime class ID and maximum priority.

Code Example 11-7 Program to Set Highest Valid Priority - 1

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

Code Example 11-7 Program to Set Highest Valid Priority - 1 (Continued)

```
    main (argc, argv)
        int     argc;
        char    *argv[];
{
        pcparms_t  pcparms;
        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 realtime processes. When any of these are used, `rt_tqsecs` is ignored. These values are defined in the header file `rtpriocntl.h`.

Table 11-5 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 realtime priority being set with the PC_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.

Using priocntlset

The priocntlset function changes scheduler parameters of a set of processes, just like priocntl. The priocntlset function also 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 following is the format of the procntlset 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 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;
```

The `p_lidtype` and `p_lid` parameters 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`, enables you 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);
```

The `priocntlset` function is useful in a situation where you have a program running both realtime and time-sharing processes under a single user ID and you want to change the priority of only its realtime processes. The following example does this using the `schedinfo` function defined in the "PC_GETCID and PC_GETCLINFO Commands" section.

Code Example 11-8   Changing Realtime Processes Using the schedinfo Function

```
/*
 * Change realtime 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 */
```

Code Example 11-8 Changing Realtime Processes Using the schedinfo Function (Continued)

```
        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);
        }

    /* right set: select realtime processes */
        procset.p_ridtype = P_CID;
        procset.p_rid = rtclassID;

    /* select only my RT processes */
        procset.p_op = POP_AND;

    /* specify new scheduler parameters */
        pcparms.pc_cid = rtclassID;
        rtparmsp = (struct rtparms *) pcparms.pc_clparms;
        rtparmsp->rt_pri = maxrtpri - 1;
        rtparmsp->rt_tqnsecs = RT_NOCHANGE;
        if (priocntlset (&procset, PC_SETPARMS, &pcparms) == -1) {
            perror ("priocntlset failed");
            exit (2);
        }
}
```

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

Utilities that Control Scheduling

The administrative utilities which control process scheduling are `dispadmin` and `priocntl`. Both these utilities support the `priocntl` system call with compatible options and loadable modules. Using these utilities provides system administration functions which control realtime process scheduling during runtime. For more details about using these utilities, refer to the *SunOS 5.2 Reference Manual* and the *SunOS 5.2 Administering Security, Performance, and Accounting* guide.

Using the priocntl Command

The `priocntl` command displays or sets scheduling parameters of the specified process(es). It can also be used to display the current configuration information for the system's process scheduler or execute a command with specified scheduling parameters.

There four options can be used with `priocntl`:

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

The `priocnt -l` command takes no arguments and displays a list of classes currently configured in the system along with class-specific information about each class.

The command:

```
priocntl -d [-i idtype] [idlist]
```

displays the class and class-specific scheduling parameters of the process(es) specified by `idtype and idlist.`

The syntax for the `-s` option is:

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

This command sets the class and class-specific parameters of the specified processes to the values given on the command line. where `class` can be either the `RT` or the `TS` class. The `class_options` can include one of the choices shown in the following table:

Table 11-6 Choices for `class_option` *for* `priocntl`

class_option	**Meaning**
`-p`	priority
`-t`	time quantum
`-r`	resolution

Changing timesharing processes into realtime processes gives them a default priority and time quantum unless otherwise specified.

The command:

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

executes the specified command with the class and scheduling parameters specified on the command line where `arguments` are the arguments to `command`.

Using dispadmin

The `dispadmin` utility displays all current process scheduling classes when you include the `-l` command line option during runtime. Process scheduling can also be changed for the class specified after the `-c` option using `RT` as the argument for the realtime class.

Table 11-7 lists additional options for the `dispadmin` utility.

Table 11-7 Class Options for the dispadmin Utility

Option	Meaning
`-l`	lists scheduler classes currently configured
`-c`	specifies the class whose parameters are to be displayed or changed
`-g`	gets the dispatch parameters for the specified class
`-r`	when using `-g`, specifies time quantum resolution
`-s`	specifies a file where values may be located

A class-specific file containing the dispatch parameters can also be loaded during runtime. Such a file could be used to establish a new set of priorities replacing the default values established during boot time.This class-specific file must assert the arguments in the format used by the `-g` option. Parameters for the `RT` class are found in the realtime dispatcher parameter table, `rt_dptbl`, which is described in the *SunOS 5.2 Reference Manual*.

To add an `RT` class file to the system, the following modules must be present:

- An `rt_init()` routine in the class module which loads the `rt_dptbl` structure.
- An `rt_dptbl` module that provides the dispatch parameters and a routine to return pointers to the `config_rt_dptbl` structure.
- The `dispadmin` executable.

You then load the class-specific module by typing:

```
# modload /kernel/sched/<module_name>
```

where `<module_name>` is the class-specific module.

Next, invoke the `dispadmin` command:

```
# dispadmin -c RT -s <filename>
```

The file must describe a table with the same number of entries as the current table being overwritten. Super-user privilege is required to perform this operation.

Configuring Scheduling

Associated with each scheduling class is a parameter table: `config_rt_dptbl` (`RT`) and `config_ts_dptbl` (`TS`). These tables are configurable by using a loadable module at boot time, or with `dispadmin` during runtime.

The Dispatcher Parameter Table

The in-core table for realtime establishes the properties for `RT` scheduling. The `config_rt_dptbl` structure consists of an array of parameters, `struct rt_dpent`, one for each of the `n` priority levels. The properties of a given priority level `i` are specified by the `ith` parameter structure in the array, `config_rt_dptbl`.

A parameter structure consists of the following members, (also described in the `/usr/include/sys/rt.h` header file):

- `rt_globpri`
  The global scheduling priority associated with this priority level. The `rt_globpri` values cannot be changed with `dispadmin`.
- `rt_quantum`
  The length of the time quantum allocated to processes at this level in clock ticks. (For more information on clock ticks see `sysconf` in the *SunOS 5.2 Reference Manual*.) The time quantum value is only a default or starting value for processes at a particular level. The time quantum of a realtime process can be changed by using the `priocntl` command or the `priocntl` system call.

Reconfiguring config_rt_dptbl

A realtime administrator can change the behavior of the realtime portion of the scheduler by reconfiguring the `config_rt_dptbl` at any time. Both methods are described below:

The first method might be to reconfigure the `config_rt_dptbl` parameter table with a loadable module which contains a new dispatch table loaded at boot time. The module containing the dispatch table is a separate module. This is the only method that can be used to change the number of realtime priority levels or the set of global scheduling priorities used by the realtime class. Note that changing the `config_rt_dptbl` affects the realtime processes that you set after the table gets updated.

A second method for examining or modifying the realtime parameter table on a running system is by using the dispadmin command. Invoking dispadmin for the realtime class allows retrieval of the current rt_quantum values in the current config_rt_dptbl configuration from the kernel's in-core table. When overwriting the current in-core table, the configuration file used for input to dispadmin must conform to the specific format described in the man page for config_rt_dptbl found in the *SunOS 5.2 Reference Manual*.

Memory Locking

Locking memory is one of the most important issues for realtime applications. Applications that have strict realtime constraints may need to prevent processes from being paged or swapped out to secondary memory.

Under SunOS 5.x, the memory residency of a process is determined by its current state, the total available physical memory, the number of active processes, and the demand for memory by these processes. In a realtime environment, a process must be able to guarantee memory residence for all or part of itself to reduce its memory access and dispatch latency.

For realtime in SunOS 5.x, memory locking is provided by a set of library routines that allow a process, running with super-user privileges, to lock specified portions of its virtual address space into physical memory. Pages locked in this manner are exempt from paging until they are unlocked or the process exits.

There is a system-wide limit on the number of pages that can be locked at any given time. This is a tunable parameter whose default value is calculated at boot time. It is based on the number of page frames less another percentage (currently set at ten percent).

Locking a Page

A call to mlock requests that one segment of memory be locked into the system's physical memory. The pages that make up the specified segment are faulted in and the lock count of each is incremented. Any page with a lock count greater than 0 is exempt from paging activity.

A particular page can be locked multiple times by multiple processes through different mappings. If two different processes lock the same page, the page remains locked until both processes remove their locks. However, within a given mapping, page locks do not nest. Multiple calls of locking functions on the same address by the same process are removed by a single unlock request.

If the mapping through which a lock has been performed is removed, the memory segment is implicitly unlocked. When a page is deleted through closing or truncating the file, it is also unlocked implicitly.

Locks are not inherited by a child process after a `fork` call is made. So, if a process with memory locked forks a child, the child must perform a memory locking operation on its own behalf to lock its own pages. Otherwise, the child process incurs copy-on-write pages, which are the usual penalties associated with forking a process.

Unlocking a Page

To unlock a page of memory, a process requests that a segment of locked virtual pages be released by a call to `munlock`. The lock counts of the specified physical pages are decremented. Once the lock count of a page has been decremented to `0`, the page is swapped normally.

Locking All Pages

A super–user process may request that all mappings within its address space be locked by a call to `mlockall`. If the flag `MCL_CURRENT` is set, all of the existing memory mappings are locked. If the flag `MCL_FUTURE` is set, every mapping that is added or that replaces an existing mapping is locked into memory.

Sticky Locks

A page is permanently locked into memory when its lock count reaches 65535 (0xFFFF). The value 0xFFFF is implementation-defined and may change in future releases. Pages locked in this manner cannot be unlocked. You must reboot the system to recover.

High Performance I/O

Standard UNIX I/O is generally synchronous to the application programmer. An application that calls `read` or `write` is not usually allowed to proceed until that system call has completed, successfully or otherwise.

Realtime applications need *asynchronous* bounded I/O behavior. A process that issues an asynchronous I/O call does not wait until the I/O operation has been completed before it is allowed to proceed. Instead, the caller is notified that the I/O operation has finished at a later time while the process is doing something else.

Asynchronous I/O

Asynchronous I/O applies to any SunOS file. Files are opened in the synchronous way and no special flagging is required. An asynchronous I/O transfer is composed of three elements: call, request, and operation. The application calls an asynchronous I/O function, the request for the I/O is placed on a queue, and the call returns immediately. At some point, the system dequeues the request and initiates the I/O operation itself.

Asynchronous and standard I/O requests can be intermingled on any file descriptor. Note, however, that the system does not necessarily maintain any particular sequence of read and write requests. The system can and does arbitrarily resequence any and all pending read and write requests. If a specific sequence is required for the application, it must be planned for ahead of time.

Notification (SIGIO)

When an asynchronous I/O call returns successfully, the I/O operation is placed on the queue, and waits to be done. When the I/O is finished, the return value and error value are stored at a location given by the user at the time of the request as a pointer to an `aio_result_t`. The structure of the `aio_result_t` is defined in `sys/asynch.h`:

```
typedef struct aio_result_t
     {
     int aio_return; /* return value of read or write */
     int aio_errno;  /* errno generated by the IO */
     } aio_result_t;
```

When `aio_result_t` is updated, a `SIGIO` signal is delivered to the process that made the I/O request.

Using aioread, aiowrite, aiocancel, and aiowait

The `aioread` and `aiowrite` functions are the asynchronous versions of `read` and `write`, respectively. In addition to the normal read and write arguments, they take the arguments specifying a file position and the address of an `aio_result_t` structure at which the system is to store the result information about the operation.

The file position specifies a seek to be performed within the file before the operation. If the `aioread` (or `aiowrite`) call succeeds, the file pointer is updated to the position that would have resulted in a successful seek and read (or write). The file pointer is also updated when a read(write) fails to allow for subsequent read (or write) requests.

The `aiocancel` function attempts to cancel the asynchronous request whose `aio_result_t` structure is given as an argument. A call to the `aiowait` function blocks the calling process until at least one outstanding asynchronous I/O operation is completed.

Using poll

There may be circumstances in which you wish to poll rather than depend on a `SIGIO` interrupt. Devices may be polled using the `poll` system call. You can also poll to determine the origin of an `SIGIO` interrupt.

Using close

Files are closed by a call to `close`. The call to `close` cancels any outstanding asynchronous I/O request that can be cancelled. The `close` function waits on an operation that cannot be cancelled. When a call to `close` returns, there is no asynchronous I/O pending for the file descriptor.

Asynchronous Networking

This section discusses the techniques of asynchronous network communication using Transport Level Interface (TLI) for realtime applications. SunOS provides support for asynchronous network processing of TLI events using a combination of STREAMS asynchronous features and the non-blocking mode of the TLI library routines.

For more information on the Transport Level Interface, see the *SunOS 5.2 Network Interfaces Programmer's Guide* and the*SunOS 5.2 Reference Manual*.

Modes of Networking

The Transport Level Interface provides two modes of service: *connection-mode* and *connectionless-mode*.

Connection-Mode Service

The *connection-mode* is circuit-oriented and enables the transmission of data over an established connection in a reliable, sequenced manner. It also provides an identification procedure that avoids the overhead of address resolution and transmission during the data transfer phase. This service is attractive for applications that require relatively long-lived, datastream-oriented interactions.

Connectionless-Mode Service

Connectionless-mode is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. All information required to deliver a unit of data, including the destination address, is passed by the sender to the transport provider, together with the data, in a single service request. Connectionless-mode service is attractive for applications that involve short-term request/response interactions and do not require guaranteed, in-sequence delivery of data. It is generally assumed that connectionless transports are unreliable.

Networking Programming Models

Like file and device I/O, network transfers can be done synchronously or asynchronously with process service requests.

Synchronous Networking

Synchronous networking proceeds similarly to synchronous file and device I/O. Like the `write` function, the request to send returns after buffering the message, but might suspend the calling process if buffer space is not immediately available. Like the `read` function, a request to receive suspends execution of the calling process until data arrives to satisfy the request. Because SunOS 5.x provides no guaranteed bounds for transport services, synchronous networking is inappropriate for processes that must have realtime behavior.

Asynchronous Networking

Asynchronous networking is provided by non-blocking service requests. Additionally, applications can request asynchronous notification when a connection might be established, when data might be sent, or when data might be received.

Asynchronous Connectionless-Mode Service

Asynchronous connectionless mode networking is conducted by configuring the endpoint for non-blocking service, and either polling for or receiving asynchronous notification when data might be transferred. If asynchronous notification is used, the actual receipt of data typically takes place within a signal handler.

Making the Endpoint Asynchronous

After the endpoint has been established using `t_open`, and its identity established using `t_bind`, the endpoint can be configured for asynchronous service. This is done by using the `fcntl` function to set the `O_NONBLOCK` flag on the endpoint. Thereafter, calls to `t_sndudata` for which no buffer space is immediately available return `-1` with `t_errno` set to `TFLOW`. Likewise, calls to `t_rcvudata` for which no data are available return `-1` with `t_errno` set to `TNODATA`.

Asynchronous Network Transfers

Although an application can use the `poll` function to wait for the receipt of data on an endpoint, it might be necessary to receive asynchronous notification when data has arrived. This can be done by using the `ioctl` function with the `I_SETSIG` command to request that a `SIGPOLL` signal be sent to the process upon receipt of data at the endpoint. Applications should check for the possibility of multiple messages causing a single signal.

In the following example, `protocol` is the name of the application-chosen transport protocol.

Code Example 11-9 Example of an Asynchronous Network Transfer

```
#include <sys/types.h>
#include <tiuser.h>
#include <signal.h>
#include <stropts.h>

int     fd;
struct  t_bind   *bind;
void    sigpoll(int);

        fd = t_open(protocol, O_RDWR, (struct t_info *) NULL);

        bind = (struct t_bind *) t_alloc(fd, T_BIND, T_ADDR);
        ...     /* set up binding address */
        t_bind(fd, bind, bind);

        /* make endpoint non-blocking */
        fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | O_NONBLOCK);

        /* establish signal handler for SIGPOLL */
        signal(SIGPOLL, sigpoll);

        /* request SIGPOLL signal when receive data is available */
        ioctl(fd, I_SETSIG, S_INPUT | S_HIPRI);

        ...

void    sigpoll(int sig)
{
        int    flags:
        struct t_unitdata       ud;
```

Code Example 11-9 Example of an Asynchronous Network Transfer (Continued)

```
        for (;;) {
        ... /* initialize ud */
            if (t_rcvudata(fd, ud, &flags) < 0) {
                if (t_error == TNODATA)
                break;  /* no more messages */
            ... /* process other error conditions */
            }

        ... /* process message in ud */

        }
}
```

Asynchronous Connection-Mode Service

For connection-mode service, an application can arrange for not only the data transfer, but for the establishment of the connection itself to be done asynchronously. The sequence of operations depends on whether the process is attempting to connect to another process or is awaiting connection attempts.

Asynchronously Establishing a Connection

A process can attempt a connection and asynchronously complete the connection. The process first creates the connecting endpoint, and, using `fcntl`, configures the endpoint for non-blocking operation. As with connectionless data transfers, the endpoint can also be configured for asynchronous notification upon completion of the connection and subsequent data transfers. The connecting process then uses the `t_connect` function to initiate setting up the transfer. Then the `t_rcvconnect` function is used to confirm the establishment of the connection.

Asynchronous Use of a Connection

To asynchronously await connections, a process first establishes a non-blocking endpoint bound to a service address. When either the result of `poll` or an asynchronous notification indicates that a connection request has arrived, the process can get the connection request by using the `t_listen` function.To accept the connection, the process uses the `t_accept` function. The responding endpoint must be separately configured for asynchronous data transfers.

The following example illustrates how to request a connection asynchronously:

Code Example 11-10 How to Request a Connection Asynchronously

```
#include <tiuser.h>

int             fd;
struct t_call   *call;

      fd = .../* establish a non-blocking endpoint */

      call = (struct t_call *) t_alloc(fd, T_CALL, T_ADDR);
      .../* initialize call structure */
      t_connect(fd, call, call);

      /* connection request is now proceeding asynchronously */

      .../* receive indication that connection has been accepted */
      t_rcvconnect(fd, &call);
```

The following example illustrates listening for connections asynchronously:

Code Example 11-11 Listening for Connections Asynchronously

```
#include <tiuser.h>

int      fd, res_fd;
struct  t_call   call;

    fd = ... /* establish non-blocking endpoint */

    .../*receive indication that connection request has arrived */
    call = (struct t_call *) t_alloc(fd, T_CALL, T_ALL);
    t_listen(fd, &call);

    .../* determine whether or not to accept connection */
    res_fd = ... /* establish non-blocking endpoint for response */
    t_accept(fd, res_fd, call);
```

Asynchronous Open

Occasionally, an application might be required to dynamically open a regular file in a file system mounted from a remote host or on a device whose initialization might be prolonged. However, while such an open is in progress, the application would be unable to achieve realtime response to other events. Fortunately, SunOS 5.x provides a means of solving this problem by having a second process perform the actual open and then pass the file descriptor to the realtime process.

Transferring a File Descriptor

The STREAMS interface under SunOS 5.x provides a mechanism for passing an open file descriptor from one process to another. The process with the open file descriptor uses the `ioctl` function with a command argument of `I_SENDFD`. The second process obtains the file descriptor by calling the `ioctl` function with a command argument of `I_RECVFD`.

In the following example, the parent process first prints out information about the test file, and then it creates a pipe. Next, the parent creates a child process, which opens the test file, and passes the open file descriptor back to the parent through the pipe. The parent process then displays the status information on the new file descriptor.

Code Example 11-12 Transferring a File Descriptor

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stropts.h>
#include <stdio.h>

#define TESTFILE "/dev/null"
main(int argc, char * argv)
{
    int     fd;
    int     pipefd[2];
    struct  stat statbuf;

    stat(TESTFILE, &statbuf);
    statout(TESTFILE, &statbuf);
    pipe(pipefd);
    if (fork() == 0) {
        close(pipefd[0]);
        sendfd(pipefd[1]);
```

Code Example 11-12 Transferring a File Descriptor (Continued)

```
    } else {
        close(pipefd[1]);
        recvfd(pipefd[0]);
    }
}

sendfd(int p)
{
    int     tfd;

    tfd = open(TESTFILE, O_RDWR);
    ioctl(p, I_SENDFD, tfd);
}

recvfd(int p)
{
    struct  strrecvfd rfdbuf;
    struct  stat statbuf;
    char    fdbuf[32];

    ioctl(p, I_RECVFD, &rfdbuf);
    fstat(rfdbuf.fd, &statbuf);
    sprintf(fdbuf, "recvfd=%d", rfdbuf.fd);
    statout(fdbuf, &statbuf);
}

statout(char *f, struct stat *s)
    printf("stat: from=%s mode=0%o, ino=%d, dev=%d, rdev=%d\n",
        f, s->st_mode, s->st_ino, s->st_dev, s->st_rdev);
    fflush(stdout);
```

Interprocess Communication

Realtime processing often requires fast, high-bandwidth interprocess communication. The interprocess communication (IPC) functions of SunOS 5.x realtime processing include pipes, FIFOs (named pipes), message queues, shared memory, file mapping, and semaphores. These functions are fully described in Chapter 13, “Interprocess Communication”.

The choice of which mechanisms should be used can be dictated by functional requirements, and the relative performance will depend upon application behavior. The traditional method of interprocess communication in UNIX is the pipe. Unfortunately, pipes can have framing problems.

IPC messages mimic the reading and writing of files. They are easier to use than pipes when more than two processes must communicate by using a single medium, but messages can become intermingled by multiple writers or can be torn apart by multiple readers.

The IPC shared semaphore facility provides process synchronization. Shared memory is the fastest form of interprocess communication. The main advantage of shared memory is that the copying of message data is eliminated. The usual mechanism for synchronizing shared memory access is semaphores.

For more information about the libraries, functions, and routines useful for interprocess communication, see Chapter 13, “Interprocess Communication” and the *SunOS 5.2 Reference Manual*.

Timers

The timing functions of SunOS 5.x fall into two separate areas of functionality: timestamps and interval timers. The timestamp functions provide a measure of elapsed time and allow the application to measure the duration of a state, or the time between events. Although an application can poll a timestamp function to schedule itself, such an application would monopolize the processor to the detriment of other system functions.

Interval timers allow an application to wake up at specified times and to schedule activities based on the passage of time. For detailed information on the timing functions described in this section, see the function’s man pages in the *SunOS 5.2 Reference Manual.*

Timestamp Function

The timestamp function, `gettimeofday`, provides the current time in a `timeval` structure, representing the time in seconds and microseconds since midnight, Greenwich Meridian, on January 1, 1970.

SunOS 5.x uses a hardware periodic timer. For some workstations, this is the sole timing information, and the accuracy of timestamps is limited to the resolution of that periodic timer. For other platforms, a timer register with a resolution of one microsecond allows SunOS 5.x to provide timestamps accurate to one microsecond.

Interval Timer Functions

Realtime applications often schedule their activities through the use of interval timers. Interval timers can be either of two types: a *one-shot* type or a *periodic* type. These timers are either relative to current time, or they must establish their own absolute time.

The one-shot is an armed timer that is set with an initial expiration time relative either to current time or to an absolute time. This timer expires once, and then is disarmed. Such a timer might be useful for clearing buffers after the data has been transferred to storage, or to time-out an operation which should have completed.

The periodic timer is armed with the initial expiration time (either absolute or relative) and a repetition interval. Each time the interval timer expires, it is reloaded with the repetition interval and the timer is automatically rearmed. This timer might be useful for data logging or for servo-control. In calls to interval timer functions, time values smaller than the resolution of the system hardware periodic timer are rounded up to the next multiple of the hardware periodic timer interval (10 ms).

The system calls `getitimer` and `setitimer` respectively retrieve and establish the value of the specified interval timer. There are four timers available to a process, including a realtime timer. If a timer is armed and allowed to expire, the system sends a signal appropriate to the timer to the process that set the timer.

The four timers are defined in `sys/time.h` and are:

- `ITIMER_REAL`
  Decrements in realtime. The `SIGALRM` signal is delivered when this timer expires.
- `ITIMER_VIRTUAL`
  Decrements in process virtual time. It runs only when the process is executing. The `SIGVTALRM` signal is delivered when it expires.
- `ITIMER_PROF`
  Decrements both in process virtual time and when the system is running on behalf of the process. It is designed to be used by interpreters in statistically profiling the execution of interpreted programs. Each time the `ITIMER_PROF` timer expires, the `SIGPROF` signal is delivered. Because this signal may interrupt in-progress functions, programs using this timer must be prepared to restart interrupted functions.
- `ITIMER_REALPROF`
  Decrements in realtime. It is designed to be used for realtime profiling of multithreaded programs. Each time the `ITIMER_REALPROF` timer expires, one counter in a set of counters maintained by the system for each lightweight process (`lwp`) is incremented.

The following example illustrates how to use the `setitimer` interface to generate a periodic interrupt, and how to control the arrival of timer interrupts.

Code Example 11-13 Using the `setitimer` *Function*

```
#include<unistd.h>
#include<signal.h>
#include<sys/time.h>

#define TIMERCNT 8
void   timerhandler();
int    timercnt;
struct timeval alarmtimes[TIMERCNT];

main(argc, argv)
     int   argc;
     char  *argv[]
{
    struct itimervaltimes;
    sigset_t   sigset;
    int   i;

    /* block SIGALRM */
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGALRM);
    sigprocmask(SIG_BLOCK, &sigset, NULL);

    /* set up handler for SIGALRM */
    signal(SIGALRM, timerhandler);

    /*
     * set up interval timer, starting in three seconds,
     * then every 1/3 second
     */
     times.it_value.tv_sec = 3;
     times.it_value.tv_usec = 0;
     times.it_interval.tv_sec = 0;
     times.it_interval.tv_usec = 333333;
     setitimer (ITIMER_REAL, &times, NULL);/

     /* now wait for the alarm */
     timehandler(0, 0, NULL, NULL);
     while (timercnt < TIMERCNT) {
     sigsuspend (sigset);
     }
     printtimes();
```

Code Example 11-13 Using the `setitimer` *Function (Continued)*

```
#include<unistd.h>
}
void
timehandler(sig, code, scp, addr)
     int        sig, code;
     struct    sigcontext*scp;
     char      *addr;
{
     gettimeofday(&alarmtimes[timercnt], NULL);
     timercnt++;
}
printtimes()
{
     int     i;

     for (i=0; i <TIMERCNT; i++) {
          printf("%d.%06d0, alarmtimes[i].tv_sec,
                    alarmtimes[i].tv_usec);
     }
}
```

Cautions When Using Realtime

Realtime response is only guaranteed if certain conditions are met. This section identifies these conditions and some of the more significant design errors that can cause problems or disable a system.

Most of the potential problems described here can degrade the response time of the system. One of the potential problems can freeze a workstation. Other, more subtle mistakes are priority inversion and system overload (too much to do).

Degrading Response Time

The problems described in this section all increase the response time of the system to varying extents. The degradation can be serious enough to cause an application to miss a critical deadline.

Realtime processing can also significantly impact the operation of aspects of other applications active on a system running a realtime application. Since realtime processes have higher priority, timesharing processes can be prevented from running for significant amounts of time. This can cause interactive activities, such as displays and keyboard response time, to be noticeably slowed.

System Response Time

System response under SunOS 5.x provides no bounds to the timing of I/O events. This means that synchronous I/O calls should never be included in any program segment which has time-critical execution. Even program segments that permit very large time bounds must not perform synchronous I/O. Mass storage I/O is such a case, where causing a read or write operation hangs the system while the operation takes place.

Interrupt Servicing

Prioritizing processes does not carry through to prioritizing the services of hardware interrupts that result from the actions of the processes. This means that interrupt processing for a device controlled by a realtime process is not, a priori, done before interrupt processing for another device controlled by a timeshare process.

Dynamic Linking

Programs can save significant amounts of memory by using dynamically linked, shared libraries. Dynamic linking is typically used in a *lazy binding* mode where undefined external function references are resolved upon the initial call to that function. This linking delay improves loading performance but could negatively impact execution performance. All programs, including realtime applications, can avoid lazy binding when dynamically linking by having the environment variable `LD_BIND_NOW` set to a non-NULL value when the program is invoked. This forces all undefined external function references to be resolved at runtime before the program's `main()` receives control.

Priority Inversion

A time-sharing process can block a realtime process by acquiring a resource that is required by a realtime process. Priority inversion is a condition that occurs when a higher priority process is blocked by a lower priority process. The term *blocking* describes a situation in which a process must wait for one or more processes to relinquish control of resources. If this blocking is prolonged, deadlines might be missed, even for lower level resources.

Sticky Locks

A page will be permanently locked into memory if its lock count reaches 65535 (0xFFFF). The value 0xFFFF is implementation defined and might change in future releases. Pages locked in this manner cannot be unlocked. Reboot the system to recover.

Runaway Realtime Processes

Runaway realtime processes can cause the system to halt or can slow the system response so much that the system appears to halt.

Note – If you have a runaway process, try the `L1-A` key sequence. You might have to repeat this procedure many times. If this doesn't work, disconnect the keyboard.

When a high priority realtime process will not relinquish control of the CPU, there is no simple way to regain control of the system until the infinite loop is forced to terminate. Such a runaway process will not respond to the control-C kill sequence.

Note – Attempts to use a shell at a higher priority than a runaway process will not succeed. The STREAMS processes that govern `tty` management are running at system priority, and so will not get scheduled. Therefore, keyboard input is not received by the shell, even when the shell is running at a higher priority.

Asynchronous I/O Behavior

There is no guarantee that asynchronous I/O operations will either be done in the sequence in which they are queued to the kernel, or that asynchronous operations will be returned to the caller in the sequence in which they were done.

File Allocation and I/O

SunOS 5.x provides no facilities to assure that files will be allocated as physically contiguous. For regular files, the `read` and `write` operations are always buffered. An application can use `mmap` and `msync` to effect direct I/O transfers between secondary storage and process memory.

Other Notes

If a single buffer is specified for a rapid sequence of calls to `aioread`, there is no guarantee of the state of the buffer between the time that the first call is made and the time that the last result is signaled to the caller.

Use a single `aio_result_t` structure for one asynchronous read or write at a time.

Bibliography

SunOS 5.2 System Services, Part Number 801-4055-05, SunSoft, Inc., Mountain View, CA.

Realtime Scheduling in SunOS 5.0, Khanna, Sebree, Zolnowsky , SunSoft, Inc., Mountain View, CA.

System Transition Guide for Application Developers, Part Number 800-6444-10, SunSoft, Inc., Mountain View, CA.

Administering Security, Performance, and Accounting, Part Number, 800-7362-10, SunSoft, Inc., Mountain View, CA.