Realtime Programming and Administration

7

This chapter describes writing and porting realtime applications to run under SunOS 5.x. This chapter is written for programmers experienced in writing realtime applications and administrators familiar with realtime processing and the SunOS system.

Basic Rules of Realtime Applications

Realtime response is guaranteed when 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).

A SunOS realtime process:

- runs in the `RT` scheduling class, as described in "Scheduling" on page 112
- locks down all the memory in its process address space, as described in "Memory Locking" on page 127
- is from a statically-linked program or from a program in which all dynamic binding is completed early, as described in "Shared Libraries" on page 109

7

Realtime operations are described in this chapter in terms of single-threaded processes, but the description can also apply to multithreaded processes (for detailed information about multithreaded processes, see the *Multithreaded Programming Guide*). 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.

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

- acquires the processor within the guaranteed dispatch latency period of becoming runnable (see"Dispatch Latency" on page 112)
- 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`(2) and `close`(2).

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, time-sharing 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 whose execution is time critical. 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.

A common application mistake is to perform I/O to get error message text from disk. This should be done from an independent non-realtime process or thread.

Interrupt Servicing

Interrupt priorities are independent of process priorities. 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 necessarily done before interrupt processing for another device controlled by a timeshare process.

Shared Libraries

Time-sharing processes can save significant amounts of memory by using dynamically linked, shared libraries. This type of linking is implemented through a form of file mapping. Dynamically linked library routines cause implicit reads.

Realtime programs can use shared libraries, yet avoid dynamic binding, by setting the environment variable `LD_BIND_NOW` to a non-NULL value when the program is invoked. This forces all dynamic linking to be bound before the program begins execution. See the *Linker and Libraries Guide* for more information.

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, even for lower level resources, deadlines might be missed.

By way of illustration, consider the case in Figure 7-1 where a high priority process wanting to use a shared resource gets blocked when a lower priority process holds the resource, and the lower priority process is preempted by an intermediate priority process. This condition can persist for a long time, arbitrarily long, in fact, since the amount of time the high priority process must wait for the resource depends not only on the duration of the critical section being executed by the lower priority process, but on the duration until the intermediate process blocks. Any number of intermediate processes can be involved.

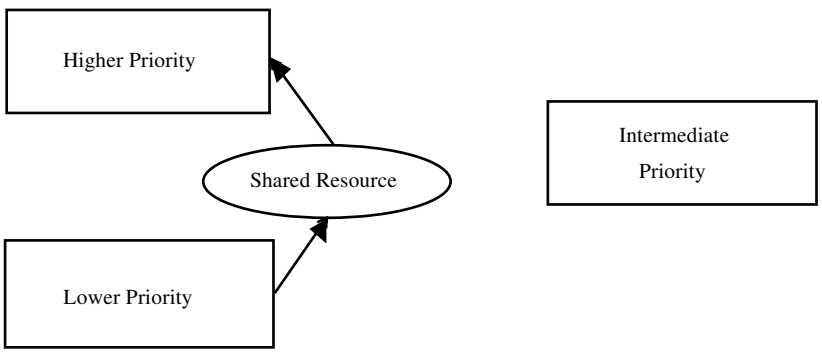

Figure 7-1 Unbounded Priority Inversion

Sticky Locks

A page is permanently locked into memory when its lock count reaches 65535 (0xFFFF). The value 0xFFFF is implementation-defined and might change in future releases. Pages locked this way cannot be unlocked.

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 on a SPARC system, try typing (Stop-A). 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.

Caution – Attempts to use a shell set 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.

I/O Behavior

Asynchronous I/O

There is no guarantee that asynchronous I/O operations will be done in the sequence in which they are queued to the kernel. Nor is there any guarantee that asynchronous operations will be returned to the caller in the sequence in which they were done.

If a single buffer is specified for a rapid sequence of calls to `aioread`(3), there is no guarantee about 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.

An individual `aio_result_t` structure can be used only for one asynchronous read or write at a time.

Realtime Files

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.

Scheduling

Realtime scheduling constraints are necessary to manage data acquisition or process control hardware. The realtime environment requires that a process be able to react to external events in a bounded amount of time. Such constraints can exceed the capabilities of a kernel designed to provide a "fair" distribution of the processing resources to a set of time-sharing processes.

This section describes the SunOS 5.x realtime scheduler, its priority queue, and how to use system calls and utilities that control scheduling. For more information about the functions described in this section, see the *man Pages(3): Library Routines*.

Dispatch Latency

The most significant element in scheduling behavior for realtime applications is the provision of a real-time scheduling class. The standard time-sharing scheduling class is not suitable for realtime applications because this scheduling class treats every process equally and has a limited notion of priority. Realtime applications require a scheduling class in which process priorities are taken as absolute and are changed only by explicit application operations.

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, realtime applications can be developed with a bounded dispatch latency.

Figure 7-2 illustrates the amount of time it takes an application to respond to a request from an external event.

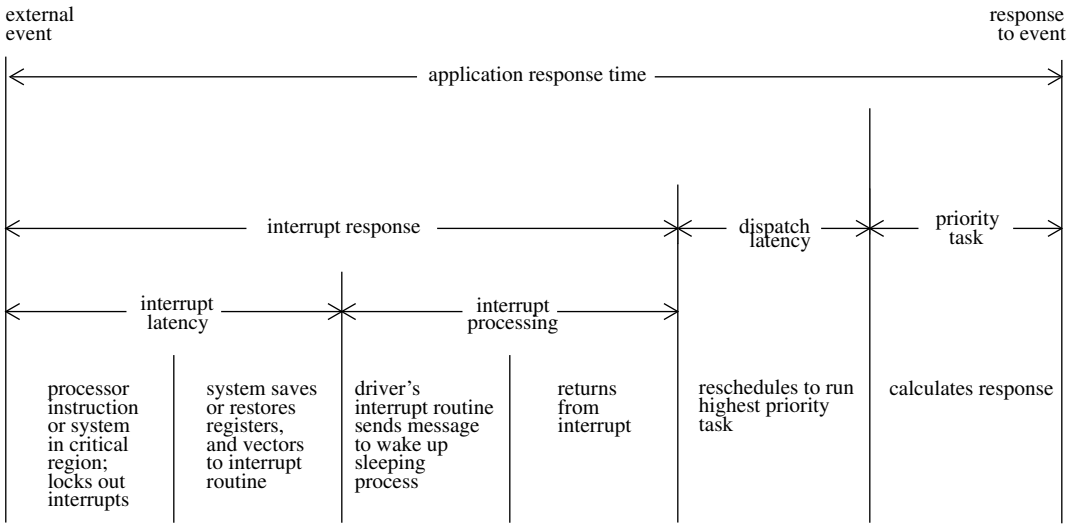

Figure 7-2 Application Response Time

The overall application response time is composed of the interrupt response time, the dispatch latency, and the time it takes the application itself 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 7-3 illustrates the internal dispatch latency/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.

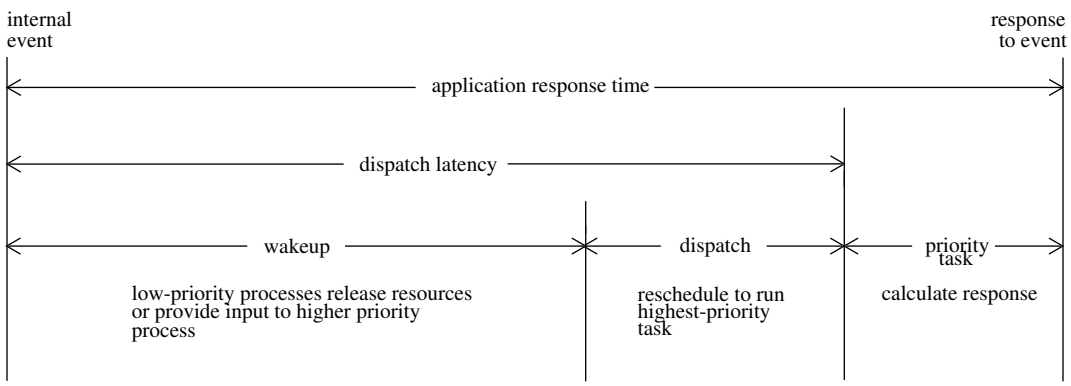

Figure 7-3 Internal Dispatch Latency

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 7-1, dispatch latency improves with a bounded number of processes.

Table 7-1 Realtime System Dispatch Latency with SunOS 5.x

	Dispatch Latency	
Workstation	**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 Sun workstation is an effective platform for the development of realtime applications. (We apologize that the examples are not of current products.)

Scheduling Classes

The SunOS 5.x kernel dispatches processes by priority. The scheduler (or dispatcher) supports the concept of scheduling classes. Classes are defined as Realtime (`RT`), System (`sys`), and 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, realtime processes have precedence over `sys` and `TS` processes, but administrators can configure systems so that `TS` and `RT` processes have overlapping priorities.

Figure 7-4 illustrates the concept of classes as viewed by the SunOS 5.x kernel.

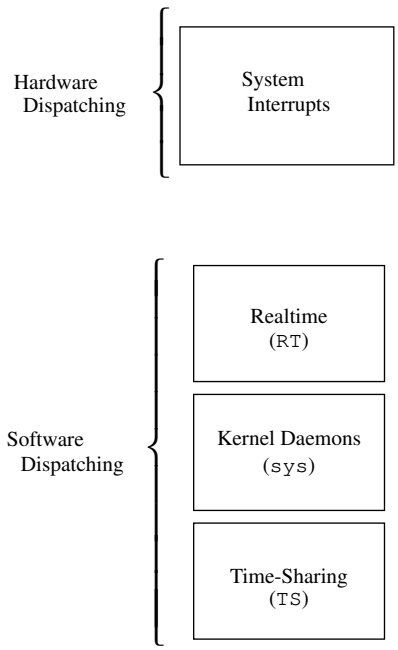

Figure 7-4 Dispatch Priorities for Scheduling Classes

At highest priority are the hardware interrupts; these cannot be controlled by 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. A process with an infinite time quantum ceases execution only when it terminates, blocks, or is preempted.

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

At lowest priority are the time-sharing (`TS`) processes. `TS` class processes are scheduled dynamically, with a few hundred milliseconds for each 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. Default time-sharing policy gives larger time slices to processes with lower priority.

A child process inherits the scheduling class and attributes of the parent process through `fork`(2). A process' scheduling class and attributes are unchanged by `exec`(2).

Different algorithms 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 7-5 illustrates, each class algorithm has its own method of nominating the highest priority process to place on the global run queue.

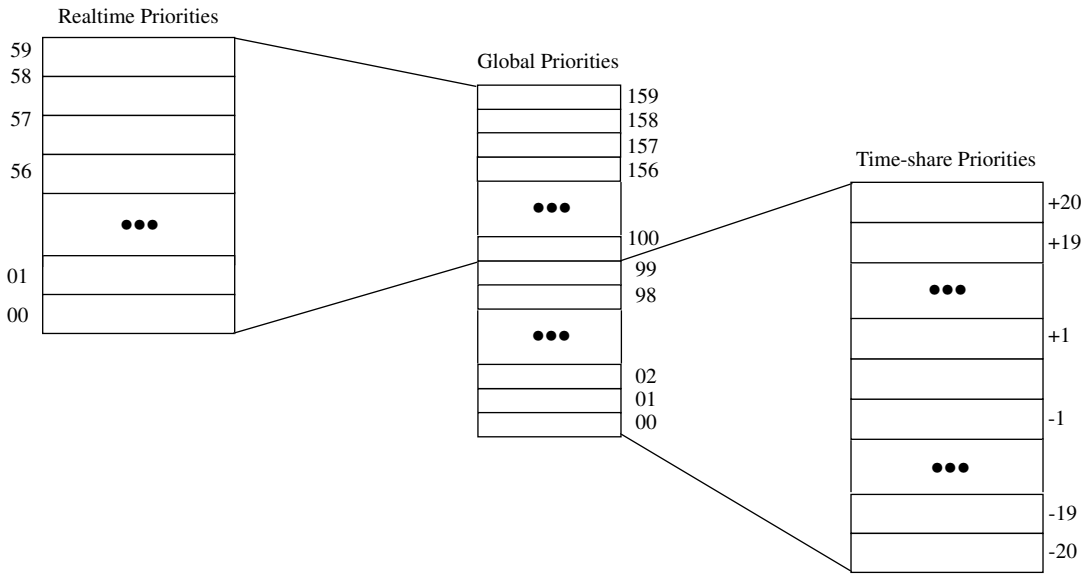

Figure 7-5 The Kernel Dispatch Queue

Each class has a set of priority levels that 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 that they be contiguous.

By default, the global priority values for time-sharing (`TS`) 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 (`RT`) 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 global priority. Each process is invoked with class specific information attached to it. A process is dispatched from the kernel dispatch table based upon its global priority.

Dispatching Processes

When a process is dispatched, the process' 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 currently running process.

Preemption

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

For example, suppose that process A performs a read from a peripheral device. Process A is put into the *sleep* state by the kernel. The kernel then finds that a lower priority process B is runnable, so process B is dispatched and begins execution. Eventually, the peripheral device interrupts, and the driver of the device is entered. The device driver makes process A runnable and returns. Rather than returning to the interrupted process B, the kernel now preempts B from processing and resumes execution of the awakened process A.

Another interesting situation occurs when several processes contend for kernel resources. When a lower priority process releases a resource for which a higher priority realtime process is waiting, the kernel immediately preempts the lower priority process and resumes execution of the higher priority process.

Kernel Priority Inversion

Priority inversion occurs when a higher priority process is blocked by one or more lower priority processes for a long 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 can lead to deadlines being missed, even for low levels of utilization.

The problem of priority inversion has been addressed for mutual-exclusion locks for the SunOS 5.x kernel by implementing a basic priority inheritance policy. The 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. User-level processes can still exhibit priority inversion, however.

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

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

/*
 * acquire synchronization object
 */
    ...
```

```
/*
 * execute the critical section
 */
    ...

/*
 * release synchronized object
 */
    ...

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

System Calls That Control Scheduling

System calls implemented for realtime scheduling include the library calls and functions listed in this section. For more detail about using these, see the *man Pages(3): Library Routines*.

Using `priocntl`(2)

Control over scheduling of active classes is handled with `priocntl`(2). Class attributes are inherited over `fork`(2) and `exec`(2), along with scheduling parameters and permissions required for priority control. These characteristics are true for both the `RT` and the `TS` classes.

`The priocntl`(2) 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`(2) system call also provides the more general interface for specifying an entire set of processes to which the system call is to apply.

The *idtype* and *id* arguments are used together to specify the set of processes on the queue. Depending upon the value of *idtype*, *id* can have values for a single process ID, a parent process ID, a process group ID, a session ID, a class ID, a user ID, a group ID, or a lightweight process ID.

The command arguments of `priocntl` can be one of: `PC_GETCID`, `PC_GETCLINFO`, `PC_GETPARMS`, or `PC_SETPARMS`. The real or effective ID of the calling process must match that of the affected process or processes, or must have super-user privilege.

PC_GETCID

This command takes the name field of a structure that contains a recognizable class name (`RT` for realtime and `TS` for time-sharing). The class ID and an array of class attribute data are returned.

PC_GETCLINFO

This command takes the ID field of a structure that contains a recognizable class identifier. The class name and an array of class attribute data are returned.

PC_GETPARMS

This command returns the scheduling class identifier and/or the class specific scheduling parameters of one of the specified processes. Even though `idtype & id might` specify a big set, `PC_GETPARMS` returns the parameter of only one process. It is up to the class to select which one.

PC_SETPARMS

This command sets the scheduling class and/or the class specific scheduling parameters of the specified process or processes.

Utilities that Control Scheduling

The administrative utilities that control process scheduling are `dispadmin`(1M) and `priocntl`(1). Both these utilities support the `priocntl`(2) system call with compatible options and loadable modules. Using these utilities provides system administration functions that control realtime process scheduling during runtime. For more details about using these utilities, see the *man Pages(1): User Commands* and the *System Administration Guide, Volume II* guide.

Using `priocntl`(1)

The `priocntl`(1) command sets and retrieves scheduler parameters for processes.

Using `dispadmin`(1M)

The `dispadmin`(1M) utility displays all current process scheduling classes by including 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.

The following options are also available:

Table 7-2 Class Options for the `dispadmin`(1M) 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 can be located

A class specific file containing the dispatch parameters can also be loaded during runtime. Use this file 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 `rt_dptbl`(4), and are listed in the example at the end of this section.

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`.
- A `rt_dptbl` module that provides the dispatch parameters and a routine to return pointers to `config_rt_dptbl`.

- The `dispadmin` executable.

Then load the class specific module with the following command, where `<module_name>` is the class specific module.

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

Then invoke the `dispadmin` command

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

The file must describe a table with the same number of entries as the table that is being overwritten.

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`(1M) 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 `i`th parameter structure in the array, `config_rt_dptbl[i]`.

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`(1M).

`rt_quantum`
The length of the time quantum allocated to processes at this level in ticks (see "Timestamp Functions" on page 145). 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`(1) command or the `priocntl`(2) 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. Two methods are described here.

The first method is 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 through using the `dispadmin`(1M) 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 manual page for `config_rt_dptbl` found in the *man Pages(1M): System Administration Commands*.

Following is an example of prioritized processes `rtdpent_t` with their associated time quantum `config_rt_dptbl[]` value as they might appear in `config_rt_dptbl[]`:

```
rtdpent_t    rt_dptbl[] = {
          /* prilevel  Time quantum */
                    100,     100,
                    101,     100,
                    102,     100,
                    103,     100,
                    104,     100,
                    105,     100,
                    106,     100,
                    107,     100,
                    108,     100,
                    109,     100,
                    110,     80,
                    111,     80,
                    112,     80,
                    113,     80,
                    114,     80,
                    115,     80,
                    116,     80,
                    117,     80,
                    118,     80,
                    119,     80,
                    120,     60,
                    121,     60,
                    122,     60,
                    123,     60,
                    124,     60,
                    125,     60,
                    126,     60,
                    127,     60,
                    128,     60,
                    129,     60,
                    130,     40,
                    131,     40,
                    132,     40,
                    133,     40,
                    134,     40,
                    135,     40,
                    136,     40,
                    137,     40,
                    138,     40
                    139,     40,
                    140,     20,
                    141,     20,
                    142,     20,
                    143,     20,
                    144,     20,
                    145,     20,
                    146,     20,
                    147,     20,
                    148,     20,
                    149,     20,
                    150,     10,
                    151,     10,
                    152,     10,
                    153,     10,
                    154,     10,
                    155,     10,
                    156,     10,
                    157,     10,
                    158,     10,
                    159,     10,
}
```

Memory Locking

Locking memory is one of the most important issues for realtime applications. In a realtime environment, a process must be able to guarantee continuous memory residence to reduce latency and to prevent paging and swapping.

This section describes the memory locking mechanisms available to realtime applications in SunOS 5.x. For more details about using memory management functions and calls, see the *man Pages(3): Library Routines* for pertinent manual pages.

Overview

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 processes' demand for memory. This is appropriate in a time-share environment, but it is often unacceptable for a realtime process. 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 superuser 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 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`(3) 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`(2) call is made. So, if a process with memory locked forks a child, the child must perform a memory locking operation in its own behalf to lock its own pages. Otherwise, the child process incurs copy-on-write page faults, 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`(3). 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 superuser process can request that all mappings within its address space be locked by a call to `mlockall`(3). If the flag `MCL_CURRENT` is set, all the existing memory mappings are locked. If the flag `MCL_FUTURE` is set, every mapping that is added to 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 might change in future releases. Pages locked in this manner cannot be unlocked. Reboot the system to recover.

7

High Performance I/O

This section describes I/O with realtime processes. With SunOS 5.x, several functions and calls are available within the libraries supplied to perform fast, asynchronous I/O operations. For robustness, SunOS provides file synchronization operations and modes to prevent information loss and data inconsistency. See the *man Pages(3): Library Routines* for more detailed information.

Asynchronous I/O

Standard `UNIX` I/O is generally synchronous to the application programmer. An application that calls `read`(2) or `write`(2) is not usually allowed to proceed until that system call has finished, 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 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, the application must assure the completion of prior operations before issuing the dependent requests.

Notification (SIGIO)

When an asynchronous I/O call returns successfully, the I/O operation has only been placed on the queue, waiting to be done. The actual operation also has a return value and a potential error identifier. These are the values that would have been returned to the caller as the result of a synchronous call.

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

Note that a person with two or more asynchronous I/O operations pending has no certain way to determine which request or even whether either request is the cause of the `SIGIO` signal. A process receiving a SIGIO should check all its conditions which could be generating the `SIGIO` signal.

Using aioread(3)

The `aioread`(3) function is the asynchronous version of `read`(2). In addition to the normal read arguments, `aioread` takes 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` call succeeds, the file pointer is updated to the position that would have resulted in a successful seek and read. The file pointer is also updated when a read fails to allow for subsequent read requests.

Using aiowrite(3)

The `aiowrite`(3) function is the asynchronous version of `write`(2). In addition to the normal write arguments, `aiowrite` takes 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 `aiowrite` call succeeds, the file pointer is updated to the position that would have resulted in a successful seek and write. The file pointer is also updated when a write fails to allow for subsequent write requests.

Using aiocancel(3)

The `aiocancel`(3) function attempts to cancel the asynchronous request whose `aio_result_t` structure is given as an argument. An `aiocancel` call succeeds only if the request is still queued. If the operation is in progress, `aiocancel` fails.

Using aiowait(3)

A call to the `aiowait`(3) function blocks the calling process until at least one outstanding asynchronous I/O operation is completed. The timeout parameter points to a maximum interval to wait for I/O completion. A timeout value of zero specifies that no wait is wanted. The `aiowait` function returns a pointer to the `aio_result_t` structure for the completed operation.

Using poll(2)

When you prefer to poll devices rather than to depend on a `SIGIO` interrupt, use the `poll`(2) system call. You can also poll to determine the origin of an `SIGIO` interrupt.

Using close(2)

Files are closed by a call to `close`(2). 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.

Only asynchronous I/O requests that are queued to the specified file descriptor are cancelled when a file is closed. Any I/O requests that are pending for other file descriptors are not cancelled.

Synchronized I/O

Applications may need to guarantee that information has been written to stable storage, or that file updates are performed in a particular order. Synchronized I/O provides for these needs.

Modes of Synchronization

Under SunOS 5.x, data is successfully transferred for a write operation to a regular file when the system ensures that all data written is readable on any

subsequent open of the file (even one that follows a system or power failure) in the absence of a failure of the physical storage medium. Data is successfully

transferred for a read operation when an image of the data on the physical storage medium is available to the requesting process. An I/O operation is complete when either the associated data been successfully transferred or the operation has been diagnosed as unsuccessful.

An I/O operation has reached synchronized I/O data integrity completion when:

For reads, the operation has been completed or diagnosed if unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests affecting the data to be read at the time that the synchronized read operation was requested, these write requests are successfully transferred prior to reading the data.

For writes, the operation has been completed or diagnosed if unsuccessful. The write is complete only when the data specified in the write request is successfully transferred, and all file system information required to retrieve the data is successfully transferred.

File attributes that are not necessary for data retrieval (access time, modification time, status change time) are not successfully transferred prior to returning to the calling process.

Synchronized I/O file integrity completion is identical to synchronized I/O data integrity completion with the addition that all file attributes relative to the I/O operation (including access time, modification time, status change time) must be successfully transferred prior to returning to the calling process.

Synchronizing a File

The `fsync(3C)` and `fdatasync(3R)` functions explicitly synchronize a file to secondary storage:

```
int fsync (int fildes);
int fdatasync (int fildes);
```

The `fsync()` guarantees the function is synchronized at the I/O file integrity completion level, while The `fdatasync()` guarantees the function is synchronized at the I/O data integrity completion level.

Applications can arrange that each I/O operation is synchronized before the operation completes. Setting the `O_DSYNC` flag on the file description via `open(2)` or `fcntl(2)` ensures that all I/O writes `(write(2), aiowrite(3))` have reached I/O data completion before the operation is indicated as completed. Setting the O_SYNC flag on the file description ensures that all I/O writes have reached I/O file completion before the operation is indicated as completed. Setting the `O_RSYNC` flag on the file description ensures that all I/O reads `(read(2),aioread(3))` have reached the same level of completion as request for writes by the setting `O_DSYNC` or `O_SYNC` on the descriptor.

Interprocess Communication

This section describes the interprocess communication (IPC) functions of SunOS 5.x as they relate to realtime processing. Signals, pipes, FIFOs (named pipes), message queues, shared memory, file mapping, and semaphores are described here. For more information about the libraries, functions, and routines useful for interprocess communication, see chapter three, "Interprocess Communication," and the *man Pages(3): Library Routines*.

Overview

Realtime processing often requires fast, high-bandwidth 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. Messages can become intermingled by multiple writers or can be torn apart by multiple readers.

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.

7

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.

Signals

Signals may be used to send a small amount of information between processes. The sender can use the `sigqueue(3R)` function to send a signal together with a small amount of information to a target process:

```
int sigqueue(pid_t pid, int signo, const union sigval value);
union sigval {
    int     sival_int;      /* integer value */
    void    *sival_ptr;     /* pointer value */
};
```

The target process must have the `SA_SIGINFO` bit set for the given signal number (see `sigaction(2)`), in order that occurrences of the signal occurring when that signal is already pending will be queued.

The target process can receive the signals either synchronously or asynchronously. By leaving that signal blocked `(sigprocmask(2))` and calling either `sigwaitinfo(3R)` or `sigtimedwait(3R)`, the signal will be received synchronously, with the value sent by the caller of `sigqueue()` being stored in the `si_value` member of the `siginfo_t` argument. By leaving the signal unblocked, the arrival will be delivered to the signal handler specified by `sigaction()`, with the value appearing in the `si_value` of the `siginfo_t` argument to the handler.

Only a fixed number of signals with associated values can be sent by a process and remain undelivered. Storage for {`SIGQUEUE_MAX`} signals is allocated at the first call to `sigqueue()`. Thereafter, a call to `sigqueue()` either successfully enqueues at the target process or fails within a bounded amount of time.

Pipes

Pipes provide one-way communication between processes. Pipes are created by a process using the `pipe`(2) system call. The `pipe`(2) system call returns two file descriptors, the first for reading and the second for writing. Once the pipe

is created, the process must create other processes with the `fork`(2) system call, which allows the processes to communicate among themselves. Processes must have a common ancestor in order to communicate with pipes.

Data passed through a pipe is treated as a conventional UNIX byte stream. Data is sent into the pipe by calls to `write`(2V) using the writing file descriptor.

Data is received from the pipe by calls to `read`(2V) using the reading file descriptor. The `read` call is usually a blocking function: it does not return to the caller until some data can be returned. To get a non-blocking `read`, the pipe can be set so that it doesn't block by using the `ioctl`(2) or `fcntl`(2) functions.

A `read` on an empty, non-blocking pipe returns with an indication that no data is available.

Named Pipes

SunOS 5.x provides *named pipes* or *FIFO*s. The FIFO is more flexible than the pipe because it is a named entity in a directory. Once created, a FIFO can be opened by any process that has legitimate access to it. Processes do not have to share a parent and there is no need for a parent to initiate the pipe and pass it to the descendants. A FIFO can be created with `mknod`(2).

A process connects to a FIFO through a call to `open`(2V). A process that opens a FIFO for a read is blocked until that FIFO has been opened by a process for writing. The decision about whether or not reads block is made in the `open` call or by using a subsequent call to `fcntl`.

As with pipes, data in a FIFO is treated as a byte stream. Input is obtained from a FIFO with calls to `read` and output is sent with calls to `write`. A process ends use of a FIFO through a call to `close`(2).

IPC Message Queues

IPC message queues provide a powerful means of communicating between processes by allowing any number of processes to send and receive from the same message queue. Messages are passed as blocks of arbitrary size, not as byte streams. Each message includes an integer type, which can be used by application convention as a message priority, or as message categories. The

latter usage provides multiple flows of messages with a single message queue. This can be simpler than opening an arbitrary number of pipes or FIFOs when a large number are required. Note that IPC insertion is strictly FIFO.

IPC message queue structures are initiated by a call to `msgget`(2). A message is sent by a call to `msgsnd`(2), and `msgrcv`(2) is called to extract a message from the queue structure. The `msgctl`(2) system call controls various functions on a message queue structure, including removal.

IPC Semaphores

The IPC semaphore is a mechanism that synchronizes access to shared resources. IPC semaphores are created in arrays, each element of which can be used to control the execution of processes that call for operations on the array elements.

Create an array of IPC semaphores with a call to `semget`(2). Query or set individual semaphores or the complete array of semaphores with calls to `semctl`(2). Acquire and release a semaphore or the array of semaphores with calls to `semop`(2). Look in `intro`(2) for more information about information structures and the operation of IPC semaphores.

Note that using IPC semaphores can cause priority inversions unless these are explicitly avoided by the techniques mentioned earlier in this chapter.

Shared Memory

The fastest way for processes to communicate is directly, through a shared segment of memory. A common memory area is added to the address space of processes wishing to communicate. Applications use stores to send data and fetches to receive communicated data. SunOS 5.x provides two mechanisms for shared memory: memory mapped files and IPC shared memory.

The major difficulty with shared memory is that results can be wrong when more than two processes are trying to read and write in it at the same time. See “Shared Memory Synchronization” on page 138 for more information.

Memory Mapped Files

The system call `mmap`(2) connects a shared memory segment to the caller's memory. The caller specifies the shared segment by address and length. The caller must also specify access protection flags and how the mapped pages are managed.

The `mmap`(2) system call can also be used to map a file or a segment of a file to a process's memory. While this technique is very convenient in some applications, it is easy to forget that any access to the mapped file segment might result in implicit I/O. This can make an otherwise bounded process have unpredictable response times. The function `msync`(3) forces immediate or eventual copies of the specified memory segment to its permanent storage location(s).

The process can later change the access protection of the segment by the system call `mprotect`(2). The segment is specified by address and length.

The system call `munmap`(2) disconnects a mapped memory segment. The segment is specified by address and length.

Fileless Memory Mapping

The zero special file, `/dev/zero`(4S), can be used to create an unnamed, zero initialized memory object. The length of the memory object is the least number of pages that contain the mapping. The object can be shared only by descendants of a common ancestor process.

IPC Shared Memory

A `shmget`(2) call can be used either to create and obtain a shared memory segment or to obtain an existing shared memory segment. The call specifies an identifying key, the size of the segment, and a flag parameter. The flags contain the usual access permission bits and can contain a flag to create a new segment. The `shmget` function returns an identifier that is analogous to a file identifier.

The shared memory segment is made accessible to the process by a call to `shmat`(2). The shared memory segment becomes a virtual segment of the process memory space and can be freely written to and read from depending on creating permissions. The shared memory segment is detached from a

process's memory space by a call to `shmdt`(2). The `shmctl` system call can be used to control a variety of functions on an IPC shared memory object, including removal.

Shared Memory Synchronization

In sharing memory, a portion of memory can be mapped into the address space of one or more processes. This allows shared access to that portion of memory by the attached processes. No method of coordinating access is automatically provided, so nothing prevents two processes from writing to the shared memory at the same time. For this reason, it is typically used with semaphores, which are used to synchronize processes.

Choice of IPC Mechanism

Applications can have specific functional requirements that determine which IPC mechanism to use. If one of several mechanisms can be used, the application writer determines which mechanism performs best for the application. The SunOS 5.x interprocess communication facilities are sensitive to application behavior. Determine which mechanism provides the best response capabilities by measuring the throughput capacity of each mechanism for the particular combination of message sizes used in the application

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 *Transport Interfaces Programming Guide* and the*man Pages(3): Library Routines*.

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`(2) 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`(2) 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 with respect to other devices.

7

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`(3), and its identity established using `t_bind`(3), the endpoint can be configured for asynchronous service. This is done by using the `fcntl`(2) function to set the `O_NONBLOCK` flag on the endpoint. Thereafter, calls to `t_sndudata`(3) for which no buffer space is immediately available return `-1` with `t_errno` set to `TFLOW`. Likewise, calls to `t_rcvudata`(3) for which no data are available return `-1` with `t_errno` set to `TNODATA`.

Asynchronous Network Transfers

Although an application can use the `poll`(2) 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`(2) 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 7-1 Asynchornous Network Transfers

```
#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, bin

    /* 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;

    for (;;) {
        ... /* initialize ud */
        if (t_rcvudata(fd, &ud, &flags) < 0) {
            if (t_errno == 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`(3) function to initiate setting up the transfer. Then the `t_rcvconnect`(3) 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`(3) function. To accept the connection, the process uses the `t_accept`(3) function. The responding endpoint must be separately configured for asynchronous data transfers.

The following example illustrates how to request a connection asynchronously

Code Example 7-2 Asynchronous Connection Request

```
#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 7-3 Asynchronous LIstening

```
#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`(2) 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 this example, the parent process prints out information about the test file, and 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 7-4 File Descriptor Transfer

```
#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]);
    } 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);
}
```

Timers

This section describes the timing facilities available for realtime applications under SunOS 5.x. Realtime applications that want to take advantage of these mechanisms will require detailed information from the manual pages of the routines listed in this section. These can be found in the *man Pages(3): Library Routines*.

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. Interval timers allow an application to wake up at specified times and to schedule activities based on the passage of time. 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.

Timestamp Functions

Two functions provide timestamps. The `gettimeofday`(2) function provides the current time in a *timeval* structure, representing the time in seconds and microseconds since midnight, Greenwich Mean Time, on January 1, 1970. The `clock_gettime`(`3R`) function, with a clockid of `CLOCK_REALTIME`, provides the current time in a *timespec* structure, representing in seconds and nanoseconds the same time interval returned by `gettimeofday`().

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.

7

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.

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 is then disarmed. Such a timer might be useful for clearing buffers after the data has been transferred to storage, or to time-out an operation that should have finished.

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 (typically 10 ms).

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.

There are twos set of timers interfaces in SunOS 5.x. The `setitimer(2)` and `getitimer(2)` interfaces provide access to fixed set timers, called the BSD timers, using the timeval structure to specify time intervals. The POSIX timers are specifically related to POSIX clocks; the only POSIX clock currently supported is CLOCK_REALTIME. POSIX timer operations are expressed in terms of the timespec structure.

The functions `getitimer(2)` and `setitimer(2)` respectively retrieve and establish the value of the specified BSD interval timer. There are three BSD interval timers available to a process, including a realtime timer designated `ITIMER_REAL`. If a BSD timer is armed and allowed to expire, the system sends a signal appropriate to the timer to the process that set the timer.

The `timer_create(3R)` function can create up to {TIMER_MAX} POSIX timers. At the time of creation, the caller can specify what signal and what associated value will be sent to the process upon timer expiration. The `timer_gettime(3R)` and `timer_settime(3R)` functions respectively retrieve and establish the value of the specified POSIX interval timer.

Expirations of POSIX timers while the required signal is pending delivery are counted, and the function `timer_getoverrun(3R)` retrieves the count of such expirations. The function `timer_delete(3R)` deallocates a POSIX timer.

Code Example 7-5 illustrates how to use the `setitimer` interface to generate a periodic interrupt, and how to control the arrival of timer interrupts.

Code Example 7-5 Controlling Timer Interrupts

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

#define TIMERCNT 8

voidtimerhandler();
int timercnt;
structtimeval alarmtimes[TIMERCNT];

main()
{
    struct itimerval times;
    sigset_tsigset;
    int     i, ret;
    struct sigaction act;

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

    /* set up handler for SIGALRM */
    act.sa_handler = timerhandler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_SIGINFO;
    sigaction(SIGALRM, &act, NULL);
    /*
     * 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;
    ret = setitimer(ITIMER_REAL, &times, NULL);
    printf("main:setitimer ret = %d\n", ret);
```

```
    /* now wait for the alarms */
    sigemptyset(&sigset);
    timerhandler(0, 0, NULL, NULL);
    while (timercnt < TIMERCNT) {
        ret = sigsuspend(&sigset);
    }
    printtimes();
}

void timerhandler(sig, siginfo, context)
    int     sig;
    siginfo_tsiginfo;
    void    *context;
{
    printf("timerhandler:start\n");
    gettimeofday(&alarmtimes[timercnt], NULL);
    timercnt++;
    printf("timerhandler:timercnt = %d\n", timercnt);
}

printtimes()
{
    int i;

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