Programming Interfaces Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 816–5042–10
December 2003

030929@6671

Contents

Preface

This book describes the SunOS™ 5.9 network and system interfaces used by application developers.

SunOS 5.9 is fully compatible with UNIX® System V, Release 4 (SVR4) and conforms to the third edition of the System V Interface Description (SVID). SunOS 5.9 supports all System V network services.

All utilities, their options, and library functions in this manual reflect SunOS Release 5.8.

Audience

This book is intended for programmers who are new to the SunOS™ platform or want more familiarity with some portion of the interfaces provided. Additional interfaces and facilities for networked applications are described in the *ONC+ Developer's Guide*.

This manual assumes basic competence in programming, a working familiarity with the C programming language, and familiarity with the UNIX operating system, particularly networking concepts. For more information on UNIX networking basics, see W. Richard Stevens' *UNIX Network Programming*, second edition, Upper Saddle River, Prentice Hall, 1998.

Organization of the Manual

The services and capabilities of the basic system interfaces and basic network interfaces of the SunOS 5.9 platform are described in the following chapters.

Chapter 1 describes the interfaces that create and manage memory mappings, do high performance file I/O, and control other aspects of memory management.

Chapter 2 describes the Application Programming Interface (API) framework and library functions for remote shared memory.

Chapter 3 describes the operation of the SunOS process scheduler, modification of the scheduler's behavior, the scheduler's interactions with process management interfaces, and performance effects.

Chapter 5 describes basic and old-style buffered file I/O and other elements of I/O.

Chapter 6 describes older forms of non-networked interprocess communication.

Chapter 7 describes the use of sockets, which are the basic mode of networked communication.

Chapter 8 describes the use of XTI and TLI to do transport-independent networked communication.

Chapter 9 describes the network selection mechanisms used by applications to select a network transport and its configuration.

Chapter 10 describes real-time programming facilities in the SunOS environment and their use.

Chapter 11 describes the Solaris™ Application Binary Interface (ABI) and the tools used to verify an application's compliance with the Solaris™ ABI, `appcert` and `apptrace`.

Appendix A describes UNIX domain sockets.

Accessing Sun Documentation Online

The docs.sun.com℠ Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Ordering Sun Documentation

Sun Microsystems offers select product documentation in print. For a list of documents and how to order them, see "Buy printed documentation" at http://docs.sun.com.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Memory Management

This chapter describes an application developer's view of virtual memory in SunOS.

- "Memory Management Interfaces" on page 15 describes interfaces and cache control.
- Library level dynamic memory allocation and debugging are described in "Library-Level Dynamic Memory" on page 17.
- "Other Memory Control Interfaces" on page 20 describes other memory control interfaces.

Memory Management Interfaces

Applications use the virtual memory facilities through several sets of interfaces. This section summarizes these interfaces. This section also provides examples of the interfaces' use.

Creating and Using Mappings

`mmap`(2) establishes a mapping of a named file system object into a process address space. A named file system object can also be partially mapped into a process address space. This basic memory management interface is very simple. Use `open`(2) to open the file, then use `mmap`(2) to create the mapping with appropriate access and sharing options. Then, proceed with your application.

The mapping established by `mmap`(2) replaces any previous mappings for the specified address range.

The flags MAP_SHARED and MAP_PRIVATE specify the type of mapping. You must specify a mapping type. If the MAP_SHARED flag is set, write operations modify the mapped object. No further operations on the object are needed to make the change. If the MAP_PRIVATE flag is set, the first write operation to the mapped area creates a copy of the page. All further write operations reference the copy. Only modified pages are copied.

A mapping type is retained across a fork(2).

After you have established the mapping through mmap(2), the file descriptor used in the call is no longer used. If you close the file, the mapping remains until munmap(2) undoes the mapping. Creating a new mapping replaces an existing mapping.

A mapped file can be shortened by a call to truncate. An attempt to access the area of the file that no longer exists causes a SIGBUS signal.

Mapping /dev/zero gives the calling program a block of zero-filled virtual memory. The size of the block is specified in the call to mmap(2). The following code fragment demonstrates a use of this technique to create a block of zeroed storage in a program. The block's address is chosen by the system.

```
removed to fr.ch4/pl1.create.mapping.c
```

Some devices or files are useful only when accessed by mapping. Frame buffer devices used to support bit-mapped displays are an example of this phenomenon. Display management algorithms are much simpler to implement when the algorithms operate directly on the addresses of the display.

Removing Mappings

munmap(2) removes all mappings of pages in the specified address range of the calling process. munmap(2) has no affect on the objects that were mapped.

Cache Control

The virtual memory system in SunOS is a cache system, in which processor memory buffers data from file system objects. Interfaces are provided to control or interrogate the status of the cache.

Using mincore

The mincore(2) interface determines the residency of the memory pages in the address space covered by mappings in the specified range. Because the status of a page can change after mincore checks the page but before mincore returns the data, returned information can be outdated. Only locked pages are guaranteed to remain in memory.

Using mlock and munlock

mlock(3C) causes the pages in the specified address range to be locked in physical memory. References to locked pages in this process or in other processes do not result in page faults that require an I/O operation. Because this I/O operation interferes with normal operation of virtual memory, as well as slowing other processes, the use of mlock is limited to the superuser. The limit to the number of pages that can be locked in memory is dependent on system configuration. The call to mlock fails if this limit is exceeded.

munlock releases the locks on physical pages. If multiple mlock calls are made on an address range of a single mapping, a single munlock call releases the locks. However, if different mappings to the same pages are locked by mlock, the pages are not unlocked until the locks on all the mappings are released.

Removing a mapping also releases locks, either through being replaced with an mmap(2) operation or removed with munmap(2).

The copy-on-write event that is associated with a MAP_PRIVATE mapping transfers a lock on the source page to the destination page. Thus locks on an address range that includes MAP_PRIVATE mappings are retained transparently along with the copy-on-write redirection. For a discussion of this redirection, see "Creating and Using Mappings" on page 15.

Using mlockall and munlockall

mlockall(3C) and munlockall(3C) are similar to mlock and munlock, but mlockall and munlockall operate on entire address spaces. mlockall sets locks on all pages in the address space and munlockall removes all locks on all pages in the address space, whether established by mlock or mlockall.

Using msync

msync(3C) causes all modified pages in the specified address range to be flushed to the objects mapped by those addresses. This command is similar to fsync(3C), which operates on files.

Library-Level Dynamic Memory

Library-level dynamic memory allocation provides an easy-to-use interface to dynamic memory allocation.

Dynamic Memory Allocation

The most often used interfaces are:

- malloc(3C)
- free(3C)
- calloc(3C)
- cfree(3MALLOC)

Other dynamic memory allocation interfaces are memalign(3C), valloc(3C), and realloc(3C)

- malloc returns a pointer to a block of memory at least as large as the amount of memory that is requested. The block is aligned to store any type of data.
- free returns the memory that is obtained from malloc, calloc, realloc, memalign, or valloc to system memory. Trying to free a block that was not reserved by a dynamic memory allocation interface is an error that can cause a process to crash.
- calloc returns a pointer to a block of memory that is initialized to zeros. Memory reserved by calloc can be returned to the system through either cfree or free. The memory is allocated and aligned to contain an array of a specified number of elements of a specified size.
- memalign allocates a specified number of bytes on a specified alignment boundary. The alignment boundary must be a power of 2.
- valloc allocates a specified number of bytes that are aligned on a page boundary.
- realloc changes the size of the memory block allocated to a process. realloc can be used to increase or reduce the size of an allocated block of memory. realloc is the only way to shrink a memory allocation without causing a problem. The location in memory of the reallocated block might be changed, but the contents up to the point of the allocation size change remain the same.

Dynamic Memory Debugging

The Sun™ WorkShop package of tools is useful in finding and eliminating errors in dynamic memory use. The Run Time Checking (RTC) facility of the Sun WorkShop uses the functions that are described in this section to find errors in dynamic memory use.

RTC does not require the program be compiled using -g in order to find all errors. However, symbolic (-g) information is sometimes needed to guarantee the correctness of certain errors, particularly errors that are read from uninitialized memory. For this reason, certain errors are suppressed if no symbolic information is available. These errors are rui for a.out and rui + aib + air for shared libraries. This behavior can be changed by using suppress and unsuppress.

check -access

The `-access` option turns on access checking. RTC reports the following errors:

`baf`	Bad free
`duf`	Duplicate free
`maf`	Misaligned free
`mar`	Misaligned read
`maw`	Misaligned write
`oom`	Out of memory
`rua`	Read from unallocated memory
`rui`	Read from uninitialized memory
`rwo`	Write to read-only memory
`wua`	Write to unallocated memory

The default behavior is to stop the process after detecting each access error. This behavior can be changed using the `rtc_auto_continue` `dbxenv` variable. When set to `on`, RTC logs access errors to a file. The file name is determined by the value of the `rtc_error_log_file_name` `dbxenv` variable. By default, each unique access error is only reported the first time the error happens. Change this behavior using the `rtc_auto_suppress` `dbxenv` variable. The default setting of this variable is `on`.

check -leaks [-frames *n*] [-match *m*]

The `-leaks` option turns on leak checking. RTC reports the following errors:

`aib`	Possible memory leak – The only pointer points in the middle of the block
`air`	Possible memory leak – The pointer to the block exists only in register
`mel`	Memory leak – No pointers to the block

With leak checking turned on, you get an automatic leak report when the program exits. All leaks, including potential leaks, are reported at that time. By default, a non-verbose report is generated. This default is controlled by the `dbxenv` `rtc_mel_at_exit`. However, you can ask for a leak report at any time.

The `-frames` *n* variable displays up to *n* distinct stack frames when reporting leaks. The `-match` *m* variable combines leaks. If the call stack at the time of allocation for two or more leaks matches *m* frames, these leaks are reported in a single combined leak report. The default value of *n* is the larger of 8 or the value of *m*. The maximum value of *n* is 16. The default value of *m* is 2.

`check -memuse [-frames` *n*`] [-match` *m*`]`

The `-memuse` option turns on memory use (`memuse`) checking. Using `check -memuse` implies using `check -leaks`. In addition to a leak report at program exit, you also get a report listing blocks in use, `biu`. By default, a non-verbose report on blocks in use is generated. This default is controlled by the `dbxenv rtc_biu_at_exit`. At any time during program execution, you can see where the memory in your program has been allocated.

The `-frames` *n* and `-match` *m* variables function as described in the following section.

`check -all [-frames` *n*`] [-match` *m*`]`

Equivalent to `check -access; check -memuse [-frames` *n*`] [-match` *m*`]`. The value of `rtc_biu_at_exit` `dbxenv` variable is not changed with `check -all`. So, by default, no memory use report is generated at exit.

`check [`*funcs*`] [`*files*`] [`*loadobjects*`]`

Equivalent to `check -all; suppress all; unsuppress all in` *funcs files loadobjects*. You can use this option to focus RTC on places of interest.

Other Memory Control Interfaces

This section discusses additional memory control interfaces.

Using `sysconf`

`sysconf`(3C) returns the system dependent size of a memory page. For portability, applications should not embed any constants that specify the size of a page. Note that varying page sizes are not unusual, even among implementations of the same instruction set.

Using `mprotect`

`mprotect`(2) assigns the specified protection to all pages in the specified address range. The protection cannot exceed the permissions that are allowed on the underlying object.

Using brk and sbrk

A *break* is the greatest valid data address in the process image that is not in the stack. When a program starts executing, the break value is normally set by execve(2) to the greatest address defined by the program and its data storage.

Use brk(2) to set the break to a greater address. You can also use sbrk(2) to add an increment of storage to the data segment of a process. You can get the maximum possible size of the data segment by a call to getrlimit(2).

```
caddr_t
 brk(caddr_t addr);

 caddr_t
 sbrk(intptr_t incr);
```

brk identifies the lowest data segment location not used by the caller as *addr*. This location is rounded up to the next multiple of the system page size.

sbrk, the alternate interface, adds *incr* bytes to the caller data space and returns a pointer to the start of the new data area.

CHAPTER 2

Remote Shared Memory API for Solaris Clusters

Solaris Cluster OS™ systems can be configured with a memory-based interconnect such as Dolphin-SCI and layered system software components. These components implement a mechanism for user-level inter-node messaging that is based on direct access to memory residing on remote nodes. This mechanism is referred to as Remote Shared Memory (RSM). This chapter defines the RSM Application Programming Interface (RSMAPI).

- "API Framework" on page 24 describes the RSM API framework.
- "API Library Functions" on page 25 covers RSM API library functions.
- "RSMAPI Usage Example" on page 48 shows an example of use.

Overview of the Shared Memory Model

In the shared memory model, an application process creates an RSM export segment from the process's local address space. One or more remote application processes create an RSM import segment with a virtual connection between export and import segments across the interconnect. All processes make memory references for the shared segment with addresses local to their specific address space.

An application process creates an RSM export segment by allocating locally addressable memory to the export segment. This allocation is done by using one of the standard Solaris interfaces, such as System V Shared Memory, `mmap`(2), or `valloc`(3C). The process then calls on the RSMAPI for the creation of a segment, which provides a reference handle for the allocated memory. The RSM segment is published through one or more interconnect controllers. A published segment is remotely accessible. A list of access privileges for the nodes that are permitted to import the segment is also published.

A segment ID is assigned to the exported segment. This segment ID, along with the cluster node ID of the creating process, allows an importing process to uniquely specify an export segment. Successfully creating an export segment returns a segment handle to the process for use in subsequent segment operations.

An application process obtains access to a published segment by using the RSMAPI to create an import segment. After creating the import segment, the application process forms a virtual connection across the interconnect. Successfully creating this import segment returns an RSM import segment handle to the application process for use in subsequent segment import operations. After establishing the virtual connection, the application might request RSMAPI to provide a memory map for local access, if supported by the interconnect. If memory mapping is not supported, the application can use memory access primitives provided by RSMAPI.

The RSMAPI provides a mechanism to support remote access error detection and to resolve write-order memory model issues. This mechanism is called a *barrier*.

RSMAPI provides a notification mechanism to synchronize local and remote accesses. An export process can call a function to block while an import process finishes a data write operation. When the import process finishes writing, the process unblocks the export process by calling a signal function. Once unblocked, the export process processes the data.

API Framework

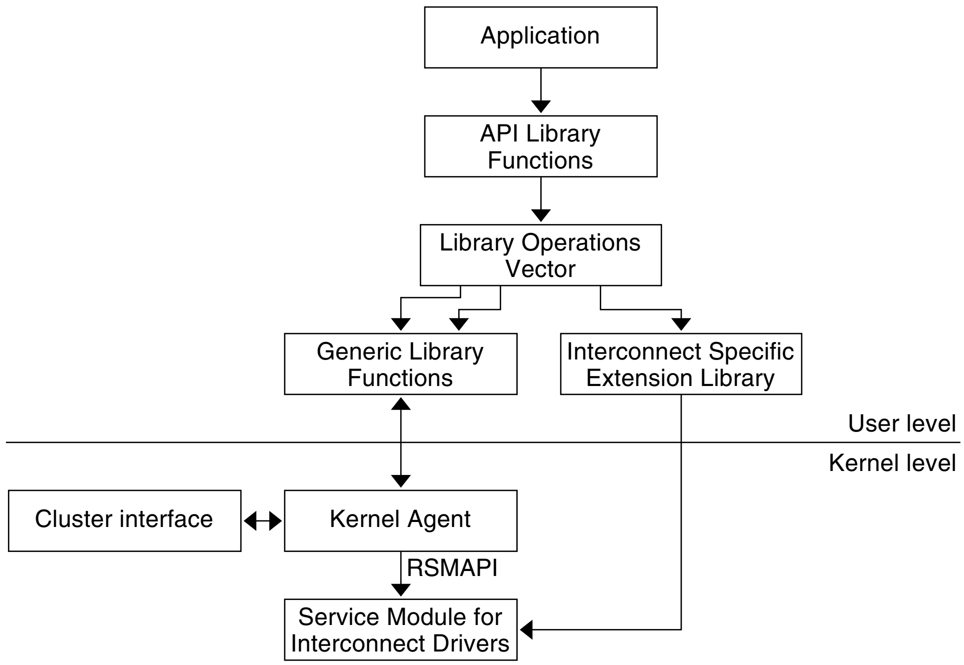
The RSM application support components are delivered in software packages as follows:

- `SUNWrsm`
  - A shared library (`/usr/lib/librsm.so`) that exports the RSMAPI functions.
  - A Kernel Agent (KA) pseudo device driver (`/usr/kernel/drv/rsm`) that interfaces with the memory interconnect driver through the RSMAPI interface on behalf of the user library.
  - A cluster interface module for obtaining interconnect topology.
- `SUNWrsmop`

  Interconnect driver service module (`/kernel/misc/rsmops`).
- `SUNWrsmdk`

  Header files providing API function and data structure prototypes (`/opt/SUNWrsmdk/include`).
- `SUNW`*interconnect*

An optional extension to `librsm.so` that provides RSM support for the specific interconnect is configured in the system. The extension is provided in the form of a library, `librsm`*interconnect*`.so`.



API Library Functions

The API library functions support the following operations:

- Interconnect controller operations
- Cluster topology operations
- Memory segment operations, including segment management and data access
- Barrier operations
- Event operations

Interconnect Controller Operations

The controller operations provide mechanisms for obtaining access to a controller. Controller operations can also determine the characteristics of the underlying interconnect. The following list contains information on controller operations:

- Get controller
- Get controller attributes
- Release controller

rsm_get_controller

`int` **`rsm_get_controller`**`(`*char *name, rsmapi_controller_handle_t *controller*`);`

The `rsm_get_controller` operation acquires a controller handle for the given controller instance, such as `sci0` or `loopback`. The returned controller handle is used for subsequent RSM library calls.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_CTLR_HNDL`	Invalid controller handle
`RSMERR_CTLR_NOT_PRESENT`	Controller not present
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory
`RSMERR_BAD_LIBRARY_VERSION`	Invalid library version
`RSMERR_BAD_ADDR`	Bad address

rsm_release_controller

`int` **`rsm_release_controller`**`(`*rsmapi_controller_handle_t chdl*`);`

This function releases the controller associated with the given controller handle. Each call to `rsm_release_controller` must have a matching `rsm_get_controller`. When all the controller handles associated with a controller are released, the system resources associated with the controller are freed. Attempting to access a controller handle, or attempting to access import or export segments on a released controller handle, is not legal. The results of such an attempt are undefined.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_CTLR_HNDL`	Invalid controller handle

rsm_get_controller_attr

`int`
`rsm_get_controller_attr``(`*rsmapi_controller_handle_t chdl, rsmapi_controller_attr_t *attr*`);`

This function retrieves attributes for the specified controller handle. The following list describes the currently defined attributes for this function:

```
typedef struct {
     uint_t       attr_direct_access_sizes;
     uint_t       attr_atomic_sizes;
     size_t       attr_page_size;
     size_t       attr_max_export_segment_size;
     size_t       attr_tot_export_segment_size;
     ulong_t      attr_max_export_segments;
     size_t       attr_max_import_map_size;
     size_t       attr_tot_import_map_size;
     ulong_t      attr_max_import_segments;
 } rsmapi_controller_attr_t;
```

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_CTLR_HNDL	Invalid controller handle
RSMERR_BAD_ADDR	Bad address

Cluster Topology Operations

The key interconnect data required for export operations and import operations are:

- Export cluster node ID
- Import cluster node ID
- Controller name

As a fundamental constraint, the controller specified for a segment import must have a physical connection with the controller used for the associated segment export. This interface defines the interconnect topology, which helps applications establish efficient export and import policies. The data that is provided includes local node ID, local controller instance name, and remote connection specification for each local controller.

An application component that exports memory can use the data provided by the interface to find the set of existing local controllers. The data provided by the interface can also be used to correctly assign controllers for the creation and publishing of segments. Application components can efficiently distribute exported segments over the set of controllers that is consistent with the hardware interconnect and with the application software distribution.

An application component that is importing memory must be informed of the segment IDs and controllers used in the memory export. This information is typically conveyed by a predefined segment and controller pair. The importing component can use the topology data to determine the appropriate controllers for the segment import operations.

rsm_get_interconnect_topology

int **rsm_get_interconnect_topology**(*rsm_topology_t **topology_data*);

This function returns a pointer to the topology data in a location specified by an application pointer. The topology data structure is defined next.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_TOPOLOGY_PTR	Invalid topology pointer
RSMERR_INSUFFICIENT_MEM	Insufficient memory
RSMERR_BAD_ADDR	Insufficient memory

rsm_free_interconnect_topology

void **rsm_free_interconnect_topology**(*rsm_topology_t *topology_data*);

The rsm_free_interconnect_topology operation frees the memory allocated by rsm_get_interconnect_topology.

Return Values: None.

Data Structures

The pointer returned from rsm_get_topology_data references a rsm_topology_t structure. This structure provides the local node ID and an array of pointers to a connections_t structure for each local controller.

```
typedef struct rsm_topology {
   rsm_nodeid_t    local_nodeid;
   uint_t          local_cntrl_count;
   connections_t   *connections[1];
} rsm_topology_t;
```

Administrative Operations

RSM segment IDs can be specified by the application or generated by the system using the rsm_memseg_export_publish() function. Applications that specify segment IDs require a reserved range of segment IDs to use. To reserve a range of segment IDs, use rsm_get_segmentid_range and define the reserved range of segment IDs in the segment ID configuration file /etc/rsm/rsm.segmentid. The rsm_get_segmentid_range function can be used by applications to obtain the segment ID range that is reserved for the applications. This function reads the segment ID range defined in the /etc/rsm/rsm.segmentid file for a given application ID.

An application ID is a null-terminated string that identifies the application. The application can use any value equal to or greater than `baseid` and less than `baseid+length`. If `baseid` or `length` are modified, the segment ID returned to the application might be outside the reserved range. To avoid this problem, use an offset within the range of reserved segment IDs to obtain a segment ID.

Entries in the `/etc/rsm/rsm.segmentid` file are of the form:

```
#keyword      appid      baseid       length
reserve       SUNWfoo    0x600000     100
```

The entries are composed of strings, which can be separated by tabs or blanks. The first string is the keyword `reserve`, followed by the application identifier, which is a string without spaces. Following the application identifier is the `baseid`, which is the starting segment ID of the reserved range in hexadecimal. Following the `baseid` is the `length`, which is the number of segment IDs that are reserved. Comment lines have a # in the first column. The file should not contain blank or empty lines. Segment IDs that are reserved for the system are defined in the `/usr/include/rsm/rsm_common.h` header file. The segment IDs that are reserved for the system cannot be used by the applications.

The `rsm_get_segmentid_range` function returns 0 to indicate success. If the function fails, the function returns one of the following error values:

`RSMERR_BAD_ADDR`	The address that is passed is invalid
`RSMERR_BAD_APPID`	Application ID not defined in the`/etc/rsm/rsm.segmentid` file
`RSMERR_BAD_CONF`	The configuration file `/etc/rsm/rsm.segmentid` is not present or not readable. The file format's configuration is incorrect

Memory Segment Operations

An RSM segment represents a set of (generally) non-contiguous physical memory pages mapped to a contiguous virtual address range. RSM segment export and segment import operations enable the sharing of regions of physical memory among systems on an interconnect. A process of the node on which the physical pages reside is referred to as the *exporter* of the memory. An exported segment that is published for remote access will have a segment identifier that is unique for the given node. The segment ID might be specified by the exporter or assigned by the RSMAPI framework.

Processes of nodes on the interconnect obtain access to exported memory by creating an RSM import segment. The RSM import segment has a connection with an exported segment, rather than local physical pages. When the interconnect supports memory mapping, importers can read and write the exported memory by using the local memory-mapped addresses of the import segment. When the interconnect does not support memory mapping, the importing process uses memory access primitives.

Export-Side Memory Segment Operations

When exporting a memory segment, the application begins by allocating memory in its virtual address space through the normal operating system interfaces such as the System V Shared Memory Interface, `mmap`, or `valloc`. After allocating memory, the application calls the RSMAPI library interfaces to create and label a segment. After labelling the segment, the RSMAPI library interfaces bind physical pages to the allocated virtual range. After binding the physical pages, the RSMAPI library interfaces publish the segment for access by importing processes.

Note – If virtual address space is obtained by using `mmap`, the mapping must be `MAP_PRIVATE`.

Export side memory segment operations include:

- Memory segment creation and destruction
- Memory segment publishing and unpublishing
- Rebinding backing store for a memory segment

Memory Segment Creation and Destruction

Establishing a new memory segment with `rsm_memseg_export_create` enables the association of physical memory with the segment at creation time. The operation returns an export-side memory segment handle to the new memory segment. The segment exists for the lifetime of the creating process or until destroyed with `rsm_memseg_export_destroy`.

Note – If destroy operation is performed before an import side disconnect, the disconnect is forced.

Segment Creation

`int` **`rsm_memseg_export_create`**`(`*rsmapi_controller_handle_t controller*`,` *rsm_memseg_export_handle_t *memseg*`,` *void *vaddr*`,` *size_t size*`,` *uint_t flags*`);`

This function creates a segment handle. After the segment handle is created, the segment handle is bound to the specified virtual address range [vaddr..vaddr+size]. The range must be valid and aligned on the controller's alignment property. The flags argument is a bitmask, which enables:

- Unbinding on the segment
- Rebinding on the segment
- Passing RSM_ALLOW_REBIND to flags
- Support of lock operations
- Passing RSM_LOCK_OPS to flags

Note – The RSM_LOCK_OPS flag is not included in the initial release of RSMAPI.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_CTLR_HNDL	Invalid controller handle
RSMERR_CTLR_NOT_PRESENT	Controller not present
RSMERR_BAD_SEG_HNDL	Invalid segment handle
RSMERR_BAD_LENGTH	Length zero or length exceeds controller limits
RSMERR_BAD_ADDR	Invalid address
RSMERR_PERM_DENIED	Permission denied
RSMERR_INSUFFICIENT_MEM	Insufficient memory
RSMERR_INSUFFICIENT_RESOURCES	Insufficient resources
RSMERR_BAD_MEM_ALIGNMENT	Address not aligned on page boundary
RSMERR_INTERRUPTED	Operation interrupted by signal

Segment Destruction

int **rsm_memseg_export_destroy**(*rsm_memseg_export_handle_t memseg*);

This function deallocates segment and its free resources. All importing processes are forcibly disconnected.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SEG_HNDL	Invalid segment handle
RSMERR_POLLFD_IN_USE	pollfd in use

Memory Segment Publish, Republish, and Unpublish

The publish operation enables the importing of a memory segment by other nodes on the interconnect. An export segment might be published on multiple interconnect adapters.

The segment ID might be specified from within authorized ranges or specified as zero, in which case a valid segment ID is generated by the RSMAPI framework and is passed back.

The segment access control list is composed of pairs of node ID and access permissions. For each node ID specified in the list, the associated read/write permissions are provided by three octal digits for owner, group and other, as with Solaris file permissions. In the access control list, each octal digit can have the following values:

2 Write access.

4 Read only access.

6 Read and write access.

An access permission value of 0624 specifies the following kind of access:

- An importer with the same `uid` as the exporter has both read and write access.
- An importer with the same `gid` as the exporter has write access only.
- All other importers have read access only.

When an access control list is provided, nodes not included in the list cannot import the segment. However, if the access list is null, any node can import the segment. The access permissions on all nodes equal the owner-group-other file creation permissions of the exporting process.

Note – Node applications have the responsibility of managing the assignment of segment identifiers to ensure uniqueness on the exporting node.

Publish Segment

`int` **`rsm_memseg_export_publish`**`(`*rsm_memseg_export_handle_t memseg*`,` *rsm_memseg_id_t *segment_id*`,` *rsmapi_access_entry_t access_list[]*`,` *uint_t access_list_length*`);`

```
typedef struct {
rsm_node_id_t    ae_node;    /* remote node id allowed to access resource */
rsm_permission_t ae_permissions;    /* mode of access allowed */
}rsmapi_access_entry_t;.
```

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL` Invalid segment handle

RSMERR_SEG_ALREADY_PUBLISHED	Segment already published
RSMERR_BAD_ACL	Invalid access control list
RSMERR_BAD_SEGID	Invalid segment identifier
RSMERR_SEGID_IN_USE	Segment identifier in use
RSMERR_RESERVED_SEGID	Segment identifier reserved
RSMERR_NOT_CREATOR	Not creator of segment
RSMERR_BAD_ADDR	Bad address
RSMERR_INSUFFICIENT_MEM	Insufficient memory
RSMERR_INSUFFICIENT_RESOURCES	Insufficient resources

Authorized Segment ID Ranges:

`#define RSM_DRIVER_PRIVATE_ID_BASE`
0

`#define RSM_DRIVER_PRIVATE_ID_END`
0x0FFFFF

`#define RSM_CLUSTER_TRANSPORT_ID_BASE`
0x100000

`#define RSM_CLUSTER_TRANSPORT_ID_END`
0x1FFFFF

`#define RSM_RSMLIB_ID_BASE`
0x200000

`#define RSM_RSMLIB_ID_END`
0x2FFFFF

`#define RSM_DLPI_ID_BASE`
0x300000

`#define RSM_DLPI_ID_END`
0x3FFFFF

`#define RSM_HPC_ID_BASE`
0x400000

`#define RSM_HPC_ID_END`
0x4FFFFF

The following range is reserved for allocation by the system when the publish value is zero.

#define RSM_USER_APP_ID_BASE	0x80000000
#define RSM_USER_APP_ID_END	0xFFFFFFF

Republish Segment

`int` **`rsm_memseg_export_republish`**`(`*rsm_memseg_export_handle_t memseg*`,` *rsmapi_access_entry_t access_list[]*`,` *uint_t access_list_length*`);`

This function establishes a new node access list and segment access mode. These changes only affect future import calls and do not revoke already granted import requests.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_SEG_NOT_PUBLISHED`	Segment not published
`RSMERR_BAD_ACL`	Invalid access control list
`RSMERR_NOT_CREATOR`	Not creator of segment
`RSMERR_INSUFFICIENT_MEMF`	Insufficient memory
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources
`RSMERR_INTERRUPTED`	Operation interrupted by signal

Unpublish Segment

`int` **`rsm_memseg_export_unpublish`**`(`*rsm_memseg_export_handle_t memseg*`);`

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_SEG_NOT_PUBLISHED`	Segment not published
`RSMERR_NOT_CREATOR`	Not creator of segment
`RSMERR_INTERRUPTED`	Operation interrupted by signal

Memory Segment Rebind

The rebind operation releases the current backing store for an export segment. After releasing the current backing store for an export segment, the rebind operation allocates a new backing store. The application must first obtain a new virtual memory allocation for the segment. This operation is transparent to importers of the segment.

Note – The application has the responsibility of preventing access to segment data until the rebind operation is complete. Retrieving data from a segment during rebinding does not cause a system failure, but the results of such an operation are undefined.

Rebind Segment

`int` **`rsm_memseg_export_rebind`**`(`*rsm_memseg_export_handle_t memseg*`,` *void *vaddr, offset_t off*`,` *size_t size*`);`

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_BAD_LENGTH`	Invalid length
`RSMERR_BAD_ADDR`	Invalid address
`RSMERR_REBIND_NOT_ALLOWED`	Rebind not allowed
`RSMERR_NOT_CREATOR`	Not creator of segment
`RSMERR_PERM_DENIED`	Permission denied
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources
`RSMERR_INTERRUPTED`	Operation interrupted by signal

Import-Side Memory Segment Operations

The following list describes Import-side operations:

- Memory segment connection and disconnection
- Access to imported segment memory
- Barrier operations used to impose order on data access operations and for access error detection

The connect operation is used to create an RSM import segment and form a logical connection with an exported segment.

Access to imported segment memory is provided by three interface categories:

- Segment access.
- Data transfer.
- Segment memory mapping.

Memory Segment Connection and Disconnection

Connect to Segment

`int` **`rsm_memseg_import_connect`**`(`*rsmapi_controller_handle_t controller*`,` *rsm_node_id_t node_id*`,` *rsm_memseg_id_t segment_id*`,` *rsm_permission_t perm*`,` *rsm_memseg_import_handle_t *im_memseg*`);`

This function connects to segment *segment_id* on remote node *node_id* by using the specified permission *perm*. The function returns a segment handle after connecting to the segment.

The argument *perm* specifies the access mode requested by the importer for this connection. To establish the connection, the access permissions specified by the exporter are compared to the access mode, user ID, and group ID used by the importer. If the request mode is not valid, the connection request is denied. The *perm* argument is limited to the following octal values:

0400	Read mode
0200	Write mode
0600	Read/write mode

The specified controller must have a physical connection to the controller that is used in the export of the segment.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_CTLR_HNDL`	Invalid controller handle
`RSMERR_CTLR_NOT_PRESENT`	Controller not present
`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_PERM_DENIED`	Permission denied
`RSMERR_SEG_NOT_PUBLISHED_TO_NODE`	Segment not published to node
`RSMERR_SEG_NOT_PUBLISHED`	No such segment published
`RSMERR_REMOTE_NODE_UNREACHABLE`	Remote node not reachable
`RSMERR_INTERRUPTED`	Connection interrupted
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources
`RSMERR_BAD_ADDR`	Bad address

Disconnect from Segment

`int` **`rsm_memseg_import_disconnect`**`(`*rsm_memseg_import_handle_t im_memseg*`);`

This function disconnects a segment. This function frees a segment's resources after disconnecting a segment. All existing mappings to the disconnected segment are removed. The handle `im_memseg` is freed.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_SEG_STILL_MAPPED`	Segment still mapped

`RSMERR_POLLFD_IN_USE`	`pollfd` in use

Memory Access Primitives

The following interfaces provide a mechanism for transferring between 8 bits and 64 bits of data. The `get` interfaces use a repeat count (*rep_cnt*) to indicate the number of data items of a given size the process will read from successive locations. The locations begin at byte offset *offset* in the imported segment. The data is written to successive locations that begin at *datap*. The `put` interfaces use a repeat count (*rep_cnt*). The count indicates the number of data items the process will read from successive locations. The locations begin at *datap*. The data is then written to the imported segment at successive locations. The locations begin at the byte offset specified by the *offset* argument.

These interfaces also provide byte swapping in case the source and destination have incompatible endian characteristics.

Function Prototypes:

`int` **`rsm_memseg_import_get8`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint8_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_get16`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint16_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_get32`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint32_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_get64`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint64_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_put8`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint8_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_put16`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint16_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_put32`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint32_t *datap*`,` *ulong_t rep_cnt*`);`

`int` **`rsm_memseg_import_put64`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *uint64_t *datap*`,` *ulong_t rep_cnt*`);`

The following interfaces are intended for data transfers that are larger than the ones supported by the segment access operations.

Segment Put

`int` **`rsm_memseg_import_put`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *off_t offset*`,` *void *src_addr*`,` *size_t length*`);`

This function copies data from local memory, specified by the *src_addr* and *length*, to the corresponding imported segment locations specified by the handle and offset.

Segment Get

`int` **`rsm_memseg_import_get`**`(`*rsm_memseg_import_handle_t im_memseg*`,`
off_t offset`,` *void *dst_addr*`,` *size_t length*`);`

This function is similar to `rsm_memseg_import_put()`, but data flows from the imported segment into local regions defined by the *dest_vec* argument

The `put` and `get` routines write or read the specified quantity of data from the byte offset location specified by the argument *offset*. The routines begin at the base of the segment. The offset must align at the appropriate boundary. For example, `rsm_memseg_import_get64()` requires that *offset* and *datap* align at a double-word boundary, while `rsm_memseg_import_put32()` requires an offset that is aligned at a word boundary.

By default, the barrier mode attribute of a segment is `implicit`. Implicit barrier mode means that the caller assumes the data transfer has completed or has failed upon return from the operation. Because the default barrier mode is implicit, the application must initialize the barrier. The application initializes the barrier by using the `rsm_memseg_import_init_barrier()` function before calling `put` or `get` routines when using the default mode. To use the explicit operation mode, the caller must use a barrier operation to force the completion of a transfer. After forcing the completion of the transfer, the caller must determine if any errors have occurred as a result of the forced completion.

Note – An import segment can be partially mapped by passing an offset in the `rsm_memseg_import_map()` routine. If the import segment is partially mapped, the *offset* argument in the `put` or `get` routines is from the base of the segment. The user must make sure that the correct byte offset is passed to `put` and `get` routines.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_BAD_ADDR`	Bad address
`RSMERR_BAD_MEM_ALIGNMENT`	Invalid memory alignment
`RSMERR_BAD_OFFSET`	Invalid offset
`RSMERR_BAD_LENGTH`	Invalid length
`RSMERR_PERM_DENIED`	Permission denied
`RSMERR_BARRIER_UNINITIALIZED`	Barrier not initialized
`RSMERR_BARRIER_FAILURE`	I/O completion error

RSMERR_CONN_ABORTED	Connection aborted
RSMERR_INSUFFICIENT_RESOURCES	Insufficient resources

Scatter-Gather Access

The rsm_memseg_import_putv() and rsm_memseg_import_getv() functions allow the use of a list of I/O requests instead of a single source and single destination address.

Function Prototypes:

int **rsm_memseg_import_putv**(*rsm_scat_gath_t *sg_io*);

int **rsm_memseg_import_getv**(*rsm_scat_gath_t *sg_io*);

The I/O vector component of the scatter-gather list (sg_io) enables the specification of local virtual addresses or local_memory_handles. Handles are an efficient way to repeatedly use a local address range. Allocated system resources, such as locked down local memory, are maintained until the handle is freed. The supporting functions for handles are rsm_create_localmemory_handle() and rsm_free_localmemory_handle().

You can gather virtual addresses or handles into the vector in order to write to a single remote segment. You can also scatter the results of reading from a single remote segment to the vector of virtual addresses or handles.

I/O for the entire vector is initiated before returning. The barrier mode attribute of the import segment determines whether the I/O has completed before the function returns. Setting the barrier mode attribute to implicit guarantees that data transfer is completed in the order entered in the vector. An implicit barrier open and close surrounds each list entry. If an error is detected, I/O for the vector is terminated and the function returns immediately. The residual count indicates the number of entries for which the I/O either did not complete or was not initiated.

You can specify that a notification event be sent to the target segment when a putv or getv operation is successful. To specify the delivery of a notification event, specify the RSM_IMPLICIT_SIGPOST value in the flags entry of the rsm_scat_gath_t structure. The flags entry can also contain the value RSM_SIGPOST_NO_ACCUMULATE, which is passed on to the signal post operation if RSM_IMPLICIT_SIGPOST is set.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SGIO	Invalid scatter-gather structure pointer
RSMERR_BAD_SEG_HNDL	Invalid segment handle
RSMERR_BAD_CTLR_HNDL	Invalid controller handle
RSMERR_BAD_ADDR	Bad address

`RSMERR_BAD_OFFSET`	Invalid offset
`RSMERR_BAD_LENGTH`	Invalid length
`RSMERR_PERM_DENIED`	Permission denied
`RSMERR_BARRIER_FAILURE`	I/O completion error
`RSMERR_CONN_ABORTED`	Connection aborted
`RSMERR_INSUFFICIENT_RESOURCES`	Insufficient resources
`RSMERR_INTERRUPTED`	Operation interrupted by signal

Get Local Handle

`int` **`rsm_create_localmemory_handle`**`(`*rsmapi_controller_handle_t cntrl_handle*`,` *rsm_localmemory_handle_t *local_handle*`,` *caddr_t local_vaddr*`,` *size_t length*`);`

This function obtains a local handle for use in the I/O vector for subsequent calls to `putv` or `getv`. Freeing the handle as soon as possible conserves system resources, notably the memory spanned by the local handle, which might be locked down.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_CTLR_HNDL`	Invalid controller handle
`RSMERR_BAD_LOCALMEM_HNDL`	Invalid local memory handle
`RSMERR_BAD_LENGTH`	Invalid length
`RSMERR_BAD_ADDR`	Invalid address
`RSMERR_INSUFFICIENT_MEM`	Insufficient memory

Free Local Handle

`rsm_free_localmemory_handle``(`*rsmapi_controller_handle_t cntrl_handle*`,` *rsm_localmemory_handle_t handle*`);`

This function releases the system resources associated with the local handle. While all handles that belong to a process are freed when the process exits, calling this function conserves system resources.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_CTLR_HNDL`	Invalid controller handle
`RSMERR_BAD_LOCALMEM_HNDL`	Invalid local memory handle

The following example demonstrates the definition of primary data structures.

EXAMPLE 2–1 Primary Data Structures

```
typedef void *rsm_localmemory_handle_t
typedef struct {
   ulong_t    io_request_count;    number of rsm_iovec_t entries
```

EXAMPLE 2–1 Primary Data Structures *(Continued)*

```
   ulong_t    io_residual_count;    rsm_iovec_t entries not completed

   in    flags;
   rsm_memseg_import_handle_t        remote_handle; opaque handle for
                                                    import segment
   rsm_iovec_t                            *iovec;          pointer to
                                                    array of io_vec_t
} rsm_scat_gath_t;

typedef struct {
   int    io_type;        HANDLE or VA_IMMEDIATE
   union {
        rsm_localmemory_handle_t    handle;            used with HANDLE
        caddr_t                        virtual_addr;        used with
                                                         VA_IMMEDIATE
    } local;
   size_t         local_offset;              offset from handle base vaddr
   size_t         import_segment_offset;    offset from segment base vaddr
   size_t         transfer_length;
} rsm_iovec_t;
```

Segment Mapping

Mapping operations are only available for native architecture interconnects such as Dolphin-SCI or NewLink. Mapping a segment grants CPU memory operations access to that segment, saving the overhead of calling memory access primitives.

Imported Segment Map

`int` **`rsm_memseg_import_map`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *void **address*`,` *rsm_attribute_t attr*`,` *rsm_permission_t perm*`,` *off_t offset*`,` *size_t length*`);`

This function maps an imported segment into the caller address space. If the attribute `RSM_MAP_FIXED` is specified, the function maps the segment at the value specified in ***address*.

```
typedef enum {
RSM_MAP_NONE = 0x0,   /* system will choose available virtual address */
RSM_MAP_FIXED = 0x1,  /* map segment at specified virtual address */
} rsm_map_attr_t;
```

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_BAD_ADDR`	Invalid address
`RSMERR_BAD_LENGTH`	Invalid length
`RSMERR_BAD_OFFSET`	Invalid offset

`RSMERR_BAD_PERMS`	Invalid permissions
`RSMERR_SEG_ALREADY_MAPPED`	Segment already mapped
`RSMERR_SEG_NOT_CONNECTED`	Segment not connected
`RSMERR_CONN_ABORTED`	Connection aborted
`RSMERR_MAP_FAILED`	Error during mapping
`RSMERR_BAD_MEM_ALIGNMENT`	Address not aligned on page boundary

Unmap segment

`int` **`rsm_memseg_import_unmap`**`(`*rsm_memseg_import_handle_t im_memseg*`);`

This function unmaps an imported segment from user virtual address space.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle

Barrier Operations

Use Barrier operations to resolve order-of-write-access memory model issues. Barrier operations also provide remote memory access error detection.

The barrier mechanism is made up of the following operations:

- Initialization
- Open
- Close
- Order

The open and close operations define a span-of-time interval for error detection and ordering. The initialization operation enables barrier creation for each imported segment, as well as barrier type specification. The only barrier type currently supported has a span-of-time scope per segment. Use a type argument value of `RSM_BAR_DEFAULT`.

Successfully performing a close operation guarantees the successful completion of covered access operations, which take place between the barrier open and the barrier close. After a barrier open operation, failures of individual data access operations, both reads and writes, are not reported until the barrier close operation.

To impose a specific order of write completion within a barrier's scope, use an explicit barrier-order operation. A write operation that is issued before the barrier-order operation finishes before operations that are issued after the barrier-order operation. Write operations within a given barrier scope are ordered with respect to another barrier scope.

Initialize Barrier

int
rsm_memseg_import_init_barrier(*rsm_memseg_import_handle_t im_memseg*, *rsm_barrier_type_t type*, *rsmapi_barrier_t *barrier*);

Note – At present, RSM_BAR_DEFAULT is the only supported type.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SEG_HNDL	Invalid segment handle
RSMERR_BAD_BARRIER_PTR	Invalid barrier pointer
RSMERR_INSUFFICIENT_MEM	Insufficient memory

Open Barrier

int **rsm_memseg_import_open_barrier**(*rsmapi_barrier_t *barrier*);

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SEG_HNDL	Invalid segment handle
RSMERR_BAD_BARRIER_PTR	Invalid barrier pointer

Close Barrier

int **rsm_memseg_import_close_barrier**(*rsmapi_barrier_t *barrier*);

This function closes the barrier and flushes all store buffers. This call assumes that the calling process will retry all remote memory operations since the last rsm_memseg_import_open_barrier call if the call to rsm_memseg_import_close_barrier() fails.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SEG_HNDL	Invalid segment handle
RSMERR_BAD_BARRIER_PTR	Invalid barrier pointer
RSMERR_BARRIER_UNINITIALIZED	Barrier not initialized
RSMERR_BARRIER_NOT_OPENED	Barrier not opened
RSMERR_BARRIER_FAILURE	Memory access error
RSMERR_CONN_ABORTED	Connection aborted

Order Barrier

int **rsm_memseg_import_order_barrier**(*rsmapi_barrier_t *barrier*);

This function flushes all store buffers.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SEG_HNDL	Invalid segment handle
RSMERR_BAD_BARRIER_PTR	Invalid barrier pointer
RSMERR_BARRIER_UNINITIALIZED	Barrier not initialized
RSMERR_BARRIER_NOT_OPENED	Barrier not opened
RSMERR_BARRIER_FAILURE	Memory access error
RSMERR_CONN_ABORTED	Connection aborted

Destroy Barrier

`int` **`rsm_memseg_import_destroy_barrier`**`(`*rsmapi_barrier_t *barrier*`);`

This function deallocates all barrier resources.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SEG_HNDL	Invalid segment handle
RSMERR_BAD_BARRIER_PTR	Invalid barrier pointer

Set Mode

`int` **`rsm_memseg_import_set_mode`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *rsm_barrier_mode_t mode*`);`

This function supports the optional explicit barrier scoping that is available in the `put` routines. The two valid barrier modes are `RSM_BARRIER_MODE_EXPLICIT` and `RSM_BARRIER_MODE_IMPLICIT`. The default value of the barrier mode is `RSM_BARRIER_MODE_IMPLICIT`. While in implicit mode, an implicit barrier open and barrier close is applied to each `put` operation. Before setting the barrier mode value to `RSM_BARRIER_MODE_EXPLICIT`, use the `rsm_memseg_import_init_barrier` routine to initialize a barrier for the imported segment `im_memseg`.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SEG_HNDL	Invalid segment handle

Get Mode

`int` **`rsm_memseg_import_get_mode`**`(`*rsm_memseg_import_handle_t im_memseg*`,` *rsm_barrier_mode_t *mode*`);`

This function obtains the current mode value for barrier scoping in the `put` routines.

Return Values: Returns 0 if successful. Returns an error value otherwise.

RSMERR_BAD_SEG_HNDL	Invalid segment handle.

Event Operations

Event operations enable processes synchronization on memory access events. If a process cannot use the `rsm_intr_signal_wait()` function, it can multiplex event waiting by obtaining a poll descriptor with `rsm_memseg_get_pollfd()` and using the `poll` system call.

Note – Using the `rsm_intr_signal_post()` and `rsm_intr_signal_wait()` operations incurs the need to process of `ioctl` calls to the kernel.

Post Signal

`int` **`rsm_intr_signal_post`**`(`*void *memseg, uint_t flags*`);`

The void pointer `*memseg` can be type cast to either an import segment handle or an export segment handle. If `*memseg` refers to an import handle, this function sends a signal the exporting process. If `*memseg` refers to an export handle, this function sends a signal to all importers of that segment. Setting the `flags` argument to `RSM_SIGPOST_NO_ACCUMULATE` discards this event if an event is already pending for the target segment.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_REMOTE_NODE_UNREACHABLE`	Remote node not reachable

Wait for Signal

`int` **`rsm_intr_signal_wait`**`(`*void * memseg, int timeout*`);`

The void pointer `*memseg` can be type cast to either an import segment handle or an export segment handle. The process blocks for up to *timeout* milliseconds or until an event occurs. If the value is -1, the process blocks until an event occurs or until interrupted.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle
`RSMERR_TIMEOUT`	Timer expired
`RSMERR_INTERRUPTED`	Wait interrupted

Get pollfd

`int` **`rsm_memseg_get_pollfd`**`(`*void *memseg, struct pollfd *pollfd*`);`

This function initializes the specified `pollfd` structure with a descriptor for the specified segment and the singular fixed event generated by `rsm_intr_signal_post()`. Use the `pollfd` structure with the `poll` system call to wait for the event signalled by `rsm_intr_signal_post`. If the memory segment is not currently published, the `poll` system call does not return a valid `pollfd`. Each successful call increments a `pollfd` reference count for the specified segment.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle

Release pollfd

`int` **`rsm_memseg_release_pollfd`**`(`*oid *memseg*`);`

This call decrements the `pollfd` reference count for the specified segment. If the reference count is nonzero, operations that unpublish, destroy, or unmap the segment fail.

Return Values: Returns 0 if successful. Returns an error value otherwise.

`RSMERR_BAD_SEG_HNDL`	Invalid segment handle

RSMAPI General Usage Notes

These usage notes describe general considerations for the export and import sides of a shared-memory operation. These usage notes also contain general information regarding segments, file descriptors, and RSM configurable parameters.

Segment Allocation and File Descriptor Usage

The system allocates a file descriptor, which is inaccessible to the application importing or exporting memory, for each export operation or import operation. The default limit on file descriptor allocation for each process is 256. The importing or exporting application must adjust the allocation limit appropriately. If the application increases the file descriptor limit beyond 256, the values of the file descriptors that are allocated for export segments and import segments starts at 256. These file descriptor values are chosen to avoid interfering with normal file descriptor allocation by the application. This behavior accommodates the use of certain `libc` functions in 32-bit applications that only work with file descriptor values lower than 256.

Export-Side Considerations

The application must prevent access to segment data until the rebind operation is complete. Segment data access during rebind does not cause a system failure, but data content results are undefined. The virtual address space must be currently mapped and valid.

Import-Side Considerations

The controller that is specified for a segment import must have a physical connection with the controller that is used in the export of the segment.

RSM Configurable Parameters

The `SUNWrsm` software package includes an `rsm.conf` file. This file is located in `/usr/kernel/drv`. This file is a configuration file for RSM. The `rsm.conf` file can be used to specify values for certain configurable RSM properties. The configurable properties currently defined in `rsm.conf` include `max-exported-memory` and `enable-dynamic-reconfiguration`.

`max-exported-memory`
: This property specifies an upper limit on the amount of exportable memory. The upper limit is expressed as a percentage of total available memory. Giving this property a value of zero indicates that the amount of exportable memory is unlimited.

`enable-dynamic-reconfiguration`
: The value of this property indicates whether dynamic reconfiguration is enabled. A value of zero indicates dynamic reconfiguration is disabled. A value of one enables dynamic reconfiguration support. The default value for this property is one.

RSMAPI Usage Example

This section provides a simple program to illustrate the usage of the RSMAPI. The program runs on two nodes: an exporter node and an importer node. The exporter node creates and publishes a memory segment, then waits for a message to be written in the segment. The importer node connects to the exported segment, writes a message, and then signals the exporter.

```
/*
 * Copyright (c) 1998 by Sun Microsystems, Inc.
 * All rights reserved.
 */
#include <stdio.h>
#include <rsm/rsmpai.h>
#include <errno.h>


/*
        To run this program do the following:

        First node(assuming node id = 1):
                rsmtest -e -n 2

        Second node(assuming node id = 2):
                rsmtest -i -n 1

       The program will prompt the importer for a message at the
       console. Enter any message and hit return. The message will
       be displayed on the export console.
*/

typedef struct {
        char    out;
        char    in;
        char    data[1];
}msg_t;

#define SEG_ID 0x400000
#define EXPORT  0
#define IMPORT  1

#define BUFSIZE (1024 * 8)
#define DEFAULT_SEGSZ BUFSIZE

#define RSM_PERM_READ                         0400
#define RSM_PERM_WRITE                        0200
#define RSM_PERM_RDWR                         (RSM_PERM_READ|RSM_PERM_WRITE)
#define
RSM_ACCESS_TRUSTED
```

```
0666

rsm_topology_t *tp;

int iterations = 10;

int mode = EXPORT;
int test = 0;

char *buf;
int buflen = BUFSIZE;
int offset = 0;
volatile char *iva;

int status;
rsm_memseg_id_t segid;
rsmapi_controller_handle_t ctrl;
rsmapi_controller_attr_t attr;
rsm_memseg_export_handle_t seg;
rsm_memseg_import_handle_t imseg;
rsm_access_entry_t list[2];
rsm_node_id_t dest;

extern void *valloc(size_t);
extern void exit();
extern void sleep();
extern int atoi(const char *);

/* The following function exports a segment and publishes it.
 */
static int
export()
{
        int i;

        /* allocate and clear memory */
        buf = (char *)valloc(buflen);


if (!buf) {


(void) fprintf(stderr, "Unable to allocate memory\n");


exit (1);

}
        for (i = 0; i < buflen; i++)
                buf[i] = 0;


/* Create an export memory segment */
```

```
        status = rsm_memseg_export_create(ctrl, &seg, (void *)buf, buflen);
        if (status != 0) {
                (void) fprintf(stderr,
                      "unable to create an exported segment %d\n", status);
                exit(1);
        }

        /* Set up access list for publishing to nodes 1 and 2 */
        list[0].ae_node = tp->topology_hdr.local_nodeid ;

/* Allow read and write permissions */
        list[0].ae_permission = RSM_ACCESS_TRUSTED;
        list[1].ae_node = tp->topology_hdr.local_nodeid + 1;

/* Allow read and write permissions */
        list[1].ae_permission = RSM_ACCESS_TRUSTED;


/* Publish the created export segment */
        status = rsm_memseg_export_publish(seg, &segid, list, 0);
        if (status != 0) {
              (void) fprintf(stderr, "unable to pub segment %d\n", status);
              exit(1);
        }
        return (0);
}

/* The following function is used to connect to an exported memory segment.
 */
static void
import()
{
        /* Connect to exported segment and set up mapping for
         * access through local virtual addresses.
         */
again:
        status = rsm_memseg_import_connect(ctrl, dest, segid, RSM_PERM_RDWR,
            &imseg);
        if (status != 0) {
                (void) fprintf(stderr,
                                 "unable to conect to segment %x err %x\n",
                                 segid, status);
                sleep(1);
                goto again;
        }

        iva = NULL;
        status = rsm_memseg_import_map(imseg, (void **)&iva,
                                  RSM_MAP_NONE, RSM_PERM_RDWR, 0, buflen);
        if (status != 0) {
             (void) fprintf(stderr, "unable to mmap segment %d\n", status);
             exit(1);
        }
}
```

```
/* Unpublish and destroy the export segment */
static void
export_close()
{
again:
        status = rsm_memseg_export_unpublish(seg);
        if (status != 0) {
                (void) fprintf(stderr,
                        "unable to create an unpub segment %d\n", status);
                sleep(10);
                goto again;
        }

        status = rsm_memseg_export_destroy(seg);
        if (status != 0) {
                (void) fprintf(stderr, "unable to destroy segment %d\n",
                                status);
                exit(1);
        }
}

/* Unmap the virtual address mapping and disconnect the segment */
static void
import_close()
{
        status = rsm_memseg_import_unmap(imseg);
        if (status != 0) {
            (void) fprintf(stderr, "unable to unmap segment %d\n", status);


                exit(1);
        }

        status = rsm_memseg_import_disconnect(imseg);
        if (status != 0) {
                (void) fprintf(stderr,
                        "unable to disconnect segment %d\n", status);
                exit(1);
        }
}


static void
test0()
{
        volatile msg_t  *mbuf;
        /* Barrier to report error */
        rsmapi_barrier_t bar;
        int i;

        if (mode == EXPORT) {
                (void) export();
                mbuf = (msg_t *)(buf + offset);
```

```
                mbuf->in = mbuf->out = 0;
        } else {
             import();
             mbuf = (msg_t *)(iva + offset);
             rsm_memseg_import_init_barrier(imseg, RSM_BARRIER_NODE, &bar);
        }

        (void) printf("Mbuf is %x\n", (uint_t)mbuf);
        while (iterations-- > 0) {


int e;

                switch (mode) {
                case EXPORT:
                    while (mbuf->out == mbuf->in) {
                            (void) rsm_intr_signal_wait(seg, 1000);
                    }
                    (void) printf("msg [0x%x %d %d] ",
                            (uint_t)mbuf, (int)mbuf->out, mbuf->in);
                    for (i = 0; mbuf->data[i] != '\0' && i < buflen; i++) {
                            (void) putchar(mbuf->data[i]);
                            mbuf->data[i] = '?';
                    }
                    (void) putchar('\n');

                    mbuf->out++;
                    break;
                case IMPORT:
                    (void) printf("Enter msg [0x%x %d]: ",
                                    (uint_t)mbuf, mbuf->out, mbuf->in);
retry:
                    e = rsm_memseg_import_open_barrier(&bar);
                    if (e != 0) {
                        (void) printf("Barrier open failed %x\n", e);
                        exit(1);
                    }
                    for (i = 0; (mbuf->data[i] = getchar()) != '\n'; i++)
                            ;
                    mbuf->data[i] = '\0';
                    rsm_memseg_import_order_barrier(&bar);
                    mbuf->in++;

                    e = rsm_memseg_import_close_barrier(&bar);
                    if (e != 0) {
                        (void) printf("Barrier close failed, %d\n", e);
                        goto retry;
                    }



(void)rsm_intr_signal_post(imseg);
                        break;
```

```
                }
        }

        if (mode == IMPORT) {
                import_close();
        } else {
                export_close();
        }
}


void
main(int argc, char *argv[])
{
        int unit = 0;
        char *device = "sci0";
        int i;

        segid = SEG_ID;
        buflen = DEFAULT_SEGSZ;
        while ((i = getopt(argc, argv, "OCGeid:b:sl:n:k:t:c:u:v")) != -1) {
                switch (i) {
                case 'e':
                        mode = EXPORT;
                        break;
                case 'i':
                        mode = IMPORT;
                        break;
                case 'n':
                        dest = atoi(optarg);
                        if ((int)dest < 0) dest = 0;
                        break;
                default:



  (void) fprintf(stderr, "Usage: %s -ei  -n dest\n",






 argv[0]);




  exit(1);
                }
        }

        status = rsm_get_controller(device, &ctrl);
        if (status != 0) {
                (void) fprintf(stderr, "Unable to get controller\n");
                exit(1);
```

```
        }

        status = rsm_get_controller_attr(ctrl, &attr);

status = rsm_get_interconnect_topology(&tp);

if (status != 0) {

(void) fprintf(stderr, "Unable to get topology\n");

exit(1);

} else {

    (void) printf("Local node id = %d\n",

  tp->topology_hdr.local_nodeid);

}

if (dest == 0) {

  dest = tp->topology_hdr.local_nodeid;

(void) printf("Dest is adjusted to %d\n", dest);

}

        switch (test) {
        case 0:
                test0();
                break;
        default:
                (void) printf("No test executed\n");
                break;
        }
}
```

CHAPTER 3

Process Scheduler

This chapter describes the scheduling of processes and how to modify scheduling.

- "Overview of the Scheduler" on page 55 contains an overview of the scheduler and the time-sharing scheduling class. Other scheduling classes are briefly described.
- "Commands and Interfaces" on page 59 describes the commands and interfaces that modify scheduling.
- "Interactions With Other Interfaces" on page 62 describes the effects of scheduling changes on kernel processes and certain interfaces.
- Performance issues to consider when using these commands or interfaces are covered in "Scheduling and System Performance" on page 63.

The chapter is for developers who need more control over the order of process execution than default scheduling provides. See *Multithreaded Programming Guide* for a description of multithreaded scheduling.

Overview of the Scheduler

When a process is created, the system assigns a lightweight process (LWP) to the process. If the process is multithreaded, more LWPs might be assigned to the process. An LWP is the object that is scheduled by the UNIX system scheduler, which determines when processes run. The scheduler maintains process priorities that are based on configuration parameters, process behavior, and user requests. The scheduler uses these priorities to determine which process runs next. The six priority classes are real-time, system, interactive (IA), fixed-priority (FX), fair-share (FSS), and time-sharing (TS).

The default scheduling is a time-sharing policy. This policy dynamiccally adjusts process priorities to balance the response time of interactive processes. The policy also dynamically adjusts priorities to balance the throughput of processes that use a lot of CPU time. The time-sharing class has the lowest priority.

The SunOS 5.9 scheduler also provides a real-time scheduling policy. Real-time scheduling enables the assigning of fixed priorities to specific processes by users. The highest-priority real-time user process always gets the CPU as soon as the process is runnable .

The SunOS 5.9 scheduler also provides a policy for fixed-priority scheduling. Fixed-priority scheduling enables the assignment of fixed priorities to specific processes by users. Fixed-priority scheduling uses the same priority range as the time-sharing scheduling class by default.

A program can be written so that its real-time processes have a guaranteed response time from the system. See Chapter 10 for detailed information.

The control of process scheduling provided by real-time scheduling is rarely needed. However, when the requirements for a program include strict timing constraints, real-time processes might be the only way to satisfy those constraints.

Caution – Careless use of real-time processes can have a dramatic negative effect on the performance of time-sharing processes.

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

- `dispadmin`(1M) displays or changes scheduler configuration in a running system.
- `ts_dptbl`(4) and `rt_dptbl`(4) are tables that contain the time-sharing and real-time parameters that are used to configure the scheduler.

A process inherits its scheduling parameters, including scheduling class and priority within that class, when the process is created. A process changes class only by user request. The system bases its adjustments of a process' priority on user requests and the policy associated with the scheduler class of the process.

In the default configuration, the initialization process belongs to the time-sharing class. Therefore, all user login shells begin as time-sharing processes.

The scheduler converts class-specific priorities into global priorities. The global priority of a process determines when the process runs. The scheduler always runs the runnable process with the highest global priority. Higher priorities run first. A process assigned to the CPU runs until the process sleeps, uses its time slice, or is pre-empted by a higher-priority process. Processes with the same priority run in sequence, around a circle.

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

Note – In a single processor system, no kernel process and no time-sharing process runs while a runnable real-time process exists.

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

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

The following sections describe the scheduling policies of the six scheduling classes.

Time-Sharing Class

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

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

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

Global process priorities and user-supplied priorities are in ascending order: higher priorities run first. The user priority runs from the negative of a configuration-dependent maximum to the positive of that maximum. A process inherits its user priority. Zero is the default initial user priority.

The “user priority limit” is the configuration-dependent maximum value of the user priority. You can set a user priority to any value lower than the user priority limit. With appropriate permission, you can raise the user priority limit. Zero is the user priority limit by default.

You can lower the user priority of a process to give the process reduced access to the CPU. Alternately, with the appropriate permission, raise the user priority to get faster service. The user priority cannot be set to a value that is higher than the user priority limit. Therefore, you must raise the user priority limit before raising the user priority if both have their default values at zero.

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

The scheduler manages time-sharing processes by using configurable parameters in the time-sharing parameter table `ts_dptbl`(4). This table contains information specific to the time-sharing class.

System Class

The system class uses a fixed-priority policy to run kernel processes such as servers and housekeeping processes like the paging daemon. The system class is reserved to the kernel. Users cannot add a process to the system class. Users cannot remove a process from the system class. Priorities for system class processes are set up in the kernel code. The priorities of system processes do not change once established. User processes that run in kernel mode are not in the system class.

Real-time Class

The real-time class uses a scheduling policy with fixed priorities so that critical processes run in predetermined order. Real-time priorities never change except when a user requests a change. Privileged users can use the `priocntl`(1) command or the `priocntl`(2) interface to assign real-time priorities.

The scheduler manages real-time processes by using configurable parameters in the real-time parameter table `rt_dptbl`(4). This table contains information specific to the real-time class.

Interactive Class

The IA class is very similar to the TS class. When used in conjunction with a windowing system, processes have a higher priority while running in a window with the input focus. The IA class is the default class while the system runs a windowing system. The IA class is otherwise identical to the TS class, and the two classes share the same `ts_dptbl` dispatch parameter table.

Fair-Share Class

The FSS class is used by the Fair-Share Scheduler (FSS(7)) to manage application performance by explicitly allocating shares of CPU resources to projects. A share indicates a project's entitlement to available CPU resources. The system tracks resource usage over time. The system reduces entitlement when usage is heavy. The system increases entitlement when usage is light. The FSS schedules CPU time among processes according to their owners' entitlements, independent of the number of processes each project owns. The FSS class uses the same priority range as the TS and IA classes. See the FSS man page for more details.

Fixed-Priority Class

The FX class provides a fixed-priority pre-emptive scheduling policy. This policy is used by processes that require user or application control of scheduling priorities but are not dynamically adjusted by the system. By default, the FX class has the same priority range as the TS, IA, and FSS classes. The FX class allows user or application control of scheduling priorities through user priority values assigned to processes within the class. These user priority values determine the scheduling priority of a fixed-priority process relative to other processes within its class.

The scheduler manages fixed-priority processes by using configurable parameters in the fixed-priority dispatch parameter table `fx_dptbl`(4). This table contains information specific to the fixed-priority class.

Commands and Interfaces

The following figure illustrates the default process priorities.

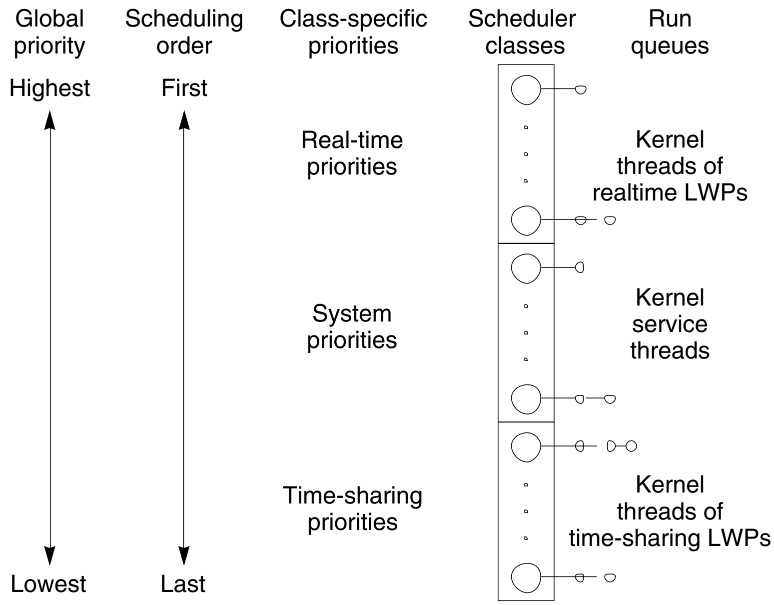


FIGURE 3–1 Process Priorities (Programmer's View)

A process priority has meaning only in the context of a scheduler class. You specify a process priority by specifying a class and a class-specific priority value. The class and class-specific value are mapped by the system into a global priority that the system uses to schedule processes.

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

The `ps`(1) command with `-cel` options reports global priorities for all active processes. The `priocntl`(1) command reports the class-specific priorities that users and programmers use.

The `priocntl`(1) command and the `priocntl`(2) and `priocntlset`(2) interfaces are used to set or retrieve scheduler parameters for processes. Setting priorities generally follows the same sequence for the command and both interfaces:

1. Specify the target processes.
2. Specify the scheduler parameters that you want for those processes.
3. Execute the command or interface to set the parameters for the processes.

Process IDs are basic properties of UNIX processes. See `Intro`(2) for more information. The class ID is the scheduler class of the process. `priocntl`(2) works only for the time-sharing and the real-time classes, not for the system class.

`priocntl` Usage

The `priocntl`(1) utility performs four different control interfaces on the scheduling of a process:

`priocntl -l`	Displays configuration information
`priocntl -d`	Displays the scheduling parameters of processes
`priocntl -s`	Sets the scheduling parameters of processes
`priocntl -e`	Executes a command with the specified scheduling parameters

The following examples demonstrate the use of `priocntl`(1).

- The `-l` option for the default configuration produces the following output:

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

SYS (System Class)

TS (Time Sharing)
Configured TS User Priority Range -60 through 60

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

- To display information on all processes, do the following:

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

- To display information on all time-sharing processes:

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

- To display information on all processes with user ID 103 or 6626, do the following:

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

- To make the process with ID 24668 a real-time process with default parameters, do the following:

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

- To make 3608 RT with priority 55 and a one-fifth second time slice:

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

- To change all processes into time-sharing processes, do the following:

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

- To reduce TS user priority and user priority limit to -10 for uid 1122:

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

- To start a real-time shell with default real-time priority, do the following:

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

- To run make with a time-sharing user priority of -10, do the following:

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

priocntl(1) includes the interface of nice(1). nice works only on time-sharing processes and uses higher numbers to assign lower priorities. The previous example is equivalent to using nice(1) to set an increment of 10:

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

priocntl Interface

priocntl(2) manages the scheduling parameters of a process or set of processes. An invocation of priocntl(2) can act on a LWP, on a single process, or on a group of processes. A group of processes can be identified by parent process, process group, session, user, group, class, or all active processes. For more details, see the priocntl man page.

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

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

Interactions With Other Interfaces

Altering the priority of a process in the TS class can affect the behavior of other processes in the TS class. This section identifies ways in which a scheduling change can affect other processes.

Kernel Processes

The kernel's daemon and housekeeping processes are members of the system scheduler class. Users can neither add processes to nor remove processes from this class, nor can users change the priorities of these processes. The command ps -cel lists the scheduler class of all processes. A SYS entry in the CLS column identifies processes in the system class when you run ps(1) with the -f option.

Using `fork` and `exec`

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

Using `nice`

The `nice`(1) command and the `nice`(2) interface work as in previous versions of the UNIX system. These commands enable you to change the priority of a time-sharing process. Use lower numeric values to assign higher time-sharing priorities with these interfaces.

To change the scheduler class of a process or to specify a real-time priority, use `priocntl`(2). Use higher numeric values to assign higher priorities.

`init`(1M)

The`init`(1M) process is a special case to the scheduler. To change the scheduling properties of `init`(1M), `init` must be the only process specified by `idtype` and `id` or by the `procset` structure.

Scheduling and System Performance

The scheduler determines when and for how long processes run. Therefore, the scheduler's behavior strongly affects a system's performance.

By default, all user processes are time-sharing processes. A process changes class only by a `priocntl`(2) call.

All real-time process priorities have a higher priority than any time-sharing process. Time-sharing processes or system processes cannot run while any real-time process is runnable. A real-time application that occasionally fails to relinquish control of the CPU can completely lock out other users and essential kernel housekeeping.

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

Process State Transition

Applications that have strict real-time constraints might need to prevent processes from being swapped or paged out to secondary memory. A simplified overview of UNIX process states and the transitions between states is shown in the following figure.

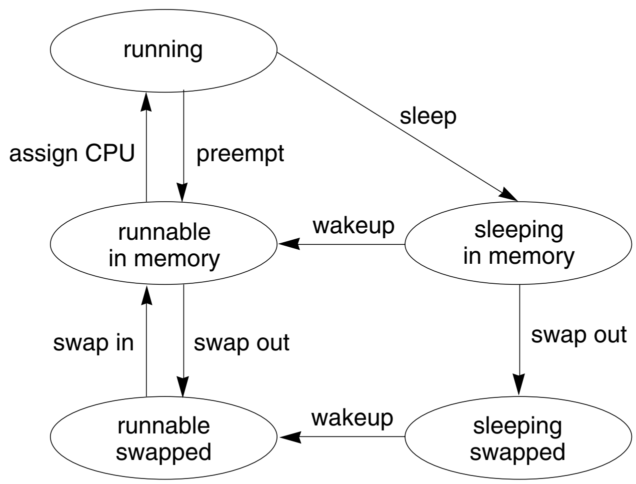


FIGURE 3–2 Process State Transition Diagram

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

- A process is running if the process is assigned to a CPU. A process is removed from the running state by the scheduler if a process with a higher priority becomes runnable. A process is also pre-empted if a process of equal priority is runnable when the original process consumes its entire time slice.
- A process is runnable in memory if the process is in primary memory and ready to run, but is not assigned to a CPU.
- A process is sleeping in memory if the process is in primary memory but is waiting for a specific event before continuing execution. For example, a process sleeps while waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, a wakeup call is sent to the process. If the reason for its sleep is gone, the process becomes runnable.
- When a process' address space has been written to secondary memory, and that process is not waiting for a specific event, the process is runnable and swapped.
- If a process is waiting for a specific event and has had its whole address space written to secondary memory, the process is sleeping and swapped.

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

- When the system is short of primary memory, the system writes individual pages of some processes to secondary memory but leaves those processes runnable. When a running process, accesses those pages, the process sleeps while the pages are read back into primary memory.
- When the system encounters a more serious shortage of primary memory, the system writes all the pages of some processes to secondary memory. The system marks the pages that have been written to secondary memory as swapped. Such processes can only be scheduled when the system scheduler daemon selects these processes to be read back into memory.

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

To avoid swapping delays, real-time processes are never swapped, though parts of such processes can be paged. A program can prevent paging and swapping by locking its text and data into primary memory. For more information, see the `memcntl`(2) man page. How much memory can be locked is limited by how much memory is configured. Also, locking too much can cause intolerable delays to processes that do not have their text and data locked into memory.

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

Note – See "Dispatch Latency" on page 211 for information about latencies in real-time applications.

CHAPTER 4

Locality Group APIs

This chapter describes the APIs that applications use to interact with locality groups.

"Locality Groups Overview" on page 68 describes the locality group abstraction.

"Verifying the Interface Version" on page 70 describes the functions that give information about the interface.

"Initializing the Locality Group Interface" on page 70 describes function calls that initialize and shut down the portion of the interface that is used to traverse the locality group hierarchy and to discover the contents of a locality group.

"Locality Group Hierarchy" on page 72 describes function calls that navigate the locality group hierarchy and get characteristics of the locality group hierarchy.

"Locality Group Contents" on page 74 describes function calls that retrieve information about a locality group's contents.

"Locality Group Characteristics" on page 75 describes function calls that retrieve information about a locality group's characteristics.

"Locality Groups and Thread and Memory Placement" on page 76 describes how to affect a thread's memory placement and other memory management techniques.

"Examples of API usage" on page 84 contains code that performs example tasks by using the APIs that are described in this chapter.

Locality Groups Overview

Shared memory multiprocessor computers contain multiple CPUs. Each CPU can access all of the memory in the machine. In some shared memory multiprocessors, the memory architecture enables each CPU to access some areas of memory more quickly than other areas.

When a machine with such a memory architecture runs Solaris, giving the kernel information about the shortest access times between a given CPU and a given area of memory can improve the system's performance. The locality group (lgroup) abstraction has been introduced in Solaris to handle this information. The lgroup abstraction is part of the Memory Placement Optimization (MPO) feature.

An lgroup is a set of CPU–like and memory–like devices in which each member of the set can access another member of that set within a bounded latency interval. The latency value of each lgroup is chosen by the operating system.

Lgroups are hierarchical. The lgroup hierarchy is a Directed Acyclic Graph (DAG) and is similar to a tree, except that an lgroup may have more than one parent. Like a tree, there is a root. The root lgroup contains all the resources in the system and can include child lgroups. Furthermore, the root lgroup can be characterized as having the highest latency value of all the lgroups in the system. All of its child lgroups will have lower latency values. The lgroups closer to the root have a higher latency while lgroups closer to leaves have lower latency.

A computer in which all the CPUs can access all the memory in the same amount of time can be represented with a single lgroup (see Figure 4–1). A computer in which some of the CPUs can access some areas of memory in a shorter time than other areas can be represented using multiple lgroups (see Figure 4–2).

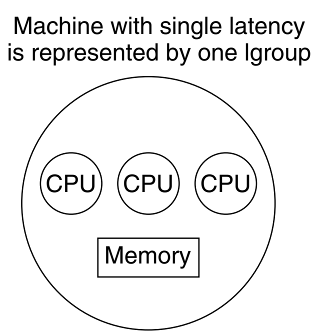


FIGURE 4–1 Single Locality Group Schematic

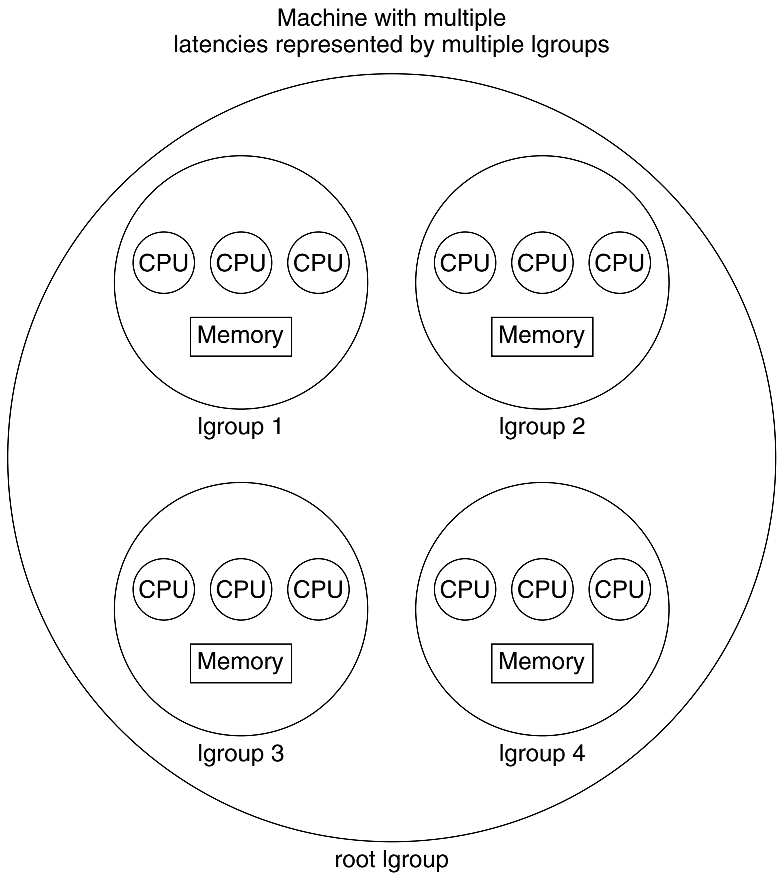


FIGURE 4–2 Multiple Locality Groups Schematic

The organization of the lgroup hierarchy simplifies the task of finding the nearest resources in the system. Each thread is assigned a home lgroup upon creation. The operating system attempts to allocate resources for the thread from the thread's home lgroup by default. For example, the Solaris kernel attempts to schedule a thread to run on the CPUs in the thread's home lgroup and allocate the thread's memory in a way that optimizes for locality. If the desired resources are not available from the thread's home lgroup, the kernel can traverse the lgroup hierarchy to find the next nearest resources from parents of the home lgroup.

The lgroup APIs export the lgroup abstraction for applications to use for observability and operformance tuning. Applications can use the APIs to traverse the lgroup hierarchy, discover the contents and characteristics of a given lgroup, and affect the thread and memory placement on lgroups. A new library, called `liblgrp`, contains the new APIs.

Verifying the Interface Version

The `lgrp_version()` function discussed in this section must be used to verify the presence of a supported lgroup interface before using the lgroup API.

Using `lgrp_version()`

```
#include <sys/lgrp_user.h>
int lgrp_version(const int version);
```

The `lgrp_version()` function takes a version number for the lgroup interface as an argument and returns the lgroup interface version that the system supports. When the current implementation of the lgroup API supports the version number in the `version` argument, the `lgrp_version()` function returns that version number. Otherwise, the `lgrp_version()` function returns `LGRP_VER_NONE`.

EXAMPLE 4–1 Example of `lgrp_version()` use

```
#include <sys/lgrp_user.h>
if (lgrp_version(LGRP_VER_CURRENT) != LGRP_VER_CURRENT) {
    fprintf(stderr, "Built with unsupported lgroup interface %d\n",
        LGRP_VER_CURRENT);
    exit (1);
    }
```

Initializing the Locality Group Interface

Applications must call `lgrp_init()` in order to use the APIs for traversing the lgroup hierarchy and discover the contents of the lgroup hierarchy. The call to `lgrp_init()` gives the application a consistent snapshot of the lgroup hierarchy. The application developer can specify whether the snapshot contains only the resources available to the calling thread specifically or the resources available to the operating system in general. The `lgrp_init()` function returns a cookie that is used for the following tasks:

- Navigating the lgroup hierarchy
- Determining the contents of an lgroup
- Determining whether or not the snapshot is current

Using `lgrp_init()`

The `lgrp_init()` function initializes the lgroup interface and takes a snapshot of the lgroup hierarchy.

```
#include <sys/lgrp_user.h>
lgrp_cookie_t lgrp_init(lgrp_view_t view);
```

When the `lgrp_init()` function is called with `LGRP_VIEW_CALLER` as the view, the function returns a snapshot that contains only the resources available to the calling thread. When the `lgrp_init()` function is called with `LGRP_VIEW_OS` as the view, the function returns a snapshot that contains the resources that are available to the operating system. When a thread successfully calls the `lgrp_init()` function, the function returns a cookie that is used by any function that interacts with the lgroup hierarchy.

The lgroup hierarchy consists of a root lgroup that contains all of the machine's CPU and memory resources. The root lgroup may contain other locality groups defined by bounded latency intervals.

The `lgrp_init()` function can return two errors. When a view is invalid, the function returns `EINVAL`. When there is insufficient memory to allocate the snapshot of the lgroup hierarchy, the function returns `ENOMEM`.

Using `lgrp_fini()`

The `lgrp_fini()` function ends the usage of a given cookie and frees the corresponding lgroup hierarchy snapshot.

```
#include <sys/lgrp_user.h>
int lgrp_fini(lgrp_cookie_t cookie);
```

The `lgrp_fini()` function takes a cookie which represents an lgroup hierarchy snapshot created by a previous call to `lgrp_init()`. The `lgrp_fini()` function frees the memory that is allocated to that snapshot. After the call to `lgrp_fini()`, the cookie is invalid. Do not use that cookie again.

When the cookie passed to the `lgrp_fini()` function is invalid, `lgrp_fini()` returns `EINVAL`.

Locality Group Hierarchy

The APIs that are described in this section enable the calling thread to navigate the lgroup hierarchy. The lgroup hierarchy is a directed acyclic graph that is similar to a tree, except that a node may have more than one parent. The root lgroup represents the whole machine and is the lgroup with the highest latency value in the system. Each of the child lgroups contains a subset of the hardware in the root lgroup and is bounded by a lower latency value. Locality groups that are closer to the root have more resources and a higher latency. Locality groups that are closer to the leaves have fewer resources and a lower latency.

Using `lgrp_cookie_stale()`

The `lgrp_cookie_stale()` function determines whether the snapshot of the lgroup hierarchy represented by the given cookie is current.

```
#include <sys/lgrp_user.h>
int lgrp_cookie_stale(lgrp_cookie_t cookie);
```

The cookie returned by the `lgrp_init()` function can become stale due to several reasons that depend on the view the snapshot represents. A cookie returned by calling the `lgrp_init()` function with the view set to `LGRP_VIEW_OS` can become stale due to changes in the lgroup hierarchy such as dynamic reconfiguration or a change in a CPU's online status. A cookie returned by calling the `lgrp_init()` function with the view set to `LGRP_VIEW_CALLER` can become stale due to changes in the calling thread's processor set or changes in the lgroup hierarchy. A stale cookie is refreshed by calling the `lgrp_fini()` function with the old cookie, followed by calling `lgrp_init()` to generate a new cookie.

The `lgrp_cookie_stale()` function returns `EINVAL` when the given cookie is invalid.

Using `lgrp_view()`

The `lgrp_view()` function determines the view with which a given lgroup hierarchy snapshot was taken.

```
#include <sys/lgrp_user.h>
lgrp_view_t lgrp_view(lgrp_cookie_t cookie);
```

The `lgrp_view()` function takes a cookie representing a snapshot of the lgroup hierarchy and returns the snapshot's view of the lgroup hierarchy. Snapshots taken with the view `LGRP_VIEW_CALLER` contain only the resources that are available to the calling thread. Snapshots taken with the view `LGRP_VIEW_OS` contain all the resources that are available to the operating system.

The `lgrp_view()` function returns `EINVAL` when the given cookie is invalid.

Using `lgrp_nlgrps()`

The `lgrp_nlgrps()` function returns the number of locality groups in the system. If a system has only one locality group, memory placement optimizations have no effect.

```
#include <sys/lgrp_user.h>
int lgrp_nlgrps(lgrp_cookie_t cookie);
```

The `lgrp_nlgrps()` function takes a cookie representing a snapshot of the lgroup hierarchy and returns the number of lgroups available in the hierarchy.

The `lgrp_nlgrps()` function returns `EINVAL` when the cookie is invalid.

Using `lgrp_root()`

The `lgrp_root()` function returns the root lgroup ID.

```
#include <sys/lgrp_user.h>
lgrp_id_t lgrp_root(lgrp_cookie_t cookie);
```

The `lgrp_root()` function takes a cookie representing a snapshot of the lgroup hierarchy and returns the root lgroup ID.

Using `lgrp_parents()`

The `lgrp_parents()` function takes a cookie representing a snapshot of the lgroup hierarchy and returns the number of parent `lgroups` for the specified lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_parents(lgrp_cookie_t cookie, lgrp_id_t child,
                 lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

If `lgrp_array()` is not `NULL` and the value of `lgrp_array_size` is not zero, the `lgrp_parents()` function fills the array with parent lgroup IDs until the array is full or all parent lgroup IDs are in the array. The root lgroup has zero parents. When the `lgrp_parents()` function is called for the root lgroup, `lgrp_array` will not be filled in.

The `lgrp_parents()` function returns `EINVAL` when the cookie is invalid. The `lgrp_parents()` function returns `ESRCH` when the specified lgroup ID is not found.

Using lgrp_children()

The lgrp_children() function takes a cookie representing the calling thread's snapshot of the lgroup hierarchy and returns the number of child lgroups for the specified lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_children(lgrp_cookie_t cookie, lgrp_id_t parent,
                  lgrp_id_t *lgrp_array, uint_t lgrp_array_size);
```

If lgrp_array is not NULL and the value of lgrp_array_size is not zero, the lgrp_children() function fills the array with child lgroup IDs until the array is full or all child lgroup IDs are in the array.

The lgrp_children() function returns EINVAL when the cookie is invalid. The lgrp_children() function returns ESRCH when the specified lgroup ID is not found.

Locality Group Contents

The following APIs retrieve information about the contents of a given lgroup.

Using lgrp_cpus()

The lgrp_cpus() function takes a cookie representing a snapshot of the lgroup hierarchy and returns the number of CPUs in a given lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_cpus(lgrp_cookie_t cookie, lgrp_id_t lgrp, processorid_t *cpuids,
              uint_t count, int content);
```

If the *cpuid[]* argument is not NULL and the CPU count is not zero, the lgrp_cpus() function fills the array with CPU IDs until the array is full or all the CPU IDs are in the array.

The *content* argument can have the following two values:

LGRP_CONTENT_HIERARCHY	The lgrp_cpus() function returns IDs for the CPUs in this lgroup and this lgroup's descendants.
LGRP_CONTENT_DIRECT	The lgrp_cpus() function returns IDs for the CPUs in this lgroup only.

The lgrp_cpus() function returns EINVAL when the cookie, lgroup ID, or one of the flags is not valid. The lgrp_cpus() function returns ESRCH when the specified lgroup ID is not found.

Using `lgrp_mem_size()`

The `lgrp_mem_size()` function takes a cookie representing a snapshot of the lgroup hierarchy and returns the size of installed or free memory in the given lgroup. The `lgrp_mem_size()` function reports memory sizes in bytes.

```
#include <sys/lgrp_user.h>
lgrp_mem_size_t lgrp_mem_size(lgrp_cookie_t cookie, lgrp_id_t lgrp,
                              int type, int content)
```

The *type* argument can have the following two values:

`LGRP_MEM_SZ_FREE`	The `lgrp_mem_size()` function returns the amount of free memory in bytes.
`LGRP_MEM_SZ_INSTALLED`	The `lgrp_mem_size()` function returns the amount of installed memory in bytes.

The *content* argument can have the following two values:

`LGRP_CONTENT_HIERARCHY`	The `lgrp_mem_size()` function returns the amount of memory in this lgroup and this lgroup's descendants.
`LGRP_CONTENT_DIRECT`	The `lgrp_mem_size()` function returns the amount of memory in this lgroup only.

The `lgrp_mem_size()` function returns `EINVAL` when the cookie, lgroup ID, or one of the flags is not valid. The `lgrp_mem_size()` function returns `ESRCH` when the specified lgroup ID is not found.

Locality Group Characteristics

The following API retrieves information about the characteristics of a given lgroup.

Using `lgrp_latency()`

The `lgrp_latency()` function returns the latency between a CPU in one lgroup to the memory in another lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_latency(lgrp_id_t from, lgrp_id_t to);
```

The `lgrp_latency()` function returns a value that represents the latency between a CPU in the lgroup given by the value of the *from* argument and the memory in the lgroup given by the value of the *to* argument. If both arguments point to the same lgroup, the `lgrp_latency()` function returns the latency value within that lgroup.

Note – The latency value returned by the `lgrp_latency()` function is defined by the operating system and is platform-specific. This value does not necessarily represent the actual latency between hardware devices and may only be used for comparison within one domain.

The `lgrp_latency()` function returns `EINVAL` when the lgroup ID is not valid. When the `lgrp_latency()` function does not find the specified lgroup ID, the 'from' lgroup does not contain any CPUs, or the 'to' lgroup does not have any memory, the `lgrp_latency()` function returns `ESRCH`.

Locality Groups and Thread and Memory Placement

This section discusses the APIs used to discover and affect thread and memory placement with respect to lgroups. The `lgrp_home()` function is used to discover thread placement. The `meminfo`(2) system call is used to discover memory placement. The `MADV_ACCESS` flags to the `madvise`(3C) function are used to affect memory allocation among lgroups. The `lgrp_affinity_set()` function can affect thread and memory placement by setting a thread's affinity for a given lgroup. The affinities of an lgroup may specify an order of preference for lgroups from which to allocate resources. The kernel needs information about the likely pattern of an application's memory use in order to allocate memory resources efficiently. The `madvise()` function, and its shared object analogue `madv.so.1`, provide this information to the kernel. A running process can gather memory usage information about itself by using the `meminfo()` system call.

Using `lgrp_home()`

The `lgrp_home()` function returns the home lgroup for the specified process or thread.

```
#include <sys/lgrp_user.h>
lgrp_id_t lgrp_home(idtype_t idtype, id_t id);
```

The `lgrp_home()` function returns `EINVAL` when the ID type is not valid. The `lgrp_home()` function returns `EPERM` when the effective user of the calling process is not the superuser and the calling process' real or effective user ID does not match the real or effective user ID of one of the threads. The `lgrp_home()` function returns `ESRCH` when the specified process or thread is not found.

Using `madvise()`

The `madvise()` function advises the kernel that a region of user virtual memory in the range starting at the address specified in *addr* and with length equal to the value of the *len* parameter is expected to follow a particular pattern of use. The kernel uses this information to optimize the procedure for manipulating and maintaining the resources associated with the specified range. Use of the `madvise()` function can increase system performance when used by programs that have specific knowledge of their access patterns over memory.

```
#include <sys/types.h>
#include <sys/mman.h>
int madvise(caddr_t addr, size_t len, int advice);
```

The `madvise()` function provides the following flags to affect how a thread's memory is allocated among lgroups:

`MADV_ACCESS_DEFAULT` This flag resets the kernel's expected access pattern for the specified range to the default.

`MADV_ACCESS_LWP` This flag advises the kernel that the next LWP to touch the specified address range is the LWP that will access that range the most. The kernel allocates the memory and other resources for this range and the LWP accordingly.

`MADV_ACCESS_MANY` This flag advises the kernel that many processes or LWPs will access the specified address range randomly across the system. The kernel allocates the memory and other resources for this range accordingly.

The `madvise()` function returns `EAGAIN` when some or all of the mappings in the specified address range, from *addr* to *addr+len*, are locked for I/O. The `madvise()` function returns `EINVAL` when the value of the *addr* parameter is not a multiple of the page size as returned by `sysconf`(3C). The `madvise()` function returns `EINVAL` when the length of the specified address range is less than or equal to zero. The `madvise()` function returns `EINVAL` when the advice is invalid. The `madvise()` function returns `EIO` when an I/O error occurs while reading from or writing to the file system. The `madvise()` function returns `ENOMEM` when addresses in the specified address range are outside the valid range for the address space of a process, or the addresses in the specified address range specifiy one or more pages that are not mapped. The `madvise()` function returns `ESTALE` when the NFS file handle is stale.

Using madv.so.1

The madv.so.1 shared object enables the selective configuration of virtual memory advice for launched processes and their descendants. To use the shared object, the following string must be present in the environment:

```
LD_PRELOAD=$LD_PRELOAD:madv.so.1
```

The madv.so.1 shared object applies memory advice as specified by the value of the MADV environment variable. The MADV environment variable specifies the virtual memory advice to use for all heap, shared memory, and mmap regions in the process address space. This advice is applied to all created processes. The following values of the MADV environment variable affect resource allocation among lgroups:

access_default — This value resets the kernel's expected access pattern to the default.

access_lwp — This value advises the kernel that the next LWP to touch an address range is the LWP that will access that range the most. The kernel allocates the memory and other resources for this range and the LWP accordingly.

access_many — This value advises the kernel that many processes or LWPs will access memory randomly across the system. The kernel allocates the memory and other resources accordingly.

The value of the MADVCFGFILE environment variable is the name of a text file that contains one or more memory advice configuration entries in the form <exec-name>:<advice-opts>.

The value of <exec-name> is the name of an application or executable. The value of <exec-name> can be a full pathname, a base name, or a pattern string.

The value of <advice-opts> is of the form <region>=<advice>. The values of <advice> are the same as the values for the MADV environment variable. Replace <region> with any of the following legal values:

madv — Advice applies to all heap, shared memory, and mmap(2) regions in the process address space.

heap — The heap is defined to be the brk(2) area. Advice applies to the existing heap and to any additional heap memory allocated in the future.

shm — Advice applies to shared memory segments. See shmat(2) for more information on shared memory operations.

ism — Advice applies to shared memory segments that are using the SHM_SHARE_MMU flag. The ism option takes precedence over shm.

dsm — Advice applies to shared memory segments that are using the SHM_PAGEABLE flag. The dsm option takes precedence over shm.

`mapshared`	Advice applies to mappings established by the `mmap()` system call using the `MAP_SHARED` flag.
`mapprivate`	Advice applies to mappings established by the `mmap()` system call using the `MAP_PRIVATE` flag.
`mapanon`	Advice applies to mappings established by the `mmap()` system call using the `MAP_ANON` flag. The `mapanon` option takes precendence when multiple options apply.

The value of the `MADVERRFILE` environment variable is the pathname where error messages are logged. In the absence of a `MADVERRFILE` location, the `madv.so.1` shared object logs errors by using `syslog`(3C) with a `LOG_ERR` as the severity level and `LOG_USER` as the facility descriptor.

Memory advice is inherited. A child process has the same advice as its parent. The advice is set back to the system default advice after a call to `exec`(2) unless a different level of advice is configured via the `madv.so.1` shared object. Advice is only applied to `mmap()` regions explicitly created by the user program. Regions established by the run-time linker or by system libraries that make direct system calls are not affected.

`madv.so.1` Usage Examples

The following examples illustrate specific aspects of the `madv.so.1` shared object.

EXAMPLE 4–2 Setting Advice for a Set of Applications

This configuration applies advice to all ISM segments for applications with exec names that begin with `foo`.

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        foo*:ism=access_lwp
```

EXAMPLE 4–3 Excluding a Set of Applications From Advice

This configuration sets advice for all applications with the exception of `ls`.

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADV=access_many
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADV MADVCFGFILE
$ cat $MADVCFGFILE
        ls:
```

EXAMPLE 4–4 Pattern Matching in a Configuration File

Because the configuration specified in MADVCFGFILE takes precedence over the value set in MADV, specifying * as the <exec-name> of the last configuration entry is equivalent to setting MADV. This example is equivalent to the previous example.

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        ls:
        *:madv=access_many
```

EXAMPLE 4–5 Advice for Multiple Regions

This configuration applies one type of advice for mmap() regions and different advice for heap and shared memory regions for applications whose exec() names begin with foo.

```
$ LD_PRELOAD=$LD_PRELOAD:madv.so.1
$ MADVCFGFILE=madvcfg
$ export LD_PRELOAD MADVCFGFILE
$ cat $MADVCFGFILE
        foo*:madv=access_many,heap=sequential,shm=access_lwp
```

Using meminfo()

The meminfo() function gives the calling process information about the virtual memory and physical memory that the system has allocated to that process.

```
#include <sys/types.h>
#include <sys/mman.h>
int meminfo(const uint64_t inaddr[], int addr_count,
    const uint_t info_req[], int info_count, uint64_t outdata[],
    uint_t validity[]);
```

The meminfo() function can return the following types of information:

MEMINFO_VPHYSICAL	The physical memory address corresponding to the given virtual address.
MEMINFO_VLGRP	The lgroup to which the physical page corresponding to the given virtual address belongs.
MEMINFO_VPAGESIZE	The size of the physical page corresponding to the given virtual address.
MEMINFO_VREPLCNT	The number of replicated physical pages that correspond to the given virtual address.
MEMINFO_VREPL|n	The *n*th physical replica of the given virtual address.

`MEMINFO_VREPL_LGRP|n`	The lgroup to which the *n*th physical replica of the given virtual address belongs.
`MEMINFO_PLGRP`	The lgroup to which the given physical address belongs.

The `meminfo()` function takes the following parameters:

inaddr — An array of input addresses.

addr_count — The number of addresses that are passed to `meminfo()`.

info_req — An array listing the types of information that are being requested.

info_count — The number of pieces of information that are requested for each address in the *inaddr* array.

outdata — An array where the `meminfo()` function places the results. The array's size is equal to the product of the values of the *info_req* and *addr_count* parameters.

validity — An array of size equal to the value of the *addr_count* parameter. The *validity* array contains bitwise result codes. The *0*th bit of the result code evaluates the validity of the corresponding input address. Each successive bit in the result code evaluates the validity of the response to the members of the *info_req* array in turn.

The `meminfo()` function returns `EFAULT` when the area of memory that the *outdata* or *validity* arrays point to cannot be written to. The `meminfo()` function returns `EFAULT` when the area of memory that the *info_req* or *inaddr* arrays point to cannot be read from. The `meminfo()` function returns `EINVAL` when the value of *info_count* exceeds 31 or is less than 1. The `meminfo()` function returns `EINVAL` when the value of *addr_count* is less than zero.

EXAMPLE 4–6 Use of `meminfo()` to print out physical pages and page sizes corresponding to a set of virtual addresses

```
void
print_info(void **addrvec, int how_many)
{
        static const int info[] = {
                MEMINFO_VPHYSICAL,
                MEMINFO_VPAGESIZE};
        uint64_t * inaddr = alloca(sizeof(uint64_t) * how_many);
        uint64_t * outdata = alloca(sizeof(uint64_t) * how_many * 2;
        uint_t * validity = alloca(sizeof(uint_t) * how_many);

        int i;

        for (i = 0; i < how_many; i++)
                inaddr[i] = (uint64_t *)addr[i];

        if (meminfo(inaddr, how_many,  info,
                    sizeof (info)/ sizeof(info[0]),
```

EXAMPLE 4–6 Use of `meminfo()` to print out physical pages and page sizes corresponding to a set of virtual addresses *(Continued)*

```
                    outdata, validity) < 0)
                ...

        for (i = 0; i < how_many; i++) {
                if (validity[i] & 1 == 0)
                        printf("address 0x%llx not part of address
                                        space\n",
                                inaddr[i]);

                else if (validity[i] & 2 == 0)
                        printf("address 0x%llx has no physical page
                                        associated with it\n",
                                inaddr[i]);

                else {
                        char buff[80];
                        if (validity[i] & 4 == 0)
                                strcpy(buff, "<Unknown>");
                        else
                                sprintf(buff, "%lld", outdata[i * 2 +
                                                1]);
                        printf("address 0x%llx is backed by physical
                                        page 0x%llx of size %s\n",
                                        inaddr[i], outdata[i * 2], buff);
                }
        }
}
```

Locality Group Affinity

The kernel assigns a thread to a locality group when the light weight process (LWP) for that thread is created. That lgroup is called the thread's *home lgroup*. The kernel runs the thread on the CPUs in the thread's home lgroup and allocates memory from that lgroup whenever possible. If resources from the home lgroup are unavailable, the kernel allocates resources from other lgroups. When a thread has affinity for more than one lgroup, the operating system allocates resources from lgroups chosen in order of affinity strength. There are three affinity levels:

1. `LGRP_AFF_STRONG` indicates strong affinity. If this lgroup is the thread's home lgroup, the operating system avoids rehoming the thread to another lgroup if possible. Events such as dynamic reconfiguration, processor, offlining, processor binding, and processor set binding and manipulation may still result in thread rehoming.
2. `LGRP_AFF_WEAK` indicates weak affinity. If this lgroup is the thread's home lgroup, the operating system rehomes the thread if necessary for load balancing purposes.

3. `LGRP_AFF_NONE` indicates no affinity. If a thread has no affinity to any lgroup, the operating system assigns the thread a home lgroup.

The operating system uses lgroup affinities as advice when allocating resources for a given thread. The advice is factored in with the other system constraints. Processor binding and processor sets do not change lgroup affinities, but may restrict the lgroups on which a thread can run.

Using `lgrp_affinity_get()`

The `lgrp_affinity_get()` function returns the affinity that a LWP or set of LWPs have for a given lgroup.

```
#include <sys/lgrp_user.h>
lgrp_affinity_t lgrp_affinity_get(idtype_t idtype, id_t id, lgrp_id_t lgrp);
```

The *idtype* and *id* arguments specify the LWP or set of LWPs that the `lgrp_affinity_get()` function examines. If the value of *idtype* is `P_PID`, the `lgrp_affinity_get()` function gets the lgroup affinity for one of the LWPs in the process whose process ID matches the value of the *id* argument. If the value of *idtype* is `P_LWPID`, the `lgrp_affinity_get()` function gets the lgroup affinity for the LWP of the current process whose LWP ID matches the value of the *id* argument. If the value of *idtype* is `P_MYID`, the `lgrp_affinity_get()` function gets the lgroup affinity for the current LWP or process.

The `lgrp_affinity_get()` function returns `EINVAL` when the given lgroup, affinity, or ID type is not valid. The `lgrp_affinity_get()` function returns `EPERM` when the effective user of the calling process is not the superuser and the calling process' ID does not match the real or effective user ID of one of the LWPs. The `lgrp_affinity_get()` function returns `ESRCH` when a given lgroup or LWP is not found.

Using `lgrp_affinity_set()`

The `lgrp_affinity_set()` function sets the affinity that a LWP or set of LWPs have for a given lgroup.

```
#include <sys/lgrp_user.h>
int lgrp_affinity_set(idtype_t idtype, id_t id, lgrp_id_t lgrp,
                      lgrp_affinity_t affinity);
```

The *idtype* and *id* arguments specify the LWP or set of LWPs the `lgrp_affinity_set()` function examines. If the value of *idtype* is `P_PID`, the `lgrp_affinity_set()` function sets the lgroup affinity for all of the LWPs in the process whose process ID matches the value of the *id* argument to the affinity level specified in the *affinity* argument. If the value of *idtype* is `P_LWPID`, the `lgrp_affinity_set()` function sets the lgroup affinity for the LWP of the current process whose LWP ID matches the value of the *id* argument to the affinity level specified in the *affinity* argument. If the value of *idtype* is `P_MYID`, the `lgrp_affinity_set()` function sets the lgroup affinity for the current LWP or process to the affinity level specified in the *affinity* argument.

The `lgrp_affinity_set()` function returns `EINVAL` when the given lgroup, affinity, or ID type is not valid. The `lgrp_affinity_set()` function returns `EPERM` when the effective user of the calling process is not the superuser and the calling process' ID does not match the real or effective user ID of one of the LWPs. The `lgrp_affinity_set()` function returns `ESRCH` when a given lgroup or LWP is not found.

Examples of API usage

This section contains code that performs example tasks by using the APIs that are described in this chapter.

EXAMPLE 4–7 Move Memory to a Thread

The following code sample moves the memory in the range from the address specified by *addr* to the address specified by *addr+len* to the thread specified by `MADV_ACCESS_LWP`.

```
#include <sys/mman.h>
#include <sys/types.h>

/*
 * Move memory to thread
 */
mem_to_thread(caddr_t addr, size_t len)
{
    if (madvise(addr, len, MADV_ACCESS_LWP) < 0)
        perror("madvise");
}
```

EXAMPLE 4–8 Move a Thread to Memory

This sample code uses the `meminfo()` function to return the lgroup of a specified memory page and raises the specified thread's affinity to that lgroup with the `lgrp_affinity_set function()`.

```
#include <sys/lgrp_user.h>
#include <sys/mman.h>
#include <sys/types.h>
```

EXAMPLE 4–8 Move a Thread to Memory *(Continued)*

```
/*
 * Move a Thread to Memory
 */
int
thread_to_memory(caddr_t va)
{
    uint64_t    addr;
    ulong_t        count;
    lgrp_id_t    home;
    uint64_t    lgrp;
    uint_t        request;
    uint_t        valid;

    addr = (uint64_t)va;
    count = 1;
    request = MEMINFO_VLGRP;
    if (meminfo(&addr, 1, &request, 1, &lgrp, &valid) != 0) {
        perror("meminfo");
        return (1);
    }

    if (lgrp_affinity_set(P_LWPID, P_MYID, lgrp, LGRP_AFF_STRONG) != 0) {
        perror("lgrp_affinity_set");
        return (2);
    }

    home = lgrp_home(P_LWPID, P_MYID);
    if (home == -1) {
        perror ("lgrp_home");
        return (3);
    }

    if (home != lgrp)
        return (-1);

    return (0);
}
```

EXAMPLE 4–9 Walk the lgroup Hierarchy

The following sample code walks through and prints out the lgroup hierarchy.

```
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>


/*
 * Walk and print lgroup hierarchy from given lgroup
 * through all its descendants
 */
```

```
int
lgrp_walk(lgrp_cookie_t cookie, lgrp_id_t lgrp, lgrp_content_t content)
{
    lgrp_affinity_t    aff;
    lgrp_id_t    *children;
    processorid_t    *cpuids;
    int        i;
    int        ncpus;
    int        nchildren;
    int        nparents;
    lgrp_id_t    *parents;
    lgrp_mem_size_t    size;

    /*
     * Print given lgroup, caller's affinity for lgroup,
     * and desired content specified
     */
    printf("LGROUP #%d:\n", lgrp);

    aff = lgrp_affinity_get(P_MYID, P_MYID, lgrp);
    if (aff == -1)
        perror ("lgrp_affinity_get");
    printf("\tAFFINITY: %d\n", aff);

    printf("CONTENT %d:\n", content);

    /*
     * Get CPUs
     */
    ncpus = lgrp_cpus(cookie, lgrp, NULL, 0, content);
    printf("\t%d CPUS: ", ncpus);
    if (ncpus == -1) {
        perror("lgrp_cpus");
        return (-1);
    } else if (ncpus > 0) {
        cpuids = malloc(ncpus * sizeof (processorid_t));
        ncpus = lgrp_cpus(cookie, lgrp, cpuids, ncpus, content);
        if (ncpus == -1) {
            free(cpuids);
            perror("lgrp_cpus");
            return (-1);
        }
        for (i = 0; i < ncpus; i++)
            printf("%d ", cpuids[i]);
        free(cpuids);
    }
    printf("\n");

    /*
     * Get memory size
     */
    printf("\tMEMORY: ");
```

```
    size = lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_INSTALLED, content);
    if (size == -1) {
        perror("lgrp_mem_size");
        return (-1);
    }
    printf("installed bytes 0x%llx, ", size);
    size = lgrp_mem_size(cookie, lgrp, LGRP_MEM_SZ_FREE, content);
    if (size == -1) {
        perror("lgrp_mem_size");
        return (-1);
    }
    printf("free bytes 0x%llx\n", size);

    /*
     * Get parents
     */
    nparents = lgrp_parents(cookie, lgrp, NULL, 0);
    printf("\t%d PARENTS: ", nparents);
    if (nparents == -1) {
        perror("lgrp_parents");
        return (-1);
    } else if (nparents > 0) {
        parents = malloc(nparents * sizeof (lgrp_id_t));
        nparents = lgrp_parents(cookie, lgrp, parents, nparents);
        if (nparents == -1) {
            free(parents);
            perror("lgrp_parents");
            return (-1);
                   }
        for (i = 0; i < nparents; i++)
            printf("%d ", parents[i]);
        free(parents);
    }
    printf("\n");

    /*
     * Get children
     */
    nchildren = lgrp_children(cookie, lgrp, NULL, 0);
    printf("\t%d CHILDREN: ", nchildren);
    if (nchildren == -1) {
        perror("lgrp_children");
        return (-1);
    } else if (nchildren > 0) {
        children = malloc(nchildren * sizeof (lgrp_id_t));
        nchildren = lgrp_children(cookie, lgrp, children, nchildren);
        if (nchildren == -1) {
            free(children);
            perror("lgrp_children");
            return (-1);
                   }
        printf("Children: ");
```

EXAMPLE 4–9 Walk the lgroup Hierarchy *(Continued)*

```
        for (i = 0; i < nchildren; i++)
            printf("%d ", children[i]);
        printf("\n");

        for (i = 0; i < nchildren; i++)
            lgrp_walk(cookie, children[i], content);

        free(children);
    }
    printf("\n");

    return (0);
}
```

EXAMPLE 4–10 Find the Closest lgroup With Available Memory Outside a Given lgroup

```
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>

#define    INT_MAX    2147483647

/*
 * Find next closest lgroup outside given one with available memory
 */
lgrp_id_t
lgrp_next_nearest(lgrp_cookie_t cookie, lgrp_id_t from)
{
    lgrp_id_t    closest;
    int         i;
    int         latency;
    int         lowest;
    int         nparents;
    lgrp_id_t    *parents;
    lgrp_mem_size_t    size;


    /*
     * Get number of parents
     */
    nparents = lgrp_parents(cookie, from, NULL, 0);
    if (nparents == -1) {
        perror("lgrp_parents");
        return (LGRP_NONE);
    }

    /*
     * No parents, so current lgroup is next nearest
     */
    if (nparents == 0) {
        return (from);
```

EXAMPLE 4–10 Find the Closest lgroup With Available Memory Outside a Given lgroup *(Continued)*

```
    }

    /*
     * Get parents
     */
    parents = malloc(nparents * sizeof (lgrp_id_t));
    nparents = lgrp_parents(cookie, from, parents, nparents);
    if (nparents == -1) {
        perror("lgrp_parents");
        free(parents);
        return (LGRP_NONE);
        }

    /*
     * Find closest parent (ie. the one with lowest latency)
     */
    closest = LGRP_NONE;
    lowest = INT_MAX;
    for (i = 0; i < nparents; i++) {
        lgrp_id_t    lgrp;

        /*
         * See whether parent has any free memory
         */
        size = lgrp_mem_size(cookie, parents[i], LGRP_MEM_SZ_FREE,
            LGRP_CONTENT_HIERARCHY);
        if (size > 0)
            lgrp = parents[i];
        else {
            if (size == -1)
                perror("lgrp_mem_size");

            /*
             * Find nearest ancestor if parent doesn't
             * have any memory
             */
            lgrp = lgrp_next_nearest(cookie, parents[i]);
            if (lgrp == LGRP_NONE)
                continue;
        }

        /*
         * Get latency within parent lgroup
         */
        latency = lgrp_latency(lgrp, lgrp);
        if (latency == -1) {
            perror("lgrp_latency");
            continue;
        }

        /*
```

EXAMPLE 4–10 Find the Closest lgroup With Available Memory Outside a Given lgroup *(Continued)*

```
         * Remember lgroup with lowest latency
         */
        if (latency < lowest) {
            closest = lgrp;
            lowest = latency;
        }
    }

    free(parents);
    return (closest);
}


/*
 * Find lgroup with memory nearest home lgroup of current thread
 */
lgrp_id_t
lgrp_nearest(lgrp_cookie_t cookie)
{
    lgrp_id_t    home;
    longlong_t    size;

    /*
     * Get home lgroup
     */
    home = lgrp_home(P_LWPID, P_MYID);

    /*
     * See whether home lgroup has any memory available in its hierarchy
     */
    size = lgrp_mem_size(cookie, home, LGRP_MEM_SZ_FREE,
        LGRP_CONTENT_HIERARCHY);
    if (size == -1)
        perror("lgrp_mem_size");

    /*
     * It does, so return the home lgroup.
     */
    if (size > 0)
        return (home);

    /*
     * Otherwise, find next nearest lgroup outside of the home.
     */
    return (lgrp_next_nearest(cookie, home));
}
```

EXAMPLE 4–11 Find Nearest lgroup With Free Memory

This example code finds the nearest lgroup with free memory to a given thread's home lgroup.

```
#include <stdlib.h>
#include <sys/lgrp_user.h>
#include <sys/types.h>


#define    INT_MAX    2147483647


/*
 * Find next closest lgroup outside given one with available memory
 */
lgrp_id_t
lgrp_next_nearest(lgrp_cookie_t cookie, lgrp_id_t from)
{
    lgrp_id_t    closest;
    int        i;
    int        latency;
    int        lowest;
    int        nparents;
    lgrp_id_t    *parents;
    lgrp_mem_size_t    size;


    /*
     * Get number of parents
     */
    nparents = lgrp_parents(cookie, from, NULL, 0);
    if (nparents == -1) {
        perror("lgrp_parents");
        return (LGRP_NONE);
    }

    /*
     * No parents, so current lgroup is next nearest
     */
    if (nparents == 0) {
        return (from);
    }

    /*
     * Get parents
     */
    parents = malloc(nparents * sizeof (lgrp_id_t));
    nparents = lgrp_parents(cookie, from, parents, nparents);
    if (nparents == -1) {
        perror("lgrp_parents");
        free(parents);
        return (LGRP_NONE);
        }
```

EXAMPLE 4–11 Find Nearest lgroup With Free Memory *(Continued)*

```
    /*
     * Find closest parent (ie. the one with lowest latency)
     */
    closest = LGRP_NONE;
    lowest = INT_MAX;
    for (i = 0; i < nparents; i++) {
        lgrp_id_t    lgrp;

        /*
         * See whether parent has any free memory
         */
        size = lgrp_mem_size(cookie, parents[i], LGRP_MEM_SZ_FREE,
            LGRP_CONTENT_HIERARCHY);
        if (size > 0)
            lgrp = parents[i];
        else {
            if (size == -1)
                perror("lgrp_mem_size");

            /*
             * Find nearest ancestor if parent doesn't
             * have any memory
             */
            lgrp = lgrp_next_nearest(cookie, parents[i]);
            if (lgrp == LGRP_NONE)
                continue;
        }

        /*
         * Get latency within parent lgroup
         */
        latency = lgrp_latency(lgrp, lgrp);
        if (latency == -1) {
            perror("lgrp_latency");
            continue;
        }

        /*
         * Remember lgroup with lowest latency
         */
        if (latency < lowest) {
            closest = lgrp;
            lowest = latency;
        }
    }

    free(parents);
    return (closest);
}
```

EXAMPLE 4–11 Find Nearest lgroup With Free Memory *(Continued)*

```
/*
 * Find lgroup with memory nearest home lgroup of current thread
 */
lgrp_id_t
lgrp_nearest(lgrp_cookie_t cookie)
{
    lgrp_id_t    home;
    longlong_t    size;

    /*
     * Get home lgroup
     */
    home = lgrp_home(P_LWPID, P_MYID);

    /*
     * See whether home lgroup has any memory available in its hierarchy
     */
    size = lgrp_mem_size(cookie, home, LGRP_MEM_SZ_FREE,
        LGRP_CONTENT_HIERARCHY);
    if (size == -1)
        perror("lgrp_mem_size");

    /*
     * It does, so return the home lgroup.
     */
    if (size > 0)
        return (home);

    /*
     * Otherwise, find next nearest lgroup outside of the home.
     */
    return (lgrp_next_nearest(cookie, home));
}
```

CHAPTER **5**

Input/Output Interfaces

This chapter introduces file input/output operations, as provided on systems that do not provide virtual memory services. The chapter discusses the improved input/output method provided by the virtual memory facilities. The chapter describes the older method of locking files and records in "Using File and Record Locking" on page 98.

Files and I/O Interfaces

Files that are organized as a sequence of data are called *regular* files. Regular files can contain ASCII text, text in some other binary data encoding, executable code, or any combination of text, data, and code.

A regular file is made up of the following components:

- Control data, which is called the *inode*. This data includes the file type, the access permissions, the owner, the file size, and the location of the data blocks.
- File contents: a nonterminated sequence of bytes.

The Solaris operating environment provides the following basic forms of file input/output interfaces:

- The traditional, raw style of file I/O is described in "Basic File I/O" on page 96.
- The standard I/O buffering provides an easier interface and improved efficiency to an application run on a system without virtual memory. In an application running in a virtual memory environment, such as on the SunOS™ operating system, standard file I/O is outdated.
- The memory mapping interface is described in "Memory Management Interfaces" on page 15. Mapping files is the most efficient form of file I/O for most applications run under the SunOS™ platform.

Basic File I/O

The following interfaces perform basic operations on files and on character I/O devices.

TABLE 5–1 Basic File I/O Interfaces

Interface Name	Purpose
`open`(2)	Open a file for reading or writing
`close`(2)	Close a file descriptor
`read`(2)	Read from a file
`write`(2)	Write to a file
`creat`(2)	Create a new file or rewrite an existing one
`unlink`(2)	Remove a directory entry
`lseek`(2)	Move read/write file pointer

The following code sample demonstrates the use of the basic file I/O interface. `read`(2) and `write`(2) both transfer no more than the specified number of bytes, starting at the current offset into the file. The number of bytes actually transferred is returned. The end of a file is indicated on a `read`(2) by a return value of zero.

EXAMPLE 5–1 Basic File I/O Interface

```
#include            <fcntl.h>
#define            MAXSIZE            256

main()
{
    int     fd;
    ssize_t n;
    char        array[MAXSIZE];

    fd = open ("/etc/motd", O_RDONLY);
    if (fd == -1) {
        perror ("open");
        exit (1);
    }
    while ((n = read (fd, array, MAXSIZE)) > 0)
        if (write (1, array, n) != n)
            perror ("write");
    if (n == -1)
        perror ("read");
    close (fd);
}
```

When you are done reading or writing a file, always call `close`(2). Do not call `close`(2) for a file descriptor that was not returned from a call to `open`(2).

File pointer offsets into an open file are changed by using read(2), write(2), or by calls to lseek(2). The following example demonstrates the uses of lseek.

```
off_t        start, n;
 struct        record        rec;

 /* record current offset in start */
 start = lseek (fd, 0L, SEEK_CUR);

 /* go back to start */
 n = lseek (fd, -start, SEEK_SET);
 read (fd, &rec, sizeof (rec));

 /* rewrite previous record */
 n = lseek (fd, -sizeof (rec), SEEK_CUR);
 write (fd, (char *&rec, sizeof (rec));
```

Advanced File I/O

The following table lists the tasks performed by advanced file I/O interfaces.

TABLE 5–2 Advanced File I/O Interfaces

Interface Name	Purpose
link(2)	Link to a file
access(2)	Determine accessibility of a file
mknod(2)	Make a special or ordinary file
chmod(2)	Change mode of file
chown(2), lchown(2), fchown(2)	Change owner and group of a file
utime(2)	Set file access and modification times
stat(2), lstat(2), fstat(2)	Get file status
fcntl(2)	Perform file control functions
ioctl(2)	Control device
fpathconf(2)	Get configurable path name variables
opendir(3C), readdir(3C), closedir(3C)	Perform directory operations
mkdir(2)	Make a directory
readlink(2)	Read the value of a symbolic link
rename(2)	Change the name of a file
rmdir(2)	Remove a directory

TABLE 5–2 Advanced File I/O Interfaces *(Continued)*

Interface Name	Purpose
symlink(2)	Make a symbolic link to a file

File System Control

The file system control interfaces listed in the following table enable the control of various aspects of the file system.

TABLE 5–3 File System Control Interfaces

Interface Name	Purpose
ustat(2)	Get file system statistics
sync(2)	Update super block
mount(2)	Mount a file system
statvfs(2), fstatvfs(2)	Get file system information
sysfs(2)	Get file system type information

Using File and Record Locking

You do not need to use traditional file I/O to lock file elements. Use the lighter weight synchronization mechanisms that are described in *Multithreaded Programming Guide* with mapped files.

Locking files prevents errors that can occur when several users try to update a file at the same time. You can lock a portion of a file.

File locking blocks access to an entire file. Record locking blocks access to a specified segment of the file. In SunOS, all files are a sequence of bytes of data: a record is a concept of the programs that use the file.

Choosing a Lock Type

Mandatory locking suspends a process until the requested file segments are free. Advisory locking returns a result indicating whether the lock was obtained or not. A process can ignore the result of advisory locking. You cannot use both mandatory and advisory file locking on the same file at the same time. The mode of a file at the time the file is opened determines whether locks on a file are treated as mandatory or advisory.

Of the two basic locking calls, `fcntl`(2) is more portable, more powerful, and less easy to use than `lockf`(3C). `fcntl`(2) is specified in POSIX 1003.1 standard. `lockf`(3C) is provided to be compatible with older applications.

Selecting Advisory or Mandatory Locking

For mandatory locks, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory.

Set a mandatory lock as follows.

```
#include <sys/types.h>
#include <sys/stat.h>

 int mode;
 struct stat buf;
     ...
     if (stat(filename, &buf) < 0) {
         perror("program");
         exit (2);
     }
     /* get currently set mode */
     mode = buf.st_mode;
     /* remove group execute permission from mode */
     mode &= ~(S_IEXEC>>3);
         /* set 'set group id bit' in mode */
     mode |= S_ISGID;
     if (chmod(filename, mode) < 0) {
         perror("program");
         exit(2);
     }
     ...
```

The operating system ignores record locks when the system is executing a file. Any files with record locks should not have execute permissions set.

The `chmod`(1) command can also be used to set a file to permit mandatory locking.

`$` **`chmod +l`** *file*

This command sets the `O20n0` permission bit in the file mode, which indicates mandatory locking on the file. If *n* is even, the bit is interpreted as enabling mandatory locking. If *n* is odd, the bit is interpreted as "set group ID on execution."

The `ls`(1) command shows this setting when you ask for the long listing format with the `-l` option:

`$` **`ls -l`** *file*

This command displays the following information:

```
-rw---l--- 1 user group size mod_time file
```

The letter “l” in the permissions indicates that the set-group-ID bit is on. Since the set-group-ID bit is on, mandatory locking is enabled. Normal semantics of set group ID are also enabled.

Cautions About Mandatory Locking

Keep in mind the following aspects of locking:

- Mandatory locking works only for local files. Mandatory locking is not supported when accessing files through NFS.
- Mandatory locking protects only the segments of a file that are locked. The remainder of the file can be accessed according to normal file permissions.
- If multiple reads or writes are needed for an atomic transaction, the process should explicitly lock all such segments before any I/O begins. Advisory locks are sufficient for all programs that perform in this way.
- Arbitrary programs should not have unrestricted access permission to files on which record locks are used.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Supported File Systems

Both advisory and mandatory locking are supported on the file systems listed in the following table.

TABLE 5–4 Supported File Systems

File System	Description
`ufs`	The default disk-based file system
`fifofs`	A pseudo file system of named pipe files that give processes common access to data
`namefs`	A pseudo file system used mostly by STREAMS for dynamic mounts of file descriptors on top of file
`specfs`	A pseudo file system that provides access to special character devices and block devices

Only advisory file locking is supported on NFS. File locking is not supported for the `proc` and `fd` file systems.

Opening a File for Locking

You can only request a lock for a file with a valid open descriptor. For read locks, the file must be open with at least read access. For write locks, the file must also be open with write access. In the following example, a file is opened for both read and write access.

```
...
     filename = argv[1];
     fd = open (filename, O_RDWR);
     if (fd < 0) {
         perror(filename);
         exit(2);
     }
     ...
```

Setting a File Lock

To lock an entire file, set the offset to zero and set the size to zero.

You can set a lock on a file in several ways. The choice of method depends on how the lock interacts with the rest of the program, performance, and portability. This example uses the POSIX standard-compatible `fcntl`(2) interface. The interface tries to lock a file until one of the following happens:

- The file lock is set successfully.
- An error occurs.
- MAX_TRY is exceeded, and the program stops trying to lock the file.

```
#include <fcntl.h>

 ...
     struct flock lck;

 ...
     lck.l_type = F_WRLCK;    /* setting a write lock */
     lck.l_whence = 0;    /* offset l_start from beginning of file */
     lck.l_start = (off_t)0;
     lck.l_len = (off_t)0;    /* until the end of the file */
     if (fcntl(fd, F_SETLK, &lck) <0) {
         if (errno == EAGAIN || errno == EACCES) {
             (void) fprintf(stderr, "File busy try again later!\n");
             return;
         }
         perror("fcntl");
         exit (2);
     }
     ...
```

Using `fcntl`(2), you can set the type and start of the lock request by setting structure variables.

Note – You cannot lock mapped files with flock(3UCB). However, you can use the multithread-oriented synchronization mechanisms with mapped files. These synchronization mechanisms can be used in POSIX styles as well as in Soalris styles. See the mutex(3THR), condition(3THR), semaphore(3THR), mmap(2), and rwlock(3THR) man pages.

Setting and Removing Record Locks

When locking a record, do not set the starting point and length of the lock segment to zero. The locking procedure is otherwise identical to file locking.

Contention for data is why you use record locking. Therefore, you should have a failure response for when you cannot obtain all the required locks:

- Wait a certain amount of time, then try again
- Abort the procedure, warn the user
- Let the process sleep until signaled that the lock has been freed
- Do some combination of the previous

This example shows a record being locked by using fcntl(2).

```
{
     struct flock lck;
       ...
     lck.l_type = F_WRLCK;    /* setting a write lock */
     lck.l_whence = 0;    /* offset l_start from beginning of file */
     lck.l_start = here;
     lck.l_len = sizeof(struct record);

     /* lock "this" with write lock */
     lck.l_start = this;
     if (fcntl(fd, F_SETLKW, &lck) < 0) {
         /* "this" lock failed. */
         return (-1);
 ...
}
```

The next example shows the lockf(3C) interface.

```
#include <unistd.h>

{
 ...
     /* lock "this" */
     (void) lseek(fd, this, SEEK_SET);
     if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
         /* Lock on "this" failed. Clear lock on "here". */
         (void) lseek(fd, here, 0);
         (void) lockf(fd, F_ULOCK, sizeof(struct record));
```

```
        return (-1);
}
```

You remove locks in the same way the locks were set. Only the lock type is different (F_ULOCK). An unlock cannot be blocked by another process and affects only locks placed by the calling process. The unlock affects only the segment of the file specified in the preceding locking call.

Getting Lock Information

You can determine which process is holding a lock. A lock is set, as in the previous examples, and F_GETLK is used in fcntl(2).

The next example finds and prints identifying data on all the locked segments of a file.

EXAMPLE 5–2 Printing Locked Segments of a File

```
struct flock lck;

     lck.l_whence = 0;
     lck.l_start = 0L;
     lck.l_len = 0L;
     do {
         lck.l_type = F_WRLCK;
         (void) fcntl(fd, F_GETLK, &lck);
         if (lck.l_type != F_UNLCK) {
             (void) printf("%d %d %c %8ld %8ld\n", lck.l_sysid, lck.l_pid,
            (lck.l_type == F_WRLCK) ? 'W' : 'R', lck.l_start, lck.l_len);
             /* If this lock goes to the end of the address space, no
              * need to look further, so break out. */
             if (lck.l_len == 0) {
             /* else, look for new lock after the one just found. */
                     lck.l_start += lck.l_len;
             }
         }
     } while (lck.l_type != F_UNLCK);
```

fcntl(2) with the F_GETLK command can sleep while waiting for a server to respond. The command can fail, returning ENOLCK, if either the client or the server have a resource shortage.

Use lockf(3C) with the F_TEST command to test if a process is holding a lock. This interface does not return information about the lock's location or ownership.

EXAMPLE 5–3 Testing a Process With lockf

```
(void) lseek(fd, 0, 0L);
 /* set the size of the test region to zero (0). to test until the
    end of the file address space. */
```

EXAMPLE 5–3 Testing a Process With `lockf` *(Continued)*

```
if (lockf(fd, (off_t)0, SEEK_SET) < 0) {
    switch (errno) {
        case EACCES:
        case EAGAIN:
            (void) printf("file is locked by another process\n");
            break;
        case EBADF:
            /* bad argument passed to lockf */
            perror("lockf");
            break;
        default:
            (void) printf("lockf: unexpected error <%d>\n", errno);
            break;
    }
```

Process Forking and Locks

When a process forks, the child receives a copy of the file descriptors that the parent opened. Locks are not inherited by the child because the locks are owned by a specific process. The parent and child share a common file pointer for each file. Both processes can try to set locks on the same location in the same file. This problem occurs with both `lockf`(3C) and `fcntl`(2). If a program holding a record lock forks, the child process should close the file. After closing the file, the child process should reopen the file to set a new, separate file pointer.

Deadlock Handling

The UNIX locking facilities provide deadlock detection and avoidance. Deadlocks can occur only when the system is ready to put a record–locking interface to sleep. A search is made to determine whether two processes are in a deadlock. If a potential deadlock is detected, the locking interface fails and sets `errno` to indicate deadlock. Processes setting locks that use `F_SETLK` do not cause a deadlock because these processes do not wait when the lock cannot be granted immediately.

Terminal I/O Functions

Terminal I/O interfaces deal with a general terminal interface for controlling asynchronous communications ports, as shown in the following table. For more information, see the `termios`(3C) and `termio`(7I) man pages.

TABLE 5–5 Terminal I/O Interfaces

Interface Name	Purpose
`tcgetattr`(3C), `tcsetattr`(3C)	Get and set terminal attributes
`tcsendbreak`(3C), `tcdrain`(3C), `tcflush`(3C), `tcflow`(3C)	Perform line control interfaces
`cfgetospeed`(3C), `cfgetispeed`(3C)`cfsetispeed`(3C), `cfsetospeed`(3C)	Get and set baud rate
`tcsetpgrp`(3C)	Get and set terminal foreground process group ID
`tcgetsid`(3C)	Get terminal session ID

The following example shows how the server dissociates from the controlling terminal of its invoker in the non-`DEBUG` mode of operation.

EXAMPLE 5–4 Dissociating From the Controlling Terminal

```
(void) close(0);
(void) close(1);
(void) close(2);
(void) open("/", O_RDONLY);
(void) dup2(0, 1);
(void) dup2(0, 2);
setsid();
```

This operation mode prevents the server from receiving signals from the process group of the controlling terminal. A server cannot send reports of errors to a terminal after the server has dissociated. The dissociated server must log errors with `syslog`(3C).

CHAPTER 6

Interprocess Communication

This chapter is for programmers who develop multiprocess applications.

SunOS 5.9 and compatible operating environments have a large variety of mechanisms for concurrent processes to exchange data and synchronize execution. All of these mechanisms, except mapped memory, are introduced in this chapter.

- Pipes (anonymous data queues) are described in "Pipes Between Processes" on page 107.
- Named pipes (data queues with file names.) "Named Pipes" on page 109 covers named pipes.
- System V message queues, semaphores, and shared memory are described in "System V IPC" on page 112.
- POSIX message queues, semaphores, and shared memory are described in "POSIX Interprocess Communication" on page 110.
- "Sockets Overview" on page 109 describes interprocess communication using sockets.
- Mapped memory and files are described in "Memory Management Interfaces" on page 15.

Pipes Between Processes

A pipe between two processes is a pair of files that is created in a parent process. The pipe connects the resulting processes when the parent process forks. A pipe has no existence in any file name space, so it is said to be anonymous. A pipe usually connects only two processes, although any number of child processes can be connected to each other and their related parent by a single pipe.

A pipe is created in the process that becomes the parent by a call to pipe(2). The call returns two file descriptors in the array passed to it. After forking, both processes read from p[0] and write to p[1]. The processes actually read from and write to a circular buffer that is managed for them.

Because calling fork(2) duplicates the per-process open file table, each process has two readers and two writers. Closing the extra readers and writers enables the proper functioning of the pipe. For example, no end-of-file indication would ever be returned if the other end of a reader is left open for writing by the same process. The following code shows pipe creation, a fork, and clearing the duplicate pipe ends.

```
#include <stdio.h>
#include <unistd.h>
 ...
     int p[2];
 ...
     if (pipe(p) == -1) exit(1);
     switch( fork() )
     {
         case 0:                        /* in child */
             close( p[0] );
             dup2( p[1], 1);
             close P[1] );
             exec( ... );
             exit(1);
         default:                        /* in parent */
             close( p[1] );
             dup2( P[0], 0 );
             close( p[0] );
             break;
     }
     ...
```

The following table shows the results of reads from a pipe and writes to a pipe, under certain conditions.

TABLE 6–1 Read/Write Results in a Pipe

Attempt	Conditions	Result
read	Empty pipe, writer attached	Read blocked
write	Full pipe, reader attached	Write blocked
read	Empty pipe, no writer attached	EOF returned
write	No reader	SIGPIPE

Blocking can be prevented by calling fcntl(2) on the descriptor to set FNDELAY. This causes an error return (-1) from the I/O call with errno set to EWOULDBLOCK.

Named Pipes

Named pipes function much like pipes, but are created as named entities in a file system. This enables the pipe to be opened by all processes with no requirement that they be related by forking. A named pipe is created by a call to `mknod`(2). Any process with appropriate permission can then read or write to a named pipe.

In the `open`(2) call, the process opening the pipe blocks until another process also opens the pipe.

To open a named pipe without blocking, the `open`(2) call joins the `O_NDELAY` mask (found in `sys/fcntl.h`) with the selected file mode mask using the Boolean `or` operation on the call to `open`(2). If no other process is connected to the pipe when `open`(2) is called, -1 is returned with `errno` set to `EWOULDBLOCK`.

Sockets Overview

Sockets provide point-to-point, two-way communication between two processes. Sockets are a basic component of interprocess and intersystem communication. A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.

Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. Twenty three socket domains are identified (see `sys/socket.h`), of which only the UNIX and Internet domains are normally used in Solaris 9 and compatible operating environments.

You can use sockets to communicate between processes on a single system, like other forms of IPC. The UNIX domain (`AF_UNIX`) provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. UNIX domain sockets are further described in "UNIX Domain Sockets" in *Programming Interfaces Guide*. Sockets can also be used to communicate between processes on different systems. The socket address space between connected systems is called the Internet domain (`AF_INET`). Internet domain communication uses the TCP/IP internet protocol suite. Internet domain sockets are described in "Socket Interfaces" in *Programming Interfaces Guide*.

POSIX Interprocess Communication

POSIX interprocess communication (IPC) is a variation of System V interprocess communication. It was introduced in the Solaris 7 release. Like System V objects, POSIX IPC objects have read and write, but not execute, permissions for the owner, the owner's group, and for others. There is no way for the owner of a POSIX IPC object to assign a different owner. POSIX IPC includes the following features:

- Messages allow processes to send formatted data streams to arbitrary processes.
- Semaphores allow processes to synchronize execution.
- Shared memory allows processes to share parts of their virtual address space.

Unlike the System V IPC interfaces, the POSIX IPC interfaces are all multithread safe.

POSIX Messages

The POSIX message queue interfaces are listed in the following table.

TABLE 6–2 POSIX Message Queue Interfaces

Interface Name	Purpose
`mq_open`(3RT)	Connects to, and optionally creates, a named message queue
`mq_close`(3RT)	Ends the connection to an open message queue
`mq_unlink`(3RT)	Ends the connection to an open message queue and causes the queue to be removed when the last process closes it
`mq_send`(3RT)	Places a message in the queue
`mq_receive`(3RT)	Receives (removes) the oldest, highest priority message from the queue
`mq_notify`(3RT)	Notifies a process or thread that a message is available in the queue
`mq_setattr`(3RT), `mq_getattr`(3RT)	Set or get message queue attributes

POSIX Semaphores

POSIX semaphores are much lighter weight than are System V semaphores. A POSIX semaphore structure defines a single semaphore, not an array of up to 25 semaphores.

The POSIX semaphore interfaces are shown below.

TABLE 6–3 POSIX Semaphore Interfaces

`sem_open`(3RT)	Connects to, and optionally creates, a named semaphore
`sem_init`(3RT)	Initializes a semaphore structure (internal to the calling program, so not a named semaphore)
`sem_close`(3RT)	Ends the connection to an open semaphore
`sem_unlink`(3RT)	Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it
`sem_destroy`(3RT)	Initializes a semaphore structure (internal to the calling program, so not a named semaphore)
`sem_getvalue`(3RT)	Copies the value of the semaphore into the specified integer
`sem_wait`(3RT), `sem_trywait`(3RT)	Blocks while the semaphore is held by other processes or returns an error if the semaphore is held by another process
`sem_post`(3RT)	Increments the count of the semaphore

POSIX Shared Memory

POSIX shared memory is actually a variation of mapped memory (see "Creating and Using Mappings" on page 15). The major differences are:

- You use `shm_open`(3RT) to open the shared memory object instead of calling `open`(2).
- You use `shm_unlink`(3RT) to close and delete the object instead of calling `close`(2) which does not remove the object.

The options in `shm_open`(3RT) are substantially fewer than the number of options provided in `open`(2).

System V IPC

SunOS 5.9 and compatible operating environments also provide the System V inter process communication (IPC) package. System V IPC has effectively been replaced by POSIX IPC, but is maintained to support older applications.

See the `ipcrm`(1), `ipcs`(1), `Intro`(2), `msgctl`(2), `msgget`(2), `msgrcv`(2), `msgsnd`(2), `semget`(2), `semctl`(2), `semop`(2), `shmget`(2), `shmctl`(2), `shmop`(2), and `ftok`(3C) man pages for more information about System V IPC.

Permissions for Messages, Semaphores, and Shared Memory

Messages, semaphores, and shared memory have read and write permissions, but no execute permission, for the owner, group, and others, which is similar to ordinary files. Like files, the creating process identifies the default owner. Unlike files, the creating process can assign ownership of the facility to another user or revoke an ownership assignment.

IPC Interfaces, Key Arguments, and Creation Flags

Processes requesting access to an IPC facility must be able to identify the facility. To identify the facility to which the process requests access, interfaces that initialize or provide access to an IPC facility use a `key_t` *key* argument. The *key* is an arbitrary value or one that can be derived from a common seed at runtime. One way to derive such a key is by using `ftok`(3C), which converts a file name to a key value that is unique within the system.

Interfaces that initialize or get access to messages, semaphores, or shared memory return an ID number of type `int`. IPC Interfaces that perform read, write, and control operations use this ID.

If the key argument is specified as `IPC_PRIVATE`, the call initializes a new instance of an IPC facility that is private to the creating process.

When the `IPC_CREAT` flag is supplied in the flags argument appropriate to the call, the interface tries to create the facility if it does not exist already.

When called with both the`IPC_CREAT` and `IPC_EXCL` flags, the interface fails if the facility already exists. This behavior can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with `IPC_EXCL` in effect, only the first attempt succeeds.

If neither of these flags is given and the facility already exists, the interfaces return the ID of the facility to get access. If `IPC_CREAT` is omitted and the facility is not already initialized, the calls fail.

Using logical (bitwise) `OR`, `IPC_CREAT` and `IPC_EXCL` are combined with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist:

```
msqid = msgget(ftok("/tmp", 'A'), (IPC_CREAT | IPC_EXCL | 0400));
```

The first argument evaluates to a key (`'A'`) based on the string (`"/tmp"`). The second argument evaluates to the combined permissions and control flags.

System V Messages

Before a process can send or receive a message, you must initialize the queue through `msgget`(2). The owner or creator of a queue can change its ownership or permissions using `msgctl`(2). Any process with permission can use `msgctl`(2) for control operations.

IPC messaging enables processes to send and receive messages and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length.

Messages can be assigned a specific type. A server process can thus direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Operations to send and receive messages are performed by `msgsnd`(2) and `msgrcv`(2), respectively. When a message is sent, its text is copied to the message queue. `msgsnd`(2) and `msgrcv`(2) can be performed as either blocking or non-blocking operations. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Initializing a Message Queue

`msgget`(2) initializes a new message queue. It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The `MSGMNI` kernel configuration option determines the maximum number of unique message queues that the kernel supports. `msgget`(2) fails when this limit is exceeded.

The following code illustrates msgget(2).

```
#include <sys/ipc.h>
 #include <sys/msg.h>

 ...
     key_t    key;        /* key to be passed to msgget() */
     int    msgflg,    /* msgflg to be passed to msgget() */
            msqid;    /* return value from msgget() */
     ...
     key = ...
     msgflg = ...
     if ((msqid = msgget(key, msgflg)) == -1)
     {
         perror("msgget: msgget failed");
         exit(1);
     } else
         (void) fprintf(stderr, "msgget succeeded");
     ...
```

Controlling Message Queues

msgctl(2) alters the permissions and other characteristics of a message queue. The msqid argument must be the ID of an existing message queue. The cmd argument is one of the following:

IPC_STAT — Place information about the status of the queue in the data structure pointed to by buf. The process must have read permission for this call to succeed.

IPC_SET — Set the owner's user and group ID, the permissions, and the size (in number of bytes) of the message queue. A process must have the effective user ID of the owner, creator, or superuser for this call to succeed.

IPC_RMID — Remove the message queue specified by the msqid argument.

The following code illustrates msgctl(2) with all its various flags.

```
#include            <sys/types.h>
 #include            <sys/ipc.h>
 #include            <sys/msg.h>

     ...
     if (msgctl(msqid, IPC_STAT, &buf) == -1)  {
         perror("msgctl: msgctl failed");
         exit(1);
     }
     ...
     if (msgctl(msqid, IPC_SET, &buf) == -1) {
         perror("msgctl: msgctl failed");
         exit(1);
     }
```

```
...
```

Sending and Receiving Messages

msgsnd(2) and msgrcv(2) send and receive messages, respectively. The msqid argument must be the ID of an existing message queue. The msgp argument is a pointer to a structure that contains the type of the message and its text. The msgsz argument specifies the length of the message in bytes. The msgflg argument passes various control flags.

The following code illustrates msgsnd(2) and msgrcv(2).

```
#include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/msg.h>

 ...
     int                msgflg;    /* message flags for the operation */
     struct msgbuf     *msgp;    /* pointer to the message buffer */
     size_t            msgsz;    /* message size */
        size_t            maxmsgsize;
     long               msgtyp;    /* desired message type */
     int                msqid        /* message queue ID to be used */
     ...
     msgp = malloc(sizeof(struct msgbuf) - sizeof (msgp->mtext)
                            + maxmsgsz);
     if (msgp == NULL) {
         (void) fprintf(stderr, "msgop: %s %ld byte messages.\n",
                 "could not allocate message buffer for", maxmsgsz);
         exit(1);
         ...
         msgsz = ...
         msgflg = ...
         if (msgsnd(msqid, msgp, msgsz, msgflg) == -1)
             perror("msgop: msgsnd failed");
         ...
         msgsz = ...
         msgtyp = first_on_queue;
         msgflg = ...
         if (rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg) == -1)
             perror("msgop: msgrcv failed");
         ...
```

System V Semaphores

Semaphores enable processes to query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments. Semaphores can be operated on as individual units or as elements in a set.

Because System V IPC semaphores can be in a large array, they are extremely heavy weight. Much lighter-weight semaphores are available in the threads library (see the `semaphore`(3THR) man page). Also, POSIX semaphores are the most current implementation of System V semaphores (see "POSIX Semaphores" on page 110). Threads library semaphores must be used with mapped memory (see "Memory Management Interfaces" on page 15).

A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements. The semaphore set must be initialized using `semget`(2). The semaphore creator can change its ownership or permissions using `semctl`(2). Any process with permission can use `semctl`(2) to do control operations.

Semaphore operations are performed by `semop`(2). This interface takes a pointer to an array of semaphore operation structures. Each structure in the array contains data about an operation to perform on a semaphore. Any process with read permission can test whether a semaphore has a zero value. Operations to increment or decrement a semaphore require write permission.

When an operation fails, none of the semaphores are altered. The process blocks unless the `IPC_NOWAIT` flag is set, and remains blocked until:

- The semaphore operations can all finish, so the call succeeds.
- The process receives a signal.
- The semaphore set is removed.

Only one process at a time can update a semaphore. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a `semop`(2) call, no updates are done until all operations on the array can finish successfully.

If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this occurrence, the `SEM_UNDO` control flag makes `semop`(2) allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state.

If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with `SEM_UNDO` in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state.

When performing a semaphore operation with SEM_UNDO in effect, you must also have SEM_UNDO in effect for the call that performs the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This ensures that, unless the process is aborted, the values applied to the undo structure are canceled to zero. When the undo structure reaches zero, it is removed.

Using SEM_UNDO inconsistently can lead to memory leaks because allocated undo structures might not be freed until the system is rebooted.

Initializing a Semaphore Set

semget(2) initializes or gains access to a semaphore. When the call succeeds, it returns the semaphore ID (semid). The key argument is a value associated with the semaphore ID. The nsems argument specifies the number of elements in a semaphore array. The call fails when nsems is greater than the number of elements in an existing array. When the correct count is not known, supplying 0 for this argument ensures that it will succeed. The semflg argument specifies the initial access permissions and creation control flags.

The SEMMNI system configuration option determines the maximum number of semaphore arrays allowed. The SEMMNS option determines the maximum possible number of individual semaphores across all semaphore sets. Because of fragmentation between semaphore sets, allocating all available semaphores might not be possible.

The following code illustrates semget(2).

```
#include            <sys/types.h>
#include            <sys/ipc.h>
#include            <sys/sem.h>
...
     key_t        key;            /* key to pass to semget() */
     int        semflg;            /* semflg to pass to semget() */
     int        nsems;            /* nsems to pass to semget() */
     int        semid;            /* return value from semget() */
        ...
     key = ...
     nsems = ...
     semflg = ...
     ...
     if ((semid = semget(key, nsems, semflg)) == -1) {
         perror("semget: semget failed");
         exit(1);
     } else
         exit(0);
 ...
```

Controlling Semaphores

semctl(2) changes permissions and other characteristics of a semaphore set. It must be called with a valid semaphore ID. The semnum value selects a semaphore within an array by its index. The *cmd* argument is one of the following control flags.

GETVAL	Return the value of a single semaphore.
SETVAL	Set the value of a single semaphore. In this case, arg is taken as arg.val, an int.
GETPID	Return the PID of the process that performed the last operation on the semaphore or array.
GETNCNT	Return the number of processes waiting for the value of a semaphore to increase.
GETZCNT	Return the number of processes waiting for the value of a particular semaphore to reach zero.
GETALL	Return the values for all semaphores in a set. In this case, arg is taken as arg.array, a pointer to an array of unsigned short values.
SETALL	Set values for all semaphores in a set. In this case, arg is taken as arg.array, a pointer to an array of unsigned short values.
IPC_STAT	Return the status information from the control structure for the semaphore set and place it in the data structure pointed to by arg.buf, a pointer to a buffer of type semid_ds.
IPC_SET	Set the effective user and group identification and permissions. In this case, arg is taken as arg.buf.
IPC_RMID	Remove the specified semaphore set.

A process must have an effective user identification of owner, creator, or superuser to perform an IPC_SET or IPC_RMID command. Read and write permission is required, as for the other control commands.

The following code illustrates semctl(2).

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
...
     register int                i;
 ...
     i = semctl(semid, semnum, cmd, arg);
     if (i == -1) {
         perror("semctl: semctl failed");
         exit(1);
...
```

Semaphore Operations

semop(2) performs operations on a semaphore set. The semid argument is the semaphore ID returned by a previous semget(2) call. The sops argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

- The semaphore number
- The operation to be performed
- Control flags, if any

The sembuf structure specifies a semaphore operation, as defined in sys/sem.h. The nsops argument specifies the length of the array, the maximum size of which is determined by the *SEMOPM* configuration option. This option determines the maximum number of operations allowed by a single semop(2) call, and is set to 10 by default.

The operation to be performed is determined as follows:

- Positive integer increments the semaphore value by that amount.
- Negative integer decrements the semaphore value by that amount. An attempt to set a semaphore to a value less than zero fails or blocks, depending on whether IPC_NOWAIT is in effect.
- Value of zero means to wait for the semaphore value to reach zero.

The two control flags that can be used with semop(2) are IPC_NOWAIT and SEM_UNDO.

IPC_NOWAIT	Can be set for any operations in the array. Makes the interface return without changing any semaphore value if it cannot perform any of the operations for which IPC_NOWAIT is set. The interface fails if it tries to decrement a semaphore more than its current value, or tests a nonzero semaphore to be equal to zero.
SEM_UNDO	Allows individual operations in the array to be undone when the process exits.

The following code illustrates semop(2).

```
#include                <sys/types.h>
#include                <sys/ipc.h>
#include                <sys/sem.h>
...
     int                i;            /* work area */
     int                nsops;    /* number of operations to do */
     int                semid;    /* semid of semaphore set */
     struct sembuf    *sops;    /* ptr to operations to perform */
     ...
     if ((i = semop(semid, sops, nsops)) == -1) {
         perror("semop: semop failed");
```

```
    } else
        (void) fprintf(stderr, "semop: returned %d\n", i);
...
```

System V Shared Memory

In the SunOS 5.9 operating system, the most efficient way to implement shared memory applications is to rely on `mmap`(2) and on the system's native virtual memory facility. See Chapter 1 for more information.

The SunOS 5.9 platform also supports System V shared memory, which is a less efficient way to enable the attachment of a segment of physical memory to the virtual address spaces of multiple processes. When write access is allowed for more than one process, an outside protocol or mechanism, such as a semaphore, can be used to prevent inconsistencies and collisions.

A process creates a shared memory segment using `shmget`(2). This call is also used to get the ID of an existing shared segment. The creating process sets the permissions and the size in bytes for the segment.

The original owner of a shared memory segment can assign ownership to another user with `shmctl`(2). The owner can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using `shmctl`(2).

Once created, you can attach a shared segment to a process address space using `shmat`(2). You can detach it using `shmdt`(2). The attaching process must have the appropriate permissions for `shmat`(2). Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process.

A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the `shmid`. The structure definition for the shared memory segment control structure can be found in `sys/shm.h`.

Accessing a Shared Memory Segment

`shmget`(2) is used to obtain access to a shared memory segment. When the call succeeds, it returns the shared memory segment ID (*shmid*). The following code illustrates `shmget`(2).

```
#include  <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
...
    key_t        key;            /* key to be passed to shmget() */
```

```
    int        shmflg;        /* shmflg to be passed to shmget() */
    int        shmid;        /* return value from shmget() */
    size_t    size;            /* size to be passed to shmget() */
    ...
    key = ...
    size = ...
    shmflg) = ...
    if ((shmid = shmget (key, size, shmflg)) == -1) {
        perror("shmget: shmget failed");
        exit(1);
    } else {
        (void) fprintf(stderr,
                    "shmget: shmget returned %d\n", shmid);
        exit(0);
    }
...
```

Controlling a Shared Memory Segment

`shmctl`(2) is used to alter the permissions and other characteristics of a shared memory segment. The `cmd` argument is one of following control commands.

`SHM_LOCK`	Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.
`SHM_UNLOCK`	Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.
`IPC_STAT`	Return the status information contained in the control structure and place it in the buffer pointed to by `buf`. The process must have read permission on the segment to perform this command.
`IPC_SET`	Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.
`IPC_RMID`	Remove the shared memory segment. The process must have an effective ID of owner, creator, or superuser to perform this command.

The following code illustrates `shmctl`(2).

```
#include           <sys/types.h>
#include           <sys/ipc.h>
#include           <sys/shm.h>
...
 int        cmd;        /* command code for shmctl() */
 int        shmid;    /* segment ID */
 struct shmid_ds    shmid_ds; /* shared memory data structure to
                                    hold results */
    ...
    shmid = ...
```

```
cmd = ...
if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
    perror("shmctl: shmctl failed");
    exit(1);
...
```

Attaching and Detaching a Shared Memory Segment

`shmat()` and `shmdt()` are used to attach and detach shared memory segments (see the `shmop`(2) man page). `shmat`(2) returns a pointer to the head of the shared segment. `shmdt`(2) detaches the shared memory segment located at the address indicated by *shmaddr*. The following code illustrates calls to `shmat`(2) and `shmdt`(2)

```
#include            <sys/types.h>
#include            <sys/ipc.h>
#include            <sys/shm.h>

static struct state {    /* Internal record of attached segments. */
     int        shmid;        /* shmid of attached segment */
     char        *shmaddr;    /* attach point */
     int        shmflg;        /* flags used on attach */
     } ap[MAXnap];            /* State of current attached segments. */
     int        nap;            /* Number of currently attached segments. */
 ...
     char                *addr;            /* address work variable */
     register int                i;            /* work area */
     register struct state  *p;  /* ptr to current state entry */
 ...
     p = &ap[nap++];
     p->shmid = ...
     p->shmaddr = ...
     p->shmflg = ...
     p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
     if(p->shmaddr == (char *)-1) {
         perror("shmat failed");
         nap--;
     } else
          (void) fprintf(stderr, "shmop: shmat returned %p\n",
                     p->shmaddr);
     ...
     i = shmdt(addr);
     if(i == -1) {
         perror("shmdt failed");
     } else {
         (void) fprintf(stderr, "shmop: shmdt returned %d\n", i);
         for (p = ap, i = nap; i--; p++) {
             if (p->shmaddr == addr) *p = ap[--nap];
         }
     }
     ...
```

CHAPTER 7

Socket Interfaces

This chapter presents the socket interface. Sample programs are included to illustrate key points. The following topics are discussed in this chapter:

- "SunOS 4 Binary Compatibility" on page 123 discusses binary compatibility with the SunOS™ 4 environment.
- Socket creation, connection, and closure are discussed in "Socket Basics" on page 127.
- Client-Server architecture is discussed in "Client-Server Programs" on page 146.
- Advanced topics such as multicast and asynchronous sockets are discussed in "Advanced Socket Topics" on page 150.

Note – The interface that is described in this chapter is multithread safe. You can call applications that contain socket interface calls freely in a multithreaded application. Note, however, that the degree of concurrency that is available to applications is not specified.

SunOS 4 Binary Compatibility

Two major changes from the SunOS 4 environment hold true for SunOS 5.9 releases. The binary compatibility package enables dynamically linked socket applications that are based on SunOS 4 to run on SunOS 5.9.

- You must explicitly specify the socket library (`-lsocket` or `libsocket`) on the compilation line.
- You might also need to link with `libnsl` by using `-lsocket -lnsl`, not `-lnsl -lsocket`.
- You must recompile all SunOS™ 4 socket-based applications with the socket library to run in a SunOS 5.9 environment.

Overview of Sockets

Sockets have been an integral part of SunOS releases since 1981. A socket is an endpoint of communication to which a name can be bound. A socket has a *type* and an associated process. Sockets were designed to implement the client-server model for interprocess communication where:

- The interface to network protocols needs to accommodate multiple communication protocols, such as TCP/IP, Xerox internet protocols (XNS), and the UNIX family.
- The interface to network protocols needs to accommodate server code that waits for connections and client code that initiates connections.
- Operations differ depending on whether communication is connection-oriented or connectionless.
- Application programs might want to specify the destination address of the datagrams that are being delivered instead of binding the address with the `open`(2) call.

Sockets make network protocols available while behaving like UNIX files. Applications create sockets as sockets are needed. Sockets work with the `close`(2), `read`(2), `write`(2), `ioctl`(2), and `fcntl`(2) interfaces. The operating system differentiates between the file descriptors for files and the file descriptors for sockets.

Socket Libraries

The socket interface routines are in a library that must be linked with the application. The library `libsocket.so` is contained in `/usr/lib` with the rest of the system service libraries. Use `libsocket.so` for dynamic linking.

Socket Types

Socket types define the communication properties that are visible to a user. The Internet family sockets provide access to the TCP/IP transport protocols. The Internet family is identified by the value `AF_INET6`, for sockets that can communicate over both IPv6 and IPv4. The value `AF_INET` is also supported for source compatibility with old applications and for "raw" access to IPv4.

The SunOS environment supports three types of sockets:

- *Stream* sockets enable processes to communicate using TCP. A stream socket provides a bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. After the connection has been established, data can be read from and written to these sockets as a byte stream. The socket type is

SOCK_STREAM.

- *Datagram* sockets enable processes to use UDP to communicate. A datagram socket supports a bidirectional flow of messages. A process on a datagram socket can receive messages in a different order from the sending sequence. A process on a datagram socket can receive duplicate messages. Record boundaries in the data are preserved. The socket type is SOCK_DGRAM.
- *Raw* sockets provide access to ICMP. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not for most applications. Raw sockets are provided to support the development of new communication protocols, or for access to more esoteric facilities of an existing protocol. Only superuser processes can use raw sockets. The socket type is SOCK_RAW.

See "Selecting Specific Protocols" on page 156 for further information.

Interface Sets

The SunOS 5.9 platform provides two sets of socket interfaces. The BSD socket interfaces are provided and, since SunOS™ version 5.7, the XNS 5 (Unix98) socket interfaces are also provided. The XNS 5 interfaces differ slightly from the BSD interfaces.

The XNS 5 socket interfaces are documented in the following man pages:

- accept(3XNET)
- bind(3XNET)
- connect(3XNET)
- endhostent(3XNET)
- endnetent(3XNET)
- endprotoent(3XNET)
- endservent(3XNET)
- gethostbyaddr(3XNET)
- gethostbyname(3XNET)
- gethostent(3XNET)
- gethostname(3XNET)
- getnetbyaddr(3XNET)
- getnetbyname(3XNET)
- getnetent(3XNET)
- getpeername(3XNET)
- getprotobyname(3XNET)
- getprotobynumber(3XNET)
- getprotoent(3XNET)
- getservbyname(3XNET)
- getservbyport(3XNET)
- getservent(3XNET)
- getsockname(3XNET)

- `getsockopt`(3XNET)
- `htonl`(3XNET)
- `htons`(3XNET)
- `inet_addr`(3XNET)
- `inet_lnaof`(3XNET)
- `inet_makeaddr`(3XNET)
- `inet_netof`(3XNET)
- `inet_network`(3XNET)
- `inet_ntoa`(3XNET)
- `listen`(3XNET)
- `ntohl`(3XNET)
- `ntohs`(3XNET)
- `recv`(3XNET)
- `recvfrom`(3XNET)
- `recvmsg`(3XNET)
- `send`(3XNET)
- `sendmsg`(3XNET)
- `sendto`(3XNET)
- `sethostent`(3XNET)
- `setnetent`(3XNET)
- `setprotoent`(3XNET)
- `setservent`(3XNET)
- `setsockopt`(3XNET)
- `shutdown`(3XNET)
- `socket`(3XNET)
- `socketpair`(3XNET)

The traditional BSD Socket behavior is documented in the corresponding 3N man pages. In addition, the following new interfaces have been added to section 3N:

- `freeaddrinfo`(3SOCKET)
- `freehostent`(3SOCKET)
- `getaddrinfo`(3SOCKET)
- `getipnodebyaddr`(3SOCKET)
- `getipnodebyname`(3SOCKET)
- `getnameinfo`(3SOCKET)
- `inet_ntop`(3SOCKET)
- `inet_pton`(3SOCKET)

See the `standards`(5) man page for information on building applications that use the XNS 5 (Unix98) socket interface.

Socket Basics

This section describes the use of the basic socket interfaces.

Socket Creation

The `socket`(3SOCKET) call creates a socket in the specified family and of the specified type.

```
s = socket(family, type, protocol);
```

If the protocol is unspecified, the system selects a protocol that supports the requested socket type. The socket handle is returned. The socket handle is a file descriptor.

The *family* is specified by one of the constants that are defined in `sys/socket.h`. Constants that are named `AF_`*suite* specify the address format to use in interpreting names:

`AF_APPLETALK`	Apple Computer Inc. Appletalk network
`AF_INET6`	Internet family for IPv6 and IPv4
`AF_INET`	Internet family for IPv4 only
`AF_PUP`	Xerox Corporation PUP internet
`AF_UNIX`	UNIX file system

Socket types are defined in `sys/socket.h`. These types, `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`, are supported by `AF_INET6`, `AF_INET`, and `AF_UNIX`. The following example creates a stream socket in the Internet family:

```
s = socket(AF_INET6, SOCK_STREAM, 0);
```

This call results in a stream socket. The TCP protocol provides the underlying communication. Set the *protocol* argument to `0`, the default, in most situations. You can specify a protocol other than the default, as described in "Advanced Socket Topics" on page 150.

Binding Local Names

A socket is created with no name. A remote process has no way to refer to a socket until an address is bound to the socket. Processes that communicate are connected through addresses. In the Internet family, a connection is composed of local and remote addresses and local and remote ports. Duplicate ordered sets, such as: `protocol`, `local address`, `local port`, `foreign address`, `foreign port` cannot exist. In most families, connections must be unique.

The `bind`(3SOCKET) interface enables a process to specify the local address of the socket. This interface forms the `local address`, `local port` set. `connect`(3SOCKET) and `accept`(3SOCKET) complete a socket's association by fixing the remote half of the address tuple. The `bind`(3SOCKET) call is used as follows:

`bind (`*s, name, namelen*`);`

The socket handle is *s*. The bound name is a byte string that is interpreted by the supporting protocols. Internet family names contain an Internet address and port number.

This example demonstrates binding an Internet address.

```
#include <sys/types.h>
#include <netinet/in.h>
...
        struct sockaddr_in6 sin6;
...
        s = socket(AF_INET6, SOCK_STREAM, 0);
        bzero (&sin6, sizeof (sin6));
        sin6.sin6_family = AF_INET6;
        sin6.sin6_addr.s6_addr = in6addr_arg;
        sin6.sin6_port = htons(MYPORT);
        bind(s, (struct sockaddr *) &sin6, sizeof sin6);
```

The content of the address `sin6` is described in "Address Binding" on page 156, where Internet address bindings are discussed.

Connection Establishment

Connection establishment is usually asymmetric, with one process acting as the client and the other as the server. The server binds a socket to a well-known address associated with the service and blocks on its socket for a connect request. An unrelated process can then connect to the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the `connect`(3SOCKET) call initiates a connection. In the Internet family, this connection might appear as:

```
struct sockaddr_in6 server;
 ...
connect(s, (struct sockaddr *)&server, sizeof server);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. For more information, see "Address Binding" on page 156. This automatic selection is the usual way to bind local addresses to a socket on the client side.

To receive a client's connection, a server must perform two steps after binding its socket. The first step is to indicate how many connection requests can be queued. The second step is to accept a connection.

```
struct sockaddr_in6 from;
 ...
listen(s, 5);                /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

The socket handle *s* is the socket bound to the address to which the connection request is sent. The second parameter of `listen`(3SOCKET) specifies the maximum number of outstanding connections that might be queued. The `from` structure is filled with the address of the client. A `NULL` pointer might be passed. *fromlen* is the length of the structure. In the UNIX family, `from` is declared a `struct sockaddr_un`.

The `accept`(3SOCKET) routine normally blocks processes. `accept`(3SOCKET) returns a new socket descriptor that is connected to the requesting client. The value of *fromlen* is changed to the actual size of the address.

A server cannot indicate that the server accepts connections from only specific addresses. The server can check the `from` address returned by `accept`(3SOCKET) and close a connection with an unacceptable client. A server can accept connections on more than one socket, or avoid blocking on the `accept`(3SOCKET) call. These techniques are presented in "Advanced Socket Topics" on page 150.

Connection Errors

An error is returned if the connection is unsuccessful, but an address bound by the system remains. If the connection is successful, the socket is associated with the server and data transfer can begin.

The following table lists some of the more common errors returned when a connection attempt fails.

TABLE 7–1 Socket Connection Errors

Socket Errors	Error Description
`ENOBUFS`	Lack of memory available to support the call.
`EPROTONOSUPPORT`	Request for an unknown protocol.

TABLE 7–1 Socket Connection Errors *(Continued)*

Socket Errors	Error Description
EPROTOTYPE	Request for an unsupported type of socket.
ETIMEDOUT	No connection established in specified time. This error happens when the destination host is down or when problems in the network cause in lost transmissions.
ECONNREFUSED	The host refused service. This error happens when a server process is not present at the requested address.
ENETDOWN or EHOSTDOWN	These errors are caused by status information delivered by the underlying communication interface.
ENETUNREACH or EHOSTUNREACH	These operational errors can occur because no route to the network or host exists. These errors can also occur because of status information returned by intermediate gateways or switching nodes. The status information that is returned is not always sufficient to distinguish between a network that is down and a host that is down.

Data Transfer

This section describes the interfaces to send and receive data. You can send or receive a message with the normal `read`(2) and `write`(2) interfaces:

```
write(s, buf, sizeof buf);
read(s,  buf, sizeof buf);
```

You can also use `send`(3SOCKET) and `recv`(3SOCKET):

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

`send`(3SOCKET) and `recv`(3SOCKET) are very similar to `read`(2) and `write`(2), but the `flags` argument is important. The `flags` argument, which is defined in `sys/socket.h`, can be specified as a nonzero value if one or more of the following is required:

MSG_OOB	Send and receive out-of-band data
MSG_PEEK	Look at data without reading
MSG_DONTROUTE	Send data without routing packets

Out-of-band data is specific to stream sockets. When `MSG_PEEK` is specified with a `recv`(3SOCKET) call, any data present is returned to the user, but treated as still unread. The next `read`(2) or `recv`(3SOCKET) call on the socket returns the same data. The option to send data without routing packets applied to the outgoing packets is currently used only by the routing table management process.

Closing Sockets

A SOCK_STREAM socket can be discarded by a close(2) interface call. If data is queued to a socket that promises reliable delivery after a close(2), the protocol continues to try to transfer the data. The data is discarded if it remains undelivered after an arbitrary period.

A shutdown(3SOCKET) closes SOCK_STREAM sockets gracefully. Both processes can acknowledge that they are no longer sending. This call has the form:

```
shutdown(s, how);
```

where how is defined as

0 Disallows further data reception

1 Disallows further data transmission

2 Disallows further transmission and further reception

Connecting Stream Sockets

The following two examples illustrate initiating and accepting an Internet family stream connection.

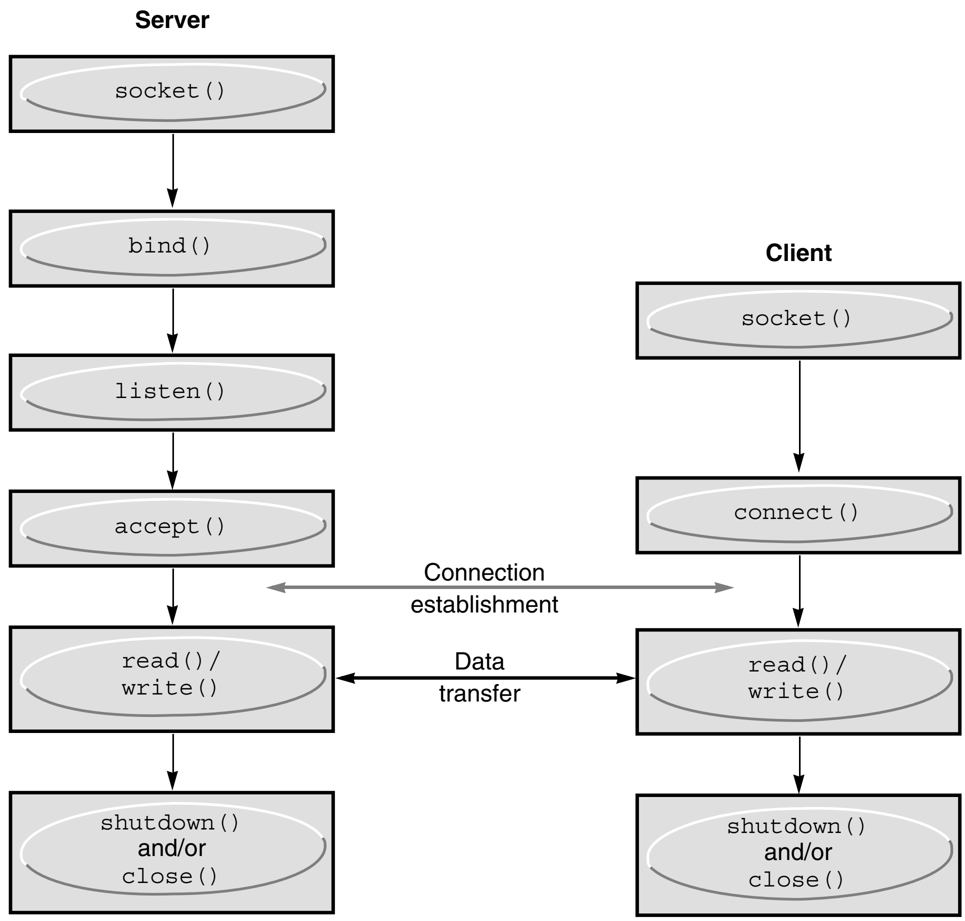


FIGURE 7–1 Connection-Oriented Communication Using Stream Sockets

The following example program is a server. The server creates a socket and binds a name to the socket, then displays the port number. The program calls `listen`(3SOCKET) to mark the socket as ready to accept connection requests and to initialize a queue for the requests. The rest of the program is an infinite loop. Each pass of the loop accepts a new connection and removes it from the queue, creating a new socket. The server reads and displays the messages from the socket and closes the socket. The use of `in6addr_any` is explained in "Address Binding" on page 156.

EXAMPLE 7–1 Accepting an Internet Stream Connection (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include >stdio.h>
```

```
#define TRUE 1
/*
 * This program creates a socket and then begins an infinite loop.
 * Each time through the loop it accepts a connection and prints
 * data from it. When the connection breaks, or the client closes
 * the connection, the program accepts a new connection.
*/
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    /* Create socket. */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
      perror("opening stream socket");
      exit(1);
   }
   /* Bind socket using wildcards.*/
   bzero (&server, sizeof(server));
/* bzero (&sin6, sizeof (sin6)); */
   server.sin6_family = AF_INET6;
   server.sin6_addr = in6addr_any;
   server.sin6_port = 0;
   if (bind(sock, (struct sockaddr *) &server, sizeof server)
         == -1) {
      perror("binding stream socket");
      exit(1);
   }
   /* Find out assigned port number and print it out. */
   length = sizeof server;
   if (getsockname(sock,(struct sockaddr *) &server, &length)
         == -1) {
      perror("getting socket name");
      exit(1);
   }
   printf("Socket port #%d\n", ntohs(server.sin6_port));
   /* Start accepting connections. */
   listen(sock, 5);
   do {
      msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
      if (msgsock == -1)
         perror("accept");
      else do {
         memset(buf, 0, sizeof buf);
         if ((rval = read(msgsock,buf, 1024)) == -1)
            perror("reading stream message");
         if (rval == 0)
            printf("Ending connection\n");
         else
            /* assumes the data is printable */
```

EXAMPLE 7–1 Accepting an Internet Stream Connection (Server) *(Continued)*

```
            printf("-->%s\n", buf);
      } while (rval > 0);
      close(msgsock);
   } while(TRUE);
   /*
    * Since this program has an infinite loop, the socket "sock" is
    * never explicitly closed. However, all sockets are closed
    * automatically when a process is killed or terminates normally.
    */
   exit(0);
}
```

To initiate a connection, the client program in Example 7–2 creates a stream socket, then calls `connect`(3SOCKET), specifying the address of the socket for connection. If the target socket exists, and the request is accepted, the connection is complete. The program can now send data. Data is delivered in sequence with no message boundaries. The connection is destroyed when either socket is closed. For more information about data representation routines in this program, such as `ntohl`(3SOCKET), `ntohs`(3SOCKET), `htons`(3SOCKET), and `htonl`(3XNET), see the `byteorder`(3SOCKET) man page.

EXAMPLE 7–2 Internet Family Stream Connection (Client)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "Half a league, half a league . . ."
/*
 * This program creates a socket and initiates a connection with
 * the socket given in the command line. Some data are sent over the
 * connection and then the socket is closed, ending the connection.
 * The form of the command line is: streamwrite hostname portnumber
 * Usage: pgm host port
 */
main(argc, argv)
    int argc;
    char *argv[];
{
    int sock, errnum, h_addr_index;
    struct sockaddr_in6 server;
    struct hostent *hp;
    char buf[1024];
    /* Create socket. */
    sock = socket( AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
```

EXAMPLE 7–2 Internet Family Stream Connection (Client) *(Continued)*

```
    /* Connect socket using name specified by command line. */
    bzero (&server, sizeof (server));
    server.sin6_family = AF_INET6;
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
/*
 * getipnodebyname returns a structure including the network address
 * of the specified host.
 */
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    h_addr_index = 0;
    while (hp->h_addr_list[h_addr_index] != NULL) {
        bcopy(hp->h_addr_list[h_addr_index], &server.sin6_addr,
                    hp->h_length);
        server.sin6_port = htons(atoi(argv[2]));
        if (connect(sock, (struct sockaddr *) $server,
                    sizeof (server)) == -1) {
            if (hp->h_addr_list[++h_addr_index] != NULL) {
                /* Try next address */
                continue;
            }
            perror("connecting stream socket");
            freehostent(hp);
            exit(1);
        }
        break;
    }
    freehostent(hp);
    if (write( sock, DATA, sizeof DATA) == -1)
        perror("writing on stream socket");
    close(sock);
    freehostent (hp);
    exit(0);
}
```

Input/Output Multiplexing

Requests can be multiplexed among multiple sockets or multiple files. Use `select`(3C) to multiplex:

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
 ...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
 ...
```

```
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

The first argument of `select`(3C) is the number of file descriptors in the lists pointed to by the next three arguments.

The second, third, and fourth arguments of `select`(3C) point to three sets of file descriptors: a set of descriptors to read on, a set to write on, and a set on which exception conditions are accepted. Out-of-band data is the only exceptional condition. You can designate any of these pointers as a properly cast null. Each set is a structure that contains an array of long integer bit masks. Set the size of the array with `FD_SETSIZE`, which is defined in `select.h`. The array is long enough to hold one bit for each `FD_SETSIZE` file descriptor.

The macros `FD_SET` (*fd*, &*mask*) and `FD_CLR` (*fd*, &*mask*) add and delete, respectively, the file descriptor *fd* in the set `mask`. The set should be zeroed before use and the macro `FD_ZERO` (*&mask*) clears the set `mask`.

The fifth argument of `select`(3C) enables the specification of a time-out value. If the `timeout` pointer is `NULL`, `select`(3C) blocks until a descriptor is selectable, or until a signal is received. If the fields in `timeout` are set to `0`, `select`(3C) polls and returns immediately.

The `select`(3C) routine normally returns the number of file descriptors that are selected, or a zero if the time-out has expired. The `select`(3C) routine returns `-1` for an error or interrupt, with the error number in *errno* and the file descriptor masks unchanged. For a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending.

Test the status of a file descriptor in a select mask with the `FD_ISSET` (*fd*, &*mask*) macro. The macro returns a nonzero value if *fd* is in the set `mask`. Otherwise, the macro returns zero. Use `select`(3C) followed by a `FD_ISSET` (*fd*, *&mask*) macro on the read set to check for queued connect requests on a socket.

The following example shows how to select on a listening socket for readability to determine when a new connection can be picked up with a call to `accept`(3SOCKET). The program accepts connection requests, reads data, and disconnects on a single socket.

EXAMPLE 7–3 Using `select(3C)` to Check for Pending Connections

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time/h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
/*
 * This program uses select to check that someone is
 * trying to connect before calling accept.
```

```
 */
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    fd_set ready;
    struct timeval to;
    /* Open a socket and bind it as in previous examples. */
    /* Start accepting connections. */
    listen(sock, 5);
    do {
        FD_ZERO(&ready);
        FD_SET(sock, &ready);
        to.tv_sec = 5;
        to.tv_usec = 0;
        if (select(sock + 1, &ready, (fd_set *)0,
                   (fd_set *)0, &to) == -1) {
            perror("select");
            continue;
        }
        if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
            if (msgsock == -1)
                perror("accept");
            else do {
                memset(buf, 0, sizeof buf);
                if ((rval = read(msgsock, buf, 1024)) == -1)
                    perror("reading stream message");
                else if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
        } else
            printf("Do something else\n");
        } while (TRUE);
    exit(0);
}
```

In previous versions of the `select`(3C) routine, its arguments were pointers to integers instead of pointers to `fd_sets`. This style of call still works if the number of file descriptors is smaller than the number of bits in an integer.

The `select`(3C) routine provides a synchronous multiplexing scheme. The `SIGIO` and `SIGURG` signals, which is described in “Advanced Socket Topics” on page 150, provide asynchronous notification of output completion, input availability, and exceptional conditions.

Datagram Sockets

A datagram socket provides a symmetric data exchange interface without requiring connection establishment. Each message carries the destination address. The following figure shows the flow of communication between server and client.

The `bind`(3SOCKET) step for the server is optional.

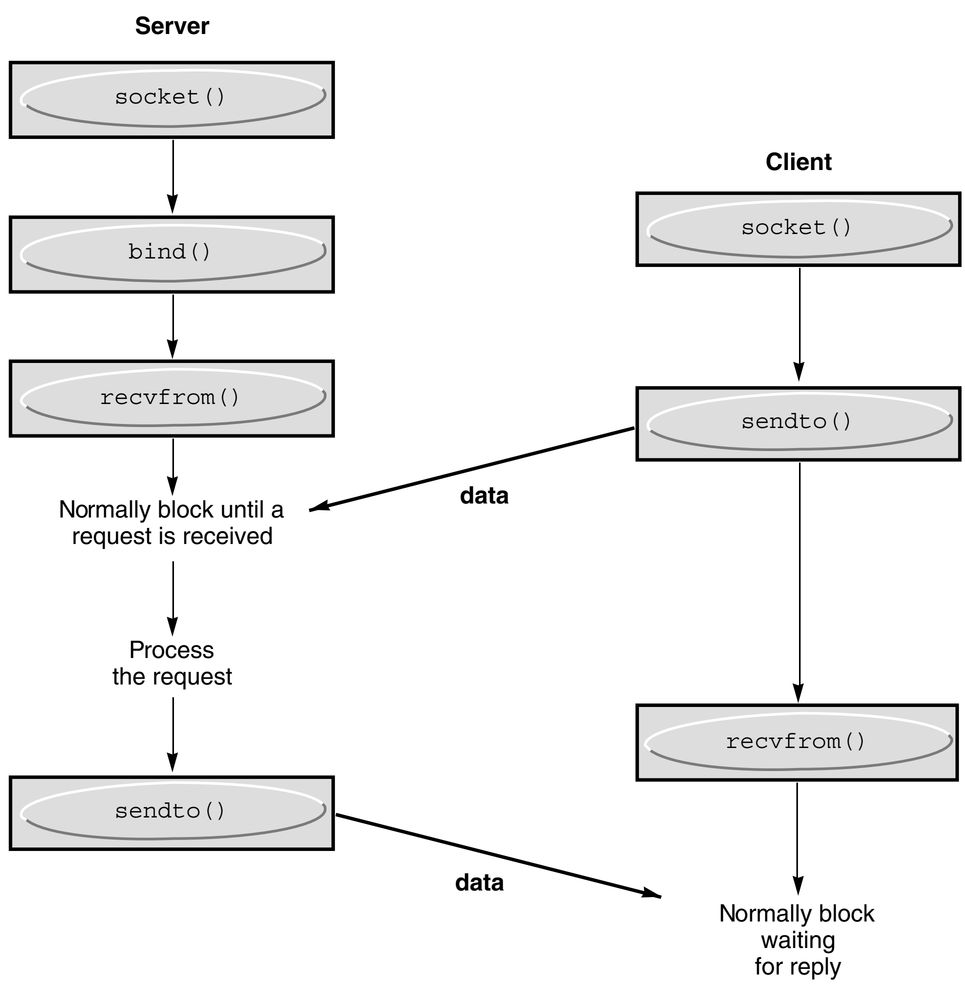


FIGURE 7–2 Connectionless Communication Using Datagram Sockets

Create datagram sockets as described in "Socket Creation" on page 127. If a particular local address is needed, the `bind`(3SOCKET) operation must precede the first data transmission. Otherwise, the system sets the local address or port when data is first sent. Use `sendto`(3SOCKET) to send data.

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are the same as in connection-oriented sockets. The *to* and *tolen* values indicate the address of the intended recipient of the message. A locally detected error condition, such as an unreachable network, causes a return of `−1` and *errno* to be set to the error number.

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from, &fromlen);
```

To receive messages on a datagram socket, `recvfrom`(3SOCKET) is used. Before the call, *fromlen* is set to the size of the *from* buffer. On return, `fromlen` is set to the size of the address from which the datagram was received.

Datagram sockets can also use the `connect`(3SOCKET) call to associate a socket with a specific destination address. The socket can then use the `send`(3SOCKET) call. Any data that is sent on the socket that does not explicitly specify a destination address is addressed to the connected peer. Only the data that is received from that peer is delivered. A socket can have only one connected address at a time. A second `connect`(3SOCKET) call changes the destination address. Connect requests on datagram sockets return immediately. The system records the peer's address. Neither `accept`(3SOCKET) nor `listen`(3SOCKET) are used with datagram sockets.

A datagram socket can return errors from previous `send`(3SOCKET) calls asynchronously while the socket is connected. The socket can report these errors on subsequent socket operations. Alternately, the socket can use an option of `getsockopt`(3SOCKET), `SO_ERROR` to interrogate the error status.

The following example code shows how to send an Internet call by creating a socket, binding a name to the socket, and sending the message to the socket.

EXAMPLE 7–4 Sending an Internet Family Datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "The sea is calm, the tide is full . . ."
/*
 * Here I send a datagram to a receiver whose name I get from
 * the command line arguments. The form of the command line is:
 * dgramsend hostname portnumber
 */
main(argc, argv)
    int argc;
    char *argv[];
{
```

EXAMPLE 7–4 Sending an Internet Family Datagram *(Continued)*

```
    int sock, errnum;
    struct sockaddr_in6 name;
    struct hostent *hp;
    /* Create socket on which to send. */
    sock = socket(AF_INET6,SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /*
     * Construct name, with no wildcards, of the socket to ''send''
     * to. getinodebyname returns a structure including the network
     * address of the specified host. The port number is taken from
     * the command line.
     */
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    bzero (&name, sizeof (name));
    memcpy((char *) &name.sin6_addr, (char *) hp->h_addr,
       hp->h_length);
    name.sin6_family = AF_INET6;
    name.sin6_port = htons(atoi(argv[2]));
    /* Send message. */
    if (sendto(sock,DATA, sizeof DATA ,0,
        (struct sockaddr *) &name,sizeof name) == -1)
        perror("sending datagram message");
    close(sock);
    exit(0);
}
```

The following sample code shows how to read an Internet call by creating a socket, binding a name to the socket, and then reading from the socket.

EXAMPLE 7–5 Reading Internet Family Datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
/*
 * This program creates a datagram socket, binds a name to it, then
 * reads from the socket.
 */
 main()
{
    int sock, length;
    struct sockaddr_in6 name;
    char buf[1024];
```

```
    /* Create socket from which to read. */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
   }
   /* Create name with wildcards. */
   bzero (&name, sizeof (name));
   name.sin6_family = AF_INET6;
   name.sin6_addr = in6addr_any;
   name.sin6_port = 0;
   if (bind (sock, (struct sockaddr *)&name, sizeof (name)) == -1) {
       perror("binding datagram socket");
       exit(1);
   }
   /* Find assigned port value and print it out. */
   length = sizeof(name);
   if (getsockname(sock,(struct sockaddr *) &name, &length)
         == -1)     {
       perror("getting socket name");
       exit(1);
   }
   printf("Socket port #%d\n", ntohs(name.sin6_port));
   /* Read from the socket. */
   if (read(sock, buf, 1024) == -1 )
       perror("receiving datagram packet");
   /* Assumes the data is printable */
   printf("-->%s\n", buf);
   close(sock);
   exit(0);
}
```

Standard Routines

This section describes the routines that you can use to locate and construct network addresses. Unless otherwise stated, interfaces presented in this section apply only to the Internet family.

Locating a service on a remote host requires many levels of mapping before the client and server communicate. A service has a name for human use. The service and host names must translate to network addresses. Finally, the network address must be usable to locate and route to the host. The specifics of the mappings can vary between network architectures. Preferably, a network does not require that hosts be named, thus protecting the identity of their physical locations.

Standard routines map host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers. The standard routines also indicate the appropriate protocol to use in communicating with the server process. The file `netdb.h` must be included when using any of these routines.

Host and Service Names

The interfaces `getaddrinfo`(3SOCKET), `getnameinfo`(3SOCKET), and `freeaddrinfo`(3SOCKET) provide a simplified way to translate between the names and addresses of a service on a host. For IPv6, you can use these interfaces instead of calling `getipnodebyname`(3SOCKET) and `getservbyname`(3SOCKET). Similarly, for IPv4, you can use these interfaces instead of `gethostbyname`(3NSL) and `getservbyname`(3SOCKET). Both IPv6 and IPv4 addresses are handled transparently.

The `getaddrinfo`(3SOCKET) routine returns the combined address and port number of the specified host and service names. Because the information returned by `getaddrinfo`(3SOCKET) is dynamically allocated, the information must be freed by `freeaddrinfo`(3SOCKET) to prevent memory leaks. `getnameinfo`(3SOCKET) returns the host and services names associated with a specified address and port number. Call `gai_strerror`(3SOCKET) to print error messages based on the `EAI_xxx` codes returned by `getaddrinfo`(3SOCKET) and `getnameinfo`(3SOCKET).

An example of using `getaddrinfo`(3SOCKET) follows.

```
struct addrinfo         *res, *aip;
struct addrinfo         hints;
int                     sock = -1;
int                     error;

/* Get host address.  Any type of address will do. */
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);
if (error != 0) {
  (void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",
  gai_strerror(error), hostname, servicename);
 return (-1);
}
```

After processing the information returned by `getaddrinfo`(3SOCKET) in the structure pointed to by `res`, the storage should be released by `freeaddrinfo(res)`.

The getnameinfo(3SOCKET) routine is particularly useful in identifying the cause of an error, as in the following example:

```
struct sockaddr_storage faddr;
int                     sock, new_sock, sock_opt;
socklen_t               faddrlen;
int                     error;
char                    hname[NI_MAXHOST];
char                    sname[NI_MAXSERV];

 ...
     faddrlen = sizeof (faddr);
     new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
     if (new_sock == -1) {
         if (errno != EINTR && errno != ECONNABORTED) {
             perror("accept");
         }
         continue;
     }
     error = getnameinfo((struct sockaddr *)&faddr, faddrlen, hname,
                 sizeof (hname), sname, sizeof (sname), 0);
     if (error) {
       (void) fprintf(stderr, "getnameinfo: %s\n",
                   gai_strerror(error));
     } else {
         (void) printf("Connection from %s/%s\n", hname, sname);
     }
```

Host Names – hostent

An Internet host-name-to-address mapping is represented by the hostent structure as defined in gethostent(3NSL):

```
struct hostent {
       char  *h_name;            /* official name of host */
       char  **h_aliases;        /* alias list */
       int   h_addrtype;         /* hostaddrtype(e.g.,AF_INET6) */
       int   h_length;           /* length of address */
       char  **h_addr_list;      /* list of addrs, null terminated */
};
/*1st addr, net byte order*/
#define h_addr h_addr_list[0]
```

getipnodebyname(3SOCKET) — Maps an Internet host name to a hostent structure

getipnodebyaddr(3SOCKET) — Maps an Internet host address to a hostent structure

freehostent(3SOCKET) — Frees the memory of a hostent structure

inet_ntop(3SOCKET)	Maps an Internet host address to a displayable string

The routines return a hostent structure that contains the name of the host, its aliases, the address type, and a NULL-terminated list of variable length addresses. The list of addresses is required because a host can have many addresses. The h_addr definition is for backward compatibility, and is the first address in the list of addresses in the hostent structure.

Network Names – netent

The routines to map network names to numbers and the reverse return a netent structure:

```
/*
 * Assumes that a network number fits in 32 bits.
 */
struct netent {
   char     *n_name;      /* official name of net */
   char     **n_aliases;  /* alias list */
   int      n_addrtype;   /* net address type */
   int      n_net;        /* net number, host byte order */
};
```

getnetbyname(3SOCKET), getnetbyaddr_r(3SOCKET), and getnetent(3SOCKET) are the network counterparts to the host routines previously described.

Protocol Names – protoent

The protoent structure defines the protocol-name mapping used with getprotobyname(3SOCKET), getprotobynumber(3SOCKET), and getprotoent(3SOCKET) and defined in getprotoent(3SOCKET):

```
struct protoent {
   char     *p_name;          /* official protocol name */
   char     **p_aliases       /* alias list */
   int      p_proto;          /* protocol number */
};
```

Service Names – servent

An Internet family service resides at a specific, well-known port, and uses a particular protocol. A service-name-to-port-number mapping is described by the servent structure that is defined in getprotoent(3SOCKET):

```
struct servent {
   char     *s_name;         /* official service name */
   char     **s_aliases;     /* alias list */
   int      s_port;          /* port number, network byte order */
   char     *s_proto;        /* protocol to use */
};
```

getservbyname(3SOCKET) maps service names and, optionally, a qualifying protocol to a servent structure. The call:

```
sp = getservbyname("telnet", (char *) 0);
```

returns the service specification of a telnet server that is using any protocol. The call:

```
sp = getservbyname("telnet", "tcp");
```

returns the telnet server that uses the TCP protocol. getservbyport(3SOCKET) and getservent(3SOCKET) are also provided. getservbyport(3SOCKET) has an interface that is similar to the interface used by getservbyname(3SOCKET). You can specify an optional protocol name to qualify lookups.

Other Routines

Several other routines that simplify manipulating names and addresses are available. The following table summarizes the routines for manipulating variable-length byte strings and byte-swapping network addresses and values.

TABLE 7–2 Runtime Library Routines

Interface	Synopsis
memcmp(3C)	Compares byte-strings; 0 if same, not 0 otherwise
memcpy(3C)	Copies *n* bytes from *s2* to *s1*
memset(3C)	Sets *n* bytes to value starting at base
htonl(3SOCKET)	32-bit quantity from host into network byte order
htons(3SOCKET)	16-bit quantity from host into network byte order
ntohl(3SOCKET)	32-bit quantity from network into host byte order
ntohs(3SOCKET)	16-bit quantity from network into host byte order

The byte-swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, the host byte ordering is different from network byte order, so programs must sometimes byte-swap values. Routines that return network addresses do so in network order. Byte-swapping problems occur only when interpreting network addresses. For example, the following code formats a TCP or UDP port:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On machines that do not need these routines, the routines are defined as null macros.

Client-Server Programs

The most common form of distributed application is the client/server model. In this scheme, client processes request services from a server process.

An alternate scheme is a service server that can eliminate dormant server processes. An example is `inetd`(1M), the Internet service daemon. `inetd`(1M) listens at a variety of ports, determined at startup by reading a configuration file. When a connection is requested on an `inetd`(1M) serviced port, `inetd`(1M) spawns the appropriate server to serve the client. Clients are unaware that an intermediary has played any part in the connection. `inetd`(1M) is described in more detail in "`inetd` Daemon" on page 160.

Sockets and Servers

Most servers are accessed at well-known Internet port numbers or UNIX family names. The service `rlogin` is an example of a well-known UNIX family name. The main loop of a remote login server is shown in Example 7–6.

The server dissociates from the controlling terminal of its invoker unless the server is operating in `DEBUG` mode.

```
(void) close(0);
(void) close(1);
(void) close(2);
(void) open("/", O_RDONLY);
(void) dup2(0);
(void) dup2(0);
setsid();
```

Dissociating prevents the server from receiving signals from the process group of the controlling terminal. After a server has dissociated from the controlling terminal, the server cannot send reports of errors to the terminal. The dissociated server must log errors with `syslog`(3C).

The server gets its service definition by calling `getaddrinfo`(3SOCKET).

```
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo(NULL, "rlogin", &hints, &aip);
```

The result, which is returned in `aip`, defines the Internet port at which the program listens for service requests. Some standard port numbers are defined in `/usr/include/netinet/in.h`.

The server then creates a socket, and listens for service requests. The `bind`(3SOCKET) routine ensures that the server listens at the expected location. Because the remote login server listens at a restricted port number, the server runs as superuser. The main body of the server is the following loop.

EXAMPLE 7–6 Server Main Loop

```
/* Wait for a connection request. */
for (;;) {
    faddrlen = sizeof (faddr);
    new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
    if (new_sock == -1) {
        if (errno != EINTR && errno != ECONNABORTED) {
            perror("rlogind: accept");
        }
        continue;
    }
    if (fork() == 0) {
        close (sock);
        doit (new_sock, &faddr);
    }
    close (new_sock);
}
/*NOTREACHED*/
```

`accept`(3SOCKET) blocks messages until a client requests service. Furthermore, `accept`(3SOCKET) returns a failure indication if `accept` is interrupted by a signal, such as `SIGCHLD`. The return value from `accept`(3SOCKET) is checked, and an error is logged with `syslog`(3C), if an error occurs.

The server then forks a child process, and invokes the main body of the remote login protocol processing. The socket used by the parent to queue connection requests is closed in the child. The socket created by `accept`(3SOCKET) is closed in the parent. The address of the client is passed to the server application's `doit()` routine, which authenticates the client.

Sockets and Clients

This section describes the steps taken by a client process. As in the server, the first step is to locate the service definition for a remote login.

```
bzero(&hints, sizeof (hints));
    hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
    hints.ai_socktype = SOCK_STREAM;

    error = getaddrinfo(hostname, servicename, &hints, &res);
```

```
        if (error != 0) {
          (void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",
                         gai_strerror(error), hostname, servicename);
          return (-1);
        }
```

getaddrinfo(3SOCKET) returns the head of a list of addresses in res. The desired address is found by creating a socket and trying to connect to each address returned in the list until one works.

```
for (aip = res; aip != NULL; aip = aip->ai_next) {
        /*
         * Open socket.  The address type depends on what
         * getaddrinfo() gave us.
         */
        sock = socket(aip->ai_family, aip->ai_socktype,
            aip->ai_protocol);
        if (sock == -1) {
            perror("socket");
            freeaddrinfo(res);
            return (-1);
        }

        /* Connect to the host. */
        if (connect(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
            perror("connect");
            (void) close(sock);
            sock = -1;
            continue;
        }
        break;
    }
```

The socket has been created and has been connected to the desired service. The connect(3SOCKET) routine implicitly binds sock, because sock is unbound.

Connectionless Servers

Some services use datagram sockets. The rwho(1) service provides status information on hosts that are connected to a local area network. Avoid running in.rwhod(1M) because in.rwho causes heavy network traffic. The rwho service broadcasts information to all hosts connected to a particular network. The rwho service is an example of datagram socket use.

A user on a host that is running the rwho(1) server can get the current status of another host with ruptime(1). Typical output is illustrated in the following example.

EXAMPLE 7–7 Output of ruptime(1) Program

```
itchy up 9:45, 5 users, load 1.15, 1.39, 1.31
scratchy up 2+12:04, 8 users, load 4.67, 5.13, 4.59
click up 10:10, 0 users, load 0.27, 0.15, 0.14
```

EXAMPLE 7–7 Output of `ruptime(1)` Program (*Continued*)

```
clack up 2+06:28, 9 users, load 1.04, 1.20, 1.65
ezekiel up 25+09:48, 0 users, load 1.49, 1.43, 1.41
dandy 5+00:05, 0 users, load 1.51, 1.54, 1.56
peninsula down 0:24
wood down 17:04
carpediem down 16:09
chances up 2+15:57, 3 users, load 1.52, 1.81, 1.86
```

Status information is periodically broadcast by the `rwho`(1) server processes on each host. The server process also receives the status information. The server also updates a database. This database is interpreted for the status of each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Use of broadcast is fairly inefficient because broadcast generates a lot of net traffic. Unless the service is used widely and frequently, the expense of periodic broadcasts outweighs the simplicity.

The following example shows a simplified version of the `rwho`(1) server. The sample code receives status information broadcast by other hosts on the network and supplies the status of the host on which the sample code is running. The first task is done in the main loop of the program: Packets received at the `rwho`(1) port are checked to be sure they were sent by another `rwho`(1) server process and are stamped with the arrival time. The packets then update a file with the status of the host. When a host has not been heard from for an extended time, the database routines assume the host is down and logs this information. Because a server might be down while a host is up, this application is prone to error.

EXAMPLE 7–8 `rwho`(1) Server

```
main()
{
   ...
   sp = getservbyname("who", "udp");
   net = getnetbyname("localnet");
   sin.sin6_addr = inet_makeaddr(net->n_net, in6addr_any);
   sin.sin6_port = sp->s_port;
   ...
   s = socket(AF_INET6, SOCK_DGRAM, 0);
   ...
   on = 1;
   if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on)
         == -1) {
      syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
      exit(1);
   }
   bind(s, (struct sockaddr *) &sin, sizeof sin);
   ...
   signal(SIGALRM, onalrm);
   onalrm();
   while(1) {
```

EXAMPLE 7–8 `rwho`(1) Server *(Continued)*

```
        struct whod wd;
            int cc, whod, len = sizeof from;
        cc = recvfrom(s, (char *) &wd, sizeof(struct whod), 0,
           (struct sockaddr *) &from, &len);
        if (cc <= 0) {
        if (cc == -1 && errno != EINTR)
           syslog(LOG_ERR, "rwhod: recv: %m");
        continue;
        }
        if (from.sin6_port != sp->s_port) {
           syslog(LOG_ERR, "rwhod: %d: bad from port",
              ntohs(from.sin6_port));
           continue;
        }
        ...
        if (!verify( wd.wd_hostname)) {
           syslog(LOG_ERR, "rwhod: bad host name from %x",
              ntohl(from.sin6_addr.s6_addr));
           continue;
        }
        (void) sprintf(path, "%s/whod.%s", RWHODIR, wd.wd_hostname);
        whod = open(path, O_WRONLY|O_CREAT|O_TRUNC, 0666);
        ...
        (void) time(&wd.wd_recvtime);
        (void) write(whod, (char *) &wd, cc);
        (void) close(whod);
    }
    exit(0);
}
```

The second server task is to supply the status of its host. This requires periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other `rwho`(1) servers to hear. This task is run by a timer. This task is triggered by a signal.

Status information is broadcast on the local network. For networks that do not support broadcast, use multicast.

Advanced Socket Topics

For most programmers, the mechanisms already described are enough to build distributed applications. This section describes additional features.

Out-of-Band Data

The stream socket abstraction includes out-of-band data. Out-of-band data is a logically independent transmission channel between a pair of connected stream sockets. Out-of-band data is delivered independent of normal data. The out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message can contain at least one byte of data. At least one message can be pending delivery at any time.

With in-band signaling, urgent data is delivered in sequence with normal data, and the message is extracted from the normal data stream. The extracted message is stored separately. Users can choose between receiving the urgent data in order and receiving the data out of sequence, without having to buffer the intervening data.

Using `MSG_PEEK`, you can peek at out-of-band data. If the socket has a process group, a `SIGURG` signal is generated when the protocol is notified of its existence. A process can set the process group or process ID to deliver `SIGURG` to with the appropriate `fcntl`(2) call, as described in "Interrupt-Driven Socket I/O" on page 154 for `SIGIO`. If multiple sockets have out-of-band data waiting for delivery, a `select`(3C) call for exceptional conditions can determine which sockets have such data pending.

A logical mark is placed in the data stream at the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal is received, all data up to the mark in the data stream is discarded.

To send an out-of-band message, apply the `MSG_OOB` flag to `send`(3SOCKET) or `sendto`(3SOCKET). To receive out-of-band data, specify `MSG_OOB` to `recvfrom`(3SOCKET) or `recv`(3SOCKET). If out-of-band data is taken in line the `MSG_OOB` flag is not needed. The `SIOCATMARK` `ioctl`(2) indicates whether the read pointer currently points at the mark in the data stream:

```
int yes;
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is `1` on return, the next read returns data after the mark. Otherwise, assuming out-of-band data has arrived, the next read provides data sent by the client before sending the out-of-band signal. The routine in the remote login process that flushes output on receipt of an interrupt or quit signal is shown in the following example. This code reads the normal data up to the mark to discard the normal data, then reads the out-of-band byte.

A process can also read or peek at the out-of-band data without first reading up to the mark. Accessing this data when the underlying protocol delivers the urgent data in-band with the normal data, and sends notification of its presence only ahead of time, is more difficult. An example of this type of protocol is TCP, the protocol used to provide socket streams in the Internet family. With such protocols, the out-of-band byte might not yet have arrived when `recv`(3SOCKET) is called with the `MSG_OOB` flag. In that case, the call returns the error of `EWOULDBLOCK`. Also, the amount of in-band data in the input buffer might cause normal flow control to prevent the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data to clear the input buffer before the peer can send the urgent data.

EXAMPLE 7–9 Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ];
        int mark = 0;

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *) &out);
        while(1) {
            if (ioctl(rem, SIOCATMARK, &mark) == −1) {
                perror("ioctl");
                break;
            }
            if (mark)
                break;
            (void) read(rem, waste, sizeof waste);
        }
        if (recv(rem, &mark, 1, MSG_OOB) == −1) {
            perror("recv");
            ...
        }
        ...
}
```

A facility to retain the position of urgent in-line data in the socket stream is available as a socket-level option, `SO_OOBINLINE`. See `getsockopt`(3SOCKET) for usage. With this socket-level option, the position of urgent data remains. However, the urgent data immediately following the mark in the normal data stream is returned without the `MSG_OOB` flag. Reception of multiple urgent indications moves the mark, but does not lose any out-of-band data.

Nonblocking Sockets

Some applications require sockets that do not block. For example, a server would return an error code, not executing a request that cannot complete immediately. This error could cause the process to be suspended, awaiting completion. After creating and connecting a socket, issuing a `fcntl`(2) call, as shown in the following example, makes the socket non-blocking.

EXAMPLE 7–10 Set Nonblocking Socket

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL, 0) == -1)
        perror("fcntl F_GETFL");
        exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == -1)
        perror("fcntl F_SETFL, FNDELAY");
        exit(1);
}
...
```

When performing I/O on a nonblocking socket, check for the error `EWOULDBLOCK` in `errno.h`, which occurs when an operation would normally block. `accept`(3SOCKET), `connect`(3SOCKET), `send`(3SOCKET), `recv`(3SOCKET), `read`(2), and `write`(2) can all return `EWOULDBLOCK`. If an operation such as a `send`(3SOCKET) cannot be done in its entirety but partial writes work, as when using a stream socket, all available data is processed. The return value is the amount of data actually sent.

Asynchronous Socket I/O

Asynchronous communication between processes is required in applications that simultaneously handle multiple requests. Asynchronous sockets must be of the `SOCK_STREAM` type. To make a socket asynchronous, you issue a `fcntl`(2) call, as shown in the following example.

EXAMPLE 7–11 Making a Socket Asynchronous

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL ) == -1)
        perror("fcntl F_GETFL");
        exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) == -1)
        perror("fcntl F_SETFL, FNDELAY | FASYNC");
```

EXAMPLE 7–11 Making a Socket Asynchronous *(Continued)*

```
        exit(1);
}
...
```

After sockets are initialized, connected, and made nonblocking and asynchronous, communication is similar to reading and writing a file asynchronously. Initiate a data transfer by using `send`(3SOCKET), `write`(2), `recv`(3SOCKET), or `read`(2). A signal-driven I/O routine completes a data transfer, as described in the next section.

Interrupt-Driven Socket I/O

The `SIGIO` signal notifies a process when a socket, or any file descriptor, has finished a data transfer. The steps in using `SIGIO` are as follows:

1. Set up a `SIGIO` signal handler with the `signal`(3C) or `sigvec`(3UCB) calls.
2. Use `fcntl`(2) to set the process ID or process group ID to route the signal to its own process ID or process group ID. The default process group of a socket is group `0`.
3. Convert the socket to asynchronous, as shown in "Asynchronous Socket I/O" on page 153.

The following sample code enables receipt of information on pending requests as the requests occur for a socket by a given process. With the addition of a handler for `SIGURG`, this code can also be used to prepare for receipt of `SIGURG` signals.

EXAMPLE 7–12 Asynchronous Notification of I/O Requests

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* Set the process receiving SIGIO/SIGURG signals to us. */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
        perror("fcntl F_SETOWN");
        exit(1);
}
```

Signals and Process Group ID

For `SIGURG` and `SIGIO`, each socket has a process number and a process group ID. These values are initialized to zero, but can be redefined at a later time with the `F_SETOWN` `fcntl`(2) command, as in the previous example. A positive third argument to fcntl(2) sets the socket's process ID. A negative third argument to fcntl(2) sets the socket's process group ID. The only allowed recipient of SIGURG and SIGIO signals is the calling process. A similar fcntl(2), F_GETOWN, returns the process number of a socket.

You can also enable reception of SIGURG and SIGIO by using ioctl(2) to assign the socket to the user's process group.

```
/* oobdata is the out-of-band data handling routine */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
        perror("ioctl: SIOCSPGRP");
}
```

Another signal that is useful in server processes is SIGCHLD. This signal is delivered to a process when any child process changes state. Normally, servers use the signal to "reap" child processes that have exited without explicitly awaiting their termination or periodically polling for exit status. For example, the remote login server loop that was shown previously can be augmented, as shown in the following example.

EXAMPLE 7–13 SIGCHLD Signal

```
int reaper();
...
sigset(SIGCHLD, reaper);
listen(f, 5);
while (1) {
        int g, len = sizeof from;
        g = accept(f, (struct sockaddr *) &from, &len);
        if (g < 0) {
            if (errno != EINTR)
                syslog(LOG_ERR, "rlogind: accept: %m");
            continue;
        }
        ...
}

#include <wait.h>

reaper()
{
        int options;
        int error;
        siginfo_t info;

        options = WNOHANG | WEXITED;
        bzero((char *) &info, sizeof(info));
        error = waitid(P_ALL, 0, &info, options);
}
```

If the parent server process fails to reap its children, zombie processes result.

Selecting Specific Protocols

If the third argument of the socket(3SOCKET) call is 0, socket(3SOCKET) selects a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. When using raw sockets to communicate directly with lower-level protocols or lower-level hardware interfaces, set up de-multiplexing with the protocol argument.

Using raw sockets in the Internet family to implement a new protocol on IP ensures that the socket only receives packets for the specified protocol. To obtain a particular protocol, determine the protocol number as defined in the protocol family. For the Internet family, use one of the library routines that are discussed in "Standard Routines" on page 141, such as getprotobyname(3SOCKET).

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
 ...
pp = getprotobyname("newtcp");
s = socket(AF_INET6, SOCK_STREAM, pp->p_proto);
```

Using getprotobyname results in a socket s by using a stream-based connection, but with a protocol type of newtcp instead of the default tcp.

Address Binding

For addressing, TCP and UDP use a 4-tuple of:

- Local IP address
- Local port number
- Foreign IP address
- Foreign port number

TCP requires these 4-tuples to be unique. UDP does not. User programs do not always know proper values to use for the local address and local port, because a host can reside on multiple networks. The set of allocated port numbers is not directly accessible to a user. To avoid these problems, leave parts of the address unspecified and let the system assign the parts appropriately when needed. Various portions of these tuples can be specified by various parts of the sockets API:

bind(3SOCKET)	Local address or local port or both
connect(3SOCKET)	Foreign address and foreign port

A call to accept(3SOCKET) retrieves connection information from a foreign client. This causes the local address and port to be specified to the system even though the caller of accept(3SOCKET) did not specify anything. The foreign address and foreign port are returned.

A call to `listen`(3SOCKET) can cause a local port to be chosen. If no explicit `bind`(3SOCKET) has been done to assign local information, `listen`(3SOCKET) assigns an ephemeral port number.

A service that resides at a particular port can `bind`(3SOCKET) to that port. Such a service can leave the local address unspecified if the service does not require local address information. The local address is set to `in6addr_any`, a variable with a constant value in `<netinet/in.h>`. If the local port does not need to be fixed, a call to `listen`(3SOCKET) causes a port to be chosen. Specifying an address of `in6addr_any` or a port number of 0 is known as "wildcarding." For `AF_INET`, `INADDR_ANY` is used in place of `in6addr_any`.

The wildcard address simplifies local address binding in the Internet family. The following sample code binds a specific port number that was returned by a call to `getaddrinfo`(3SOCKET) to a socket and leaves the local address unspecified:

```
#include <sys/types.h>
#include <netinet/in.h>
...
    struct addrinfo        *aip;
...
    if (bind(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
        perror("bind");
        (void) close(sock);
        return (-1);
    }
```

Each network interface on a host typically has a unique IP address. Sockets with wildcard local addresses can receive messages that are directed to the specified port number. Messages that are sent to any of the possible addresses that are assigned to a host are also received by sockets with wildcard local addresses. To allow only hosts on a specific network to connect to the server, a server binds the address of the interface on the appropriate network.

Similarly, a local port number can be left unspecified, in which case the system selects a port number. For example, to bind a specific local address to a socket, but to leave the local port number unspecified, you could use `bind` as follows:

```
bzero (&sin, sizeof (sin));
(void) inet_pton (AF_INET6, ":ffff:127.0.0.1", sin.sin6_addr.s6_addr);
sin.sin6_family = AF_INET6;
sin.sin6_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The system uses two criteria to select the local port number:

- Internet port numbers less than 1024 (`IPPORT_RESERVED`) are reserved for privileged users. Nonprivileged users can use any Internet port number that is greater than 1024. The largest Internet port number is 65535.
- The port number is not currently bound to some other socket.

The port number and IP address of the client are found through either `accept`(3SOCKET) or `getpeername`(3SOCKET).

In certain cases, the algorithm used by the system to select port numbers is unsuitable for an application due to the two-step creation process for associations. For example, the Internet file transfer protocol specifies that data connections must always originate from the same local port. However, duplicate associations are avoided by connecting to different foreign ports. In this situation, the system would disallow binding the same local address and local port number to a socket if a previous data connection's socket still existed.

To override the default port selection algorithm, you must perform an option call before address binding:

```
 ...
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

With this call, local addresses already in use can be bound. This binding does not violate the uniqueness requirement. The system still verifies at connect time that any other sockets with the same local address and local port do not have the same foreign address and foreign port. If the association already exists, the error `EADDRINUSE` is returned.

Zero Copy and Checksum Off-load

In SunOS version 5.6 and compatible versions, the TCP/IP protocol stack has been enhanced to support two new features: zero copy and TCP checksum off-load.

- Zero copy uses virtual memory MMU remapping together with a copy-on-write technique to move data between the application and the kernel space.
- Checksum off-loading relies on special hardware logic to off-load the TCP checksum calculation.

Although zero copy and checksum off-loading are functionally independent of each other, these functions must work together to obtain optimal performance. Checksum off-loading requires hardware support from the network interface. Without this hardware support, zero copy is not enabled.

Zero copy requires that the applications supply page-aligned buffers before applying virtual memory page remapping. Applications should use large, circular buffers on the transmit side to avoid expensive copy-on-write faults. A typical buffer allocation is sixteen 8K buffers.

Socket Options

You can set and get several options on sockets through `setsockopt`(3SOCKET) and `getsockopt`(3SOCKET). For example, you can change the send or receive buffer space. The general forms of the calls are in the following list:

```
setsockopt(s, level, optname, optval, optlen);
```

and

```
getsockopt(s, level, optname, optval, optlen);
```

The operating system can adjust the values appropriately at any time.

The arguments of `setsockopt`(3SOCKET) and `getsockopt`(3SOCKET) calls are in the following list:

s — Socket on which the option is to be applied

level — Specifies the protocol level, such as socket level, indicated by the symbolic constant `SOL_SOCKET` in `sys/socket.h`

optname — Symbolic constant defined in `sys/socket.h` that specifies the option

optval — Points to the value of the option

optlen — Points to the length of the value of the option

For `getsockopt`(3SOCKET), *optlen* is a value-result argument. This argument is initially set to the size of the storage area pointed to by *optval*. On return, the argument's value is set to the length of storage used.

When a program needs to determine an existing socket's type, the program should invoke `inetd`(1M) by using the `SO_TYPE` socket option and the `getsockopt`(3SOCKET) call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) <0) {
     ...
}
```

After `getsockopt`(3SOCKET), `type` is set to the value of the socket type, as defined in `sys/socket.h`. For a datagram socket, `type` would be `SOCK_DGRAM`.

inetd Daemon

The inetd(1M) daemon is invoked at startup time and gets the services for which the daemon listens from the /etc/inet/inetd.conf file. The daemon creates one socket for each service that is listed in /etc/inet/inetd.conf, binding the appropriate port number to each socket. See the inetd(1M) man page for details.

The inetd(1M) daemon polls each socket, waiting for a connection request to the service corresponding to that socket. For SOCK_STREAM type sockets, inetd(1M) accepts (accept(3SOCKET)) on the listening socket, forks (fork(2)), duplicates (dup(2)) the new socket to file descriptors 0 and 1 (stdin and stdout), closes other open file descriptors, and executes (exec(2)) the appropriate server.

The primary benefit of using inetd(1M) is that services not in use do not consume machine resources. A secondary benefit is that inetd(1M) does most of the work to establish a connection. The server started by inetd(1M) has the socket connected to its client on file descriptors 0 and 1. The server can immediately read, write, send, or receive. Servers can use buffered I/O as provided by the stdio conventions, as long as the servers use fflush(3C) when appropriate.

The getpeername(3SOCKET) routine returns the address of the peer (process) connected to a socket. This routine is useful in servers started by inetd(1M). For example, you could use this routine to log the Internet address such as fec0::56:a00:20ff:fe7d:3dd2, which is conventional for representing the IPv6 address of a client. An inetd(1M) server could use the following sample code:

```
struct sockaddr_storage name;
int namelen = sizeof (name);
char abuf[INET6_ADDRSTRLEN];
struct in6_addr addr6;
struct in_addr addr;

if (getpeername(fd, (struct sockaddr *)&name, &namelen) == -1) {
    perror("getpeername");
    exit(1);
} else {
    addr = ((struct sockaddr_in *)&name)->sin_addr;
    addr6 = ((struct sockaddr_in6 *)&name)->sin6_addr;
    if (name.ss_family == AF_INET) {
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6 &&
               IN6_IS_ADDR_V4MAPPED(&addr6)) {
            /* this is a IPv4-mapped IPv6 address */
            IN6_MAPPED_TO_IN(&addr6, &addr);
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6) {
            (void) inet_ntop(AF_INET6, &addr6, abuf, sizeof (abuf));

    }
    syslog("Connection from %s\n", abuf);
}
```

Broadcasting and Determining Network Configuration

Broadcasting is not supported in IPv6. Broadcasting is supported only in IPv4.

Messages sent by datagram sockets can be broadcast to reach all of the hosts on an attached network. The network must support broadcast because the system provides no simulation of broadcast in software. Broadcast messages can place a high load on a network because broadcast messages force every host on the network to service the broadcast messages. Broadcasting is usually used for either of two reasons:

- To find a resource on a local network without having its address
- For functions that require information to be sent to all accessible neighbors

To send a broadcast message, create an Internet datagram socket:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

Bind a port number to the socket:

```
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = htonl(INADDR_ANY);
sin.sin_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The datagram can be broadcast on only one network by sending to the network's broadcast address. A datagram can also be broadcast on all attached networks by sending to the special address `INADDR_BROADCAST`, which is defined in `netinet/in.h`.

The system provides a mechanism to determine a number of pieces of information about the network interfaces on the system. This information includes the IP address and broadcast address. The `SIOCGIFCONF` `ioctl`(2) call returns the interface configuration of a host in a single `ifconf` structure. This structure contains an array of `ifreq` structures. Every address family supported by every network interface to which the host is connected has its own `ifreq` structure.

The following example shows the `ifreq` structures defined in `net/if.h`.

EXAMPLE 7–14 `net/if.h` Header File

```
struct ifreq {
#define IFNAMSIZ 16
char ifr_name[IFNAMSIZ]; /* if name, e.g., "en0" */
union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        char ifru_oname[IFNAMSIZ]; /* other if name */
        struct sockaddr ifru_broadaddr;
        short ifru_flags;
        int ifru_metric;
        char ifru_data[1]; /* interface dependent data */
```

EXAMPLE 7–14 `net/if.h` Header File *(Continued)*

```
        char ifru_enaddr[6];
} ifr_ifru;
#define ifr_addr ifr_ifru.ifru_addr
#define ifr_dstaddr ifr_ifru.ifru_dstaddr
#define ifr_oname ifr_ifru.ifru_oname
#define ifr_broadaddr ifr_ifru.ifru_broadaddr
#define ifr_flags ifr_ifru.ifru_flags
#define ifr_metric ifr_ifru.ifru_metric
#define ifr_data ifr_ifru.ifru_data
#define ifr_enaddr ifr_ifru.ifru_enaddr
};
```

The call that obtains the interface configuration is:

```
/*
 * Do SIOCGIFNUM ioctl to find the number of interfaces
 *
 * Allocate space for number of interfaces found
 *
 * Do SIOCGIFCONF with allocated buffer
 *
 */
if (ioctl(s, SIOCGIFNUM, (char *)&numifs) == -1) {
        numifs = MAXIFS;
}
bufsize = numifs * sizeof(struct ifreq);
reqbuf = (struct ifreq *)malloc(bufsize);
if (reqbuf == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
}
ifc.ifc_buf = (caddr_t)&reqbuf[0];
ifc.ifc_len = bufsize;
if (ioctl(s, SIOCGIFCONF, (char *)&ifc) == -1) {
        perror("ioctl(SIOCGIFCONF)");
        exit(1);
}
...
}
```

After this call, *buf* contains an array of `ifreq` structures. Every network to which the host connects has an associated `ifreq` structure. The sort order these structures appear in is:

- Alphabetical by interface name
- Numerical by supported address families

The value of `ifc.ifc_len` is set to the number of bytes used by the `ifreq` structures.

Each structure has a set of interface flags that indicate whether the corresponding network is up or down, point-to-point or broadcast, and so on. The following example shows ioctl(2) returning the SIOCGIFFLAGS flags for an interface specified by an ifreq structure.

EXAMPLE 7–15 Obtaining Interface Flags

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); --n >= 0; ifr++) {
   /*
    * Be careful not to use an interface devoted to an address
    * family other than those intended.
    */
   if (ifr->ifr_addr.sa_family != AF_INET)
      continue;
   if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
      ...
   }
   /* Skip boring cases */
   if ((ifr->ifr_flags & IFF_UP) == 0 ||
      (ifr->ifr_flags & IFF_LOOPBACK) ||
      (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
      continue;
}
```

The following example uses the SIOGGIFBRDADDR ioctl(2) command to obtain the broadcast address of an interface.

EXAMPLE 7–16 Broadcast Address of an Interface

```
if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {
        ...
}
memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,
        sizeof ifr->ifr_broadaddr);
```

You can also use SIOGGIFBRDADDR ioctl(2) to get the destination address of a point-to-point interface.

After the interface broadcast address is obtained, transmit the broadcast datagram with sendto(3SOCKET):

```
sendto(s, buf, buflen, 0, (struct sockaddr *)&dst, sizeof dst);
```

Use one sendto(3SOCKET) for each interface to which the host is connected, if that interface supports the broadcast or point-to-point addressing.

Using Multicast

IP multicasting is supported only on AF_INET6 and AF_INET sockets of type SOCK_DGRAM and SOCK_RAW. IP multicasting is only supported on subnetworks for which the interface driver supports multicasting.

Sending IPv4 Multicast Datagrams

To send a multicast datagram, specify an IP multicast address in the range 224.0.0.0 to 239.255.255.255 as the destination address in a sendto(3SOCKET) call.

By default, IP multicast datagrams are sent with a time-to-live (TTL) of 1. This value prevents the datagrams from being forwarded beyond a single subnetwork. The socket option IP_MULTICAST_TTL allows the TTL for subsequent multicast datagrams to be set to any value from 0 to 255. This ability is used to control the scope of the multicasts.

```
u_char ttl;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &ttl,sizeof(ttl))
```

Multicast datagrams with a TTL of 0 are not transmitted on any subnet, but can be delivered locally if the sending host belongs to the destination group and if multicast loopback has not been disabled on the sending socket. Multicast datagrams with a TTL greater than one can be delivered to more than one subnet if one or more multicast routers are attached to the first-hop subnet. To provide meaningful scope control, the multicast routers support the notion of TTL thresholds. These thresholds prevent datagrams with less than a certain TTL from traversing certain subnets. The thresholds enforce the conventions for multicast datagrams with initial TTL values as follows:

0	Are restricted to the same host
1	Are restricted to the same subnet
32	Are restricted to the same site
64	Are restricted to the same region
128	Are restricted to the same continent
255	Are unrestricted in scope

Sites and regions are not strictly defined and sites can be subdivided into smaller administrative units as a local matter.

An application can choose an initial TTL other than the ones previously listed. For example, an application might perform an expanding-ring search for a network resource by sending a multicast query, first with a TTL of 0 and then with larger and larger TTLs until a reply is received.

The multicast router does not forward any multicast datagram with a destination address between 224.0.0.0 and 224.0.0.255 inclusive, regardless of its TTL. This range of addresses is reserved for the use of routing protocols and other low-level topology discovery or maintenance protocols, such as gateway discovery and group membership reporting.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. If the host is also a multicast router and the TTL is greater than 1, a multicast can be *forwarded* to interfaces other than the originating interface. A socket option is available to override the default for subsequent transmissions from a given socket:

```
struct in_addr addr;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_IF, &addr, sizeof(addr))
```

where `addr` is the local IP address of the desired outgoing interface. Revert to the default interface by specifying the address `INADDR_ANY`. The local IP address of an interface is obtained with the `SIOCGIFCONF` `ioctl`. To determine if an interface supports multicasting, fetch the interface flags with the `SIOCGIFFLAGS` `ioctl` and test if the `IFF_MULTICAST` flag is set. This option is intended primarily for multicast routers and other system services specifically concerned with Internet topology.

If a multicast datagram is sent to a group to which the sending host itself belongs, a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether subsequent datagrams are looped back:

```
u_char loop;
setsockopt(sock, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop))
```

where `loop` is 0 to disable loopback and 1 to enable loopback. This option provides a performance benefit for applications that have only one instance on a single host by eliminating the overhead of receiving their own transmissions. Applications that can have more than one instance on a single host, or for which the sender does not belong to the destination group, should not use this option.

If the sending host belongs to the destination group on another interface, a multicast datagram sent with an initial TTL greater than 1 can be delivered to the sending host on the other interface. The loopback control option has no effect on such delivery.

Receiving IPv4 Multicast Datagrams

Before a host can receive IP multicast datagrams, the host must become a member of one or more IP multicast groups. A process can ask the host to join a multicast group by using the following socket option:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq))
```

where `mreq` is the structure:

```
struct ip_mreq {
    struct in_addr imr_multiaddr;   /* multicast group to join */
    struct in_addr imr_interface;   /* interface to join on */
}
```

Each membership is associated with a single interface. You can join the same group on more than one interface. Specify the `imr_interface` address as `in6addr_any` to choose the default multicast interface. You can also specify one of the host's local addresses to choose a particular multicast-capable interface.

To drop a membership, use:

```
struct ip_mreq mreq;
setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq))
```

where `mreq` contains the same values used to add the membership. Closing a socket or killing the process that holds the socket drops the memberships associated with that socket. More than one socket can claim a membership in a particular group, and the host remains a member of that group until the last claim is dropped.

If any socket claims membership in the destination group of the datagram, the kernel IP layer accepts incoming multicast packets. A given socket's receipt of a multicast datagram depends on the socket's associated destination port and memberships, or the protocol type for raw sockets. To receive multicast datagrams sent to a particular port, bind to the local port, leaving the local address unspecified, such as `INADDR_ANY`.

More than one process can bind to the same `SOCK_DGRAM` UDP port if the `bind`(3SOCKET) is preceded by:

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

In this case, every incoming multicast or broadcast UDP datagram destined for the shared port is delivered to all sockets bound to the port. For backwards compatibility reasons, this delivery does *not* apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port. `SOCK_RAW` sockets do not require the `SO_REUSEADDR` option to share a single IP protocol type.

The definitions required for the new, multicast-related socket options are found in `<netinet/in.h>`. All IP addresses are passed in network byte-order.

Sending IPv6 Multicast Datagrams

To send an IPv6 multicast datagram, specify an IP multicast address in the range `ff00::0/8` as the destination address in a `sendto`(3SOCKET) call.

By default, IP multicast datagrams are sent with a hop limit of one, which prevents the datagrams from being forwarded beyond a single subnetwork. The socket option `IPV6_MULTICAST_HOPS` allows the hop limit for subsequent multicast datagrams to be set to any value from 0 to 255. This ability is used to control the scope of the multicasts:

```
uint_l;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS, &hops,sizeof(hops))
```

You cannot transmit multicast datagrams with a hop limit of zero on any subnet, but you can deliver the dtagrams locally if:

- The sending host belongs to the destination group
- Multicast loopback on the sending socket is enabled

You can deliver multicast datagrams with a hop limit that is greater than one to more than one subnet if the first-hop subnet attaches to one or more multicast routers. The IPv6 multicast addresses, unlike their IPv4 counterparts, contain explicit scope information that is encoded in the first part of the address. The defined scopes are, where `X` is unspecified:

`ffX1::0/16`	Node–local scope — restricted to the same node
`ffX2::0/16`	Link–local scope
`ffX5::0/16`	Site–local scope
`ffX8::0/16`	Organization–local scope
`ffXe::0/16`	Global scope

An application can, separately from the scope of the multicast address, use different hop limit values. For example, an application might perform an expanding-ring search for a network resource by sending a multicast query, first with a hop limit of 0, and then with larger and larger hop limits, until a reply is received.

Each multicast transmission is sent from a single network interface, even if the host has more than one multicast-capable interface. If the host is also a multicast router, and the hop limit is greater than 1, a multicast can be *forwarded* to interfaces other than the originating interface. A socket option is available to override the default for subsequent transmissions from a given socket:

```
uint_t ifindex;

ifindex = if_nametoindex )"hme3");
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_IF, &ifindex,
      sizeof(ifindex))
```

where `ifindex` is the interface index for the desired outgoing interface. Revert to the default interface by specifying the value `0`.

If a multicast datagram is sent to a group to which the sending host itself belongs, a copy of the datagram is, by default, looped back by the IP layer for local delivery. Another socket option gives the sender explicit control over whether to loop back subsequent datagrams:

```
uint_t loop;
setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_LOOP, &loop,
        sizeof(loop))
```

where `loop` is zero to disable loopback and one to enable loopback. This option provides a performance benefit for applications that have only one instance on a single host (such as a router or a mail demon), by eliminating the overhead of receiving their own transmissions. Applications that can have more than one instance on a single host (such as a conferencing program) or for which the sender does not belong to the destination group (such as a time querying program) should not use this option.

If the sending host belongs to the destination group on another interface, a multicast datagram sent with an initial hop limit greater than 1 can be delivered to the sending host on the other interface. The loopback control option has no effect on such delivery.

Receiving IPv6 Multicast Datagrams

Before a host can receive IP multicast datagrams, the host must become a member of one, or more IP multicast groups. A process can ask the host to join a multicast group by using the following socket option:

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP, &mreq, sizeof(mreq))
```

where `mreq` is the structure:

```
struct ipv6_mreq {
    struct in6_addr ipv6mr_multiaddr;   /* IPv6 multicast addr */
    unsigned int    ipv6mr_interface;   /* interface index */
}
```

Each membership is associated with a single interface. You can join the same group on more than one interface. Specify `ipv6_interface` to be `0` to choose the default multicast interface. Specify an interface index for one of the host's interfaces to choose that multicast-capable interface.

To leave a group, use:

```
struct ipv6_mreq mreq;
setsockopt(sock, IPPROTO_IPV6, IP_LEAVE_GROUP, &mreq, sizeof(mreq))
```

where `mreq` contains the same values used to add the membership. The socket drops associated memberships when the socket is closed, or when the process that holds the socket is killed. More than one socket can claim a membership in a particular group. The host remains a member of that group until the last claim is dropped.

The kernel IP layer accepts incoming multicast packets if any socket has claimed a membership in the destination group of the datagram. Delivery of a multicast datagram to a particular socket is determined by the destination port and the memberships associated with the socket, or by the protocol type for raw sockets. To receive multicast datagrams sent to a particular port, bind to the local port, leaving the local address unspecified, such as `INADDR_ANY`.

More than one process can bind to the same `SOCK_DGRAM` UDP port if the `bind`(3SOCKET) is preceded by:

```
int one = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one))
```

In this case, all sockets that are bound to the port receive every incoming multicast UDP datagram destined to the shared port. For backward compatibility reasons, this delivery does *not* apply to incoming unicast datagrams. Unicast datagrams are never delivered to more than one socket, regardless of how many sockets are bound to the datagram's destination port. `SOCK_RAW` sockets do not require the `SO_REUSEADDR` option to share a single IP protocol type.

The definitions required for the new, multicast-related socket options are found in `<netinet/in.h>`. All IP addresses are passed in network byte-order.

CHAPTER 8

Programming With XTI and TLI

This chapter describes the Transport Layer Interface (TLI) and the X/Open Transport Interface (XTI). Advanced topics such as asynchronous execution mode are discussed in "Advanced XTI/TLI Topics" on page 176.

Some recent additions to XTI, such as scatter/gather data transfer, are discussed in "Additions to the XTI Interface" on page 196.

The transport layer of the OSI model (layer 4) is the lowest layer of the model that provides applications and higher layers with end-to-end service. This layer hides the topology and characteristics of the underlying network from users. The transport layer also defines a set of services common to many contemporary protocol suites including the OSI protocols, Transmission Control Protocol and TCP/IP Internet Protocol Suite, Xerox Network Systems (XNS), and Systems Network Architecture (SNA).

TLI s modeled on the industry standard Transport Service Definition (ISO 8072). It also can be used to access both TCP and UDP. XTI and TLI are a set of interfaces that constitute a network programming interface. XTI is an evolution from the older TLI interface available on the SunOS 4 platform. The Solaris operating environment supports both interfaces, although XTI represents the future direction of this set of interfaces. The Solaris software implements XTI and TLI as a user library using the STREAMS I/O mechanism.

What Are XTI and TLI?

Note – The interfaces described in this chapter are multithread safe. This means that applications containing XTI/TLI interface calls can be used freely in a multithreaded application. Because these interface calls are not re-entrant, they do not provide linear scalability.

Caution – The XTI/TLI interface behavior has not been well specified in an asynchronous environment. Do not use these interfaces from signal handler routines.

TLI was introduced with AT&T System V, Release 3 in 1986. TLI provided a transport layer interface API. The ISO Transport Service Definition provided the model on which TLI is based. TLI provides an API between the OSI transport and session layers. TLI interfaces evolved further in AT&T System V, Release 4 version of UNIX and were also made available in SunOS 5.6 operating system interfaces.

XTI interfaces are an evolution of TLI interfaces and represent the future direction of this family of interfaces. Compatibility for applications using TLI interfaces is available. You do not need to port TLI applications to XTI immediately. New applications can use the XTI interfaces and you can port older applications to XTI when necessary.

TLI is implemented as a set of interface calls in a library (`libnsl`) to which the applications link. XTI applications are compiled using the c89 front end and must be linked with the `xnet` library (`libxnet`). For additional information on compiling with XTI, see the `standards`(5) man page.

Note – An application using the XTI interface uses the `xti.h` header file, whereas an application using the TLI interface includes the `tiuser.h` header file.

XTI/TLI code can be independent of current transport providers when used in conjunction with some additional interfaces and mechanisms described in Chapter 4. The SunOS 5 product includes some transport providers (TCP, for example) as part of the base operating system. A transport provider performs services, and the transport user requests the services. The transport user issues service requests to the transport provider. An example is a request to transfer data over a connection TCP and UDP.

XTI/TLI can also be used for transport-independent programming by taking advantage of two components:

- Library routines that perform the transport services, in particular, transport selection and name-to-address translation. The network services library includes a set of interfaces that implement XTI/TLI for user processes. See Chapter 9.

  Programs using TLI should be linked with the `libnsl` network services library by specifying the `-l nsl` option at compile time.

  Programs using XTI should be linked with the `xnet` library by specifying the `-l xnet` option at compile time.
- State transition rules that define the sequence in which the transport routines can be invoked. For more information on state transition rules, see "State Transitions" on page 187. The state tables define the legal sequence of library calls based on the state and the handling of events. These events include user-generated library calls, as well as provider-generated event indications. XTI/TLI programmers should understand all state transitions before using the interface.

XTI/TLI Read/Write Interface

A user might want to establish a transport connection using `exec`(2) on an existing program (such as `/usr/bin/cat`) to process the data as it arrives over the connection. Existing programs use `read`(2) and `write`(2). XTI/TLI does not directly support a read/write interface to a transport provider, but one is available. The interface enables you to issue `read`(2) and `write`(2) calls over a transport connection in the data transfer phase. This section describes the read/write interface to the connection mode service of XTI/TLI. This interface is not available with the connectionless mode service.

EXAMPLE 8–1 Read/Write Interface

```
#include <stropts.h>
   .
   ./*
     Same local management and connection establishment steps.
     */
   .
   if (ioctl(fd, I_PUSH, "tirdwr") == −1) {
      perror("I_PUSH of tirdwr failed");
      exit(5);
     }
   close(0);
   dup(fd);
   execl("/usr/bin/cat", "/usr/bin/cat", (char *) 0);
   perror("exec of /usr/bin/cat failed");
   exit(6);
```

EXAMPLE 8–1 Read/Write Interface *(Continued)*

```
}
```

The client invokes the read/write interface by pushing `tirdwr` onto the stream associated with the transport endpoint. See the description of `I_PUSH` in the `streamio`(7I) man page. The `tirdwr` module converts XTI/TLI above the transport provider into a pure read/write interface. With the module in place, the client calls `close`(2) and `dup`(2) to establish the transport endpoint as its standard input file, and uses `/usr/bin/cat` to process the input.

Pushing `tirdwr` onto the transport provider forces XTI/TLI to use `read`(2) and `write`(2) semantics. XTI/TLI does not preserve message boundaries when using `read` and `write` semantics. Pop `tirdwr` from the transport provider to restore XTI/TLI semantics (see the description of `I_POP` in the `streamio`(7I) man page.

Caution – Push the `tirdwr` module onto a stream only when the transport endpoint is in the data transfer phase. After pushing the module, the user cannot call any XTI/TLI routines. If the user invokes an XTI/TLI routine, `tirdwr` generates a fatal protocol error, `EPROTO`, on the stream, rendering it unusable. If you then pop the `tirdwr` module off the stream, the transport connection aborts. See the description of `I_POP` in the `streamio`(7I) man page.

Write Data

After you send data over the transport connection with `write`(2), `tirdwr` passes data through to the transport provider. If you send a zero-length data packet, which the mechanism allows, `tirdwr` discards the message. If the transport connection is aborted, a hang-up condition is generated on the stream, further `write`(2) calls fail, and `errno` is set to `ENXIO`. This problem might occur, for example, because the remote user aborts the connection using `t_snddis`(3NSL). You can still retrieve any available data after a hang-up.

Read Data

Receive data that arrives at the transport connection with `read`(2). `tirdwr` passes data from the transport provider. The `tirdwr` module processes any other event or request passed to the user from the provider as follows:

- `read`(2) cannot identify expedited data to the user. If `read`(2) receives an expedited data request, `tirdwr` generates a fatal protocol error, `EPROTO`, on the stream. The error causes further system calls to fail. Do not use `read`(2) to receive expedited data.

- tirdwr discards an abortive disconnect request and generates a hang-up condition on the stream. Subsequent read(2) calls retrieve any remaining data, then return zero for all further calls, indicating end of file.
- tirdwr discards an orderly release request and delivers a zero-length message to the user. As described in the read(2) man page, this notifies the user of end of file by returning 0.
- If read(2) receives any other XTI/TLI request, tirdwr generates a fatal protocol error, EPROTO, on the stream. This causes further system calls to fail. If a user pushes tirdwr onto a stream after establishing the connection, tirdwr generates no request.

Close Connection

With tirdwr on a stream, you can send and receive data over a transport connection for the duration of the connection. Either user can terminate the connection by closing the file descriptor associated with the transport endpoint or by popping the tirdwr module off the stream. In either case, tirdwr does the following:

- If tirdwr receives an orderly release request, it passes the request to the transport provider to complete the orderly release of the connection. The remote user who initiated the orderly release procedure receives the expected request when data transfer completes.
- If tirdwr receives a disconnect request, it takes no special action.
- If tirdwr receives neither an orderly release nor a disconnect request, it passes a disconnect request to the transport provider to abort the connection.
- If an error occurs on the stream and tirdwr does not receive a disconnect request, it passes a disconnect request to the transport provider.

A process cannot initiate an orderly release after pushing tirdwr onto a stream. tirdwr handles an orderly release if the user on the other side of a transport connection initiates the release. If the client in this section is communicating with a server program, the server terminates the transfer of data with an orderly release request. The server then waits for the corresponding request from the client. At that point, the client exits and closes the transport endpoint. After closing the file descriptor, tirdwr initiates the orderly release request from the client's side of the connection. This release generates the request on which the server blocks.

Some protocols, like TCP, require this orderly release to ensure intact delivery of the data.

Advanced XTI/TLI Topics

This section presents additional XTI/TLI concepts:

- “Asynchronous Execution Mode” on page 176 describes optional nonblocking (asynchronous) mode for some library calls.
- “Advanced XTI/TLI Programming Example” on page 176 is a program example of a server supporting multiple outstanding connect requests and operating in an event-driven manner.

Asynchronous Execution Mode

Many XTI/TLI library routines block to wait for an incoming event. However, some time-critical applications should not block for any reason. An application can do local processing while waiting for some asynchronous XTI/TLI event.

Applications can access asynchronous processing of XTI/TLI events through the combination of asynchronous features and the non-blocking mode of XTI/TLI library routines. See the *ONC+ Developer’s Guide* for information on use of the `poll`(2) system call and the `I_SETSIG` `ioctl`(2) command to process events asynchronously.

You can run each XTI/TLI routine that blocks for an event in a special non-blocking mode. For example, `t_listen`(3NSL) normally blocks for a connect request. A server can periodically poll a transport endpoint for queued connect requests by calling `t_listen`(3NSL) in the non-blocking (or asynchronous) mode. You enable the asynchronous mode by setting `O_NDELAY` or `O_NONBLOCK` in the file descriptor. Set these modes as a flag through `t_open`(3NSL), or by calling `fcntl`(2) before calling the XTI/TLI routine. Use `fcntl`(2) to enable or disable this mode at any time. All program examples in this chapter use the default synchronous processing mode.

Use of `O_NDELAY` or `O_NONBLOCK` affects each XTI/TLI routine differently. You need to determine the exact semantics of `O_NDELAY` or `O_NONBLOCK` for a particular routine.

Advanced XTI/TLI Programming Example

Example 8–2 demonstrates two important concepts. The first is a server’s ability to manage multiple outstanding connect requests. The second is event-driven use of XTI/TLI and the system call interface.

By using XTI/TLI, a server can manage multiple outstanding connect requests. One reason to receive several simultaneous connect requests is to prioritize the clients. A server can receive several connect requests, and accept them in an order based on the priority of each client.

The second reason for handling several outstanding connect requests is to overcome the limits of single-threaded processing. Depending on the transport provider, while a server is processing one connect request, other clients see the server as busy. If multiple connect requests are processed simultaneously, the server is busy only if more than the maximum number of clients try to call the server simultaneously.

The server example is event-driven: the process polls a transport endpoint for incoming XTI/TLI events and takes the appropriate actions for the event received. The example following demonstrates the ability to poll multiple transport endpoints for incoming events.

EXAMPLE 8–2 Endpoint Establishment (Convertible to Multiple Connections)

```
#include <tiuser.h>
#include <fcntl.h>
#include <stdio.h>
#include <poll.h>
#include <stropts.h>
#include <signal.h>

#define NUM_FDS 1
#define MAX_CONN_IND 4
#define SRV_ADDR 1                 /* server's well known address */

int conn_fd;                       /* server connection here */
extern int t_errno;
/* holds connect requests */
struct t_call *calls[NUM_FDS][MAX_CONN_IND];

main()
{
   struct pollfd pollfds[NUM_FDS];
   struct t_bind *bind;
   int i;

   /*
    * Only opening and binding one transport endpoint, but more can
    * be supported
    */
   if ((pollfds[0].fd = t_open("/dev/tivc", O_RDWR,
         (struct t_info *) NULL)) == -1) {
      t_error("t_open failed");
      exit(1);
   }
   if ((bind = (struct t_bind *) t_alloc(pollfds[0].fd, T_BIND,
         T_ALL)) == (struct t_bind *) NULL) {
      t_error("t_alloc of t_bind structure failed");
      exit(2);
   }
   bind->qlen = MAX_CONN_IND;
   bind->addr.len = sizeof(int);
   *(int *) bind->addr.buf = SRV_ADDR;
   if (t_bind(pollfds[0].fd, bind, bind) == -1) {
      t_error("t_bind failed");
```

EXAMPLE 8–2 Endpoint Establishment (Convertible to Multiple Connections) *(Continued)*

```
      exit(3);
   }
   /* Was the correct address bound? */
   if (bind->addr.len != sizeof(int) ||
      *(int *)bind->addr.buf != SRV_ADDR) {
      fprintf(stderr, "t_bind bound wrong address\n");
      exit(4);
   }
}
```

The file descriptor returned by `t_open`(3NSL) is stored in a `pollfd` structure that controls polling of the transport endpoints for incoming data. See the `poll`(2) man page. Only one transport endpoint is established in this example. However, the remainder of the example is written to manage multiple transport endpoints. Several endpoints could be supported with minor changes to Example 8–2.

This server sets `qlen` to a value greater than `1` for `t_bind`(3NSL). This value specifies that the server should queue multiple outstanding connect requests. The server accepts the current connect request before accepting additional connect requests. This example can queue up to `MAX_CONN_IND` connect requests. The transport provider can negotiate the value of `qlen` to be smaller if the provider cannot support `MAX_CONN_IND` outstanding connect requests.

After the server binds its address and is ready to process connect requests, it behaves as shown in the following example.

EXAMPLE 8–3 Processing Connection Requests

```
pollfds[0].events = POLLIN;

while (TRUE) {
    if (poll(pollfds, NUM_FDS, -1) == -1) {
   perror("poll failed");
   exit(5);
    }
    for (i = 0; i < NUM_FDS; i++) {
   switch (pollfds[i].revents) {
      default:
         perror("poll returned error event");
      exit(6);
      case 0:
         continue;
      case POLLIN:
         do_event(i, pollfds[i].fd);
         service_conn_ind(i, pollfds[i].fd);
       }
   }
}
```

The events field of the pollfd structure is set to POLLIN, which notifies the server of any incoming XTI/TLI events. The server then enters an infinite loop in which it polls the transport endpoints for events, and processes events as they occur.

The poll(2) call blocks indefinitely for an incoming event. On return, the server checks the value of revents for each entry, one per transport endpoint, for new events. If revents is 0, the endpoint has generated no events and the server continues to the next endpoint. If revents is POLLIN, there is an event on the endpoint. The server calls do_event to process the event. Any other value in revents indicates an error on the endpoint, and the server exits. With multiple endpoints, the server should close this descriptor and continue.

Each time the server iterates the loop, it calls service_conn_ind to process any outstanding connect requests. If another connect request is pending, service_conn_ind saves the new connect request and responds to it later.

The server calls do_event in the following example to process an incoming event.

EXAMPLE 8–4 Event Processing Routine

```
do_event( slot, fd)
int slot;
int fd;
{
   struct t_discon *discon;
   int i;

   switch (t_look(fd)) {
   default:
      fprintf(stderr, "t_look: unexpected event\n");
      exit(7);
   case T_ERROR:
      fprintf(stderr, "t_look returned T_ERROR event\n");
      exit(8);
   case -1:
      t_error("t_look failed");
      exit(9);
   case 0:
      /* since POLLIN returned, this should not happen */
      fprintf(stderr,"t_look returned no event\n");
      exit(10);
   case T_LISTEN:
      /* find free element in calls array */
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (calls[slot][i] == (struct t_call *) NULL)
            break;
      }
      if ((calls[slot][i] = (struct t_call *) t_alloc( fd, T_CALL,
               T_ALL)) == (struct t_call *) NULL) {
         t_error("t_alloc of t_call structure failed");
         exit(11);
      }
      if (t_listen(fd, calls[slot][i] ) == -1) {
```

EXAMPLE 8–4 Event Processing Routine *(Continued)*

```
         t_error("t_listen failed");
         exit(12);
      }
      break;
   case T_DISCONNECT:
      discon = (struct t_discon *) t_alloc(fd, T_DIS, T_ALL);
      if (discon == (struct t_discon *) NULL) {
         t_error("t_alloc of t_discon structure failed");
         exit(13)
      }
      if(t_rcvdis( fd, discon) == -1) {
         t_error("t_rcvdis failed");
         exit(14);
      }
      /* find call ind in array and delete it */
      for (i = 0; i < MAX_CONN_IND; i++) {
         if (discon->sequence == calls[slot][i]->sequence) {
            t_free(calls[slot][i], T_CALL);
            calls[slot][i] = (struct t_call *) NULL;
         }
      }
      t_free(discon, T_DIS);
      break;
   }
}
```

The arguments in Example 8–4 are a number (*slot*) and a file descriptor (*fd*). A *slot* is the index into the global array `calls`, which has an entry for each transport endpoint. Each entry is an array of `t_call` structures that hold incoming connect requests for the endpoint.

The `do_event` module calls `t_look`(3NSL) to identify the XTI/TLI event on the endpoint specified by *fd*. If the event is a connect request (`T_LISTEN` event) or disconnect request (`T_DISCONNECT` event), the event is processed. Otherwise, the server prints an error message and exits.

For connect requests, `do_event` scans the array of outstanding connect requests for the first free entry. A `t_call` structure is allocated for the entry, and the connect request is received by `t_listen`(3NSL). The array is large enough to hold the maximum number of outstanding connect requests. The processing of the connect request is deferred.

A disconnect request must correspond to an earlier connect request. The `do_event` module allocates a `t_discon` structure to receive the request. This structure has the following fields:

```
struct t_discon {
     struct netbuf udata;
     int reason;
     int sequence;
```

```
}
```

The udata structure contains any user data sent with the disconnect request. The value of reason contains a protocol-specific disconnect reason code. The value of sequence identifies the connect request that matches the disconnect request.

The server calls t_rcvdis(3NSL) to receive the disconnect request. The array of connect requests is scanned for one that contains the sequence number that matches the sequence number in the disconnect request. When the connect request is found, its structure is freed and the entry is set to NULL.

When an event is found on a transport endpoint, service_conn_ind is called to process all queued connect requests on the endpoint, as the following example shows.

EXAMPLE 8–5 Process All Connect Requests

```
service_conn_ind(slot, fd)
{
   int i;

    for (i = 0; i < MAX_CONN_IND; i++) {
      if (calls[slot][i] == (struct t_call *) NULL)
         continue;
      if((conn_fd = t_open( "/dev/tivc", O_RDWR,
            (struct t_info *) NULL)) == -1) {
         t_error("open failed");
         exit(15);
      }
      if (t_bind(conn_fd, (struct t_bind *) NULL,
            (struct t_bind *) NULL) == -1) {
         t_error("t_bind failed");
         exit(16);
      }
      if (t_accept(fd, conn_fd, calls[slot][i]) == -1) {
         if (t_errno == TLOOK) {
            t_close(conn_fd);
            return;
         }
         t_error("t_accept failed");
         exit(167);
      }
      t_free(calls[slot][i], T_CALL);
      calls[slot][i] = (struct t_call *) NULL;
      run_server(fd);
   }
}
```

For each transport endpoint, the array of outstanding connect requests is scanned. For each request, the server opens a responding transport endpoint, binds an address to the endpoint, and accepts the connection on the endpoint. If another connect or disconnect request arrives before the current request is accepted, `t_accept`(3NSL) fails and sets `t_errno` to `TLOOK`. You cannot accept an outstanding connect request if any pending connect request events or disconnect request events exist on the transport endpoint.

If this error occurs, the responding transport endpoint is closed and `service_conn_ind` returns immediately, saving the current connect request for later processing. This activity causes the server's main processing loop to be entered, and the new event is discovered by the next call to `poll`(2). In this way, the user can queue multiple connect requests.

Eventually, all events are processed, and `service_conn_ind` is able to accept each connect request in turn.

Asynchronous Networking

This section discusses the techniques of asynchronous network communication using XTI/TLI for real-time applications. The SunOS platform provides support for asynchronous network processing of XTI/TLI events using a combination of STREAMS asynchronous features and the non-blocking mode of the XTI/TLI library routines.

Networking Programming Models

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

Synchronous networking proceeds similar to synchronous file and device I/O. Like the `write`(2) interface, the send request returns after buffering the message, but might suspend the calling process if buffer space is not immediately available. Like the `read`(2) interface, a receive request suspends execution of the calling process until data arrives to satisfy the request. Because there are no guaranteed bounds for transport services, synchronous networking is inappropriate for processes that must have real-time behavior with respect to other devices.

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

Asynchronous Network Transfers

Although an application can use `poll`(2) to check periodically for the arrival of data or to wait for the receipt of data on an endpoint, receiving asynchronous notification when data arrives might be necessary. Use `ioctl`(2) 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.

```
#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 not only for the data transfer, but also 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`(2), 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 `t_connect`(3NSL) to initiate setting up the transfer. Then `t_rcvconnect`(3NSL) 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`(2) or an asynchronous notification indicates that a connection request has arrived, the process can get the connection request by using `t_listen`(3NSL) . To accept the connection, the process uses `t_accept`(3NSL) . The responding endpoint must be separately configured for asynchronous data transfers.

The following example illustrates 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.

```
#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 a request to open a file is being processed, the application is unable to achieve real-time response to other events. The SunOS software solves this problem by having a second process handle the actual opening of the file, then passes the file descriptor to the real-time process.

Transferring a File Descriptor

The STREAMS interface provided by the SunOS platform provides a mechanism for passing an open file descriptor from one process to another. The process with the open file descriptor uses `ioctl`(2) with a command argument of `I_SENDFD`. The second process obtains the file descriptor by calling `ioctl`(2) with a command argument of `I_RECVFD`.

In the following example, the parent process prints out information about the test file, and creates a pipe. Next, the parent creates a child process that 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.

EXAMPLE 8–6 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];
```

EXAMPLE 8–6 File Descriptor Transfer *(Continued)*

```
    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=%ld, dev=%lx, rdev=%lx\n",
        f, s->st_mode, s->st_ino, s->st_dev, s->st_rdev);
    fflush(stdout);
}
```

State Transitions

The tables in the following sections describe all state transitions associated with XTI/TLI.

XTI/TLI States

The following table defines the states used in XTI/TLI state transitions, along with the service types.

TABLE 8–1 XTI/TLI State Transitions and Service Types

State	Description	Service Type
T_UNINIT	Uninitialized–initial and final state of interface	T_COTS, T_COTS_ORD, T_CLTS
T_UNBND	Initialized but not bound	T_COTS, T_COTS_ORD, T_CLTS
T_IDLE	No connection established	T_COTS, T_COTS_ORD, T_CLTS
T_OUTCON	Outgoing connection pending for client	T_COTS, T_COTS_ORD
T_INCON	Incoming connection pending for server	T_COTS, T_COTS_ORD
T_DATAXFER	Data transfer	T_COTS, T_COTS_ORD

TABLE 8–1 XTI/TLI State Transitions and Service Types *(Continued)*

State	Description	Service Type
T_OUTREL	Outgoing orderly release (waiting for orderly release request)	T_COTS_ORD
T_INREL	Incoming orderly release (waiting to send orderly release request)	T_COTS_ORD

Outgoing Events

The outgoing events described in the following table correspond to the status returned from the specified transport routines, where these routines send a request or response to the transport provider. In the table, some events, such as "accept," are distinguished by the context in which they occur. The context is based on the values of the following variables:

- *ocnt* – Count of outstanding connect requests
- *fd* – File descriptor of the current transport endpoint
- *resfd* – File descriptor of the transport endpoint where a connection is accepted

TABLE 8–2 Outgoing Events

Event	Description	Service Type
opened	Successful return of t_open(3NSL)	T_COTS, T_COTS_ORD, T_CLTS
bind	Successful return of t_bind(3NSL)	T_COTS, T_COTS_ORD, T_CLTS
optmgmt	Successful return of t_optmgmt(3NSL)	T_COTS, T_COTS_ORD, T_CLTS
unbind	Successful return of t_unbind(3NSL)	T_COTS, T_COTS_ORD, T_CLTS
closed	Successful return of t_close(3NSL)	T_COTS, T_COTS_ORD, T_CLT
connect1	Successful return of t_connect(3NSL) in synchronous mode	T_COTS, T_COTS_ORD
connect2	TNODATA error on t_connect(3NSL) in asynchronous mode, or TLOOK error due to a disconnect request arriving on the transport endpoint	T_COTS, T_COTS_ORD
accept1	Successful return of t_accept(3NSL) with ocnt == 1, fd == resfd	T_COTS, T_COTS_ORD
accept2	Successful return of t_accept(3NSL) with ocnt== 1, fd!= resfd	T_COTS, T_COTS_ORD

TABLE 8–2 Outgoing Events *(Continued)*

Event	Description	Service Type
accept3	Successful return of t_accept(3NSL) with ocnt > 1	T_COTS, T_COTS_ORD
snd	Successful return of t_snd(3NSL)	T_COTS, T_COTS_ORD
snddis1	Successful return of t_snddis(3NSL) with ocnt <= 1	T_COTS, T_COTS_ORD
snddis2	Successful return of t_snddis(3NSL) with ocnt > 1	T_COTS, T_COTS_ORD
sndrel	Successful return of t_sndrel(3NSL)	T_COTS_ORD
sndudata	Successful return of t_sndudata(3NSL)	T_CLTS

Incoming Events

The incoming events correspond to the successful return of the specified routines. These routines return data or event information from the transport provider. The only incoming event not associated directly with the return of a routine is pass_conn, which occurs when a connection is transferred to another endpoint. The event occurs on the endpoint that is being passed the connection, although no XTI/TLI routine is called on the endpoint.

In the following table, the rcvdis events are distinguished by the value of ocnt, the count of outstanding connect requests on the endpoint.

TABLE 8–3 Incoming Events

Event	Description	Service Type
listen	Successful return of t_listen(3NSL)	T_COTS, T_COTS_ORD
rcvconnect	Successful return of t_rcvconnect(3NSL)	T_COTS, T_COTS_ORD
rcv	Successful return of t_rcv(3NSL)	T_COTS, T_COTS_ORD
rcvdis1	Successful return of t_rcvdis(3NSL) rcvdis1t_rcvdis(), onct <= 0	T_COTS, T_COTS_ORD
rcvdis2	Successful return of t_rcvdis(3NSL), ocnt == 1	T_COTS, T_COTS_ORD
rcvdis3	Successful return of t_rcvdis(3NSL) with ocnt > 1	T_COTS, T_COTS_ORD
rcvrel	Successful return of t_rcvrel(3NSL)	T_COTS_ORD

TABLE 8–3 Incoming Events *(Continued)*

Event	Description	Service Type
rcvudata	Successful return of t_rcvudata(3NSL)	T_CLTS
rcvuderr	Successful return of t_rcvuderr(3NSL)	T_CLTS
pass_conn	Receive a passed connection	T_COTS, T_COTS_ORD

State Tables

The state tables describe the XTI/TLI state transitions. Each box contains the next state, given the current state (column) and the current event (row). An empty box is an invalid state/event combination. Each box can also have an action list. Actions must be done in the order specified in the box.

You should understand the following when studying the state tables:

- t_close(3NSL) terminates an established connection for a connection-oriented transport provider. The connection termination will be either orderly or abortive, depending on the service type supported by the transport provider. See the t_getinfo(3NSL) man page.
- If a transport user issues a interface call out of sequence, the interface fails and t_errno is set to TOUTSTATE. The state does not change.
- The error codes TLOOK or TNODATA after t_connect(3NSL) can result in state changes. The state tables assume correct use of XTI/TLI.
- Any other transport error does not change the state, unless the man page for the interface says otherwise.
- The support interfaces t_getinfo(3NSL), t_getstate(3NSL), t_alloc(3NSL), t_free(3NSL), t_sync(3NSL), t_look(3NSL), and t_error(3NSL) are excluded from the state tables because they do not affect the state.

Some of the state transitions listed in the tables below offer actions the transport user must take. Each action is represented by a digit derived from the list below:

- Set the count of outstanding connect requests to zero
- Increment the count of outstanding connect requests
- Decrement the count of outstanding connect requests
- Pass a connection to another transport endpoint, as indicated in thet_accept(3NSL) man page

The following table shows endpoint establishment states.

TABLE 8–4 Connection Establishment State

Event/State	T_UNINIT	T_UNBND	T_IDLE
opened	T_UNBND		
bind		T_IDLE[1]	
optmgmt (TLI only)			T_IDLE
unbind			T_UNBND
closed		T_UNINIT	

The following table shows data transfer in connectionless mode.

TABLE 8–5 Connection Mode State—Part 1

Event/State	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
connect1	T_DATAXFER			
connect2	T_OUTCON			
rcvconnect		T_DATAXFER		
listen	T_INCON [2]		T_INCON [2]	
accept1			T_DATAXFER [3]	
accept2			T_IDLE [3] [4]	
accept3			T_INCON [3] [4]	
snd				T_DATAXFER
rcv				T_DATAXFER
snddis1		T_IDLE	T_IDLE [3]	T_IDLE
snddis2			T_INCON [3]	
rcvdis1		T_IDLE		T_IDLE
rcvdis2			T_IDLE [3]	
rcvdis3			T_INCON [3]	
sndrel				T_OUTREL
rcvrel				T_INREL
pass_conn	T_DATAXFER			
optmgmt	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER

TABLE 8–5 Connection Mode State—Part 1 *(Continued)*

Event/State	T_IDLE	T_OUTCON	T_INCON	T_DATAXFER
closed	T_UNINIT	T_UNINIT	T_UNINIT	T_UNINIT

The following table shows connection establishment/connection release/data transfer in connection mode.

TABLE 8–6 Connection Mode State—Part 2

Event/State	T_OUTREL	T_INREL	T_UNBND
connect1			
connect2			
rcvconnect			
listen			
accept1			
accept2			
accept3			
snd		T_INREL	
rcv	T_OUTREL		
snddis1	T_IDLE	T_IDLE	
snddis2			
rcvdis1	T_IDLE	T_IDLE	
rcvdis2			
rcvdis3			
sndrel		T_IDLE	
rcvrel	T_IDLE		
pass_conn			T_DATAXFER
optmgmt	T_OUTREL	T_INREL	T_UNBND
closed	T_UNINIT	T_UNINIT	

The following table shows connectionless mode states.

TABLE 8–7 Connectionless Mode State

Event/State	T_IDLE
snudata	T_IDLE
rcvdata	T_IDLE
rcvuderr	T_IDLE

Guidelines to Protocol Independence

The set of XTI/TLI services, common to many transport protocols, offers protocol independence to applications. Not all transport protocols support all XTI/TLI services. If software must run in a variety of protocol environments, use only the common services.

The following is a list of services that might not be common to all transport protocols.

- In connection mode service, a transport service data unit (TSDU) might not be supported by all transport providers. Make no assumptions about preserving logical data boundaries across a connection.
- Protocol and implementation-specific service limits are returned by the `t_open`(3NSL) and `t_getinfo`(3NSL) routines. Use these limits to allocate buffers to store protocol-specific transport addresses and options.
- Do not send user data with connect requests or disconnect requests, such as `t_connect`(3NSL) and `t_snddis`(3NSL). Not all transport protocols can use this method.
- The buffers in the `t_call` structure used for `t_listen`(3NSL) must be large enough to hold any data sent by the client during connection establishment. Use the `T_ALL` argument to `t_alloc`(3NSL) to set maximum buffer sizes to store the address, options, and user data for the current transport provider.
- Do not specify a protocol address on `t_bind`(3NSL) on a client-side endpoint. The transport provider should assign an appropriate address to the transport endpoint. A server should retrieve its address for `t_bind`(3NSL) in a way that does not require knowledge of the transport provider's name space.
- Do not make assumptions about formats of transport addresses. Transport addresses should not be constants in a program. Chapter 9 contains detailed information about transport selection.
- The reason codes associated with `t_rcvdis`(3NSL) are protocol-dependent. Do not interpret these reason codes if protocol independence is important.
- The `t_rcvuderr`(3NSL) error codes are protocol dependent. Do not interpret these error codes if protocol independence is a concern.

- Do not code the names of devices into programs. The device node identifies a particular transport provider and is not protocol independent. See Chapter 9 for details regarding transport selection.
- Do not use the optional orderly release facility of the connection mode service, provided by `t_sndrel`(3NSL) and `t_rcvrel`(3NSL), in programs targeted for multiple protocol environments. This facility is not supported by all connection-based transport protocols. Using the facility can prevent programs from successfully communicating with open systems.

XTI/TLI Versus Socket Interfaces

XTI/TLI and sockets are different methods of handling the same tasks. Although they provide mechanisms and services that are functionally similar, they do not provide one-to-one compatibility of routines or low-level services. Observe the similarities and differences between the XTI/TLI and socket-based interfaces before you decide to port an application.

The following issues are related to transport independence, and can have some bearing on RPC applications:

- Privileged ports – Privileged ports are an artifact of the Berkeley Software Distribution (BSD) implementation of the TCP/IP Internet Protocols. These ports are not portable. The notion of privileged ports is not supported in the transport-independent environment.
- Opaque addresses – Separating the portion of an address that names a host from the portion of an address that names the service at that host cannot be done in a transport-independent fashion. Be sure to change any code that assumes it can discern the host address of a network service.
- Broadcast – No transport-independent form of broadcast address exists.

Socket-to-XTI/TLI Equivalents

The following table shows approximate equivalents between XTI/TLI interfaces and socket interfaces. The comment field describes the differences. If the comment column is blank, either the interfaces are similar or no equivalent interface exists in either interface.

TABLE 8–8 TLI and Socket Equivalent Functions

TLI interface	Socket interface	Comments
t_open(3NSL)	socket(3SOCKET)	
–	socketpair(3SOCKET)	
t_bind(3NSL)	bind(3SOCKET)	t_bind(3NSL) sets the queue depth for passive sockets, but bind(3SOCKET) does not. For sockets, the queue length is specified in the call to listen(3SOCKET).
t_optmgmt(3NSL)	getsockopt(3SOCKET) setsockopt(3SOCKET)	t_optmgmt(3NSL) manages only transport options. getsockopt(3SOCKET) and setsockopt(3SOCKET) can manage options at the transport layer, but also at the socket layer and at the arbitrary protocol layer.
t_unbind(3NSL)	–	
t_close(3NSL)	close(2)	
t_getinfo(3NSL)	getsockopt(3SOCKET)	t_getinfo(3NSL) returns information about the transport. getsockopt(3SOCKET) can return information about the transport and the socket.
t_getstate(3NSL)	-	
t_sync(3NSL)	-	
t_alloc(3NSL)	-	
t_free(3NSL)	-	
t_look(3NSL)	-	getsockopt(3SOCKET) with the SO_ERROR option returns the same kind of error information as t_look(3NSL)t_look().
t_error(3NSL)	perror(3C)	
t_connect(3NSL)	connect(3SOCKET)	You do not need to bind the local endpoint before invoking connect(3SOCKET). Bind the endpoint before calling t_connect(3NSL). You can use connect(3SOCKET) on a connectionless endpoint to set the default destination address for datagrams. You can send data using connect(3SOCKET).
t_rcvconnect(3NSL)	-	

TABLE 8–8 TLI and Socket Equivalent Functions *(Continued)*

TLI interface	Socket interface	Comments
t_listen(3NSL)	listen(3SOCKET)	t_listen(3NSL) waits for connection indications. listen(3SOCKET) sets the queue depth.
t_accept(3NSL)	accept(3SOCKET)	
t_snd(3NSL)	send(3SOCKET)	
	sendto(3SOCKET)	
	sendmsg(3SOCKET)	sendto(3SOCKET) and sendmsg(3SOCKET) operate in connection mode as well as in datagram mode.
t_rcv(3NSL)	recv(3SOCKET)	
	recvfrom(3SOCKET)	
	recvmsg(3SOCKET)	recvfrom(3SOCKET) and recvmsg(3SOCKET) operate in connection mode as well as datagram mode.
t_snddis(3NSL)	-	
t_rcvdis(3NSL)	-	
t_sndrel(3NSL)	shutdown(3SOCKET)	
t_rcvrel(3NSL)	-	
t_sndudata(3NSL)	sendto(3SOCKET)	
	recvmsg(3SOCKET)	
t_rcvuderr(3NSL)	-	
read(2), write(2)	read(2), write(2)	In XTI/TLI you must push the tirdwr(7M) module before calling read(2) or write(2). In sockets, calling read(2) or write(2) suffices.

Additions to the XTI Interface

The XNS 5 (Unix98) standard introduces some new XTI interfaces. These are briefly described below. You can find the details in the relevant manual pages. These interfaces are not available for TLI users. The scatter-gather data transfer interfaces are:

t_sndvudata(3NSL) Send a data unit from one or more non-contiguous buffers

`t_rcvvudata`(3NSL)	Receive a data unit into one or more non-contiguous buffers
`t_sndv`(3NSL)	Send data or expedited data from one or more non-contiguous buffers on a connection
`t_rcvv`(3NSL)	Receive data or expedited data sent over a connection and put the data into one or more non-contiguous buffers

The XTI utility interface `t_sysconf`(3NSL) gets configurable XTI variables. The `t_sndreldata`(3NSL) interface initiates and responds to an orderly release with user data. The `t_rcvreldata`(3NSL) receives an orderly release indication or confirmation containing user data.

Note – The additional interfaces `t_sndreldata`(3NSL) and `t_rcvreldata`(3NSL) are used only with a specific transport called “minimal OSI,” which is not available on the Solaris platform. These interfaces are not available for use in conjunction with Internet Transports (TCP or UDP).

CHAPTER 9

Transport Selection and Name-to-Address Mapping

This chapter describes selecting transports and resolving network addresses. This chapter further describes interfaces that enable you to specify the available communication protocols for an application. The chapter also explains additional interfaces that provide direct mapping of names to network addresses.

- "Transport Selection" on page 199
- "Name-to-Address Mapping" on page 200

Note – In this chapter, the terms *network* and *transport* are used interchangeably. The terms refer to the programmatic interface that conforms to the transport layer of the OSI Reference Mode. The term *network* is also used to refer to the physical collection of computers that are connected through some electronic medium.

Transport Selection

Caution – The interfaces that are described in this chapter are multithread safe. "Multithread safe" means that you can use applications that contain transport selection interface calls freely in a multithreaded application. These interface calls do not provide linear scalability because the calls are not re-entrant.

A distributed application must use a standard interface to the transport services to be portable to different protocols. Transport selection services provide an interface that allows an application to select which protocols to use. This interface makes an application independent of protocol and medium.

Transport selection means that a client application can easily try each available transport until the client establishes communication with a server. Transport selection enables request acceptance on multiple transports by server applications. The applications can then communicate over a number of protocols. Transports can be tried in either the order specified by the local default sequence or in an order specified by the user.

Choosing from the available transports is the responsibility of the application. The transport selection mechanism makes that selection uniform and simple.

Name-to-Address Mapping

Name-to-address mapping enables an application to obtain the address of a service on a specified host independent of the transport used. Name-to-address mapping consists of the following interfaces:

`netdir_getbyname`(3NSL)	Maps the host and service name to a set of addresses
`netdir_getbyaddr`(3NSL)	Maps addresses into host and service names
`netdir_free`(3NSL)	Frees structures allocated by the name-to-address translation routines
`taddr2uaddr`(3NSL)	Translates an address and returns a transport-independent character representation of the address
`uaddr2taddr`(3NSL)	The universal address is translated into a `netbuf` structure
`netdir_options`(3NSL)	Interfaces to transport-specific capabilities such as the broadcast address and reserved port facilities of TCP and UDP
`netdir_perror`(3NSL)	Displays a message stating why one of the routines that map name-to-address failed on `stderr`.
`netdir_sperror`(3NSL)	Returns a string containing the error message stating why one of the routines that map name-to-address failed.

The first argument of each routine points to a `netconfig`(4) structure that describes a transport. The routine uses the array of directory-lookup library paths in the `netconfig`(4) structure to call each path until the translation succeeds.

The name-to-address libraries are described in Table 9–1. The routines that are described in “Using the Name-to-Address Mapping Routines” on page 202 are defined in the `netdir`(3NSL) man page.

Note – The following libraries no longer exist in the Solaris™ environment: `tcpip.so`, `switch.so`, and `nis.so`. For more information on this change, see the `nsswitch.conf`(4) man page and the NOTES section of the `gethostbyname`(3NSL) man page.

TABLE 9–1 Name-to-Address Libraries

Library	Transport Family	Description
-	`inet`	The name-to-address mapping for networks of the protocol family `inet` is provided by the name service switch based on the entries for *hosts* and *services* in the file `nsswitch.conf`(4). For networks of other families, the dash indicates a nonfunctional name-to-address mapping.
`straddr.so`	`loopback`	Contains the routines that map name-to-address in any protocol that accepts strings as addresses, such as the loopback transports.

`straddr.so` Library

Name–to–address translation files for the `straddr.so` library are created by the system administrator. The system administrator also maintains these translation files. The `straddr.so` files are `/etc/net/`*transport-name*`/hosts` and `/etc/net/`*transport-name*`/services`. *transport-name* is the local name of the transport that accepts string addresses, which is specified in the *network ID* field of the `/etc/netconfig` file. For example, the host file for `ticlts` would be `/etc/net/ticlts/hosts`, and the service file for `ticlts` would be `/etc/net/ticlts/services`.

Most string addresses do not distinguish between *host* and *service*. However, separating the string into a host part and a service part is consistent with other transports. The `/etc/net/`*transport-name*`/hosts` file contains a text string that is assumed to be the host address, followed by the host name:

```
joyluckaddr      joyluck
carpediemaddr    carpediem
thehopaddr       thehop
pongoaddr        pongo
```

Because loopback transports cannot go outside the containing host, listing other hosts makes no sense.

The /etc/net/transport-name/services file contains service names followed by strings that identify the service address:

```
rpcbind    rpc
listen    serve
```

The routines create the full-string address by concatenating the host address, a period (.), and the service address. For example, the address of the listen service on pongo is pongoaddr.serve.

When an application requests the address of a service on a particular host on a transport that uses this library, the host name must be in /etc/net/*transport*/hosts. The service name must be in /etc/net/*transport*/services. If either name is missing, the name-to-address translation fails.

Using the Name-to-Address Mapping Routines

This section is an overview of the mapping routines that are available for use. The routines return or convert the network names to their respective network addresses. Note that netdir_getbyname(3NSL), netdir_getbyaddr(3NSL), and taddr2uaddr(3NSL) return pointers to data that must be freed by calls to netdir_free(3NSL).

```
int netdir_getbyname(struct netconfig *nconf,
        struct nd_hostserv *service, struct nd_addrlist **addrs);
```

netdir_getbyname(3NSL) maps the host and service name specified in *service* to a set of addresses that are consistent with the transport identified in *nconf*. The nd_hostserv and nd_addrlist structures are defined in the netdir(3NSL) man page. A pointer to the addresses is returned in *addrs*.

To find all addresses of a host and service on all available transports, call netdir_getbyname(3NSL) with each netconfig(4) structure returned by either getnetpath(3NSL) or getnetconfig(3NSL).

```
int netdir_getbyaddr(struct netconfig *nconf,
        struct nd_hostservlist **service, struct netbuf *netaddr);
```

netdir_getbyaddr(3NSL) maps addresses into host and service names. The interface is called with an address in *netaddr* and returns a list of host-name and service-name pairs in *service*. The nd_hostservlist structure is defined in netdir(3NSL).

```
void netdir_free(void *ptr, int struct_type);
```

The netdir_free(3NSL) routine frees structures allocated by the name-to-address translation routines. The parameters can take the values that are shown in the following table.

TABLE 9–2 `netdir_free`(3NSL) Routines

struct_type	ptr
ND_HOSTSERV	Pointer to an `nd_hostserv` structure
ND_HOSTSERVLIST	Pointer to an `nd_hostservlist` structure
ND_ADDR	Pointer to a `netbuf` structure
ND_ADDRLIST	Pointer to an `nd_addrlist` structure

```
char *taddr2uaddr(struct netconfig *nconf, struct netbuf *addr);
```

`taddr2uaddr`(3NSL) translates the address pointed to by *addr* and returns a transport-independent character representation of the address. This character representation is called a universal address. The value that is given in *nconf* specifies the transport for which the address is valid. The universal address can be freed by `free`(3C).

```
struct netbuf *uaddr2taddr(struct netconfig *nconf, char *uaddr);
```

The universal address pointed to by *uaddr* is translated into a `netbuf` structure. *nconf* specifies the transport for which the address is valid.

```
int netdir_options(const struct netconfig *config,
          const int option, const int fildes, char *point_to_args);
```

`netdir_options`(3NSL) provides interfaces to transport-specific capabilities, such as the broadcast address and reserved port facilities of TCP and UDP. The value of *nconf* specifies a transport, while *option* specifies the transport-specific action to take. The value in *option* might disable consideration of the value in *fd*. The fourth argument points to operation-specific data.

The following table shows the values used for `option`.

TABLE 9–3 Values for `netdir_options`

Option	Description
ND_SET_BROADCAST	Sets the transport for broadcast if the transport supports broadcast
ND_SET_RESERVEDPORT	Enables application binding to reserved ports if allowed by the transport
ND_CHECK_RESERVEDPORT	Verifies that an address corresponds to a reserved port if the transport supports reserved ports
ND_MERGEADDR	Transforms a locally meaningful address into an address to which client hosts can connect

The `netdir_perror`(3NSL) routine displays a message stating why one of the routines that map name-to-address failed on `stderr`.

```
void netdir_perror(char *s);
```

The `netdir_sperror`(3NSL) routine returns a string containing the error message stating why one of the routines that map name-to-address failed.

```
char *netdir_sperror(void);
```

The following example shows network selection and name-to-address mapping.

EXAMPLE 9–1 Network Selection and Name-to-Address Mapping

```
#include <netconfig.h>
#include <netdir.h>
#include <sys/tiuser.h>

struct nd_hostserv nd_hostserv;   /* host and service information */
struct nd_addrlist *nd_addrlistp; /* addresses for the service */
struct netbuf *netbufp;           /* the address of the service */
struct netconfig *nconf;          /* transport information*/
int i;                            /* the number of addresses */
char *uaddr;                      /* service universal address */
void *handlep;                    /* a handle into network selection */
/*
 * Set the host structure to reference the "date"
 * service on host "gandalf"
 */
nd_hostserv.h_host = "gandalf";
nd_hostserv.h_serv = "date";
/*
 * Initialize the network selection mechanism.
 */
if ((handlep = setnetpath()) == (void *)NULL) {
   nc_perror(argv[0]);
   exit(1);
}
/*
 * Loop through the transport providers.
 */
while ((nconf = getnetpath(handlep)) != (struct netconfig *)NULL)
{
   /*
    * Print out the information associated with the
    * transport provider described in the "netconfig"
    * structure.
    */
   printf("Transport provider name: %s\n", nconf->nc_netid);
   printf("Transport protocol family: %s\n", nconf->nc_protofmly);
   printf("The transport device file: %s\n", nconf->nc_device);
   printf("Transport provider semantics: ");
       switch (nconf->nc_semantics) {
   case NC_TPI_COTS:
      printf("virtual circuit\n");
      break;
   case NC_TPI_COTS_ORD:
      printf("virtual circuit with orderly release\n");
```

```
      break;

   case NC_TPI_CLTS:
      printf("datagram\n");
      break;
   }
   /*
    * Get the address for service "date" on the host
    * named "gandalf" over the transport provider
    * specified in the netconfig structure.
    */
   if (netdir_getbyname(nconf, &nd_hostserv, &nd_addrlistp) != ND_OK) {
      printf("Cannot determine address for service\n");
      netdir_perror(argv[0]);
      continue;
   }
   printf("<%d> addresses of date service on gandalf:\n",
      nd_addrlistp->n_cnt);
   /*
    * Print out all addresses for service "date" on
    * host "gandalf" on current transport provider.
    */
   netbufp = nd_addrlistp->n_addrs;
   for (i = 0; i < nd_addrlistp->n_cnt; i++, netbufp++) {
      uaddr = taddr2uaddr(nconf,netbufp);
      printf("%s\n",uaddr);
      free(uaddr);
   }
       netdir_free( nd_addrlistp, ND_ADDRLIST );

}
endnetconfig(handlep);
```

CHAPTER 10

Real-time Programming and Administration

This chapter describes writing and porting real-time applications to run under SunOS. This chapter is written for programmers that are experienced in writing real-time applications and for administrators familiar with real-time processing and the Solaris system.

This chapter discusses the following topics:

- Scheduling needs of real-time applications, which are covered in "The Real-Time Scheduler" on page 211.
- "Memory Locking" on page 222.
- "Asynchronous Network Communication" on page 230.

Basic Rules of Real-time Applications

Real-time response is guaranteed when certain conditions are met. This section identifies these conditions and some of the more significant design errors.

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.

A Solaris real-time process has the following characteristics:

- Runs in the RT scheduling class, as described in "The Real-Time Scheduler" on page 211
- Locks down all the memory in its process address space, as described in "Memory Locking" on page 222
- 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 209

Real-time 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 real-time scheduling of a thread, the thread must be created as a bound thread. Furthermore, 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 real-time process, the process acquires the processor within the guaranteed dispatch latency period of becoming runnable. For more information, see "Dispatch Latency" on page 211. The process continues to run for as long as it remains the highest priority runnable process.

A real-time process can lose control of the processor because of other system events. A real-time process can also be unable to gain control of the processor because of other system events. These events include external events, such as interrupts, resource starvation, waiting on external events such as synchronous I/O, and pre-emption by a higher priority process.

Real-time scheduling generally does not apply to system initialization and termination services such as `open`(2) and `close`(2).

Factors that Degrade 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.

Real-time processing can also impair the operation of aspects of other applications that are active on a system that is running a real-time application. Because real-time processes have higher priority, time-sharing processes can be prevented from running for significant amounts of time. This phenomenon can cause interactive activities, such as displays and keyboard response time, to slow noticeably.

Synchronous I/O Calls

System response under SunOS 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. Performing I/O in this fashion should be done from an independent process or independent thread. This independent process or independent thread should not run in real time.

Interrupt Servicing

Interrupt priorities are independent of process priorities. The priorities that are set for a group of processes are not inherited by the services of hardware interrupts that result from those processes' actions. As a consequence, devices controlled by high-priority real-time processes do not necessarily have high-priority interrupt processing.

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.

Real-time programs can set the environment variable `LD_BIND_NOW` to a non-`NULL` value when the program is invoked. Setting the value of this environment value allows the use of shared libraries while avoiding dynamic binding. This procedure also 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 real-time process by acquiring a resource that is required by a real-time process. Priority inversion 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. Real-time processes might miss their deadlines if this blocking is prolonged.

Consider the case that is depicted in the following figure, where a high-priority process requires a shared resource. A lower priority process holds the resource and is pre-empted by an intermediate priority process, blocking the high-priority process. Any number of intermediate processes can be involved. All intermediate processes must finish executing, as well as the lower-priority process' critical section. This series of executions can take an arbitrarily long time.

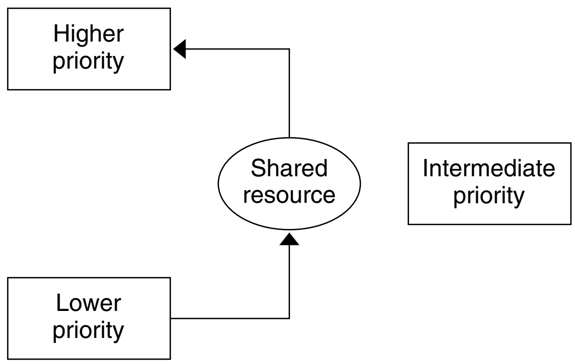


FIGURE 10–1 Unbounded Priority Inversion

This issue and the methods of dealing with this issue are described in “Mutual Exclusion Lock Attributes” in *Multithreaded Programming Guide*.

Sticky Locks

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

Runaway Real-time Processes

Runaway real-time processes can cause the system to halt. Such runaway processes can also slow the system response so much that the system appears to halt.

Note – If you have a runaway process on a SPARC system, press Stop-A. You might have to do press Stop-A more than one time. If pressing Stop-A does not work, turn the power off, wait a moment, then turn the power back on. If you have a runaway process on a non-SPARC system, turn the power off, wait a moment, then turn the power back on.

When a high priority real-time process does not relinquish control of the CPU, you must break the infinite loop in order to regain control of the system. Such a runaway process does not respond to Control-C. Attempts to use a shell set at a higher priority than the priority of the runaway process do not work.

Asynchronous I/O Behavior

Asynchronous I/O operations do not always execute in the sequence in which the operations are queued to the kernel. Asynchronous operations do not necessarily return to the caller in the sequence in which the operations were performed.

If a single buffer is specified for a rapid sequence of calls to `aioread`(3AIO), the buffer's state is uncertain. The uncertainty of the buffer's state is from the time the first call is made to the time the last result is signaled to the caller.

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

Real-time Files

SunOS provides no facilities to ensure that files are allocated as physically contiguous.

For regular files, the `read`(2) and `write`(2) operations are always buffered. An application can use `mmap`(2) and `msync`(3C) to effect direct I/O transfers between secondary storage and process memory.

The Real-Time Scheduler

Real-time scheduling constraints are necessary to manage data acquisition or process control hardware. The real-time 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 that is designed to provide a fair distribution of the processing resources to a set of time-sharing processes.

This section describes the SunOS real-time scheduler, its priority queue, and how to use system calls and utilities that control scheduling.

Dispatch Latency

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

The term *dispatch latency* describes the amount of time a system takes to respond to a request for a process to begin operation. With a scheduler that is written specifically to honor application priorities, real-time applications can be developed with a bounded dispatch latency.

The following figure illustrates the amount of time an application takes to respond to a request from an external event.

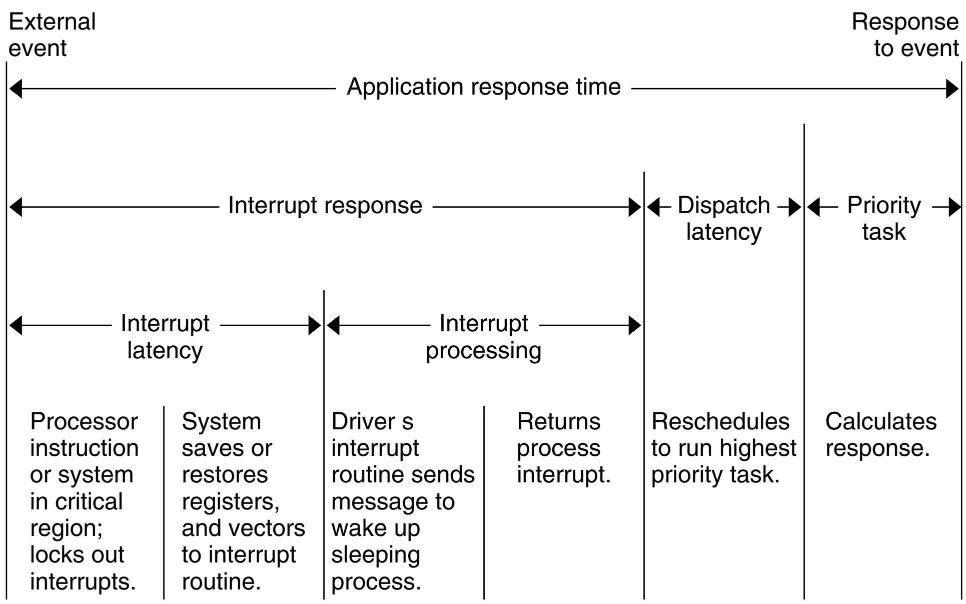


FIGURE 10–2 Application Response Time

The overall application response time consists of the interrupt response time, the dispatch latency, and the application's response time.

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 time is minimized in SunOS using synchronization primitives that do not commonly require a raised processor interrupt level.

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

Figure 10–3 illustrates the internal dispatch latency and application response time of a system. The response time is defined in terms of the amount of time a system takes to respond to an internal event. The dispatch latency of an internal event represents the amount of time that a process needs to wake up a higher priority process. The dipatch latency also includes the time that the system takes to dispatch the higher priority process.

The application response time is the amount of time that a driver takes to: wake up a higher-priority process, release resources from a low-priority process, reschedule the higher-priority task, calculate the response, and dispatch the task.

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. Therefore, this processing is not bounded by the dispatch latency guarantee.

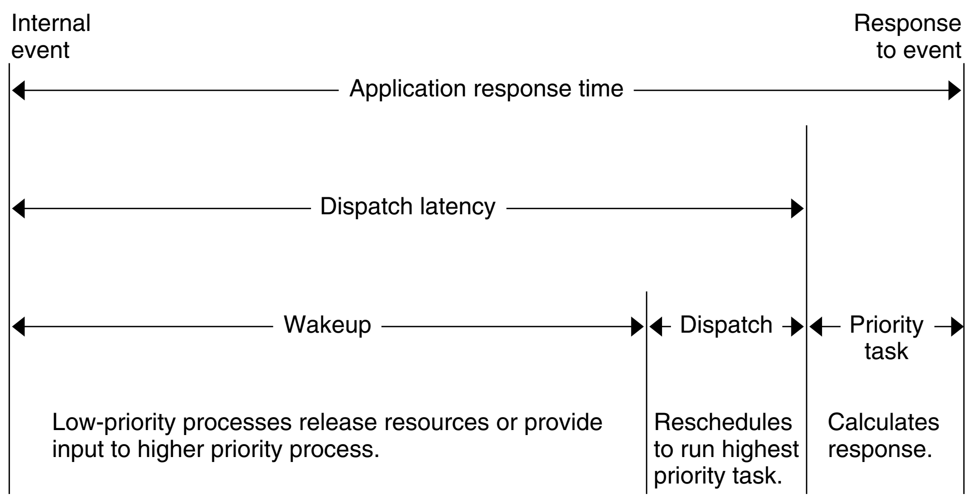


FIGURE 10–3 Internal Dispatch Latency

With the new scheduling techniques provided with real-time SunOS, the system dispatch latency time is within specified bounds. As you can see in the following table, dispatch latency improves with a bounded number of processes.

TABLE 10–1 Real-time System Dispatch Latency

Workstation	Bounded Number of Processes	Arbitrary Number of Processes
SPARCstation 2	<0.5 milliseconds in a system with fewer than 16 active processes	1.0 milliseconds

TABLE 10–1 Real-time System Dispatch Latency *(Continued)*

Workstation	Bounded Number of Processes	Arbitrary Number of Processes
SPARCstation 5	<0.3 millisecond	0.3 millisecond
Ultra 1-167	<0.15 millisecond	<0.15 millisecond

Scheduling Classes

The SunOS kernel dispatches processes by priority. The scheduler or dispatcher supports the concept of scheduling classes. Classes are defined as real-time (`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, real-time processes have precedence over `sys` and `TS` processes. Administrators can configure systems so that the priorities for `TS` processes and `RT` processes overlap.

The following figure illustrates the concept of classes as viewed by the SunOS kernel.

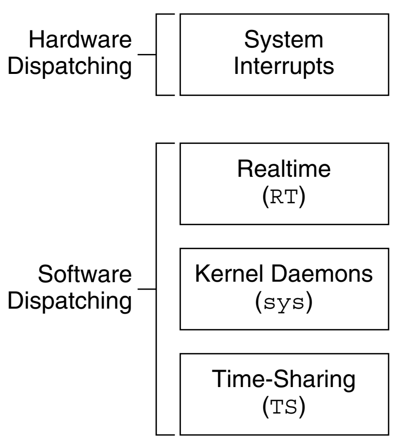


FIGURE 10–4 Dispatch Priorities for Scheduling Classes

Hardware interrupts, which cannot be controlled by software, have the highest priority. The routines that process interrupts are dispatched directly and immediately from interrupts, without regard to the priority of the current process.

Real-time 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 enables critical processes to run in a predetermined order until completion. These priorities never change unless they are changed by an application.

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. A process with a finite time quantum also stops running if the process blocks while waiting for an I/O event or is pre-empted by a higher-priority runnable real-time process. A process with an infinite time quantum ceases execution only when the process terminates, blocks, or is pre-empted.

The `SYS` class exists to schedule the execution of special system processes, such as paging, STREAMS, and the swapper. You cannot 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.

The time-sharing (`TS`) processes have the lowest priority. `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:

- The time slice value
- The process history, which records when the process was last put to sleep
- 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's 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's priority value for its class. This value is placed into the dispatch priority variable of that process.

As the following figure illustrates, each class algorithm has its own method of nominating the highest priority process to place on the global run queue.

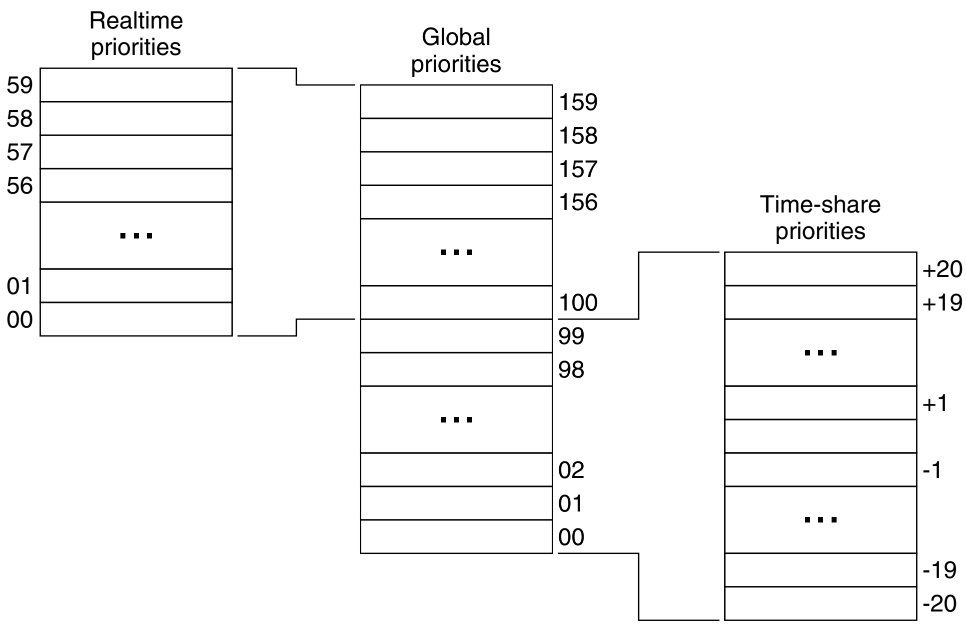


FIGURE 10–5 Kernel Dispatch Queue

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. A set of global scheduling priority maps is not required to start with zero or be contiguous.

By default, the global priority values for time-sharing (`TS`) processes range from -20 to +20. These global priority values are mapped into the kernel from 0-40, with temporary assignments as high as 99. The default priorities for real-time (`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 has class-specific information attached to the process upon invocation. A process is dispatched from the kernel dispatch table in an order that is based on the process' global priority.

Dispatching Processes

When a process is dispatched, the context of the process is mapped into memory along with its memory management information, its registers, and its stack. Execution begins after the context mapping is done. Memory management information is in the form of hardware registers that contain the data that is needed to perform virtual memory translations for the currently running process.

Process Pre-emption

When a higher priority process becomes dispatchable, the kernel interrupts its computation and forces the context switch, pre-empting the currently running process. A process can be pre-empted 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. Process B is dispatched and begins execution. Eventually, the peripheral device sends an interrupt, 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 pre-empts B from processing, resuming execution of the awakened process A.

Another interesting situation occurs when several processes contend for kernel resources. A high-priority real-time process might be waiting for a resource held by a low-priority process. When the low-priority process releases the resource, the kernel pre-empts that process to resume 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 kernel can lead to priority inversion.

A process is *blocked* when the process must wait for one or more processes to relinquish resources. Prolonged blocking can lead to missed deadlines, even for low levels of utilization.

The problem of priority inversion has been addressed for mutual-exclusion locks for the SunOS 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 inheritance 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 interface execution. User-level processes can still exhibit priority inversion, however.

User Priority Inversion

The issue of user priority inversion, and the means to deal with priority inversion, are discussed in "Mutual Exclusion Lock Attributes" in *Multithreaded Programming Guide*.

Interface Calls That Control Scheduling

The following interface calls control process scheduling.

Using `priocntl`

Control over scheduling of active classes is done with `priocntl`(2). Class attributes are inherited through `fork`(2) and `exec`(2), along with scheduling parameters and permissions required for priority control. This inheritance happens with both the `RT` and the `TS` classes.

`priocntl`(2) is the interface for specifying a real-time process, a set of processes, or a class to which the system call applies. `priocntlset`(2) also provides the more general interface for specifying an entire set of processes to which the system call applies.

The command arguments of `priocntl`(2) can be one of: `PC_GETCID`, `PC_GETCLINFO`, `PC_GETPARMS`, or `PC_SETPARMS`. The real or effective ID of the calling process must match the real or effective ID of the affected processes, or must have superuser privilege.

`PC_GETCID` — This command takes the name field of a structure that contains a recognizable class name. 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 or the class specific scheduling parameters of one of the specified processes. Even though `idtype` and `id` might specify a big set, `PC_GETPARMS` returns the parameter of only one process. The class selects the process.

`PC_SETPARMS` — This command sets the scheduling class or the class-specific scheduling parameters of the specified process or processes.

Other interface calls

`sched_get_priority_max` — Returns the maximum values for the specified policy.

sched_get_priority_min	Returns the minimum values for the specified policy. For mor einformation, see the sched_get_priority_max(3R) man page.
sched_rr_get_interval	Updates the specified timespec structure to the current execution time limit. For more information, see the sched_get_priority_max(3RT) man page.
sched_setparam, sched_getparam	Sets or gets the scheduling parameters of the specified process.
sched_yield	Blocks the calling process until the calling process returns to the head of the process list.

Utilities That Control Scheduling

The administrative utilities that control process scheduling are dispadmin(1M) and priocntl(1). Both of these utilities support the priocntl(2) system call with compatible options and loadable modules. These utilities provide system administration functions that control real-time process scheduling during runtime.

priocntl(1)

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

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 real-time class.

The class options for dispadmin(1M) are in the following list:

-l	Lists scheduler classes currently configured
-c	Specifies the class with parameters to be displayed or to be changed
-g	Gets the dispatch parameters for the specified class
-r	Used with –g, specifies time quantum resolution
-s	Specifies a file where values can be located

A class-specific file that contains the dispatch parameters can also be loaded during runtime. Use this file to establish a new set of priorities that replace the default values that were 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 Example 10–1.

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

- An `rt_init()` routine in the class module that loads the `rt_dptbl`(4).
- An `rt_dptbl`(4) module that provides the dispatch parameters and a routine to return pointers to `config_rt_dptbl`.
- The `dispadmin`(1M) executable.

The following steps install a RT class dispatch table:

1. Load the class-specific module with the following command, where *module_name* is the class-specific module:

   ```
   # modload /kernel/sched/module_name
   ```

2. 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 both scheduling classes is a parameter table, `rt_dptbl`(4), and `ts_dptbl`(4). These tables are configurable by using a loadable module at boot time, or with `dispadmin`(1M) during runtime.

Dispatcher Parameter Table

The in-core table for real-time establishes the properties for RT scheduling. The `rt_dptbl`(4) structure consists of an array of parameters, `struct rt_dpent_t`. Each of the *n* priority levels has one parameter. The properties of a given priority level are specified by the *i*th parameter structure in the array, `rt_dptbl[i]`.

A parameter structure consists of the following members, which are 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. For more information, see “Timestamp Interfaces” on page 231. The time quantum value is only a default or starting value for processes at a particular level. The time quantum of a real-time process can be changed by using the `priocntl`(1) command or the `priocntl`(2) system call.

Reconfiguring `config_rt_dptbl`

A real-time administrator can change the behavior of the real-time portion of the scheduler by reconfiguring the `config_rt_dptbl` at any time. One method is described in the `rt_dptbl`(4) man page, in the section titled “Replacing the `rt_dptbl` Loadable Module.”

A second method for examining or modifying the real-time parameter table on a running system is through the `dispadmin`(1M) command. Invoking `dispadmin`(1M) for the real-time class enables 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`(1M) must conform to the specific format described in the `rt_dptbl`(4) man page.

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

EXAMPLE 10–1 RT Class Dispatch Parameters

```
rtdpent_t  rt_dptbl[] = {             129,    60,
     /* prilevel Time quantum */                          130,    40,
        100,    100,                                          131,    40,
        101,    100,                                          132,    40,
        102,    100,                                          133,    40,
        103,    100,                                          134,    40,
        104,    100,                                          135,    40,
        105,    100,                                          136,    40,
        106,    100,                                          137,    40,
        107,    100,                                          138,    40
        108,    100,                                          139,    40,
        109,    100,                                          140,    20,
        110,    80,                                           141,    20,
        111,    80,                                           142,    20,
        112,    80,                                           143,    20,
        113,    80,                                           144,    20,
        114,    80,                                           145,    20,
        115,    80,                                           146,    20,
        116,    80,                                           147,    20,
        117,    80,                                           148,    20,
        118,    80,                                           149,    20,
        119,    80,                                           150,    10,
```

EXAMPLE 10–1 RT Class Dispatch Parameters *(Continued)*

```
        120,    60,
        121,    60,
        122,    60,
        123,    60,
        124,    60,
        125,    60,
        126,    60,
        126,    60,
        127,    60,
        128,    60,
        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 real-time applications. In a real-time 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 that are available to real-time applications in SunOS.

Under SunOS, 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 residency is appropriate in a time-share environment. This residency is often unacceptable for a real-time process. In a real-time environment, a process must guarantee a memory residence to reduce the process' memory access and dispatch latency.

Real-time memory locking in SunOS is provided by a set of library routines. These routines 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 the pages are unlocked or the process exits.

The operating system has a system-wide limit on the number of pages that can be locked at any time. This limit is a tunable parameter whose default value is calculated at boot time. The default value is based on the number of page frames minus another percentage, currently set at ten percent.

Locking a Page

A call to `mlock`(3C) 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. The lock count of each page is incremented. Any page whose lock count value is greater than zero 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 interfaces 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, the page is also implicitly unlocked.

Locks are not inherited by a child process after a `fork`(2) call. If a process that has some 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 page faults, which are the usual penalties that are associated with forking a process.

Unlocking a Page

To unlock a page of memory, a process requests the release of a segment of locked virtual pages by a calling `munlock`(3C). `munlock` decrements the lock counts of the specified physical pages. After decrementing a page's lock count to 0, the page swaps normally.

Locking All Pages

A superuser process can request that all mappings within its address space be locked by a call to `mlockall`(3C). 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 an existing mapping or that replaces an existing mapping is locked into memory.

Recovering Sticky Locks

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

High Performance I/O

This section describes I/O with real–time processes. In SunOS, the libraries supply two sets of interfaces and calls to perform fast, asynchronous I/O operations. The POSIX asynchronous I/O interfaces are the most recent standard. The SunOS environment also provides file and in-memory synchronization operations and modes to prevent information loss and data inconsistency.

Standard UNIX I/O is synchronous to the application programmer. An application that calls `read`(2) or `write`(2) usually waits until the system call has finished.

Real-time applications need asynchronous, bounded I/O behavior. A process that issues an asynchronous I/O call proceeds without waiting for the I/O operation to complete. The caller is notified when the I/O operation has finished.

Asynchronous I/O can be used with any SunOS file. Files are opened synchronously and no special flagging is required. An asynchronous I/O transfer has three elements: call, request, and operation. The application calls an asynchronous I/O interface, 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.

Asynchronous and standard I/O requests can be intermingled on any file descriptor. The system maintains no particular sequence of read and write requests. The system arbitrarily resequences all pending read and write requests. If a specific sequence is required for the application, the application must insure the completion of prior operations before issuing the dependent requests.

POSIX Asynchronous I/O

POSIX asynchronous I/O is performed using `aiocb` structures. An `aiocb` control block identifies each asynchronous I/O request and contains all of the controlling information. A control block can be used for only one request at a time. A control block can be reused after its request has been completed.

A typical POSIX asynchronous I/O operation is initiated by a call to aio_read(3RT) or aio_write(3RT). Either polling or signals can be used to determine the completion of an operation. If signals are used for completing operations, each operation can be uniquely tagged. The tag is then returned in the si_value component of the generated signal. See the siginfo(3HEAD) man page.

aio_read	aio_read(3RT) is called with an asynchronous I/O control block to initiate a read operation.
aio_write	aio_write(3RT) is called with an asynchronous I/O control block to initiate a write operation.
aio_return, aio_error	aio_return(3RT) and aio_error(3RT) are called to obtain return and error values, respectively, after an operation is known to have completed.
aio_cancel	aio_cancel(3RT) is called with an asynchronous I/O control block to cancel pending operations. aio_cancel can be used to cancel a specific request, if a request is specified by the control block. aio_cancel can also cancel all of the requests that are pending for the specified file descriptor.
aio_fsync	aio_fsync(3RT) queues an asynchronous fsync(3C) or fdatasync(3RT) request for all of the pending I/O operations on the specified file.
aio_suspend	aio_suspend(3RT) suspends the caller as though one or more of the preceding asynchronous I/O requests had been made synchronously.

Solaris Asynchronous I/O

This section discusses asynchronous I/O operations in the Solaris operating environment.

Notification (SIGIO)

When an asynchronous I/O call returns successfully, the I/O operation has only been queued and waits to be done. The actual operation has a return value and a potential error identifier. This return value and potential error identifier would have been returned to the caller if the call had been synchronous. When the I/O is finished, both the return and error values 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 {
     ssize_t    aio_return; /* return value of read or write */
     int         aio_errno;  /* errno generated by the IO */
 } aio_result_t;
```

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

Note that a process with two or more asynchronous I/O operations pending has no certain way to determine the cause of the SIGIO signal. A process that receives a SIGIO should check all its conditions that could be generating the SIGIO signal.

Using aioread

The aioread(3AIO) routine is the asynchronous version of read(2). In addition to the normal read arguments, aioread(3AIO) takes the arguments that specify a file position and the address of an aio_result_t structure. The resulting information about the operation is stored in the aio_result_t structure. The file position specifies a seek to be performed within the file before the operation. Whether the aioread(3AIO) call succeeds or fails, the file pointer is updated.

Using aiowrite

The aiowrite(3AIO) routine is the asynchronous version of write(2). In addition to the normal write arguments, aiowrite(3AIO) takes arguments that specify a file position and the address of an aio_result_t structure. The resulting information about the operation is stored in the aio_result_t structure.

The file position specifies that a seek operation is to be performed within the file before the operation. If the aiowrite(3AIO) 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

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

Using aiowait

A call to aiowait(3AIO) 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. aiowait(3AIO) returns a pointer to the aio_result_t structure for the completed operation.

Using poll()

To determine the completion of an asynchronous I/O event synchronously rather than depend on a SIGIO interrupt, use poll(2). You can also poll to determine the origin of a SIGIO interrupt.

poll(2) is slow when used on very large numbers of files. This problem is resolved by poll(7D).

Using the poll Driver

Using /dev/poll provides a highly scalable way of polling a large number of file descriptors. This scalability is provided through a new set of APIs and a new driver, /dev/poll. The /dev/poll API is an alternative to, not a replacement of, poll(2). Use poll(7D) to provide details and examples of the /dev/poll API. When used properly, the /dev/poll API scales much better than poll(2). This API is especially suited for applications that satisfy the following criteria:

- Applications that repeatedly poll a large number of file descriptors
- Polled file descriptors that are relatively stable, meaning that the descriptors are not constantly closed and reopened
- The set of file descriptors that actually have polled events pending is small, comparing to the total number of file descriptors that are being polled

Using close

Files are closed by calling close(2). The call to close(2) cancels any outstanding asynchronous I/O request that can be closed. close(2) waits for an operation that cannot be cancelled. For more information, see "Using aiocancel" on page 226. When close(2) returns, no asynchronous I/O is pending for the file descriptor. Only asynchronous I/O requests queued to the specified file descriptor are cancelled when a file is closed. Any I/O pending requests for other file descriptors are not cancelled.

Synchronized I/O

Applications might 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.

Synchronization Modes

Under SunOS, a write operation succeeds when the system ensures that all written data is readable after any subsequent open of the file. This check assumes no 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 the associated data has been successfully transferred, or when 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 the synchronized read operation is requested when pending write requests affect the data to be read, these write requests are successfully completed before the data is read.
- For writes, the operation has been completed, or diagnosed if unsuccessful. The write operation succeeds when the data specified in the write request is successfully transferred. Furthermore, all file system information required to retrieve the data must be successfully transferred.
- File attributes that are not necessary for data retrieval are not transferred prior to returning to the calling process.
- Synchronized I/O file integrity completion requires that all file attributes relative to the I/O operation be successfully transferred before returning to the calling process. Synchronized I/O file integrity completion is otherwise identical to synchronized I/O data integrity compleiton.

Synchronizing a File

`fsync`(3C) and `fdatasync`(3RT) explicitly synchronize a file to secondary storage.

The `fsync`(3C) routine guarantees that the interface is synchronized at the I/O file integrity completion level. `fdatasync`(3RT) guarantees that the interface is synchronized at level of I/O data integrity completion.

Applications can synchronize each I/O operation before the operation completes. Setting the `O_DSYNC` flag on the file description by using `open`(2) or `fcntl`(2) ensures that all I/O writes reach I/O data completion before the operation completes. Setting the O_SYNC flag on the file description ensures that all I/O writes have reached 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) and aio_read(3RT) reach the same level of completion that is requested by the descriptor setting. The descriptor setting can be either O_DSYNC or O_SYNC.

Interprocess Communication

This section describes the interprocess communication (IPC) interfaces of SunOS as the interfaces relate to real-time processing. Signals, pipes, FIFOs, message queues, shared memory, file mapping, and semaphores are described here. For more information about the libraries, interfaces, and routines that are useful for interprocess communication, see Chapter 6.

Processing Signals

The sender can use sigqueue(3RT) to send a signal together with a small amount of information to a target process.

To queue subsequent occurrences of a pending signal, the target process must have the SA_SIGINFO bit set for the specified signal. See the sigaction(2) man page.

The target process normally receive signals asynchronously. To receive signals synchronously, block the signal and call either sigwaitinfo(3RT) or sigtimedwait(3RT). See the sigprocmask(2) man page. This procedure causes the signal to be received synchronously. The value sent by the caller of sigqueue(3RT) is stored in the si_value member of the siginfo_t argument. Leaving the signal unblocked causes the signal to be delivered to the signal handler specified by sigaction(2), with the value appearing in the si_value of the siginfo_t argument to the handler.

A specified 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(3RT). Thereafter, a call to sigqueue(3RT) either successfully enqueues at the target process or fails within a bounded amount of time.

Pipes, Named Pipes, and Message Queues

Pipes, named pipes, and message queues behave similarly to character I/O devices. These interfaces have different methods of connecting. See "Pipes Between Processes" on page 107 for more information about pipes. See "Named Pipes" on page 109 for more information about named pipes. See "System V Messages" on page 113 and "POSIX Messages" on page 110 for more information about message queues.

Using Semaphores

Semaphores are also provided in both System V and POSIX styles. See "System V Semaphores" on page 115 and "POSIX Semaphores" on page 110 for more information.

Note that using semaphores can cause priority inversions unless priority inversions 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. When more than two processes attempt to read and write shared memory simultaneously, the memory contents can become inaccurate. This potential inaccuracy is the major difficulty with using shared memory.

Asynchronous Network Communication

This section introduces asynchronous network communication, using sockets or Transport-Level Interface (TLI) for real-time applications. Asynchronous networking with sockets is done by setting an open socket, of type `SOCK_STREAM`, to asynchronous and non blocking. For more information on asynchronous sockets, see "Advanced Socket Topics" on page 150. Asynchronous network processing of TLI events is supported 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 Chapter 8.

Modes of Networking

Both sockets and transport-level interface provide two modes of service: *connection-mode* and *connectionless-mode*.

Connection-mode service is circuit-oriented. This service enables the transmission of data over an established connection in a reliable, sequenced manner. This service 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 is message-oriented and supports data transfer in self-contained units with no logical relationship required among multiple units. A single service request passes all the information required to deliver a unit of data from the sender to the transport provider. This service request includes the destination address and the data to be delivered. Connectionless-mode service is attractive for applications that involve short-term interactions that do not require guaranteed, in-sequence delivery of data. Connectionless transports are generally unreliable.

Timing Facilities

This section describes the timing facilities that are available for real-time applications under SunOS. Real-time applications that use these mechanisms require detailed information from the man pages of the routines that are listed in this section.

The timing interfaces of SunOS fall into two separate areas: *timestamps* and *interval timers*. The timestamp interfaces provide a measure of elapsed time. The timestamp interfaces also enable 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.

Timestamp Interfaces

Two interfaces provide timestamps. `gettimeofday`(3C) provides the current time in a *timeval* structure, representing the time in seconds and microseconds since midnight, Greenwich Mean Time, on January 1, 1970. `clock_gettime`, 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`(3C).

SunOS uses a hardware periodic timer. For some workstations, the hardware periodic timer is the sole source of timing information. If the hardware periodic timer is the sole source of timing information, the accuracy of timestamps is limited to the timer's resolution. For other platforms, a timer register with a resolution of one microsecond means that timestamps are accurate to one microsecond.

Interval Timer Interfaces

Real-time applications often schedule actions by using interval timers. Interval timers can be either of two types: a *one-shot* type or a *periodic* type.

A one-shot is an armed timer that is set to an expiration time relative to either a current time or an absolute time. The timer expires once and is disarmed. This type of a timer is useful for clearing buffers after the data has been transferred to storage, or to time-out an operation.

A periodic timer is armed with an initial expiration time, either absolute or relative, and a repetition interval. Every time the interval timer expires, the timer is reloaded with the repetition interval. The timer is then rearmed. This timer is useful for data logging or for servo-control. In calls to interval timer interfaces, time values that are smaller than the timer's resolution are rounded up to the next multiple of the hardware timer interval. This interval is typically 10ms.

SunOS has two sets of timer interfaces. The `setitimer`(2) and `getitimer`(2) interfaces operate fixed set timers, which are called the BSD timers, using the `timeval` structure to specify time intervals. The POSIX timers, which are created with `timer_create`(3RT), operate the POSIX clock, `CLOCK_REALTIME`. POSIX timer operations are expressed in terms of the `timespec` structure.

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

The `timer_create`(3RT) routine can create up to `TIMER_MAX` POSIX timers. The caller can specify what signal and what associated value are sent to the process when the timer expires. The `timer_settime`(3RT) and `timer_gettime`(3RT) routines retrieve and establish respectively the value of the specified POSIX interval timer. POSIX timers can expire while the required signal is pending delivery. The timer expirations are counted, and `timer_getoverrun`(3RT) retrieves the count. `timer_delete`(3RT) deallocates a POSIX timer.

The following example illustrates how to use `setitimer`(2) to generate a periodic interrupt, and how to control the arrival of timer interrupts.

EXAMPLE 10–2 Controlling Timer Interrupts

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

#define TIMERCNT 8

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

main()
{
    struct itimerval times;
    sigset_t    sigset;
    int        i, ret;
    struct sigaction act;
    siginfo_t    si;

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

    /* set up handler for SIGALRM */
    act.sa_action = 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, si, NULL);
    while (timercnt < TIMERCNT) {
        ret = sigsuspend (&sigset);
    }
    printtimes();
}

void timerhandler (sig, siginfo, context)
    int        sig;
    siginfo_t    *siginfo;
    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("%ld.%06d\n", alarmtimes[i].tv_sec,
                alarmtimes[i].tv_usec);
    }
}
```

CHAPTER 11

The Solaris ABI and ABI Tools

The Solaris™ Application Binary Interface (ABI) defines the interfaces that are available for the use of application developers. Conforming to the ABI enhances an application's binary stability. This chapter discusses the Solaris ABI and the tools provided to verify an application's compliance with the ABI, including:

- The definition and purpose of the Solaris ABI, discussed in "Defining the Solaris ABI" on page 236.
- The usage of the two ABI tools, `appcert` and `apptrace`, discussed in "Solaris ABI Tools" on page 238.

What is the Solaris ABI?

The Solaris ABI is the set of supported run-time interfaces that are available for an application to use with the Solaris operating environment. The most important components of the ABI are in the following list:

- The interfaces provided by the Solaris system libraries, which are documented in section 3 of the man pages
- The interfaces provided by the Solaris kernel system calls, which are documented in section 2 of the man pages
- The locations and formats of various system files and directories, which are documented in section 4 of the man pages
- The input and output syntax and semantics of Solaris utilities, which are documented in section 1 of the man pages

The main component of the Solaris ABI is the set of system library interfaces. The term *ABI* in this chapter refers only to that component. The ABI contains exclusively C language interfaces, as C is the only language for which the Solaris operating environment provides interfaces.

C source code that is written to the Solaris API (Application Programming Interface) is transformed by the C compiler into a binary for one of three ABI versions. The three versions are 32-bit SPARC, 64-bit SPARC, or 32-bit Intel. While the ABI is very similar to the API, the source compilation process introduces several important differences:

- Compiler directives such as `#define` can alter or replace source-level constructs. The resulting binary might lack a symbol present in the source or include a symbol not present in the source.
- The compiler might generate processor-specific symbols, such as arithmetic instructions, which augment or replace source constructs.
- The compiler's binary layout might be specific to that compiler and the versions of the source language which the compiler accepts. In such cases, identical code compiled with different compilers might produce incompatible binaries.

For these reasons, source-level (API) compatibility does not provide a sufficient expectation of binary compatibility across Solaris releases.

The Solaris ABI is made up of the supported interfaces provided by the operating system. Some of the interfaces that are available in the system are intended for the exclusive use of the operating system. These exclusive interfaces are not available for use by an application. Prior to the SunOS 5.6 release, all of the interfaces in Solaris libraries were available for application developers to use. With the library symbol scoping technology available in the Solaris link editor, interfaces not intended for use outside of a library have their scope reduced to be purely local to the library. See the *Linker and Libraries Guide* for details. Due to system requirements, not all private interfaces can have such a reduced scope. These interfaces are labeled *private*, and are not included in the Solaris ABI.

Defining the Solaris ABI

The Solaris ABI is defined in the Solaris libraries. These definitions are done by means of the library versioning technology and policies used in the link editor and run-time linker.

Symbol Versioning in Solaris Libraries

The Solaris link editor and run-time linker use two kinds of library versioning: file versioning and symbol versioning. In file versioning, a library is named with an appended version number, such as `libc.so.1`. When an incompatible change is made to one or more public interfaces in that library, the version number is incremented (for example, to `libc.so.2`). In a dynamically linked application, a symbol bound to at build time might not be present in the library at run time. In symbol versioning, the Solaris linker associates a set of symbols with a name. The linker then checks for the presence of the name in the library during run-time linking to verify the presence of the associated symbols.

Library symbol versioning associates a set of symbols with a symbol version name, and number if that name has a numbering scheme, by means of a mapfile. The following is an example mapfile for a hypothetical Sun library, `libfoo.so.1`.

```
SUNW_1.2 {
    global:
        symbolD;
        symbolE
} SUNW_1.1;

SUNW_1.1 {
    global:
        symbolA;
        symbolB;
        symbolC;
};

SUNWprivate {
    global:
        __fooimpl;
    local: *;
};
```

This mapfile indicates that `symbolA`, `symbolB`, and `symbolC` are associated with version `SUNW_1.1`, `symbolD` and `symbolE` are associated with `SUNW_1.2`, and that `SUNW_1.2` inherits all the symbols associated with `SUNW_1.1`. The symbol `__fooimpl` is associated with a different named set which does not have a numbered inheritance chain.

During build time, the link editor examines the symbols used by the application. The link editor records the set names in the application on which those symbols depend. In the case of chained sets, the link editor records the smallest named set containing all the symbols used by the application. If an application uses only `symbolA` and `symbolB`, the link editor records a dependency on `SUNW_1.1`. If an application uses `symbolA`, `symbolB`, and `symbolD`, the link editor records a dependency on `SUNW_1.2`, because `SUNW_1.2` includes `SUNW_1.1`.

At run time, the linker verifies that the version names recorded as dependencies in the application are present in the libraries that are being linked. This process is a quick way to verify the presence of required symbols. For more details, see the *Linker and Libraries Guide*.

Note – The *local: ** directive in the mapfile means that any symbol in the library that is not explicitly associated with a named set is scoped locally to the library. Such locally scoped symbols are not visible outside the library. This convention ensures that symbols are only visible when associated with a symbol versioning name.

Using Symbol Versioning to Label the Solaris ABI

Since all visible symbols in a library belong to some named set, the naming scheme can be used to label the symbols' ABI status. This labeling is done by associating all private interfaces with a set name beginning with *SUNWprivate*. Public interfaces begin with other names, specifically:

- *SYSVABI*, for interfaces defined by the System V ABI definition
- *SISCD*, for interfaces defined by the SPARC International *SPARC Compliance Definition*
- *SUNW*, for interfaces defined by Sun Microsystems

These public, named sets are numbered with a *major.minor* numbering scheme. When a set includes new symbols, the set's minor version number increases. When an existing symbol changes in a way that makes the symbol incompatible with its previous behavior, the major version number of the set that includes that symbol increases. When an existing symbol changes incompatibly, the version number in the library's file name also increases.

The definition of the Solaris library ABI is therefore contained in the libraries, and consists of the set of symbols that are associated with symbol version names that do not begin with *SUNWprivate*. The `pvs` command lists the symbols in a library.

Solaris ABI Tools

The Solaris operating environment provides two tools to verify that an application's use of Solaris interfaces conforms to the Solaris ABI. The `appcert` utility statically examines the Solaris library interfaces used by ELF binaries for instances of private interface usage. The `appcert` utility produces summary and detailed reports of any potential binary stability problems it finds. The `apptrace` tool uses the link-auditing capability of the run-time linker to dynamically trace Solaris library routine calls as the application runs. This capability enables developers to examine an application's use of the Solaris system interfaces.

The ABI tools enable easy, rapid identification of binaries that might have binary compatibility problems with a given Solaris release. To check binary stability, perform the following steps:

- **Use appcert on the current Solaris release for triage.** This identifies which binaries use problematic interfaces and which do not.
- **Use apptrace on the target Solaris release for verification.** This verifies whether interface compatibility problems exist by enabling dynamic observation of those interfaces as they are used.

`appcert` Utility

The `appcert` utility is a Perl script that statically examines ELF binaries and compares the library symbols used against a model of public interfaces and private interfaces in a given Solaris release. The utility runs on either SPARC or Intel platforms. The utility can check interface usage for both SPARC and Intel 32-bit interfaces as well as the 64-bit interfaces on SPARC. Note that `appcert` only examines C language interfaces.

As new Solaris releases become available, some library interfaces might change their behavior or disappear entirely. These changes can affect the performance of applications that rely on those interfaces. The Solaris ABI defines runtime library interfaces that are safe and stable for application use. The `appcert` utility is designed to help developers verify an application's compliance with the Solaris ABI.

What `appcert` Checks

The `appcert` utility examines your applications for:

- Private symbol usage
- Static linking
- Unbound symbols

Private Symbol Usage

Private symbols are functions or data that is used by Solaris libraries to call each other. The semantic behavior of private symbols might change, and symbols might sometimes be removed. Such symbols are called *demoted symbols*. The mutable nature of private symbols introduces the potential for instability in applications that depend on private symbols.

Static Linking

The semantics of private symbol calls between Solaris libraries might change between releases. Therefore, the creation of static links to archives degrades an application's binary stability. Dynamic links to the archive's corresponding shared object file avoid this problem.

Unbound Symbols

The `appcert` utility uses the dynamic linker to resolve the library symbols that are used by the application being examined. Symbols that the dynamic linker cannot resolve are called *unbound symbols*. Unbound symbols might be caused by environment problems, such as an incorrectly set `LD_LIBRARY_PATH` variable. Unbound symbols might also be caused by build problems, such as omitting the definitions of the `-l`*lib* or `-z` switches at compile time. While these examples are minor, unbound symbols that are reported by `appcert` might indicate a more serious problem, such as a dependency on a private symbol that no longer exists.

What `appcert` Does Not Check

If the object file `appcert` is examining depends on libraries, those dependencies must be recorded in the object. To do so, be sure to use the compiler's `-l` switch when compiling the code. If the object file depends on other shared libraries, those libraries must be accessible through `LD_LIBRARY_PATH` or `RPATH` at the time you run `appcert`.

The `appcert` application cannot check 64–bit applications unless the machine is running the 64–bit Solaris kernel. Since Solaris provides no 64–bit static libraries, `appcert` does not perform static-linking checks on 64–bit applications.

The `appcert` utility cannot examine:

- Object files that are completely or partially statically linked. A completely statically linked object is reported as unstable.
- Executable files that do not have the execute permission set. The `appcert` utility skips such executables. Shared objects without the execute permission set are examined normally.
- Object files whose user ID is set to `root`.
- Non-ELF executables, such as shell scripts.
- Solaris interfaces in languages other than C. The code need not be in C, but the call to the Solaris library must be.

Working with appcert

To check your application with `appcert`, type:

`appcert` *object | directory*

replacing *object | directory* with either:

- The complete list of objects you want `appcert` to examine
- The complete list of directories that contain such objects

Note – You might run `appcert` in an environment that is different from the environment in which the application runs. If these environments are different, `appcert` might not be able to correctly resolve references to Solaris library interfaces.

The `appcert` utility uses the Solaris runtime linker to construct a profile of interface dependencies for each executable or shared object file. This profile is used to determine the Solaris system interfaces upon which the application depends. The dependencies that are outlined in the profile are compared to the Solaris ABI to verify conformance. No private interfaces should be found.

The `appcert` utility recursively searches directories for object files, ignoring non-ELF object files. After `appcert` has finished checking the application, `appcert` prints a rollup report to the standard output, usually the screen. A copy of this report is written in the working directory, which is usually `/tmp/appcert.`*pid*, in a file that is named `Report`. In the subdirectory name, *pid* represents the 1–to–6 digit process ID of that particular instantiation of `appcert`. See “`appcert` Results” on page 243 for more on the directory structure to which `appcert` writes output files.

appcert Options

The following options modify the behavior of the `appcert` utility. You can type any of these options at the command line, after the `appcert` command but before the *object | directory* operand.

`-B` Runs `appcert` in batch mode.

In batch mode, the report produced by `appcert` contains one line for each binary checked.

A line that begins with PASS indicates the binary that is named in that line did not trigger any `appcert` warnings.

A line that begins with FAIL indicates problems were found in that binary.

A line that begins with INC indicates the binary that is named in that line could not be completely checked.

`-f` *infile*	The file *infile* should contain a list of files to check, with one file name per line. These files are added to any files already specified at the command line. If you use this switch, you do not need to specify an object or directory at the command line.
`-h`	Prints usage information for `appcert`.
`-L`	By default, `appcert` notes any shared objects in an application, and appends the directories in which the shared objects reside to `LD_LIBRARY_PATH`. The `-L` switch disables this behavior.
`-n`	By default, `appcert` follows symbolic links when `appcert` searches directories for binaries to check. The `-n` switch disables this behavior.
`-S`	Appends the Solaris library directories `/usr/openwin/lib` and `/usr/dt/lib` to `LD_LIBRARY_PATH`.
`-w` *working_dir*	Specifies a directory in which to run the library components. Temporary files are also created in the directory specified by this switch. If this switch is not specified, `appcert` uses the `/tmp` directory.

Using `appcert` for Application Triage

The `appcert` utility can be used to quickly and easily discern which applications in a given set have potential stability problems. If `appcert` does not report any stability problems, the application is not likely to encounter binary stability problems in subsequent Solaris releases. The following table lists some common binary stability problems.

TABLE 11–1 Common Binary Stability Problems

Problem	Course of Action
Use of a private symbol that is known to change	Eliminate use of symbol immediately.
Use of a private symbol that has not changed yet	Application can still be run for now, but eliminate use of symbol as soon as practical.
Static linking of a library with a shared object counterpart	Use shared object counterpart instead.

TABLE 11–1 Common Binary Stability Problems *(Continued)*

Problem	Course of Action
Static linking of a library with no shared object counterpart	Convert `.a` file to `.so` file by using the command `ld -z allextract` if possible. Otherwise, continue to use static library until shared object is available.
Use of a private symbol for which no public equivalent is available	Contact Sun and request a public interface.
Use of a symbol that is deprecated, or use of a symbol that is planned for removal	Application can still be run for now, but eliminate use of symbol as soon as practical.
Use of a public symbol that has changed	Recompile.

Potential stability problems caused by the use of private interfaces might not occur on a given release. The behavior of private interfaces does not always change between releases. To verify that a private interface's behavior has changed in the target release, use the `apptrace` tool. Usage of `apptrace` is discussed in "Using `apptrace` for Application Verification" on page 245.

`appcert` Results

The results of the `appcert` utility's analysis of an application's object files are written to several files that are located in the `appcert` utility's working directory, typically `/tmp`. The main subdirectory under the working directory is `appcert.`*pid*, where *pid* is the process ID for that instantiation of `appcert`. The `appcert` utility's results are written to the following files:

`Index`
: Contains the mapping between checked binaries and the subdirectory in which `appcert` output specific to that binary is located.

`Report`
: Contains a copy of the rollup report that is displayed on `stdout` when `appcert` is run.

`Skipped`
: Contains a list of binaries that `appcert` was asked to check but was forced to skip, along with the reason each binary was skipped. These reasons are in the following list:

- File is not a binary object
- File cannot be read by the user
- File name contains metacharacters
- File does not have the execute bit set

`objects/`*object_name*
: A separate subdirectory is under the `objects` subdirectory for each object examined by `appcert`. Each of these subdirectories contains the following files:

`check.demoted.symbols`	Contains a list of symbols that `appcert` suspects are demoted Solaris symbols.
`check.dynamic.private`	Contains a list of private Solaris symbols to which the object is directly bound.
`check.dynamic.public`	Contains a list of public Solaris symbols to which the object is directly bound.
`check.dynamic.unbound`	Contains a list of symbols not bound by the dynamic linker when running `ldd -r`. Lines returned by `ldd` containing “`file not found`” are also included.
`summary.dynamic`	Contains a printer-formatted summary of dynamic bindings in the objects `appcert` examined, including tables of public and private symbols used from each Solaris library.

Returns one of four exit values.

0 No potential sources of binary instability were found by `appcert`.

1 The `appcert` utility did not run successfully.

2 Some of the objects checked by `appcert` have potential binary stability problems.

3 The `appcert` utility did not find any binary objects to check.

Correcting Problems Reported by `appcert`

- **Private Symbol Use –** An application that depends on private symbols might not run on a Solaris release different from the one in which it was developed. This phenomenon occurs because private symbols that occur in a given Solaris release might behave differently or not be present in another release. If `appcert` reports private symbol usage in your application, rewrite the application to avoid the use of private symbols.
- **Demoted Symbols –** Demoted symbols are functions or data variables in a Solaris library that have been removed or have been scoped locally in a later Solaris release. An application that directly calls such a symbol fails to run on a release whose libraries do not export that symbol.

- **Unbound Symbols –** Unbound symbols are library symbols that are referenced by the application that the dynamic linker was unable to resolve when called by appcert. While unbound symbols are not always an indicator of poor binary stability, unbound symbols might indicate a more serious problem, such as dependencies on demoted symbols.
- **Obsolete Library –** An obsolete library might be removed from the Solaris operating environment in a future release. The appcert utility flags any use of such a library. Applications that depend on such a library might not function in a future release that does not feature the library. To avoid this problem, do not use interfaces from obsolete libraries.
- **Use of sys_errlist or sys_nerr –** The use of the sys_errlist and sys_nerr symbols might degrade binary stability. A reference might be made past the end of the sys_errlist array. To avoid this risk, use strerror instead.
- **Use of strong and weak symbols –** The strong symbols that are associated with weak symbols are reserved as private because their behavior might change in future Solaris releases. Applications should only directly reference weak symbols. An example of a strong symbol is _socket, which is associated with the weak symbol socket.

Using apptrace for Application Verification

The apptrace utility is a C program which dynamically traces calls to Solaris library routines as an application runs. apptrace works on either SPARC or Intel platforms. apptrace can trace interface calls for both SPARC and Intel 32-bit interfaces, as well as the 64-bit interfaces on SPARC. As with appcert, apptrace only examines C language interfaces.

Application Verification

After using appcert to determine an application is at risk of binary instability, apptrace helps assess the degree of risk in each case. To determine an application's binary compatibility with a given release, verify the successful use of each interface used by the application with apptrace.

The apptrace utility can verify that an application is using public interfaces correctly. For example, an application that is using the open() to open the administrative file /etc/passwd directly should instead use the appropriate programmatic interfaces. This ability to inspect the usage of the Solaris ABI enables easy and rapid identification of potential interface problems.

Running apptrace

The apptrace utility does not require any modification of the application being traced. To use apptrace, type **apptrace**, followed by any desired options along with the command line used to run the application of interest. The apptrace utility works by using the link-auditing capability of the runtime linker to intercept the application's calls to Solaris library interfaces. The apptrace utility then traces the calls by printing the names and values of the call's arguments and return value. The tracing output can be on a single line or arranged across multiple lines for readability. Public interfaces are printed in human-readable form. Private interfaces are printed in hexadecimal.

The apptrace utility enables selective tracing of calls, both at the level of individual interfaces and the level of libraries. For example, apptrace can trace calls to printf() coming from libnsl, or a range of calls within a specific library. The apptrace utility can also verbosely trace user-specified calls. The specifications that dictate apptrace behavior are governed by a syntax that is consistent with the usage of truss(1). The -f option directs apptrace to follow forked child processes. The -o option specifies an output file for apptrace results.

The apptrace utility traces only library-level calls and is loaded into the running application process, gaining a performance increase over truss. With the exception of printf, apptrace cannot trace calls to functions that accept variable argument lists or examine the stack or other caller information, for example, setcontext, getcontext, setjmp, longjmp, and vfork.

Interpreting apptrace Output

The following examples contain sample apptrace output from tracing a simple one-binary application, ls.

EXAMPLE 11–1 Default Tracing Behavior

```
% apptrace ls /etc/passwd
ls       -> libc.so.1:atexit(func = 0xff3cb8f0) = 0x0
ls       -> libc.so.1:atexit(func = 0x129a4) = 0x0
ls       -> libc.so.1:getuid() = 0x32c3
ls       -> libc.so.1:time(tloc = 0x23918) = 0x3b2fe4ef
ls       -> libc.so.1:isatty(fildes = 0x1) = 0x1
ls       -> libc.so.1:ioctl(0x1, 0x540d, 0xffbff7ac)
ls       -> libc.so.1:ioctl(0x1, 0x5468, 0x23908)
ls       -> libc.so.1:setlocale(category = 0x6, locale = "") = "C"
ls       -> libc.so.1:calloc(nelem = 0x1, elsize = 0x40) = 0x23cd0
ls       -> libc.so.1:lstat64(path = "/etc/passwd", buf = 0xffbff6b0) = 0x0
ls       -> libc.so.1:acl(pathp = "/etc/passwd", cmd = 0x3, nentries = 0x0,
              aclbufp = 0x0) = 0x4
ls       -> libc.so.1:qsort(base = 0x23cd0, nel = 0x1, width = 0x40,
              compar = 0x12038)
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
```

EXAMPLE 11–1 Default Tracing Behavior *(Continued)*

```
ls       -> libc.so.1:strlen(s = "") = 0x0
ls       -> libc.so.1:strlen(s = "/etc/passwd") = 0xb
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:strlen(s = "") = 0x0
ls       -> libc.so.1:printf(format = 0x12ab8, ...) = 11
ls       -> libc.so.1:printf(/etc/passwd
format = 0x12abc, ...) = 1
ls       -> libc.so.1:exit(status = 0)
```

The previous example shows the default tracing behavior, tracing every library call on the command `ls /etc/passwd`. The `apptrace` utility prints a line of output for every system call, indicating:

- The name of the call
- The library the call is in
- The arguments and return values of the call

The output from `ls` is mixed in with the `apptrace` output.

EXAMPLE 11–2 Selective Tracing

```
% apptrace -t \*printf ls /etc/passwd
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
ls       -> libc.so.1:printf(format = 0x12ab8, ...) = 11
ls       -> libc.so.1:printf(/etc/passwd
format = 0x12abc, ...) = 1
```

The previous example shows how `apptrace` can selectively trace calls with regular-expression syntax. In the example, calls to interfaces ending in `printf`, which include `sprintf`, are traced in the same `ls` command as before. Consequently, `apptrace` only traces the `printf` and `sprintf` calls.

EXAMPLE 11–3 Verbose Tracing

```
% apptrace -v sprintf ls /etc/passwd
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
  buf =    (char *) 0x233d0 ""
  format = (char *) 0x12af8 "%s%s%s"
ls       -> libc.so.1:sprintf(buf = 0x233d0, format = 0x12af8, ...) = 0
  buf =    (char *) 0x233d0 ""
  format = (char *) 0x12af8 "%s%s%s"
/etc/passwd
```

The previous example shows the verbose tracing mode, where the arguments to `sprintf` are printed on multiple output lines for readability. At the end, `apptrace` displays the output of the `ls` command.

APPENDIX **A**

UNIX Domain Sockets

UNIX domain sockets are named with UNIX paths. For example, a socket might be named `/tmp/foo`. UNIX domain sockets communicate only between processes on a single host. Sockets in the UNIX domain are not considered part of the network protocols because they can be used to communicate only between processes on a single host.

Socket types define the communication properties visible to a user. The Internet domain sockets provide access to the TCP/IP transport protocols. The Internet domain is identified by the value `AF_INET`. Sockets exchange data only with sockets in the same domain.

Creating Sockets

The `socket`(3SOCKET) call creates a socket in the specified family and of the specified type.

```
s = socket(family, type, protocol);
```

If the protocol is unspecified (a value of `0`), the system selects a protocol that supports the requested socket type. The socket handle (a file descriptor) is returned.

The family is specified by one of the constants defined in `sys/socket.h`. Constants named `AF_`*suite* specify the address format to use in interpreting names.

The following creates a datagram socket for intramachine use:

```
s = socket(AF_UNIX, SOCK_DGRAM, 0);
```

Set the *protocol* argument to `0`, the default protocol, in most situations.

Local Name Binding

A socket is created with no name. A remote process has no way to refer to a socket until an address is bound to the socket. Communicating processes are connected through addresses. In the UNIX family, a connection is composed of (usually) one or two path names. UNIX family sockets need not always be bound to a name. If they are, bound, duplicate ordered sets such as `local pathname` or `foreign pathname` can never exist. The path names cannot refer to existing files.

The `bind`(3SOCKET) call enables a process to specify the local address of the socket. This creates the `local pathname` ordered set, while `connect`(3SOCKET) and `accept`(3SOCKET) complete a socket's association by fixing the remote half of the address. Use `bind`(3SOCKET) as follows:

```
bind (s, name, namelen);
```

The socket handle is *s*. The bound name is a byte string that is interpreted by the supporting protocols. UNIX family names contain a path name and a family. The example shows binding the name `/tmp/foo` to a UNIX family socket.

```
#include <sys/un.h>
 ...
struct sockaddr_un addr;
 ...
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind (s, (struct sockaddr *) &addr,
        strlen(addr.sun_path) + sizeof (addr.sun_family));
```

When determining the size of an `AF_UNIX` socket address, null bytes are not counted, which is why you can use `strlen`(3C).

The file name referred to in `addr.sun_path` is created as a socket in the system file name space. The caller must have write permission in the directory where `addr.sun_path` is created. The file should be deleted by the caller when it is no longer needed. Delete `AF_UNIX` sockets with `unlink`(1M).

Establishing a Connection

Connection establishment is usually asymmetric. One process acts as the client and the other as the server. The server binds a socket to a well-known address associated with the service and blocks on its socket for a connect request. An unrelated process can then connect to the server. The client requests services from the server by initiating a connection to the server's socket. On the client side, the connect(3SOCKET) call initiates a connection. In the UNIX family, this might appear as:

```
struct sockaddr_un server;
        server.sun.family = AF_UNIX;
         ...
        connect(s, (struct sockaddr *)&server, strlen(server.sun_path)
         + sizeof (server.sun_family));
```

See "Connection Errors" on page 129 for information on connection errors. "Data Transfer" on page 130 tells you how to transfer data. "Closing Sockets" on page 131 tells you how to close a socket.

Index

O

P

R

S

T

U

V

X

Z