Topics in IRIX® Programming

CONTRIBUTORS

Written by David Cortesi based on previous versions by Arthur Evans,
    Wendy Ferguson, and Jed Hartman; updated by Susan Thomas
Production by Linda Rae Sande
Engineering contributions by Ivan Bach, Greg Boyd, Joe CaraDonna, Srinivas
    Lingutla, Bill Mannell, Paul Mielke, Huy Nguyen, James Pitcairne-Hill, Paul Roy,
    Jonathan Thompson, and Ira Pramanick
St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower
    image courtesy of Xavier Berenguer, Animatica.

Topics in IRIX® Programming
Document Number 007-2478-007

Contents

List of Examples

List of Figures

List of Tables

About This Manual

This manual discusses several topics of interest to programmers writing applications for the IRIX operating system on Silicon Graphics computers. These topics include memory management, interprocess communication, models of parallel computation, file and record locking, font access, and internationalization.

What This Manual Contains

This manual contains the following major parts:

- Part I, "The Process Address Space," tells how the virtual address space of a process is created and how objects are mapped into it.

- Part II, "Interprocess Communication," covers all the facilities for communicating and coordinating among processes such as semaphores, shared memory, signals, message queues, and file and record locks.

- Part III, "Advanced File Control," describes advanced uses of disk files: file locking, asynchronous I/O, direct I/O, and guaranteed-rate I/O.

- Part IV, "Models of Parallel Computation," gives an overview of the different ways you can specify parallel execution in Silicon Graphics systems.

- Part V, "Working With Fonts," discusses typography and font use on Silicon Graphics computers, and describes the Font Manager library.

- Part VI, "Internationalizing Your Application," explains how to create an application that can be adapted for use in different countries.

- Appendix A, "ISO 3166 Country Names and Abbreviations," lists country codes for use with internationalization and localization.

What You Should Know Before Reading This Manual

This manual assumes that you are writing an application that executes under IRIX version 6.2 or later, and that you are familiar with the programming conventions of UNIX in general and IRIX in particular.

All examples are in the C language, although the descriptions are valid for C++ or any other language that provides access to IRIX kernel functions, such as Silicon Graphics Ada95 or MIPSpro Fortran 90.

Other Useful References

In addition to this manual, which covers specific IRIX features, you will need to refer to Silicon Graphics manuals that describe compilers and programming languages. Some of the most useful are listed in Table i.

Table i   Books for Further Reading in IRIX Development

Topic	Document Title	Number
Overview of the IRIX library of manuals for developers	*Programming on Silicon Graphics Systems: An Overview*	007-2476-*nnn*
Compiling, linking, and tuning programs in C, C++, or Fortran	*MIPSpro Compiling and Performance Tuning Guide*	007-2360-*nnn*
Writing modules in assembly language.	*MIPSpro Assembly Language Programmer's Guide*	007-2418-*nnn*
C language	*C Language Reference Manual*	007-0701-*nnn*
C++ language	*C++ Language System Overview*	007-1621-*nnn*
Fortran language	*MIPSpro Fortran 77 Programmer's Guide*	007-2361-*nnn*
	MIPSpro Fortran 90 Programmer's Guide	007-2761-*nnn*
System Configuration	*IRIX Admin: System Configuration and Tuning*	007-2859-*nnn*
Writing real-time applications	*REACT/Pro Real Time Programmer's Guide*	007-2499-*nnn*
Controlling devices directly	*IRIX Device Driver Programmer's Guide*	007-0911-*nnn*
Details of the MIPS processor hardware	*MIPS R4000 Microprocessor User's Manual*	

You can find additional information about internationalization from X/Open Company Limited. *X/Open Portability Guide, Volume 1, XSI Commands and Utilities, Volume 2; XSI System Interface*; and *Volume 3, XSI Supplementary Definitions*. Berkshire, United Kingdom. Prentice-Hall, Inc.

Obtaining Manuals

Silicon Graphics manuals are usually read online using IRIS InSight. This manual and many of the books in Table i are installed as part of the IRIS Development Foundation feature. When the books are installed or mounted on your workstation, use the command *iiv*, or double-click the IRIS InSight icon, to launch IRIS Insight. Then select the book you want from the "bookshelf" display.

When the manuals are not accessible to your workstation you can examine or order any Silicon Graphics manual on the World Wide Web using the following URL: http://techpubs.sgi.com/library .

If you do not have Web access, you can order a printed manual from Silicon Graphics by telephone. Inside the U.S. and Canada, call 1-800-627-9307. In other countries, call the U.S. telephone number 415-960-1980, and ask for extension 5-5007.

Conventions Used in This Manual

This manual uses the conventions and symbols shown in Table ii.

Table ii   Typographical Conventions

Type of Information	Example of Typography
Filenames and pathnames	This structure is declared in *usr/include/sys/time.h*.
IRIX command names and options used in normal text	Update these variables with *systune*; then build a new kernel with *autoconfig -vf*.
Names of program variables, structures, and data types, used in normal text	Global variable *mainSema* points to an IRIX semaphore, which has type *usema_t*.
Names of IRIX system functions, library functions, and functions in example code	Use **mmap()** to map an object into the address space, and **munmap()** to remove it.

When complete lines of example code or commands are set off from normal text, they are displayed as follows.

```
ipcrm -s semid
```

Parts of the code or command that need to be typed exactly as shown are displayed in a monospaced font. Operands that you supply are italicized.

The Process Address Space

Chapter 1

Tells how the virtual address space of a process is created under IRIX. Lists the parts of the address space and their sources; discusses memory mapping; gives tips on cache management.

Process Address Space

When planning a complex program, you must understand how IRIX creates the virtual address space of a process, and how you can modify the normal behavior of the address space. The major topics covered here are as follows:

- "Defining the Address Space" on page 3 tells what the address space is and how it is created.

- "Interrogating the Memory System" on page 11 summarizes the ways your program can get information about the address space.

- "Mapping Segments of Memory" on page 12 documents the different ways that you can create new memory segments with predefined contents.

- "Locking and Unlocking Pages in Memory" on page 23 discusses when and how to lock pages of virtual memory to avoid page faults.

- "Additional Memory Features" on page 27 summarizes functions for address space management.

- "Using Origin2000 Nonuniform Memory" on page 29 describes the use of CC-NUMA memory in the Origin2000 and Onyx2 systems.

Defining the Address Space

Each user-level process has a virtual address space. This term means simply: the set of memory addresses that the process can use without error. When 32-bit addressing is in use, addresses can range from 0 to 0x7fffffff; that is, 2^{31} possible numbers, for a total theoretical size of 2 gigabytes. (Numbers greater than 2^{31} are in the IRIX kernel's address space.)

When 64-bit addressing is used, a process's address space can encompass 2^{40} numbers. (The numbers greater than 2^{40} are reserved for kernel address spaces.) For more details on the structure of physical and virtual address spaces, see the *IRIX Device Driver Programmer's Guide* and the MIPS architecture documents listed on page xxxiv.

Although the address space includes a vast quantity of potential numbers, usually only a small fraction of the addresses are valid.

A *segment* of the address space is any range of contiguous addresses. Certain segments are created or reserved for certain uses.

The address space is called "virtual" because the address numbers are not directly related to physical RAM addresses where the data resides. The mapping from a virtual address to the corresponding real memory location is kept in a table created by the IRIX kernel and used by the MIPS processor chip.

Address Space Boundaries

A process has at least three segments of usable addresses:

- A *text* segment contains the executable image of the program. Another text segment is created for each dynamic shared object (DSO) with which a process is linked.Text segments are always read-only.

- A *data* segment contains the "heap" of dynamically allocated data space. A process can create additional data segments in various ways described later.

- A *stack* segment contains the function-call stack. This segment is extended automatically as needed.

Although the address space begins at location 0, by convention the lowest segment is allocated at 0x0040 0000 (4 MB). Addresses less than this are intentionally left undefined so that any attempt to use them (for example, through an uninitialized pointer variable) causes a hardware exception and stops the program.

Typically, text segments are at smaller virtual addresses and stack and data segments at larger ones, although you should not write code that depends on this.

Tip: The boundaries of all distributed DSOs are declared in the file */usr/lib/so_locations*. When IRIX loads a DSO that is not declared in this file, it seeks a segment of the address space that does not overlap any declared DSO and that will not interfere with growth of the stack segment. To learn more about DSOs, see the rld(1) and dso(5) reference pages, and the *MIPSpro Compiling, Linking, and Performance Tuning Guide*.

Page Numbers and Offsets

IRIX manages memory in units of a page. The size of a page can differ from one system to another. In systems such as the O2 workstation, which support only 32-bit addressing, the page size is always 4,096 bytes. In each 32-bit virtual address,

- the least-significant 12 bits specify an offset from 0 to 0x0fff within a page

- the most-significant 20 bits specify a virtual page number (VPN)

In systems that support 64-bit addressing the page size is greater than 4,096 bytes. The page size is configurable, and in fact different programs can have different page sizes, and a single program can have different size pages for the text segment, stack segment, and data segments. However, the page size is always a power of 2, and the bits of the virtual address are used in the same way: the least-significant bits of an address specify an offset within a page, while the most-significant bits specify the VPN.

You can learn the actual size of a page in the present system with **getpagesize()**, as noted under "Interrogating the Memory System" on page 11.

Page tables, built by IRIX during a **fork()** or **exec()** call, define the address space for a process by specifying which VPNs are defined. These tables are consulted by the hardware. Recently-used table entries are cached for instant lookup in the processor chip, in an array called the Translation Lookaside Buffer (TLB).

Address Definition

Most of the possible addresses in an address space are undefined; that is, not defined in the page tables, not related to contents of any kind, and not available for use. A reference to an undefined address causes a SIGSEGV error.

Addresses are *defined*—that is, made available for potential use—in one of five ways:

Fork   When a process is created using **fork()**, the new process is given a duplicate copy of the parent process's page table, so that any addresses that were defined in the parent's address space are defined the same way in the address space of the new process. (See the fork(2) reference page.)

Exec   The **exec()** function creates a new address space in which to execute a specified program or interpreter. (See the exec(2) reference page.)

Stack | The call stack is created and extended automatically. When a function is entered and more stack space is needed, IRIX makes the stack segment larger, defining new addresses if required.

Mapping | A process can ask IRIX to map (associate byte for byte) a segment of address space to one of a number of special objects, for example, the contents of a file. This is covered further under "Mapping Segments of Memory" on page 12.

Allocation | The **brk()** function extends the heap, the segment devoted to data, to a specific virtual address. The **malloc()** function allocates memory for use, calling **brk()** as required. (See the brk(2), malloc(3), and malloc(3x) reference pages).

An address is defined by an entry in the page tables. A defined address is always related to a *backing store*, a source from which its contents can be refreshed. A page in a text segment is related to the executable file. A page of a data or stack segment is related to a page in a swap partition on disk.

The total size of the defined pages in an address space is its *virtual size*, displayed by the *ps* command under the heading SZ (see the ps(1) reference page).

Once addresses have been defined in the address space by allocation, there is no way to undefine them except to terminate the process. To free allocated memory makes the freed memory available for reuse within the process, but the pages are still defined in the page tables and the swap space is still allocated.

Address Space Limits

The segments of the address space have maximum sizes that are set as resource limits on the process. Hard limits are set by these variables:

rlimit_vmem_max | Total size of the address space of a process

rlimit_data_max | Size of the portion of the address space used for data

rlimit_stack_max | Size of the portion of the address space used for stack

The limits active during a login session can be displayed and changed using the C-shell command *limits*. A program can query the limits with **getrlimit()** and change them with **setrlimit()** (see the getrlimit(2) reference page).

The initial default value and the possible range of a resource limit is established in the kernel tuning parameters. For a quick look at the kernel limits, use

```
fgrep rlimit /var/sysgen/mtune/kernel
```

To examine and change the limits, use *systune* (see the systune(1) reference page):

Example 1-1   Using systune to Check Address Space Limits

```
systune -i
Updates will be made to running system and /unix.install
systune-> rlimit_vmem_max
        rlimit_vmem_max = 536870912 (0x20000000) ll
systune-> resource
group: resource (statically changeable)
...
        rlimit_vmem_max = 536870912 (0x20000000) ll
        rlimit_vmem_cur = 536870912 (0x20000000) ll
...
        rlimit_stack_max = 536870912 (0x20000000) ll
        rlimit_stack_cur = 67108864 (0x4000000) ll
...
```

Tip: These limits interact in the following way: each time your program creates a process with **sproc()** and does not supply a stack area (see the sproc(2) reference page), an address segment equal to *rlimit_stack_max* is dedicated to the stack of the new process. When *rlimit_stack_max* is set high, a program that creates many processes can quickly run into the *rlimit_vmem_max* boundary.

Delayed and Immediate Space Definition

IRIX supports two radically different ways of defining segments of address space.

The conventional behavior of UNIX systems, and the default behavior of current releases of IRIX, is that space created using **brk()** or **malloc()** is immediately defined. Page table entries are created to define the addresses, and swap space is allocated as a backing store. Three results follow from the conventional method:

- A program can detect immediately when swap space is exhausted. A call to **malloc()** returns NULL when memory cannot be allocated. A program can test the limits of swap space by making repeated calls to **malloc()**.

- A large memory allocation by one program can fill the swap disk partition, causing other programs to see out-of-memory errors—whether the program ever uses its allocated memory or not.

- A **fork()** or **exec()** call fails unless there is free space in swap equal to the data and stack sizes of the new process.

By default in IRIX 5.2, and optionally in later releases, IRIX uses a different method sometimes called "virtual swap." In this method, the definition of new segments is delayed until the space is actually used. Functions like **brk()** and **malloc()** merely test the new size of the data segment against the resource limits. They do not actually define the new addresses, and they do not cause swap disk space to be allocated. Addresses are *reserved* with **brk()** or **malloc()**, but they are only *defined* and allocated in swap when your program references them.

When IRIX uses delayed definition ("virtual swap"), it has the following effects:

- A program cannot find the limits of swap space using **malloc()**—it never returns NULL until the program exceeds its resource limit, regardless of available swap.

  Instead, when a program finally accesses a new page of allocated space and there is *at that time* no room in the swap partition, the program receives a SIGKILL signal.

- A large memory allocation by one program cannot monopolize the swap disk until the program actually uses the allocated memory, if it ever does.

- Much less swap space is required for a successful **fork()** call.

You can test whether the system uses virtual swap with the *chkconfig* command (as described in the chkconfig(1) reference page):

```
# chkconfig vswap; echo $status
0
```

As you write a new program, assume that virtual swap may be used. Do not allocate memory merely to find out if you can. Allocate no more memory than your program needs, and use the memory immediately after allocating it.

If you are porting a program written for a conventional UNIX system, you might discover that it tests the limits of allocatable memory by calling **malloc()** until **malloc()** returns a NULL, and then does not use the memory. In this case you have several choices:

- Recode this part of the program to derive the maximum memory size in some more reasonable and portable way, for instance from an environment variable or the size of an input file.

- Using **setrlimit()**, set a lower maximum for *rlimit_data_max*, so that **malloc()** returns NULL at a reasonable allocation size, independent of the swap disk allocation (see the getrlimit(2) reference page).

- Restore the conventional UNIX behavior for the whole system. Use *chkconfig* to turn off the variable *vswap*, and reboot (see the chkconfig(1) reference page).

Note: The function **calloc()** touches all allocated pages in the course of filling them with zeros. Hence memory allocated by **calloc()** is defined as soon as it is allocated. However, you should not rely on this behavior. It is possible to implement **calloc()** in such a way that it, like **malloc()**, does not define allocated pages until they are used. This might be done in a future version of IRIX.

Page Validation

Although an address is defined, the corresponding page is not necessarily loaded in physical memory. The sum of the defined address spaces of all processes is normally far larger than available real memory. IRIX keeps selected pages in real memory. A page that is not present in real memory is marked as "invalid" in the page tables. When the program refers to an address on an invalid page, the CPU traps to the kernel, which supplies the page.

The contents of invalid pages can be supplied in one of the following ways:

Text   Pages of program text—executable code of programs and dynamically linked libraries—can be retrieved on demand from the program file or library files on disk.

Data   Pages of data from the heap and stack can be retrieved from the swap partition or file on disk.

Mapped   When a segment is created by **mmap()**, a backing store file is specified by the program (see "Mapping Segments of Memory" on page 12).

Never used Pages that have been defined but never used can be created as pages of binary zero when they are needed.

When a process refers to a VPN that is defined but invalid, a hardware interrupt occurs. The interrupt handler in the IRIX kernel chooses a page of physical RAM to hold the page. In order to acquire this space, the kernel might have to invalidate some other page belonging to your process or to another process. The contents of the needed page are read from the appropriate backing store into memory, and the process continues to execute.

Page validation takes from 10 to 50 milliseconds. Most applications are not impeded by page fault processing, but a real-time program cannot tolerate these delays.

The total size of all the defined pages in an address space is displayed by the *ps* command under the heading SZ. The aggregate size of the pages that are actually in memory is the *resident set size*, displayed by *ps* under the heading RSS.

Tip: A sophisticated IRIX user might know that the daemon responsible for reading and writing pages from disk was called *vhand*, and its activity could be monitored. However, starting with IRIX 6.4 all such system daemons became "kernel threads" and are no longer visible to commands such as *ps* or *gr_top*.

Read-Only Pages

A page of memory can be marked as valid for reading but invalid for writing. Program text is marked this way because program text is read-only; it is never changed. If a process attempts to modify a read-only page, a hardware interrupt occurs. When the page is truly read-only, the kernel turns this into a SIGSEGV signal to the program. Unless the program is handling this signal, the result is to terminate the program with a segmentation fault.

Copy-on-Write Pages

When **fork()** is executed, the new process shares the pages of the parent process under a rule of *copy-on-write*. The pages in the new address space are marked read-only. When the new process attempts to modify a page, a hardware interrupt occurs. The kernel makes a copy of that page, and changes the new address space to point to the copied page. Then the process continues to execute, modifying the page of which it now has a unique copy.

You can apply the copy-on-write discipline to the pages of an arena shared with other processes (see "Mapping a File for Shared Memory" on page 19).

Interrogating the Memory System

You can get information about the state of the memory system with the system calls shown in Table 1-1.

Table 1-1   Memory System Calls

Memory Information	System Call Invocation
Size of a page (in a data segment)	uiPageSize = getpagesize(); uiPageSize = sysconf(_SC_PAGESIZE);
Virtual and resident sizes of a process	syssgi(SGI_PROCSZ, pid, &uiSZ, &uiRSS);
Maximum stack size of a process	uiStackSize = prctl(PR_GETSTACKSIZE)
Free swap space in 512-byte units	swapctl(SC_GETFREESWAP, &uiBlocks);
Total physical swap space in 512-byte units	swapctl(SC_GETSWAPTOT, &uiBlocks);
Total real memory	sysmp(MP_KERNADDR, MPSA_RMINFO, &rmstruct);
Free real memory	sysmp(MP_KERNADDR, MPSA_RMINFO, &rmstruct);
Total real memory + swap space	sysmp(MP_KERNADDR, MPSA_RMINFO, &rmstruct);

The structure used with the **sysmp()** call shown above has this form (a more detailed layout is in *sys/sysmp.h*):

```
struct rminfo {
   __uint32_t freemem; /* pages of free memory */
   __uint32_t availsmem; /* total real+swap memory space */
   __uint32_t availrmem; /* available real memory space */
   __uint32_t bufmem; /* not useful */
   __uint32_t physmem; /* total real memory space */
};
```

Mapping Segments of Memory

Your process can create new segments within the address space. Such a "mapped" segment can represent

- the contents of a file

- a segment initialized to binary zero

- a POSIX shared memory object

- a view of the kernel's private address space or of physical memory

- a portion of VME A24 or A32 bus address space (when a VME bus exists on the system)

A mapped segment can be private to one address space, or it can be shared between address spaces. When shared, it can be

- read-only to all processes

- read-write to the creating process and read-only to others

- read-write to all sharing processes

- copy-on-write, so that any sharing process that modifies a page is given its own unique copy of that page

Note: Some of the memory-mapping capabilities described in this section are unique to IRIX and nonportable. Some of the capabilities are compatible with System V Release 4 (SVR4). IRIX also supports the POSIX 1003.1b shared memory functions. Compatibility issues with SVR4 and POSIX are noted in the text of this section.

Segment Mapping Function mmap()

The **mmap()** function (see the mmap(2) reference page) creates shared or unshared segments of memory. The syntax and most basic features of **mmap()** are compatible with SVR4 and with POSIX 1003.1b. A few features of **mmap()** are unique to IRIX.

The **mmap()** function performs many kinds of mappings based on six parameters. The function prototype is

```
void * mmap(void *addr, size_t len, int prot, int flags, int fd, off_t off)
```

The function returns the base address of a new segment, or else -1 to indicate that no segment was created. The size of the new segment is *len*, rounded up to a page. An attempt to access data beyond that point causes a SIGBUS signal.

Describing the Mapped Object

Three of the **mmap()** parameters describe the object to be mapped into memory (which is the backing store of the new segment):

fd   A file descriptor returned by **open()** or by the POSIX-defined function **shm_open()** (see the open(2) and shm_open(2) reference pages). All **mmap()** calls require a file descriptor to define the backing store for the mapped segment. The descriptor can represent a file, or it can be based on a pseudo-file that represents kernel memory or a device special file.

off   The offset into the object represented by *fd* where the mapped data begins. When *fd* describes a disk file, *off* is an offset into the file. When *fd* describes memory, *off* is an address in that memory. *off* must be an integral multiple of the memory page size (see "Interrogating the Memory System" on page 11).

len   The number of bytes of data from *fd* to be mapped. The initial size of the segment is *len*, rounded up to a multiple of whole pages.

Describing the New Segment

Three parameters of **mmap()** describe the segment to be created:

addr   Normally 0 to indicate that IRIX should pick a convenient base address, *addr* can specify a virtual address to be the base of the segment. See "Choosing a Segment Address" on page 21.

prot   Access control on the new segment. You use constants to specify a combination of read, write, and execute permission. The access control can be changed later (see "Changing Memory Protection" on page 28).

flags   Options on how the new segment is to be managed.

The elements of *flags* determine the way the segment behaves, and are as follows:

MAP_FIXED   Take *addr* literally.

MAP_PRIVATE   Changes to the mapped data are visible only to this process.

MAP_SHARED   Changes to the mapped data are visible to all processes that map the same object.

13

MAP_AUTOGROW	Extend the object when the process stores beyond its end (not a POSIX feature)
MAP_LOCAL	Map is not visible to other processes in share group (not POSIX)
MAP_AUTORESRV	Delay reserving swap space until a store is done (not POSIX).

The MAP_FIXED element of *flags* modifies the meaning of *addr*. Discussion of this is under "Choosing a Segment Address" on page 21.

The MAP_AUTOGROW element of *flags* specifies what should happen when a process stores data past the current end of the segment (provided storing is allowed by *prot*). When *flags* contains MAP_AUTOGROW, the segment is extended with zero-filled space. Otherwise the initial *len* value is a permanent limit, and an attempt to store more than *len* bytes from the base address causes a SIGSEGV signal.

Two elements of *flags* specify the rules for sharing the segment between two address spaces when the segment is writable:

- MAP_SHARED specifies that changes made to the common pages are visible to other processes sharing the segment. This is the normal setting when a memory arena is shared among multiple processes.

  When a mapped segment is writable, any changes to the segment in memory are also written to the file that is mapped. The mapped file is the backing store for the segment.

  When MAP_AUTOGROW is specified also, a store beyond the end of the segment lengthens the segment and also the file to which it is mapped.

- MAP_PRIVATE specifies that changes to shared pages are private to the process that makes the changes.

  The pages of a private segment are shared on a copy-on-write basis—there is only one copy as long as they are unmodified. When the process that specifies MAP_PRIVATE stores into the segment, that page is copied. The process has a private copy of the modified page from then on. The backing store for unmodified pages is the file, while the backing store for modified pages is the system swap space.

  When MAP_AUTOGROW is specified also, a store beyond the end of the segment lengthens only the private copy of the segment; the file is unchanged.

14

The difference between MAP_SHARED and MAP_PRIVATE is important only when the segment can be modified. When the *prot* argument does not include PROT_WRITE, there is no question of modifying or extending the segment, so the backing store is always the mapped object. However, the choice of MAP_SHARED or MAP_PRIVATE does affect how you lock the mapped segment into memory, if you do; see "Locking Program Text and Data" on page 24.

Processes created with **sproc()** normally share a single address space, including mapped segments (see the sproc(2) reference page). However, if *flags* contains MAP_LOCAL, each new process created with **sproc()** receives a private copy of the mapped segment on a copy-on-write basis.

When the segment is based on a file or on */dev/zero* (see "Mapping a Segment of Zeros" on page 19), **mmap()** normally defines all the pages in the segment. This includes allocating swap space for the pages of a segment based on */dev/zero*. However, if *flags* contains MAP_AUTOGROW, the pages are not defined until they are accessed (see "Delayed and Immediate Space Definition" on page 7).

Note: The MAP_LOCAL and MAP_AUTOGROW flag elements are IRIX features that are not portable to POSIX or to System V.

Mapping a File for I/O

You can use **mmap()** as a simple, low-overhead way of reading and writing a disk file. Open the file using **open()**, but instead of passing the file descriptor to **read()** or **write()**, use it to map the file. Access the file contents as a memory array. The memory accesses are translated into direct calls to the device driver, as follows:

- An attempt to access a mapped page, when the page is not resident in memory, is translated into a call on the read entry point of the device driver to read that page of data.

- When the kernel needs to reclaim a page of physical memory occupied by a page of a mapped file, and the page has been modified, the kernel calls the write entry point of the device driver to write the page. It also writes any modified pages when the file mapping is changed by **munmap()** or another **mmap()** call, when the program applies **msync()** to the segment, or when the program ends.

When mapping a file for input only (when the *prot* argument of **mmap()** does not contain PROT_WRITE), you can use either MAP_SHARED or MAP_PRIVATE. When writing is allowed, you must use MAP_SHARED, or changes will not be reflected in the file.

Tip: Memory mapping provides an excellent way to read a file containing precalculated, constant data used by an interactive program. Time-consuming calculation of the data elements can be done offline by another program; the other program also maps the file in order to fill it with data.

You can lock a mapped file into memory. This is discussed further under "Locking and Unlocking Pages in Memory" on page 23.

Mapped File Sizes

Because the potential 32-bit address space is more than 2000 megabytes (and the 64-bit address space vastly greater), you can in theory map very large files into memory. However, many segments of the virtual address space are preassigned to DSOs (see "Address Space Boundaries" on page 4 and the file */usr/lib/so_locations*), and this restricts the available size of maps in 32-bit space. To map an entire file, follow these steps:

1. Open the file to get a file descriptor.

2. Use **lseek**(*fd*,0,SEEK_END) to discover the size of the file (see the lseek(2) reference page).

3. Map the file with an *off* of 0 and *len* of the file size.

Apparent Process Size

When you map a large file into memory, the space is counted as part of the virtual size of the process. This can lead to very large apparent sizes. For example, under IRIX 5.3 and 6.2, the Object Server maps a large database into memory, with the result that a typical result of *ps -l* looks like this:

```
70 S 0 566 1 0 26 20 * 33481:225 80272230 ? 0:45 objectser
```

The total virtual size of 33481 certainly gets your attention! However, note the more modest real storage size of 225. Most of the mapped pages are not in physical memory. Also realize that the backing store for pages of a mapped file is the file itself—no swap space is used.

Mapping Portions of a File

You do not have to map the entire file; you can map any portion of it, from one page to the file size. Simply specify the desired length as *len* and the starting offset as *off*.

You can remap a file to a different segment by calling **mmap()** again. In this way you can use the *off* parameter of **mmap()** as the logical equivalent of **lseek()**. That is, to map a different segment of the file, specify

- the same file descriptor

- the new offset in *off*

- the current segment base address as *addr*

- MAP_FIXED in *flags* to force the use of *addr* as the base address (otherwise the new portion of the file maps to a different, additional memory segment)

The old segment is replaced with a new segment at the same address, now containing data from a different offset in the file.

Note: Each time you replace a segment with **mmap()**, the previous segment is discarded. The new segment is not locked in memory, even if the old segment was locked.

File Permissions

Access to a file for mapping is controlled by the same file permissions that control I/O to the file. The protection in *prot* must agree with the file permissions. For example, if the file is read-only to the process, **mmap()** does not allow *prot* to specify write or execute access.

Note: When a program runs with superuser privilege for other reasons, file permissions are not a protection against accidental updates.

NFS Considerations

The file that is mapped can be local to the machine, or can be mounted by NFS. In either case, be aware that changes to the file are buffered and are not immediately reflected on disk. Use **msync()** to force modified pages of a segment to be written to disk (see "Synchronizing the Backing Store" on page 28).

If IRIX needs to read a page of a mapped, NFS mounted file, and an NFS error occurs (for example, because the file server has gone down), the error is reflected to your program as a SIGBUS exception.

Caution: When two or more processes in the *same* system map an NFS-mounted file, their image of the file will be consistent. But when two or more processes in *different* systems map the same NFS-mounted file, there is no way to coordinate their updates, and the file can be corrupted.

File Integrity

Any change to a file is immediately visible in the mapped segment. This is always true when *flags* contains MAP_SHARED, and initially true when *flags* contains MAP_PRIVATE. A change to the file can be made by another process that has mapped the same file.

A mapped file can also be changed by a process that opens the file for output and then applies either **write()** to update the file or **ftruncate()** to shorten it (see the write(2) and ftruncate(3) reference pages). In particular, if any process truncates a mapped file, an attempt to access a mapped memory page that corresponds to a now-deleted portion of the file causes a bus error signal (SIGBUS) to be sent.

When MAP_PRIVATE is specified, a private copy of a page of memory is created whenever the process stores into the page (copy-on-write). This prevents the change from being seen by any other process that uses or maps the same file, and it protects the process from detecting any change made to that page by another process. However, this applies only to pages that have been written into.

Frequently you cannot use MAP_PRIVATE because it is important to see data changes and to share them with other processes that map the same file. However, it is also important to prevent an unrelated process from truncating the file and so causing SIGBUS exceptions.

The one sure way to block changes to the file is to install a mandatory file lock. You place a file lock with the **lockf()** function (see Chapter 7, "File and Record Locking"). However, a file lock is normally "advisory"; that is, it is effective only when every process that uses the file also calls **lockf()** before changing it.

You create a mandatory file lock by changing the protection mode of the file, using the **chmod()** function to set the mandatory file lock protection bit (see the chmod(2) reference page). When this is done, a lock placed with **lockf()** is recognized and enforced by **open()**.

Mapping a File for Shared Memory

You can use **mmap()** simply to create a segment of memory that can be shared among unrelated processes.

- In one process, create a file or a POSIX shared memory object to represent the segment.

  Typically a file is located in */var/tmp*, but it can be anywhere. The permissions on the file or POSIX object determine the access permitted to other processes.

- Map the file or POSIX object into memory with **mmap()**; initialize the segment contents by writing into it.

- In another process, get a file descriptor using **open()** or the POSIX function **shm_open()**, specifying the same pathname.

- In that other process, use **mmap()** specifying the file descriptor of the file.

After this procedure, both processes are using the identical segment of memory pages. Data stored by one is immediately visible to the other.

This is the most basic method of sharing a memory segment. More elaborate methods with additional services are discussed in Chapter 3, "Sharing Memory Between Processes."

Mapping a Segment of Zeros

You can use **mmap()** to create a segment of zero-filled memory. Create a file descriptor by applying **open()** to the special device file */dev/zero*. Map this descriptor with *addr* of 0, *off* of 0, and *len* set to the segment size you want.

A segment created this way cannot be shared between unrelated processes. However, it can be shared among any processes that share access to the original file descriptor—that is, processes created with **sproc()** using the PR_SFDS flag (see the sproc(2) reference page). For more information about */dev/zero*, see the zero(7) reference page.

The difference between using **mmap()** of */dev/zero* and **calloc()** is that **calloc()** defines all pages of the segment immediately. When you specify MAP_AUTOGROW, **mmap()** does not actually define a page of the segment until the page is accessed. You can create a very large segment and yet consume swap space in proportion to the pages actually used.

Note: This feature is unique to IRIX. The file */dev/zero* may not exist in other versions of UNIX. Since the feature is nonportable, you should not use the POSIX function **shm_open()** with */dev/zero* (or any device special file).

Mapping Physical Memory

You can use **mmap()** to create a segment that is a window on physical memory. To do so you create a file descriptor by opening the special file */dev/mem*. For more information, see the mem(7) reference page.

Obviously the use of such a segment is nonportable, hardware-dependent, and dependent on the OS release.

Mapping Kernel Virtual Memory

You can use **mmap()** to create a segment that is a window on the kernel's virtual address space. To do so you create a file descriptor by opening the special file */dev/mmem* (note the double "m"). For more information, see the mem(7) (single "m") reference page.

The acceptable *off* and *len* values you can use when mapping */dev/mmem* are defined by the contents of */var/sysgen/master.d/mem*. Normally this file restricts possible mappings to specific hardware registers such as the high-precision clock. For an example of mapping */dev/mmem*, see the example code in the syssgi(2) reference page under the SGI_QUERY_CYCLECNTR argument.

Mapping a VME Device

You can use **mmap()** to create a segment that is a window on the bus address space of a particular VME bus adapter. This allows you to do programmed I/O (PIO) to VME devices.

To do PIO, you create a file descriptor by opening one of the special devices in */dev/vme*. These files correspond to VME devices. For details on the naming of these files, see the usrvme(7) reference page.

The name of the device that you open and pass as the file descriptor determines the bus address space (A16, A24, or A32). The values you specify in *off* and *len* must agree with accessible locations in that VME bus space. A read or write to a location in the mapped segment causes a call to the read or write entry of the kernel device driver for VME PIO. An attempt to read or write an invalid location in the bus address space causes a SIGBUS exception to all processes that have mapped the device.

Note: On the CHALLENGE and Onyx hardware, PIO reads and writes are asynchronous. Following an invalid read or write, as much as 10 milliseconds can elapse before the SIGBUS signal is raised.

For a detailed discussion of VME PIO, see the *IRIX Device Driver Programmer's Guide*.

Note: Mapping of devices through **mmap()** is an IRIX feature that is not defined by POSIX standard. Do not use the POSIX **shm_open()** function with device special files.

Choosing a Segment Address

Normally there is no need to map a segment to any particular virtual address. You specify *addr* as 0 and IRIX picks an unused virtual address. This is the usual method and the recommended one.

You can specify a nonzero value in *addr* to request a particular base address for the new segment. You specify MAP_FIXED in *flags* to say that *addr* is an absolute requirement, and that the segment must begin at *addr* or not be created. If you omit MAP_FIXED, **mmap()** takes a nonzero *addr* as a suggestion only.

Segments at Fixed Offsets

In rare cases you may need to create two or more mapped segments with a fixed relationship between their base addresses. This would be the case when there are offset values in one segment that refer to the other segment, as diagrammed in Figure 1-1.
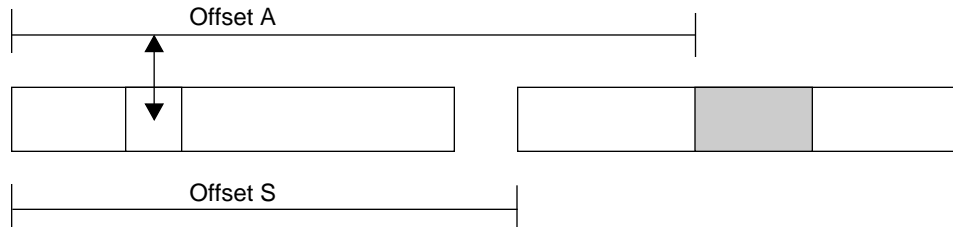
Figure 1-1   Segments With a Fixed Offset Relationship

In Figure 1-1, a word in one segment contains an offset value A giving the distance in bytes to an object in a different mapped segment. Offset A is accurate only when the two segments are separated by a known distance, offset S.

You can create segments in such a relationship using the following procedure.

1. Map a single segment large enough to encompass the lengths of all segments that need fixed offsets. Use 0 for *addr*, allowing IRIX to pick the base address. Let this base address be B.

2. Map the smaller segments over the larger one. For the first (the one at the lowest relative position), specify B for *addr* and MAP_FIXED in *flags*.

3. For the remaining segments, specify $B+S$ for *addr* and MAP_FIXED in *flags*.

The initial, large segment establishes a known base address and reserves enough address space to hold the other segments. The later mappings replace the first one, which cannot be used for its own sake.

Segments at a Fixed Address

You can specify any value for *addr*. IRIX creates the mapping if there is no conflict with an existing segment, or returns an error if the mapping is impossible. However, you cannot normally tell what virtual addresses will be available for mapping in any particular installation or version of the operating system.

There are three exceptions. First, after IRIX has chosen an address for you, you can always map a new segment of the same or shorter length at the same address. This allows you to map different parts of a file into the same segment at different times (see "Mapping Portions of a File" on page 17).

Second, the low 4 MB of the address space are unused (see "Address Space Boundaries" on page 4). It is a very bad idea to map anything into the 0 page because that makes it hard to trap the use of uninitialized pointers. But you can use other parts of the initial 4 MB for mapping.

Third, the MIPS Application Binary Interface (ABI) specification (an extension of the System V ABI published by AT&T) states that addresses from 0x3000 0000 through 0x3ffc 0000 are reserved for user-defined segment base addresses.

You may specify values in this range as *addr* with MAP_FIXED in *flags*. When you map two or more segments into this region, no two segments can occupy the same 256-KB unit. This rule ensures that segments always start in different pages, even when the maximum possible page size is in use. For example, if you want to create two segments each of 4096 bytes, you can place one at 0x30000000 through 0x3000 0fff and the other at 0x3004 0000 through 0x3004 0fff. (256 KB is 0x0004 0000.)

Note: If two programs in the same system attempt to map different objects to the same absolute address, the second attempt fails.

Locking and Unlocking Pages in Memory

A page fault interrupts a process for many milliseconds. Not only are page faults lengthy, their occurrence and frequency are unpredictable. A real-time application cannot tolerate such interruptions. The solution is to lock some or all of the pages of the address space into memory. A page fault cannot occur on a locked page.

Memory Locking Functions

You can use any of the functions summarized in Table 1-2 to lock memory.

Table 1-2   Functions for Locking Memory

Function Name	Compatibility	Purpose and Operation
mlock(3C)	POSIX	Lock a specified range of addresses.
mlockall(3C)	POSIX	Lock the entire address space of the calling process.
mpin(3C)	IRIX	Lock a specified range of addresses.
plock(3C)	SVR4	Lock all program text, or all data, or the entire address space.

Locking memory causes all pages of the specified segments to be defined before they are locked. When virtual swap is in use, it is possible to receive a SIGKILL exception while locking because there was not enough swap space to define all pages (see "Delayed and Immediate Space Definition" on page 7).

Locking pages in memory of course reduces the memory that is available for all other programs in the system. Locking a large program increases the rate of page faults for other programs.

Locking Program Text and Data

Using **mpin()** and **mlock()** you have to calculate the starting address and the length of the segment to be locked. It is relatively easy to calculate the starting address and length of global data or of a mapped segment, but it can be awkward to learn the starting address and length of program text or of stack space.

Using **mlockall()** you lock all of the program text and data as it exists at the time of the call. You specify a flag, either MCL_CURRENT or MCL_FUTURE, to give the scope in time. One possible way to lock only program text is to call **mlockall()** with MCL_CURRENT early in the initialization of a program. The program's text and static data are locked, but not any dynamic or mapped pages that may be created subsequently. Specific ranges of dynamic or mapped data can be locked with **mlock()** as they are created.

Using **plock()** you specify whether to lock text, data, or both. When you specify the text option, the function locks all executable text as loaded for the program, including shared objects (DSOs). (It does not lock segments created with **mmap()** even when you specify PROT_EXEC to **mmap()**. Use **mlock()** or **mpin()** to lock executable, mapped segments.)

When you specify the data option, **plock()** locks the default data (heap) and stack segments, and any mapped segments made with MAP_PRIVATE, as they are defined at the time of the call. If you extend these segments after locking them, the newly defined pages are also locked as they are defined.

Although new pages are locked when they are defined, you still should extend these segments to their maximum size while initializing the program. The reason is that it takes time to extend a segment: the kernel must process a page fault and create a new page frame, possibly writing other pages to backing store to make space.

One way to ensure that the full stack is created before it is locked is to call **plock()** from a function like the function in Example 1-2.

Example 1-2   Function to Lock Maximum Stack Size

```
#define MAX_STACK_DEPTH 100000 /* your best guess */
int call_plock()
{
   char dummy[MAX_STACK_DEPTH];
   return plock(PROCLOCK);
}
```

The large local variable forces the call stack to what you expect will be its maximum size before **plock()** is entered.

The **plock()** function does not lock mapped segments you create with MAP_SHARED. You must lock them individually using **mpin()**. You need to do this from only one of the processes that shares the segment.

Locking Mapped Segments

It may be better for your program to not lock the entire address space, but to lock only a particular mapped segment.

Immediately after calling **mmap()** you have the address and length of the mapped segment. This is a convenient time to call either **mpin()** or **mlock()** to lock the mapped segment.

The **mmap()** flags MAP_AUTOGROW and MAP_AUTORESRV are unique to IRIX and not defined by POSIX. However, the POSIX **mlock()** function for IRIX does recognize autogrow segments. If you lock an autogrow segment with **mpin()**, **mlock()**, or **mlockall()** with the MCL_FUTURE flag, additional pages are locked as they are added to the segment. If you lock the segment with **mlockall()** with the MCL_CURRENT flag, the segment is locked for its current size only and added pages are not locked.

Locking Mapped Files

If you map a file before you use **mlockall(**MCL_CURRENT**)** or **plock()** to lock the data segment into memory (see "Mapping a File for I/O" on page 15), the mapped file is read into the locked pages during the lock operation. If you lock the program with **mlockall(**MCL_FUTURE**)** and then map a file into memory, the mapped file is read into memory and the pages locked.

If you map a file after locking the data segment with **plock()** or **mlockall(**MCL_CURRENT**)**, the new mapped segment is not locked. Pages of file data are read on demand, as the program accesses them.

From these facts you can conclude the following:

- You should map small files before locking memory, thus getting fast access to their contents without paging delays.

- Conversely, if you map a file after locking memory, your program could be delayed for input on any access to the mapped segment.

- However, if you map a large file and then try to lock memory, the attempt to lock could fail because there is not enough physical memory to hold the entire address space including the mapped file.

One alternative is to map an entire file, perhaps hundreds of megabytes, into the address space, but to lock only the portion or portions that are of interest at any moment. For example, a visual simulator could lock the parts of a scenery file that the simulated vehicle is approaching. When the vehicle moves away from a segment of scenery, the simulator could unlock those parts of the file, and possibly use **madvise()** to release them (see "Releasing Unneeded Pages" on page 29).

Unlocking Memory

The function summarized in Table 1-3 are used to unlock memory.

Table 1-3   Functions for Unlocking Memory

Function Name	Compatibility	Purpose and Operation
munlock(3C)	POSIX	Unlock a specified range of locked addresses.
mlockall(3C)	POSIX	Unlock the entire address space of the calling process.
munpin(3C)	IRIX	Unlock a specified range of addresses.
punlock()	SVR4	Unlock addresses locked by **plock()**.

You should avoid mixing function families; for example, if you lock memory with the POSIX function **mlock()**, do not unlock the memory using **munpin()**.

The **mpin()** function maintains a counter for each locked page showing how many times it has been locked. You must call **munpin()** the same number of times before the page is unlocked. This feature is not available through the POSIX and SVR4 interfaces.

Locked pages of an address space are unlocked when the last process using the address space terminates. Locked pages of a mapped segment are unlocked when the last process that mapped the segment unmaps it or terminates.

Additional Memory Features

Your program can work with the IRIX memory manager to change the handling of the address space.

Changing Memory Protection

You can change the memory protection of specified pages using **mprotect()** (see the mprotect(2) reference page). For a segment that contains a whole number of pages, you can specify protection of these types:

Read-only   By making pages read-only, you cause a SIGSEGV signal to be generated in any process that tries to modify them. You could do this as a debugging measure, to trap an intermittent program error.

You can change read-only pages back to read-write.

Read-write   You can put read-write protection on pages of program text, but this is bad idea except in unusual cases. For example, a debugging tool makes text pages read-write in order to set breakpoints.

Executable   Normal data pages cannot be executed. This is a protection against program errors—wild branches into data are trapped quickly. If your program constructs executable code, or reads it from a file, the protection must be changed to executable before the code can be executed.

No access   You can make pages inaccessible while retaining them as part of the address space.

Note: The **mprotect()** function changes the access rights only to the memory image of a mapped file. You can apply it to the pages of a mapped file in order to control access to the file image in memory. However, **mprotect()** does not affect the access rights to the file itself, nor does it prevent other processes from opening and using the file as a file.

Synchronizing the Backing Store

IRIX writes modified pages to the backing store as infrequently as possible, in order to save time. When pages are locked, they are never written to backing store. This does not matter when the pages are ordinary data.

When the pages represent a file mapped into memory, you may want to force IRIX to write any modifications into the file. This creates a checkpoint, a known-good file state from which the program could resume.

The **msync()** function (see the msync(2) reference page) asks IRIX to write a specified segment to backing store. The segment must be a whole multiple of pages. You can optionally request

- synchronous writes, so the call does not return until the disk I/O is complete—ensuring that the data has been written

- page invalidation, so that the memory pages are released and will have to be reloaded from backing store if they are referenced again

Releasing Unneeded Pages

Using the **madvise()** function (see the madvise(2) reference page), you can tell IRIX that a range of pages is not needed by your process. The pages remain defined in the address space, so this is not a means of reducing the need for swap space. However, IRIX puts the pages at the top of its list of pages to be reclaimed when another process (or the calling process) suffers a page fault.

The **madvise()** function is rarely needed by real-time programs, which are usually more concerned with keeping pages in memory than with letting them leave memory. However, there could be a use for it in special cases

Using Origin2000 Nonuniform Memory

In the Origin2000 systems (which include the Origin200 and Onyx2 product lines) physical memory is implemented using a *cache-coherent nonuniform memory architecture*, abbreviated CC-NUMA (or sometimes simply NUMA).

For almost all programs, the CC-NUMA hardware makes no difference at all. The virtual address space as described in this chapter is implemented exactly the same in all versions of IRIX. Your program cannot tell whether the memory hardware is bus-based as in a CHALLENGE system, or uses CC-NUMA as in the Origin2000 (except that in a heavily-loaded multiprocessor, your program will run faster in an Origin than in a CHALLENGE).

However, when you implement a program that has critical performance requirements, uses multithreading, and needs a large memory space—all three conditions must be present—you may need to control the placement of virtual pages in physical memory for best performance.

About Origin Hardware

You need to understand the Origin hardware at a high level in order to understand memory placement.

Basic Building Blocks

The basic building block of an Origin system is a *node*, a single board containing

- Two MIPS R10000 CPUs, each with a secondary cache of 1 MB or 4 MB.

- Some amount of main memory, from 64 MB to 4 GB.

- One *hub* custom ASIC that manages all access to memory in the node.

Nodes are packaged into a *module*. A module contains

- One to four node boards.

- One or two routers, high-bandwidth switches that connect nodes and modules.

- *Crossbow* I/O interface chips.

The Crossbow chips are used to connect I/O devices of all sorts: SCSI, PCI, FDDI, and other types. Each Crossbow chip connects to the hub of one or two nodes, so any I/O card is closely connected to as many as two main-memory banks and as many as four CPUs.

An Origin2000 or Onyx2 system can consist of a single module, or multiple modules can be connected to make a larger system. Modules are connected by their routers. Routers in different modules are connected by special cables, the CrayLink interconnection fabric. Routers form a hypercube topology to minimize the number of hops from any node to any other.

Uniform Addressing

Physical memory is distributed throughout an Origin system, with some memory installed at each node. However, the system maintains a single, uniform, physical address space. Each CPU translates memory addresses from virtual to physical, and presents the physical address to its hub. A few high-order bits in the physical address designate the node where the physical memory is found. The hub uses these bits to direct the memory request as required: to the local memory in its own node, or through a router to another node.

All translation and routing of physical addresses is entirely transparent to software, which operates in a uniform virtual memory space.

Two aspects of memory mapping are not uniform. First, the physical memory map can contain gaps. Not all nodes have the same amount of memory installed. Indeed, there is no requirement that all nodes be present in the system, and in future releases of IRIX it will be possible to remove and replace nodes while the system remains up.

Second, the access time to memory differs, depending on the distance between the memory and the CPU that requests it:

- Memory in the same node is accessed fastest.

- Memory located on another node in the same module costs one or two router hops.

- Memory in another module costs additional router hops.

Normally, memory location relative to a program has an insignificant effect on performance, because

- IRIX takes is careful to locate a process in a CPU near the process's data.

- Most programs are so written that 90% or more of the memory accesses are satisfied from the secondary cache, which is connected directly to the CPU.

- The CrayLink interconnection fabric has an extremely high bandwidth (in excess of 600MB/sec sustained bidirectionally on every link), so each router hop adds only a small fraction of a microsecond to the access time.

Performance problems only arise when multithreaded programs defeat the caching algorithms or place high-bandwidth memory demands from multiple CPUs to a single node.

Cache Coherency

Each CPU in an Origin system has an independent secondary cache, organized as a set of 128-byte *cache lines*. The memory lines that were most recently used by the CPU are stored here for high-speed access.

When two or more CPUs access the same memory, each has an independent copy of that data. There can be as many copies of a data item as there are CPUs; and for some important tables in the IRIX kernel, this may often be the case.

Cache *coherency* means that the system hardware ensures that every cached copy remains a true reflection of the memory data, without software intervention.

Cache coherency requires no effort as long as all CPUs merely read the memory data. The hardware must intervene when a CPU attempts to modify memory. Then, that CPU must be given exclusive ownership of the modified cache line, and all other copies of the same data must be marked invalid, so that when the other CPUs need this data, they will fetch a fresh copy.

Cache Coherency in CHALLENGE Systems

The CHALLENGE and Onyx systems are designed around a central bus over which all memory requests pass. Each CPU board in a CHALLENGE system monitors the bus. When a board observes a write to memory, it checks its own cache and, if it has a copy of that same line, it invalidates the copy. This design, often called a "snoopy cache" because each CPU reads its neighbors' mail, works well when all memory access moves on a single bus.

Cache Coherency in Origin Systems

The cache coherency design of the Origin systems is fundamentally different, because in the Origin machines there is no central bus. Memory access packets can flow within a node or between any two nodes. Instead, cache coherence is implemented using what is called a *directory-based* scheme. The following is a simplified account of it.

Each 128-byte line of main memory is supplied with extra bits, one for each possible node, plus an 8-bit integer for the number of the node that owns the line exclusively. These extra bits are called directory bits. The directory bits are managed as part of memory by the hub chip in the node that contains the memory. The directory bits are not accessible to user-level software. (The kernel can read and write the directory bits using privileged instructions.)

When a CPU accesses an unmodified cache line for reading, the request is routed to the node that contains the memory. The hub chip in that node returns the memory data, and also sets the bit for the reading CPU to 1. When a CPU discards a cached line for any reason, the corresponding bit is set to 0. Thus the directory bits reflect the existence of cached copies of data. As long as all CPUs only read the data, there is no time cost for directory management.

When a CPU wants to modify a cache line, two things happen. The hub chip in the node that contains the memory sends a message to every CPU whose directory bit for that line is 1, telling the CPU to discard its copy because it is no longer valid. And the modifying CPU is noted as the exclusive owner of that line. Any further requests for that line's data are rerouted to the owning CPU, so that it can supply the latest version of the data.

Eventually the owning CPU updates memory and discards the cache line, and the directory status returns to its original condition.

About CC-NUMA Performance Issues

Most programs operate with good performance when they simply treat the system as having a single large, uniform, memory. When this is not the case, IRIX contains tools you can use to exploit the hardware.

About Default Memory Location

Clearly it is a performance advantage for a process to execute on a CPU that is as close as possible to the data used by the process. Default IRIX policies ensure this for most programs:

- Memory is usually allocated on a "first touch" basis; that is, it is allocated in the node where the program that first defines that page is executing. When that is not possible, the memory is allocated as close as possible (in router hops) to the CPU that first accessed the page.

- The IRIX scheduler maintains process affinity to CPUs based on both cache affinity (as in previous versions) and on memory affinity. When a process is ready to run it is dispatched to

    - The CPU where it last ran, if possible

    - The other CPU in the same node, if possible

    - A CPU in a nearby node

The great majority of commands and user programs have memory requirements that fit comfortably in a single node; and most execute at least as well, usually faster, than in any previous Silicon Graphics system.

About Large Memory Use

Only one memory performance issue arises with a single-threaded program. When the program allocates much more virtual memory than is physically available in its node, at least some of its virtual address space is allocated in other nodes. The program pays an access time penalty on some segments of its address space. When this occurs, the penalty is usually unnoticeable as long as the program has good cache behavior.

Typically, IRIX allocates the first-requested memory in the requester's node. When the first-requested memory is also the most-used, average access time still remains low. When this is not the case, there are tools you can use to ensure that specific memory segments are located next to specific CPUs.

About Multithreaded Memory Use

IRIX supports parallel processing under several different models (see Chapter 10, "Models of Parallel Computation"). When a program uses multiple, parallel threads of execution, additional performance issues can arise:

- Cache contention can occur as multiple threads, running in different CPUs, invalidate each other's cached data.

- Default allocation policies can place memory segments in different nodes from the CPUs the threads that use the data.

- Default allocation to a single node, when threads are running in many nodes, can saturate the node with memory requests, slowing access.

These issues are discussed in the following topics.

Dealing With Cache Contention

When one CPU updates a cache line, all other CPUs that refer to the same data must fetch a fresh copy. When a line is used often from multiple CPUs and is also updated frequently, the data is effectively not cached, but accessed at memory speeds.

In addition, when more than one CPU updates the same cache line, the CPUs are forced to execute in turn. Each waits until it can have exclusive ownership of the line. When multiple CPUs update the same line concurrently, the data is accessed at a fraction of memory speeds, and all the CPUs are forced to idle for many cycles.

An update of one 64-bit word invalidates the 15 other words in the same cache line. When the other words are not related to the new data, *false sharing* occurs; that is, variables are invalidated and have to be reloaded from memory merely by the accident of their address, with no logical need.

These cache contention issues are not new to the Origin architecture; they arise in any multiprocessor that supports cache coherency.

Detecting Cache Contention

The first problem with cache contention is to recognize that it is occurring. In earlier systems you diagnosed cache contention by elimination. Now you can use software tools and the hardware features of the MIPS R10000 CPU to detect it directly.

The R10000 includes hardware registers that can count a variety of discrete events during execution, at no performance cost. The R10000 can count individual clock cycles, numbers of loads, stores, and floating-point instructions executed, as well as cache invalidation events.

The IRIX kernel contains support for "virtualizing" the R10000 counter registers, so that each IRIX process appears to have its own set of counters (just as the kernel ensures that each process has its own unique set of other machine register contents).

Included with IRIX is the *perfex* profiling tool (see the perfex(1) reference page). It executes a specified program after setting up the kernel to count the events you specify. At the end of the test run, *perfex* displays the profile of counts. You can use *perfex* to count the number of instructions a program executes, or the number of page faults it encounters, and so on. No recompilation or relinking is required, and the program runs only fractionally slower than normal.

Using *perfex* you can discover approximately how much time a program, or a single thread of a program, loses to cache invalidations, and how many invalidations there were. This allows you to easily distinguish cache contention from other performance problems.

Correcting Cache Contention Problems

Cache contention is corrected by changing the layout of data in the source program. In general terms, the available strategies are these:

1. Minimize the number of variables that are accessed by more than one thread.

2. Segregate nonvolatile data items into different cache lines from volatile items.

3. Isolate volatile items that are not related into separate cache lines to eliminate false sharing.

4. When volatile items are updated together, group them into single cache lines.

A common design for a large program is to define a block of global status variables that is visible to all parallel threads. In the normal course of the program, every CPU caches all or most of such a common area. Read-only access does no harm, but if the items in the block are volatile, contention occurs. For example a global area might contain the anchor for a LIFO queue of some kind. Every time a thread puts or takes an item from the queue, it updates the queue anchor, and invalidates that cache line for every other thread.

It is inevitable that a queue anchor variable will be frequently invalidated. However, the time cost can be isolated to queue access by applying strategy 2: allocate the queue anchor in separate memory from the global status area. Put a nonvolatile pointer to the queue in the status area. Now the cost of fetching the queue anchor is born only by threads that access the queue.

If there are other items that are updated with the queue anchor, such as the lock that controls exclusive access to the queue (see Chapter 4, "Mutual Exclusion"), place those items adjacent to the queue anchor so that all are in the same cache line (strategy 4). However, if there are two queues that are updated at unrelated times, place each in its own cache line (strategy 3).

The locks, semaphores, and message queues that are used to synchronize threads (see "Types of Interprocess Communication Available" on page 46) are global variables that must be updated by any CPU that uses them. It is best to assume that such objects are accessed at memory speeds. Two things can be done to reduce contention:

- Minimize contention for locks and semaphores through algorithmic design. In particular, use more rather than fewer semaphores, and make each stand for the smallest possible resource. (Of course, this makes it more difficult to avoid deadlocks.)

- Never place unrelated synchronization objects in the same cache line (strategy 3). A lock or semaphore can be in the same cache line as the data that it controls, because an update of one usually follows an update of the other (strategy 4).

Carefully review the design of any data collection that is used by parallel code. For example, the root and the first few branches of a binary tree or B-tree are likely to be visited by every CPU that searches that tree, and therefore will be cached by every CPU. Elements at higher levels in the tree may be visited and cached by only a few CPUs.

Other classic data structures can cause cache contention (computer science textbooks on data structures are generally still written from the standpoint of a single-level mainframe memory architecture). For example, a hash table can be implemented compactly, with only a word or two in each entry. But that creates false sharing by putting several table entries (which are unrelated by definition) in the same cache line. To avoid false sharing in a hash table, make each table entry a full 128 bytes, cache-aligned. You can take advantage of the extra bytes in each entry to store a list of overflow hits—such a list can be quickly scanned because the entire cache line is fetched as one memory access.

Getting Optimum Memory Placement

Suppose a Fortran program allocates a 1000 by 1000 array of complex numbers. By default IRIX places this 16 MB memory allocation in the node where the program starts up. But what if the program contains the C$DOACROSS directive to parallelize the DO-loop that processes the array? (See Chapter 11, "Statement-Level Parallelism.") Some number threads—say, four—execute blocks of the DO-loop in parallel, using four CPUs located in two, three, or even four different nodes. Two problems arise:

- At least two of the threads have to pay a penalty of at least one router hop to get the data.

  It would be better to allocate parts of the array in the nodes where those threads are running.

- A single hub chip can easily keep up with the memory demands of two CPUs, but when four CPUs are generating constant memory requests, one hub may saturate, slowing access.

  It would be better to distribute the array data among other nodes—*any* other nodes—to prevent a single hub from being a bottleneck.

Detecting Memory Placement Problems

Unfortunately none of the counter values reported by *perfex* provide a direct diagnosis of bad memory placement. You can suspect memory placement problems from a combination of circumstances:

- Performance does not improve as expected when more parallel threads and CPUs are added.

- The *perfex* report shows a relatively low percentage of cache line reuse (less than 85% secondary cache hits, to pick a common number).

  This is a performance problem you can address for its own sake; but it demonstrates that the program depends on a high memory bandwidth.

- The program has a high CPU utilization, so it is not being delayed for I/O or by synchronization with other threads.

- The program has no other performance problems that can be detected with *perfex* of the Speedshop tools (see the speedshop(1) reference page).

There are two issues: to make sure that each thread concentrates memory access on some definable subset of the data; and second, to make sure that this data is allocated on or near the node where the thread executes.

The first issue is algorithmic. It is not possible for a page of data to be in two nodes at once. When data is used simultaneously by two or more threads, that data must be closer to some threads than to others, and it must be delivered to all threads from a single hub chip. (Parenthetically, what is true of data is not necessarily true of program text, which is read-only. The kernel can and does replicate the pages of common DSOs in every node so that there is no time penalty for fetching instructions from common DSOs like the C or Fortran runtime libraries.)

Programming Desired Memory Placement

When you have a clear separation of data between parallel threads, there are several tools for placing pages near the threads that use them. The tool you use depends on the model of parallel computation you use.

- Using the Fortran compiler, specify how array elements are distributed among the threads of a parallelized loop using compiler directives. The C compiler supports pragma statements for the same purpose.

- Take advantage of IRIX memory-allocation rules to ensure that memory is allocated next to the threads that use it.

- Enable dynamic page migration to handle slowly-changing access patterns.

- Use the *dprof* tool to learn the memory-use patterns of a program (see the dprof(1) reference page).

- Use the *dplace* tool to set the initial memory layout of any program, without needing to modify the source code (see the dplace(1) reference page).

- Code dynamic calls to *dplace* within the program, to request dynamic relocation of data between one program phase and the next.

Using Compiler Directives for Memory Placement

The Silicon Graphics Fortran 77 and Fortran 90 compilers support compiler directives for data placement. You use compiler directives to specify parallel processing over loops. You can supplement these with directives specifying how array sections should be distributed among the nodes that execute the parallel threads.

You use the Fortran directives to declare a static placement for array sections. You can also use directives to specify redistribution of data at runtime, when access patterns change in the course of the program. For details on these directives, see the Fortran programmer's guides cited under "Other Useful References" on page xxxiv.

The Silicon Graphics C and C++ compilers support some pragma statements for data placement. These are documented in the *C Language Reference Manual* (see "Other Useful References" on page xxxiv).

Taking Advantage of First-Touch Allocation

By default, IRIX places memory pages in the nodes where they are first "touched," that is, referenced by a CPU. In order to take advantage of this rule you have to be aware of when a first touch can take place. With reference to the different means of "Address Definition" on page 5,

- The system call **fork()** duplicates the address space, including the placement of all its pages.

- The system call **exec()** creates initial stack and data pages in the node where the new program will run.

- The system calls **brk()** and **sbrk()** extend the virtual address space but do not "touch" new, complete pages.

- The standard and optional library functions **malloc()**, when called to allocate more than a page size aligned on a page boundary, do not touch any *new* pages they allocate. (Space that has been allocated, touched, and freed can be reused, and it stays where it was first touched.)

- The system call **mmap()** does not touch the pages it maps (see "Mapping Segments of Memory" on page 12).

- The library call **calloc()** touches the pages it allocates to fill them with zero.

- The system functions to lock memory pages (see "Locking and Unlocking Pages in Memory" on page 23) do touch the pages they lock.

It is typical to allocate all memory, including work areas used by subprocesses or threads, in the parent process. This practice ensures that all memory is allocated in the node where the parent runs. Instead, the parent process should allocate and touch only data space that is used by multiple threads. Work areas that are unique to a thread should be allocated and touched first by that thread; then they are placed in the node where the thread runs.

Shared memory arenas (see Chapter 3, "Sharing Memory Between Processes") are based on memory-mapping. However, the library function or system call that creates an arena will typically touch at least the beginning of the arena in order to initialize it. If each thread is to have a private data area within an arena, make the private area at least a page in size, allocated on a page-size boundary; and allocate it from the thread that uses it.

Using Round-Robin Allocation

When a Fortran or C program uses statement-level parallelism (based on the multiprocessing library *libmp*—see "Managing Statement-Parallel Execution" on page 252), you can replace the first-touch allocation rule with round-robin allocation. When you set an environment variable _DSM_ROUND_ROBIN, *libmp* distributes all data memory for the program across the nodes in which the program runs. Each new virtual page is allocated in a different node.

Round-robin allocation does not produce optimal placement because there is no relationship between the threads and the pages they use. However, it does ensure that the data will be served by multiple hub chips.

Using Dynamic Page Migration

Dynamic page migration can be enabled for a specific program, or for all programs. When migration is enabled, IRIX keeps track of the source of the references to each page of memory. When a page is being used predominately from a different node, IRIX copies the page contents to the node that is using it, and resets the page tables to direct references to the new location.

Dynamic migration is a relatively expensive operation: besides the overhead of a daemon that uses hardware counters to monitor page usage, a migration itself entails a memory copy of data and the forced invalidation of translate lookaside registers in all affected nodes (see "Page Numbers and Offsets" on page 5). For this reason, migration is not enabled by default. (The system administrator can turn it on for all programs using the *sn* command as described in the sn(1) reference page, but this is not recommended.)

You can experiment to see whether dynamic page migration helps a particular program. It is likely to help when the initial placement of data is not optimal, and when the program maintains consistent access patterns for long periods (many seconds to minutes). When the program has variable, inconsistent access patterns, migration can hurt performance by causing frequent, unhelpful page movements.

To enable migration for a Fortran or C program using libmp, set the _DSM_MIGRATION environment variable, as described in mp(3). In order to enable migration for another type of program, run the program under the *dplace* command with the *-migration* option.

Using Explicit Memory Placement

The *dplace* execution monitor is a powerful tool that runs any program (other than programs that use *libmp*; *dplace* and *libmp* manage the same facilities and cannot be used together) using a custom memory-placement policy that you define using a simple control file. The program you run does not have to be recompiled or modified in any way to take advantage of the memory placement, and it runs at full speed once started.

The *dplace* tool is documented in three reference pages: dplace(1) describes the command syntax and options; dplace(5) documents the control file syntax; and dplace(3) describes how you can call on *dplace* dynamically, from within a program.

Using *dplace* you can:

- Establish the virtual page size of the stack, heap, and text segments individually at sizes from 16 KB to 16 MB. For example, if the *perfex* monitor shows the program is suffering many TLB misses, you can increase the size of a data page, effectively increasing the span of addresses covered by each TLB entry.

- Turn on dynamic page migration for the program, and set the threshold of local to remote accesses that triggers migration.

- Place each process within the program on a specific node, either by node number or with respect to the node where a certain I/O device is attached.

- Distribute the processes of a program among any available cluster of nodes having a specified topology (usually cube topology to minimize router distances between nodes).

- Place specified segments of the virtual address space in designated nodes.

The *dprof* profiler (see the dprof(1) reference page) complements *dplace*. You use *dprof* to run a program and get a trace report showing which pages are read and written by each process in the program.

When you have control of the source code of a program, you can place explicit calls to *dplace* within the code. The program can call *dplace* to move specific processes to specific nodes, or to migrate specific ranges of addresses to nodes.

Interprocess Communication

Chapter 2, "Interprocess Communication"

Provides an overview of the different communication mechanisms, and describes the POSIX, System V, and BSD compatibility features.

Chapter 3, "Sharing Memory Between Processes"

Describes the different ways of sharing segments of memory between different processes.

Chapter 4, "Mutual Exclusion"

Describes semaphores, locks, and other means of synchronization and exclusion between processes and threads.

Chapter 5, "Signalling Events"

Describes the different interfaces to UNIX signals, and the interval timer facilities.

Chapter 6, "Message Queues"

Describes two different facilities for creating and using message queues.

Interprocess Communication

The term *interprocess communication* (IPC) describes any method of coordinating the actions of multiple processes, or sending data from one process to another. IPC is commonly used to allow processes to coordinate the use of shared data objects; for instance, to let two programs update the same data in memory without interfering with each other, or to make data acquired by one process available to others.

This chapter provides an overview of the IPC implementations available, including:

- "Types of Interprocess Communication Available" on page 46
- "Using POSIX IPC" on page 48
- "Using IRIX IPC" on page 49
- "Using System V IPC" on page 49
- "Using 4.2 BSD IPC" on page 52

The following chapters in this Part provide details, as follows:

- Chapter 3, "Sharing Memory Between Processes," covers shared memory.
- Chapter 4, "Mutual Exclusion," covers semaphores, locks, and similar facilities.
- Chapter 5, "Signalling Events," covers the different signal facilities.
- Chapter 6, "Message Queues," describes two varieties of message queue.

Types of Interprocess Communication Available

IRIX is compatible with a broad variety of IPC mechanisms. IRIX conforms to the POSIX standards for real-time extensions (IEEE standard 1003.1b) and threads (IEEE 1003.1c). Other IPC features are compatible with the two major schools of UNIX programming: BSD UNIX and AT&T System V Release 4 (SVR4) UNIX.

Table 2-1 summarizes the types of IPC that IRIX supports, and lists the systems with which IRIX is compatible.

Table 2-1   Types of IPC and Compatibility

Type of IPC	Purpose	Compatibility
Signals	A means of receiving notice of a software or hardware event, asynchronously.	POSIX, SVR4, BSD
Shared memory	A way to create a segment of memory that is mapped into the address space of two or more processes, each of which can access and alter the memory contents.	POSIX, IRIX, SVR4
Semaphores	Software objects used to coordinate access to countable resources.	POSIX, IRIX, SVR4
Locks, Mutexes, and Condition Variables	Software objects used to ensure exclusive use of single resources or code sequences.	POSIX, IRIX
Barriers	Software objects used to ensure that all processes in a group are ready before any of them proceed.	IRIX
Message Queues	Software objects used to exchange an ordered sequence of messages.	POSIX, SVR4
File Locks	A means of gaining exclusive use of all or part of a file.	SVR4, BSD
Sockets	Virtual data connections between processes that may be in different systems.	BSD

The different implementations of these IPC features can be summarized as follows:

- POSIX compliant library calls are provided for signal handling, shared memory, semaphores, mutexes, condition variables, and message queues. The implementation is highly tuned and has low system overhead. POSIX facilities are usable from POSIX threads (see Chapter 13, "Thread-Level Parallelism").

- IRIX unique library calls are provided for shared memory, semaphores, locks, and barriers. The implementation has slightly more overhead than POSIX operations, but sometimes takes advantage of concurrent hardware in multiprocessors, and has a number of special features, such as the ability to apply **poll()** to semaphores.

- System function calls compatible with AT&T System V Release 4 are provided for signal handling, shared memory, semaphores, message queues, and file locking. The implementation is provided for ease of porting software, but is not particularly efficient.

- Library functions compatible with BSD UNIX are provided for signal handling, file locking, and socket support.

Select your IPC mechanisms based on these guidelines:

- Never mix the implementations of a given mechanism in a single program. For example, unpredictable results can follow when a single program mixes POSIX and System V signal-handling functions, or mixes both BSD and System V file locking calls.

- The POSIX libraries are the newest implementations, and in many cases they are the most efficient.

- A program based on POSIX threads should use POSIX synchronization mechanisms because they are optimized for pthreads use.

- Use System V IPC functions for code that must comply with the MIPS ABI, or code that you are porting from another System V operating system.

Using POSIX IPC

In order to use the POSIX IPC functions described in this part of the book, you must include the correct header files and libraries when compiling.

The header files required for each function are listed in the reference pages for the functions.

POSIX IPC functions are defined in the standard *libc* library. That library is included automatically in any link by the *cc* command.

POSIX IPC Name Space

POSIX shared memory segments, named semaphores, and message queues are persistent objects that survive the termination of the program that creates them (unless the program explicitly removes them). The POSIX standard specifies that these persistent names can be implemented in the filesystem, and the current IRIX implementation does use filenames in the filesystem to represent IPC objects. In order to access a named semaphore or message queue, a program opens the object using a pathname, similar to the way a program opens a disk file.

Because these persistent objects are currently implemented as files, you can display and access them using IRIX commands for files such as *ls*, *rm*, *chmod* and *chown*. However, you should keep in mind that this is an implementation choice, not a standardized behavior. Other implementations of POSIX IPC may not use the filesystem as a name space for IPC objects, and the IRIX implementation is free to change its implementation in the future. For best portability, do not assume that IPC objects are always files.

If you plan to share an object between processes that could be started from different working directories, you should always open the object using the full pathname starting with a slash ("/"). That ensures that unrelated processes always refer to the same object, regardless of their current working directory.

When a shared object is temporary, you can use the **tempnam()** library function to generate a temporary pathname (see the tempnam(3) reference page).

Other POSIX IPC objects—nameless semaphores, mutexes, and condition variables—are not persistent, but exist only in memory and are identified only by their addresses. They disappear when the programs that use them terminate.

Using IRIX IPC

The IRIX IPC facilities are designed to meet the demands of parallel programming in multiprocessor systems. They offer advantages for this use, but they are IRIX specific, so programs using them are not portable to other systems.

In order to use any IRIX IPC functions, you need to include the correct header files and link libraries when compiling. The header files required for each function are listed in the reference pages for the functions.

IRIX IPC functions are defined in the standard *libc* library (it is included automatically in any link by the *cc* command) and in the *libmpc* library, which you include with *-lmpc*.

IRIX IPC functions all require the use of a *shared arena*, a segment of memory that can be mapped into the address spaces of multiple processes. The first step in preparing to use any IRIX IPC object is to create a shared arena, as documented under "Initializing Arena Attributes" on page 61.

A shared arena is identified with a file that acts as the backing store for the arena memory. Communicating processes gain access to the arena by specifying its filename. All processes using the same arena have access to the same set of IPC objects. This makes it relatively easy for unrelated processes to communicate using IRIX IPC; they only have to know the filename of the arena to gain access.

Using System V IPC

IRIX supports SVR4 functions for signals, shared memory, semaphores, message queues, and file locking. To use them you need to include the correct header files when compiling. The header files required for each function are listed in the reference pages for the functions.

System V functions are primarily kernel functions. No special library linkage is required to access them. There is general discussion of SVR4 IPC operations in the intro(2) reference page.

SVR4 IPC Name Space

All SVR4 IPC objects are named in a special IPC name space. An object such as a shared memory segment or message queue is named by a numeric key, and has the following attributes (which are defined in the header file *sys/ipc.h*):

- the UID and GID of the creating process

- the UID and GID of the owning process (which can be different from the creator)

- access permissions in the same format as used with files

The commands and functions used to manage the IPC name space are listed in Table 2-2.

Table 2-2   SVR4 IPC Name Space Management

Function Name	Purpose and Operation
ipcs(1)	List existing shared memory segments (and other IPC objects) in the system name space with their status.
ipcrm(1)	Remove a shared memory segment (or other IPC object) from the system name space.
ftok(3)	Create a semi-unique numeric key based on a file pathname.

Configuring the IPC Name Space

SVR4 IPC objects are stored in kernel tables of limited, fixed size. You configure the size of these tables by changing kernel tunable parameters. These parameters are documented in detail in the book *IRIX Admin: System Configuration and Operation* (007-2859-*nnn*). See "Appendix A: IRIX Kernel Tunable Parameters."

Listing and Removing Persistent Objects

Objects in the IPC name space are created by programs and can be removed by programs. However, IPC objects by definition are used by multiple processes, and it is sometimes a problem to determine which process should remove an object, and when it is safe to do so.

For this reason, IPC objects are often created and never removed. In these cases, they persist until the system is rebooted, or until they are removed manually.

You can list all the components of the IPC name space using the *ipcs* command. You can remove an object with the *ipcrm* command. If you remove an object that is in use, unpredictable results will follow.

Access Permissions

IPC objects are not part of any filesystem, but access to IPC objects is controlled by rules like the rules that govern file access. For example, if the access permissions of a shared memory segment are set to 640, the segment can be read-write for processes that have the same UID as the segment owner, but the segment is read-only to processes that have only the GID of the owner, and is inaccessible to other processes.

Choosing and Communicating Key Values

The "name" of an IPC object is an integer. Two small problems are: how a program can select a unique key to use when making an IPC object, and how to communicate the key to all the processes that need access to the object. The **ftok()** library function can be used to create a predictable key based on a file pathname. For example, unrelated but cooperating programs can agree to use **ftok()** with a designated project file and project code, so that each program will arrive at the same key.

Using ID Numbers

When an IPC object is created, it has the key it is given by the creating process, but it is also assigned a second integer, the ID. The key number is chosen by the application, and is predictable. If the application creates the object each time the application starts up, the key is always the same. The ID number is arbitrary, and a new ID is created each time an object is created.

A process can gain access to an object based on either number, the key or the ID. For example, an SVR4 shared memory segment has a key and an ID. The **shmget()** function takes a key and returns the corresponding ID. The ID is used to attach the segment. However, if a process knows the ID, it can simply use it, without first calling **shmget()** to obtain it.

Private Key Values

When creating an IPC object, you can specify a key of KEY_PRIVATE (0). This causes an object to be created and recorded in the IPC name space with a key of 0. The created object cannot be accessed from another process by key, because if another process uses KEY_PRIVATE, it creates its own object. However, another process can access a key-private object using the object's ID number.

You can use the KEY_PRIVATE feature when you want to create an IPC object for use within a single process or share group (a share group is the set of processes that share one address space; see "Process Creation and Share Groups" on page 256). The IPC object can be used within the share group based on its address or by ID number. Because it has no key, it cannot be used outside the share group.

Using 4.2 BSD IPC

The 4.2 BSD functions for signals and file locking are available. To use them, you must include the correct header files and link libraries when compiling. The header files required for each function are listed in the reference pages for the functions.

One header file, *signal.h*, declares both SVR4 and BSD signal-handling functions. Some of the BSD and SVR4 functions have the same names, but different types of arguments or different results when called. In order to declare the BSD family of signal functions in your program, you must be sure to define the compiler variable _BSD_SIGNALS or _BSD_COMPAT to the compiler. You could do this directly in the source code. More often you will manage compilation with *make*, and you will include *-D_BSD_SIGNALS* as one of the compiler options in the Makefile.

The BSD compatible function for file locking, **flock()**, is defined in the standard *libc* library. That library is included automatically in any link by the *cc* command. However, when you are using C++ (not C), the function name "flock" conflicts with a structure name declared in *sys/fcntl.h*. In order to define the **flock()** function and not the structure, define the compiler variable _BSD_COMPAT.

A BSD-compatible kernel function for managing the termination of child processes, wait3(), is discussed under "Process "Reaping"" on page 259.

Sharing Memory Between Processes

There are three families of functions that let you create a segment of memory and share it among the address spaces of multiple processes. All produce the same result: a segment of memory that can be accessed or updated asynchronously by more than one process. You have to design protocols that prevent one process from changing shared data while another process is using the same data (see Chapter 4, "Mutual Exclusion").

This chapter covers three major topics:

- "POSIX Shared Memory Operations" on page 55 describes the POSIX functions for sharing memory.

- "IRIX Shared Memory Arenas" on page 61 describes IRIX shared memory arenas.

- "System V Shared Memory Functions" on page 71 describes the SVR4 functions.

Overview of Memory Sharing

The address space is the range of memory locations that a process can use without an error. (The concept of the address space is covered in detail in Chapter 1, "Process Address Space.") In a pthreads program, all threads use the same address space and share its contents. In a program that starts multiple, lightweight processes with **sproc()**, all processes share the same address space and its contents. In these programs, the entire address space is shared automatically.

Normally, distinct processes (created by the **fork()** or **exec()** system calls) have distinct address spaces, with no writable contents in common. The facilities described in this chapter allow you to define a segment of memory that can be part of the address space of more than one process. Then processes or threads running in different address spaces can share data simply by referring to the contents of the shared segment in memory.

Shared Memory Based on mmap()

The basic IRIX system operation for shared memory is the **mmap()** function, with which a process makes the contents of a file part of its address space. The fundamental uses of **mmap()** are covered under "Mapping Segments of Memory" on page 12 (see also the mmap(2) reference page). When two or more processes map the same file into memory with the MAP_SHARED option, that single segment is part of both address spaces, and the processes can update its contents concurrently.

The POSIX shared memory facility is a simple, formal interface to the use of **mmap()** to share segments. The IRIX support for shared arenas is an extension of **mmap()** to make it simpler to create a shared allocation arena and coordinate its use. The SVR4 facilities do not directly use **mmap()** but have similar results.

Sharing Memory Between 32-Bit and 64-Bit Processes

Larger Silicon Graphics systems support both 32-bit and 64-bit programs at the same time. It is possible for a memory segment to be mapped by programs using 32-bit addresses, and simultaneously mapped by programs that use 64-bit addresses. There is nothing to prevent such sharing.

However, such sharing can work satisfactorily only when the contents of the shared segment include no addresses at all. Pointer values stored by a 64-bit program can't be used by a 32-bit program and vice versa. Also the two programs will disagree about the size and offset of structure fields when structures contain addresses. For example, if you initialize an allocation arena with **acreate()** from a 64-bit program, a 32-bit program calling **amalloc()** on that same arena will almost certainly crash or corrupt the arena pointers.

You can use POSIX shared memory, SVR4 shared memory, or basic **mmap()** to share a segment between a 32-bit and a 64-bit program, provided you take pains to ensure that both programs view the data contents as having the same binary structure, and that no addresses are shared. You cannot use an IRIX shared memory arena between 32-bit and 64-bit programs at all, because the **usinit()** function stores addresses in the arena.

POSIX Shared Memory Operations

Shared-memory support specified by POSIX is based on the functions summarized in Table 3-1.

Table 3-1   POSIX Shared Memory Functions

Function Name	Purpose and Operation
mmap(2)	Map a file or shared memory object into the address space.
shm_open(2)	Create, or gain access to, a shared memory object.
shm_unlink(2)	Destroy a shared memory object when no references to it remain open.

The use of **mmap()** is described at length under "Mapping Segments of Memory" on page 12. In essence, **mmap()** takes a file descriptor and makes the contents of the described object accessible as a segment of memory in the address space. In IRIX, a file descriptor can describe a disk file, or a device, or a special pseudo-device such as */dev/kmem*. Thus **mmap()** can make a variety of objects part of the address space. POSIX adds one more type of mappable object, a persistent shared segment you create using the **shm_open()** function.

Creating a Shared Object

The **shm_open()** function is very similar to the **open()** function and takes the same arguments (compare the shm_open(2) and open(2) reference pages). The arguments are as follows:

path   Name of object, a character string in the form of a file pathname.

oflag   Option flags, detailed in the reference page and discussed in following text.

mode   Access mode for the opened object

In order to declare **shm_open()** and its arguments you need to include both *sys/mman.h* and *fcntl.h* header files.

Shared Object Pathname

The POSIX standard says that a shared object name has the form of a file pathname, but the standard leaves it "implementation defined" whether the object is actually a file or not. In the IRIX implementation, a shared memory object is also a file. The pathname you specify for a shared memory object is interpreted exactly like the pathname of a disk file that you pass to **open()**. When you create a new object, you also create a disk file of the same name. (See "POSIX IPC Name Space" on page 48.)

You can display the size, ownership, and permissions of an existing shared segment using *ls -l*. You can dump its contents with a command such as *od -X*. You can remove it with *rm*.

Shared Object Open Flags

The flags you pass to **shm_open()** control its actions, as follows:

O_RDONLY	Access can be used only for reading.
O_RDWR	Access can be read-write (however, you can enforce read-only access when calling **mmap()**).
O_CREAT	If the object does not exist, create it.
O_TRUNC	If the object does exist and O_RDWR is specified, truncate it to zero length.
O_EXCL	If the object does exist and O_CREAT is specified, return the EEXIST error code.

The flags have the same meaning when opening a disk file with **open()**. However, a number of other flags allowed by **open()** are not relevant to shared memory objects.

You can use the combination O_CREAT+O_EXCL to ensure that only one process initializes a shared object.

Shared Object Access Mode

The access mode that you specify when creating an object governs the users and groups that can open the object later, exactly as with a disk file.

Using the Shared Object File Descriptor

The value returned by **shm_open()** is a file descriptor and you can use it as such; for example you can apply the **dup()** function to make a copy of it. You can also use it as an argument to **fcntl()**, but most of the features of **fcntl()** are irrelevant to a shared memory object. (See the dup(2) and fcntl(2) reference pages.)

Using a Shared Object

In order to use a shared object, your program first opens it with **shm_open()**, then maps it into memory with **mmap()**. The arguments to **mmap()** include

- the file descriptor for the shared object

- the size of the memory segment

- access protection flags

The returned value is the base address of the segment in memory. You can then use it like any block of memory. For example, you could create an allocation arena in the segment using the **acreate()** function (see the amalloc(3) reference page). For more on the use of **mmap()**, read "Segment Mapping Function mmap()" on page 12 and "Mapping a File for Shared Memory" on page 19.

Example Program

The program in Example 3-1 allows you to experiment with **shm_open()** and **mmap()** from the command line. The program accepts the following command-line arguments:

path	The pathname of a shared memory segment (file) that exists or that is to be created.
-p *perms*	The access permissions to apply to a newly-created segment, for example -p 0664.
-s *bytes*	The initial size at which to map the segment, for example -s 0x80000.
-c	Use the O_CREAT flag with **open()**, creating the segment if it doesn't exist.
-x	Use the O_EXCL flag with **open()**, requiring the segment to not exist.
-t	Use the O_TRUNC flag with **open()**, truncating the file to zero length.

<table>
<tr><td>-r</td><td>Use the O_RDONLY flag with open() and PROT_READ with mmap(). If this option is not used, the program uses O_RDWR with open() and PROT_READ, PROT_WRITE, PROT_AUTOGROW with mmap().</td></tr>
<tr><td>-w</td><td>Wait for keyboard input before exiting, allowing you to run other copies of the program while this one has the segment mapped.</td></tr>
</table>

To create a segment named */var/tmp/test.seg*, use a command such as

```
shm_open -c -x -p 0644 -s 0x80000 /var/tmp/test.seg
```

To attach that segment read-only and then wait, use the command

```
shm_open -r -w /var/tmp/test.seg
```

From a different terminal window, enter the command

```
shm_open /var/tmp/test.seg
```

In the original window, press <Enter> and observe that the value of the first word of the shared segment changed during the wait.

Example 3-1   POSIX Program to Demonstrate shm_open()

```
/*
||   Program to test shm_open(3).
||      shm_open    [-p <perms>] [-s <bytes>] [-c] [-x] [-r] [-t] [-w] <path>
||      -p <perms>  access mode to use when creating, default 0600
||      -s <bytes>  size of segment to map, default 64K
||      -c          use O_CREAT
||      -x          use O_EXCL
||      -r          use O_RDONLY, default is O_RDWR
||      -t          use O_TRUNC
||      -w          wait for keyboard input before exiting
||      <path>      the pathname of the queue, required
*/
#include <sys/mman.h>   /* shared memory and mmap() */
#include <unistd.h>     /* for getopt() */
#include <errno.h>      /* errno and perror */
#include <fcntl.h>      /* O_flags */
#include <stdio.h>
int main(int argc, char **argv)
{
    int perms = 0600;            /* permissions */
    size_t size = 65536;         /* segment size */
```

```
int oflags = 0;                /* open flags receives -c, -x, -t */
int ropt = 0;                  /* -r option seen */
int wopt = 0;                  /* -w option seen */
int shm_fd;                    /* file descriptor */
int mprot = PROT_READ;         /* protection flags to mmap */
int mflags = MAP_SHARED;       /* mmap flags */
void *attach;                  /* assigned memory adddress */
char *path;                    /* ->first non-option argument */
int c;
while ( -1 != (c = getopt(argc,argv,"p:s:cxrtw")) )
{
    switch (c)
    {
    case 'p': /* permissions */
        perms = (int) strtoul(optarg, NULL, 0);
        break;
    case 's': /* segment size */
        size = (size_t) strtoul(optarg, NULL, 0);
        break;
    case 'c': /* use O_CREAT */
        oflags |= O_CREAT;
        break;
    case 'x': /* use O_EXCL */
        oflags |= O_EXCL;
        break;
    case 't': /* use O_TRUNC */
        oflags |= O_TRUNC;
        break;
    case 'r': /* use O_RDONLY */
        ropt = 1;
        break;
    case 'w': /* wait after attaching */
        wopt = 1;
        break;
    default: /* unknown or missing argument */
        return -1;
    } /* switch */
} /* while */
if (optind < argc)
    path = argv[optind]; /* first non-option argument */
else
    { printf("Segment pathname required\n"); return -1; }
if (0==ropt)
{ /* read-write access, reflect in mprot and mflags */
    oflags |= O_RDWR;
```

```
        mprot |= PROT_WRITE;
        mflags |= MAP_AUTOGROW + MAP_AUTORESRV;
    }
    else
    { /* read-only access, mprot and mflags defaults ok */
        oflags |= O_RDONLY;
    }
    shm_fd = shm_open(path,oflags,perms);
    if (-1 != shm_fd)
    {
        attach = mmap(NULL,size,mprot,mflags,shm_fd,(off_t)0);
        if (attach != MAP_FAILED) /* mmap worked */
        {
            printf("Attached at 0x%lx, first word = 0x%lx\n",
                                attach,      *((pid_t*)attach));
            if (mprot & PROT_WRITE)
            {
                *((pid_t *)attach) = getpid();
                printf("Set first word to 0x%lx\n",*((pid_t*)attach));
            }
            if (wopt) /* wait a while, report possibly-different value */
            {
                char inp[80];
                printf("Waiting for return key before unmapping...");
                gets(inp);
                printf("First word is now 0x%lx\n",*((pid_t*)attach));
            }
            if (munmap(attach,size))
                perror("munmap()");
        }
        else
            perror("mmap()");
    }
    else
        perror("shm_open()");
    return errno;
}
```

IRIX Shared Memory Arenas

The shared memory arena is basic to all IRIX IPC mechanisms. IRIX semaphores, locks, and barriers are all represented as objects within a shared arena.

Overview of Shared Arenas

A shared arena is a segment of memory that can be made part of the address space of more than one process. Each shared arena is associated with a disk file that acts as a backing store for the file (see "Page Validation" on page 9). Each process that wants to share access to the arena does so by specifying the file pathname of the file. The file pathname acts as the public name of the memory segment. The file access permissions determine which user IDs and group IDs can share the file.

The functions you use to manage a shared arena are discussed in the following topics and are summarized in Table 3-2.

Table 3-2   IRIX Shared Arena Management Functions

Function Name	Purpose and Operation
usconfig(3)	Establish the default size of an arena, the number of concurrent processes that can use it, and the features of IPC objects in it.
usinit(3)	Create an arena or join an existing arena.
usadd(3)	Join an existing arena.

Initializing Arena Attributes

A program creates a shared memory arena with the **usinit()** function. However, many attributes of a new arena are set by preceding calls to **usconfig()**. The normal sequence of operations is to make several calls to **usconfig()** to establish arena attributes, then to make one call to **usinit()** to create the arena.

You call **usconfig()** to establish the features summarized in Table 3-3.

Table 3-3   Arena Features Set Using usconfig()

usconfig() Flag Name	Meaning
CONF_INITSIZE	The initial size of the arena segment. The default is 64 KB. Often you know that more is needed.
CONF_AUTOGROW	Whether or not the arena can grow automatically as more IPC objects or data objects are allocated (default: yes).
CONF_INITUSERS	The largest number of concurrent processes that can use the arena. The default is 8; if more processes than this will use IPC, the limit must be set higher.
CONF_CHMOD	The effective file permissions on arena access. The default is 600, allowing only processes with the effective UID of the creating process to attach the arena.
CONF_ARENATYPE	Establish whether the arena can be attached by general processes or only by members of one program (a share group).
CONF_LOCKTYPE	Whether or not lock objects allocated in the arena collect metering statistics as they are used.
CONF_ATTACHADDR	An explicit memory base address for the next arena to be created (see "Choosing a Segment Address" on page 21).
CONF_HISTON CONF_HISTOFF	Start and stop collecting usage history (more bulky than metering information) for semaphores in a specified arena.
CONF_HISTSIZE	Set the maximum size of semaphore history records.

See the usconfig(3) reference page for a complete list of attributes. The use of metering and history information for locks and semaphores is covered in Chapter 4, "Mutual Exclusion."

Tip: In programs that use an arena and start a varying number of child processes, it is a common mistake to find that the eighth child process cannot join the arena. This occurs simply because **usconfig()** has not been called with CONF_INITUSERS to set the number of users higher than the default 8 before the arena was created.

Creating an Arena

After setting the arena attributes with **usconfig()**, the program calls **usinit()**, specifying a file pathname string.

Tip: The **mktemp()** library function can be used to create a unique temporary filename (see the mktemp(3C) reference page).

If the specified file doesn't exist, **usinit()** creates it (and gives it the access permissions specified to **usinit()** with CONF_CHMOD). If a shared arena already exists based on that name, **usinit()** joins that shared arena. If the file exists but is not yet a shared arena, **usinit()** overwrites it. In any case, **usinit()** is subject to normal filesystem permission tests, and it returns an error if the process doesn't have read and write permission on the file (if it already exists) or permission to create the file (if it doesn't exist).

Code to prepare an arena is shown in Example 3-2.

Example 3-2   Initializing a Shared Memory Arena

```
usptr_t
makeArena(size_t initSize, int nProcs)
{
   int ret;
   char * tmpname = "/var/tmp/arenaXXXXXX";
   if (ret = usconfig(CONF_INITUSERS, nProcs))
   { perror("usconfig(#users)"); return 0; }
   if (ret = usconfig(CONF_INITSIZE, initSize))
   { perror("usconfig(size)"); return 0; }
   return usinit(mktemp(tmpname));
}
```

Joining an Arena

Only one process creates a shared arena. Other processes "join" or "attach" the arena. There are three ways of doing this. When the arena is not restricted to a single process family (either by file permissions or by CONF_ARENATYPE setting), any process that calls **usinit()** and passes the same pathname string gains access to the same arena at the same virtual base address. This process need not be related in any way to the process that created the arena.

Restricting Access to an Arena

You can restrict arena access to a single process and the children it creates with **sproc()** (a share group; see "Process Creation and Share Groups" on page 256) by calling **usconfig()** to set CONF_ARENATYPE to US_SHAREDONLY before creating the arena. When this is done, the file is unlinked immediately after the arena is created. Then a call to **usinit()** with the same pathname from a different process creates a different arena, one that is not shared with the first one. This has several side-effects that are detailed in usconfig(3).

Arena Access From Processes in a Share Group

An arena is a segment in the address space of a process. When that process creates a new process using **sproc()**, the child process usually shares the same address space (see the sproc(2) reference page and Chapter 12, "Process-Level Parallelism"). The child process has access to the arena segment on the same basis as the parent process. However, the child process needs to join the arena formally.

The child process should join the arena by calling **usadd()**, passing the address of the arena. The child should test the return code of this function, since it can reflect an error in either of two cases:

- The arena has not been created, or an incorrect arena address was passed.

- The arena was not configured to allow enough using processes, and no more users can be allowed.

A child process can join an arena automatically, simply by using a semaphore, lock, or barrier that was allocated within that arena. These function calls perform an automatic call to **usadd()**. However, they can also encounter the error that too many processes are already using the arena. It is best for the child process to check for this condition with an explicit call to **usadd()**.

Allocating in an Arena

Allocating shared memory from a shared arena is much like the regular process of allocating memory using the **malloc()** and **free()** library routines. The functions related to allocation within an arena are summarized in Table 3-4.

Table 3-4   IRIX Shared Memory Arena Allocation Functions

Function Name	Purpose and Operation
usmalloc(3)	Allocate an object of specified size in an arena.
uscalloc(3)	Allocate an array of zero-filled units in an arena.
usmemalign(3)	Allocate an object of specified size on a specified alignment boundary in an arena.
usrealloc(3)	Change the allocated size of an object in an arena.
usrealloc(3)	Change the allocated size of an array created with **uscalloc()**.
usmallocblksize(3)	Query the actual size of an object as allocated.
usfree(3)	Release an object allocated in an arena.
usmallopt(3)	Tune the allocation algorithm using constants described in amallopt(3).
usmallinfo(3)	Query allocation statistics (see amallinfo(3) for structure fields).

The address of an object allocated using **usmalloc()** or a related function is a valid address in any process that is attached to the shared arena. If the address is passed to a process that has not attached the arena, the address is not valid for that process and its use will cause a SIGSEGV.

The **usmalloc()** family of functions is based on the arena-allocation function family described in the amalloc(3) reference page. The **usmallopt()** function is the same as the **amallopt()** function, and both provide several options for modifying the memory allocation methods in a particular arena. In a similar way, **usmallinfo()** is the same as **amallinfo()**, and both return detailed statistics on usage of memory allocation in one arena.

Exchanging the First Datum

The processes using a shared arena typically need to locate some fundamental data structure that has been allocated within the arena. For example, the parent process creates a foundation data structure in the arena, and initializes it with pointers to other objects within the arena. Any process starting to use the arena needs the address of the foundation structure in order to find all the other objects used by the application.

The shared arena has a special one-pointer field for storing such a basic address. This area is accessed using the functions summarized in Table 3-5.

Table 3-5   IRIX Shared Memory First-Datum Functions

Function Name	Purpose and Operation
usputinfo(3)	Set the shared-pointer field of an arena to a value.
usgetinfo(3)	Retrieve the value of the shared-pointer field of an arena.
uscasinfo(3)	Change the shared-pointer field using a compare-and-swap.

Note: The precision of the **usgetinfo()** field in an arena, 32 or 64 bits, depends on the execution model of the program that creates the arena. This is one reason that processes compiled to different models cannot share one arena (see "Sharing Memory Between 32-Bit and 64-Bit Processes" on page 54).

Often, the parent process creates and initializes the arena before it creates any of the child processes that will share the arena. In this case, you expect no race conditions. The parent can set the shared pointer using **usputinfo()** because no other process is using the arena at that time. Each child process can fetch the value with **usgetinfo()**.

The purpose of **uscasinfo()** is to change the contents of the field in an atomic fashion, avoiding any race condition between concurrent processes in a multiprocessor. All three functions are discussed in detail in the usputinfo(3P) reference page.

Tip: The data type of the shared pointer field is void*, a 64-bit value when the program is compiled to the 64-bit model. If you need to cast the value to an integer, use the type _ _*psint_t*, a pointer-sized integer in any model.

In the less-common case when an arena is shared by unrelated processes, each process that calls **usinit()** might be the first one to create the arena—or might not. If the calling process is the first, it should initialize the basic contents and set the shared pointer. If it is not the first, it should use the initialized contents that another process has already prepared. This problem is resolved with **uscasinfo()**, as sketched by the code in Example 3-3.

Example 3-3   Setting Up an Arena With uscasinfo()

```
typedef struct arenaStuff {
   ulock_t   updateLock; /* exclusive use of this structure */
   short     joinedProcs; /* number of processes joined */
   ...pointers to other things allocated by setUpArena()...
} arenaStuff_t;
/*
|| The following function performs the one-time setup of the
|| arenaStuff contents. It assumes that updateLock is held.
*/
extern void
setUpArena(usptr_t *arena, arenaStuff_t *stuff);
/*
|| The following function joins a specified arena, creating it
|| and initializing it if necessary.  It could be extended with
|| values to pass to usconfig(3) before the arena is created.
*/
usptr_t*
joinArena(char *arenaPath)
{
   usptr_t *arena;
   arenaStuff_t *stuff;
   int ret;
   /*
   || Join the arena, creating it if necessary. Exit on error.
   */
   if (!arena = usinit(arenaPath))
   {
      perror("usinit");
      return arena;
   }
   /*
   || Do the following as many times as necessary until the arena
   || has been initialized.
   */
   for(ret=0; !ret; )
```

```
{
    if (stuff = (arenaStuff_t *)usgetinfo(arena))
    {
        /*
        || Another process has created the arena, and either has
        || initialized it or is initializing it right now. Acquire
        || the lock, which will block us until initializing is done.
        */
        ussetlock(stuff->updateLock);
        /* here do anything needing exclusive use of arena */
        ++stuff->joinedProcs; /* another process has joined */
        usunsetlock(stuff->updateLock); /* release arena */
        ret = 1; /* end the loop */
    }
    else
    {
        /*
        || This process appears to be first to call usinit().
        || Allocate an arenaStuff structure with its updateLock
        || already held and 1 process joined, and try to swap it
        || into place as the active one. We expect no errors
        || in setting up arenaStuff. If one occurs, the arena is
        || simply unusable, and we return a NULL to the caller.
        */
        if (! (stuff = usmalloc(sizeof(arenaStuff_t),arena) ) )
            return stuff; /* should never occur */
        if (! (stuff->updateLock = usnewlock(arena) ) );
            return (usptr_t*)0; /* should never occur */
        if (! uscsetlock(stuff->updateLock, 1) )
            return (usptr_t*)0; /* should never occur */
        stuff->joinedProcs = 1;
        if (ret = uscasinfo(arena,0,stuff))
        {
            /*
            || Our arenaStuff is now installed. Initialize it.
            || We hold the lock in arenaStuff as setUpArena expects.
            || The loop ends because ret is now nonzero.
            */
            setUpArena(arena,stuff);
            usunsetlock(stuff->updateLock);
        }
        else
```

```
    {
        /*
        || uscasinfo() either did not find a current value of 0
        || (indicates a race with another process executing this
        || code) or it failed for some other reason. In any case,
        || release allocated stuff and repeat the loop (ret==0).
        */
        usfreelock(stuff->updatelock,arena);
        usfree(stuff,arena);
    }
  } /* usgetinfo returned 0 */
} /* while uscasinfo swap fails */
/* arena->initialized arena, updateLock not held */
return arena;
}
```

Example 3-3 assumes that everything allocated in the arena is accessed through a collection of pointers, *arenaStuff*. The two problems to be solved are these:

- Which asynchronous process is the first to call **usinit()**, and therefore should allocate *arenaStuff* and initialize it with pointers to other objects?

- How can the second and subsequent processes know when the initialization of *arenaStuff* is complete (it might take some time) and the arena is completely ready for use?

The solution in Example 3-3 is based on the discussion in the uscasinfo(3P) reference page. Each process calls function **joinArena()**. If a call to **usgetinfo()** returns nonzero, it is the address of an *arenaStuff_t* that has been allocated by some other process. Possibly that process is concurrently executing, initializing the arena. The current process waits until the lock in the *arenaStuff_t* is released. On return from the **ussetlock()** call, the process has exclusive use of *arenaStuff* until it releases the lock. It uses this exclusive control to increment the count of processes using the arena.

When **usgetinfo()** returns 0, the calling process is probably the first to create the arena, so it allocates an *arenaStuff* structure, and also allocates the essential lock and puts it in a locked state. Then it calls **uscasinfo()** to swap the *arenaStuff* address for the expected value of 0. When the swap succeeds, the process completes initializing the arena and releases the lock.

The call to **uscasinfo()** could fail if, between the time the process receives a 0 from **usgetinfo()** and the time it calls **uscasinfo()**, another process executes this same code and installs its own *arenaStuff*. The process handles this unusual event by releasing the items it allocated and repeating the whole process.

When unrelated processes join an arena with code like that shown in Example 3-3, they should terminate their use of the arena with code similar to Example 3-4.

Example 3-4   Resigning From an Arena

```
/*
|| The following function reverses the operation of joinArena.
|| Even if the calling process is the last one to hold the arena,
|| nothing drastic is done. This is because it is impossible to
|| perform {usinit(); usgetinfo(); ussetlock();} as an atomic
|| sequence.  Once an arena comes into being it must remain
|| usable until the entire application shuts down. Unlinking the
|| arena file can be the last thing that main() does.
*/
void
resignArena(usptr_t *arena)
{
   arenaStuff_t *stuff = (arenaStuff_t *)usgetinfo(arena);
   ussetlock(stuff->updateLock);
   -- stuff->joinedProcs;
   usunsetlock(stuff->updateLock);
}
```

It might seem that, when the function **resignArena()** in Example 3-4 finds that it has reduced the *joinedProcs* count to 0, it could deinitialize the arena, for example unlinking the file on which the arena is based. This is not a good idea because of the remote chance of the following sequence of events:

1.   Process A executes **joinArena()**, initializing the arena.

2.   Unrelated process B executes **joinArena()** through the **usinit()** call, but is suspended for a higher-priority process before executing **usgetinfo()**.

3.   Process A detects some error unrelated to arena use, and as part of termination, calls **resignArena()**.

4.   Process B resumes execution with the call to **usgetinfo()**.

If the **resignArena()** function did something irrevocable, such as unlinking or truncating the arena file, it would leave process B in an unexpected state.

System V Shared Memory Functions

The System V shared memory functions allow two or more processes to share memory. Unlike the IRIX method, in which the external name of a shared arena is also the name of a file, the external name of an SVR4 shared segment is an integer held in an IPC name table (see "SVR4 IPC Name Space" on page 50).

The functions and commands used with SVR4 shared memory are discussed in the following topics and summarized in Table 3-6.

Table 3-6   SVR4 Shared Memory Functions

Function Name	Purpose and Operation
shmget(2)	Create a shared memory IPC object or return the ID of one.
shmctl(2)	Get the status of a shared memory segment, change permissions or user IDs, or lock or unlock a segment in memory.
shmat(2)	Attach a shared memory segment to the address space.
shmdt(2)	Detach a shared memory segment from the address space.

Creating or Finding a Shared Memory Segment

A process creates a shared memory segment, or locates an existing segment, using the **shmget()** system function. When it creates a segment, the arguments to this function establish:

- The numeric key of the segment.

- The size of the segment.

- The user ID and group ID of the segment creator and owner.

- The access permissions to the segment.

When the function locates an existing segment, access to the segment is controlled by the access permissions and by the user ID and group ID of the calling process.

Unlike an IRIX shared arena, a shared segment does not grow automatically as it is used. The size specified when it is created is fixed. The shared segment is initialized to binary zero. (As implemented in IRIX, the pages of the segment are created as they are first referenced; see "Mapping a Segment of Zeros" on page 19.)

The value returned by **shmget()** is the ID number of the segment. It is used to identify the segment to other functions.

Attaching a Shared Segment

The **shmget()** function creates the segment, or verifies that it exists, but does not actually make it a part of the process address space. That remains to be done with a call to **shmat()** ("attach"), passing the identifier returned by **shmget()**.

You can pass a desired base address to **shmat()**, or you can pass NULL to have the system select the base address. It is best to let the system choose the base; this ensures that all processes have the same base address for the segment.

A process can detach a segment from its address space by calling **shmdt()**.

Managing a Shared Segment

The **shmctl()** function gives you the ability to get information about a segment, or to modify its attributes. These operations are summarized in Table 3-7.

Table 3-7   SVR4 Shared Segment Management Operations

Keyword	Operation	Can Be Used By
IPC_STAT	Get information about the segment.	Any process having read access.
IPC_SET	Set owner UID, owner GID, or access permissions.	Creator UID, owner UID, or superuser.
IPC_RMID	Remove the segment from the IPC name space.	Creator UID, owner UID, or superuser.
SHM_LOCK	Lock the segment pages in memory.	Superuser process only.
SHM_UNLOCK	Unlock a locked segment.	Superuser process only.

Information About Shared Memory

The information structure returned by **shmctl(**IPC_STAT**)** is declared in the *sys/shm.h* header file. The first field, *shm_perm*, is an *ipc_perm* structure. This structure is declared in the *sys/ipc.h* header file.

Shared Memory Examples

The example programs in this section illustrate the use of some of the SVR4 shared memory system functions.

Example of Creating a Shared Segment

The program in Example 3-5 illustrates the use of **shmget()**. You can specify command parameters to exercise any combination of **shmget()** function arguments.

Example 3-5   shmget() System Call Example

```
/*
||  Program to test shmget(2) for creating a segment.
||  shmget [-k <key>] [-s <size>] [-p <perms>] [-c] [-x]
||     -k <key>           the key to use, default == 0 == IPC_PRIVATE
||     -s <size>       size of segment, default is 64KB
||     -p <perms>      permissions to use, default is 0600
||     -x                 use IPC_EXCL
||     -c                 use IPC_CREAT
*/
#include <unistd.h> /* for getopt() */
#include <sys/shm.h>    /* for shmget etc */
#include <errno.h>  /* errno and perror */
#include <stdio.h>
int main(int argc, char **argv)
{
    key_t key = IPC_PRIVATE;    /* key */
    size_t size = 65536;            /* size */
    int perms = 0600;               /* permissions */
    int shmflg = 0;             /* flag values */
    struct shmid_ds ds;         /* info struct */
    int c, shmid;
    while ( -1 != (c = getopt(argc,argv,"k:s:p:cx")) )
    {
        switch (c)
```

```
                {
                case 'k': /* key */
                    key = (key_t) strtoul(optarg, NULL, 0);
                    break;
                case 's': /* size */
                    size = (size_t) strtoul(optarg, NULL, 0);
                    break;
                case 'p': /* permissions */
                    perms = (int) strtoul(optarg, NULL, 0);
                    break;
                case 'c':
                    shmflg |= IPC_CREAT;
                    break;
                case 'x':
                    shmflg |= IPC_EXCL;
                    break;
                default: /* unknown or missing argument */
                    return -1;
                }
        }
        shmid = shmget(key,size,shmflg|perms);
        if (-1 != shmid)
        {
            printf("shmid = %d (0x%x)\n",shmid,shmid);
            if (-1 != shmctl(shmid,IPC_STAT,&ds))
            {
                printf("owner uid/gid: %d/%d\n",
                            ds.shm_perm.uid,ds.shm_perm.gid);
                printf("creator uid/gid: %d/%d\n",
                            ds.shm_perm.cuid,ds.shm_perm.cgid);
            }
            else
                perror("shmctl(IPC_STAT)");
        }
        else
            perror("shmget");
        return errno;
}
```

Example of Attaching a Shared Segment

The program in Example 3-6 illustrates the process of actually attaching to and using a shared memory segment. The segment must exist, and is specified by its ID or by its key. You can use the program in Example 3-5 to create a segment for this program to use.

74

The attachment is either read-write or read-only, depending on the presence of the *-r* command parameter. When the program attaches the segment read-write, it stores its own PID in the first word of the segment. Run the program several times; each time it reports the previous PID value and sets a new PID value. This illustrates that the contents of the segment persist between uses of the segment.

You can use the *-w* parameter to have the program wait after attaching. This allows you to start more copies of the program so that multiple processes have attached the segment.

Example 3-6   shmat() System Call Example

```
/*
||  Program to test shmat().
||      shmat {-k <key> | -i <id>} [-a <addr>] [-r] [-w]
||          -k <key>        the key to use to get an ID..
||          -i <id>     ..or the ID to use
||          -a <addr>   address to attach, default=0
||          -r              attach read-only, default read/write
||          -w              wait on keyboard input before detaching
*/
#include <unistd.h> /* for getopt() */
#include <sys/shm.h> /* for shmget etc */
#include <errno.h> /* errno and perror */
#include <stdio.h>
int main(int argc, char **argv)
{
    key_t key = -1; /* key */
    int shmid = -1; /* ..or ID */
    void *addr = 0; /* address to request */
    void *attach;        /* address gotten */
    int rwflag = 0; /* read or r/w */
    int wait = 0;        /* wait before detach */
    int c, ret;
    while ( -1 != (c = getopt(argc,argv,"k:i:a:rw")) )
    {
        switch (c)
        {
        case 'k': /* key */
            key = (key_t) strtoul(optarg, NULL, 0);
            break;
        case 'i': /* id */
            shmid = (int) strtoul(optarg, NULL, 0);
            break;
```

75

```
                case 'a': /* addr */
                    addr = (void *) strtoul(optarg, NULL, 0);
                    break;
                case 'r': /* read/write */
                    rwflag = SHM_RDONLY;
                    break;
                case 'w': /* wait */
                    wait = 1;
                    break;
                default:
                    return -1;
            }
        }
        if (-1 == shmid) /* key must be given */
            shmid = shmget(key,0,0);
        if (-1 != shmid) /* we have an ID */
        {
            attach = shmat(shmid,addr,rwflag);
            if (attach != (void*)-1)
            {
                printf("Attached at 0x%lx, first word = 0x%lx\n",
                                    attach,      *((pid_t*)attach));
                if (rwflag != SHM_RDONLY)
                {
                    *((pid_t *)attach) = getpid();
                    printf("Set first word to 0x%lx\n",*((pid_t*)attach));
                }
                if (wait)
                {
                    char inp[80];
                    printf("Press return to detach...");
                    gets(inp);
                    printf("First word is now 0x%lx\n",*((pid_t*)attach));
                }
                if (shmdt(attach))
                    perror("shmdt()");
            }
            else
                perror("shmat()");
        }
        else
            perror("shmget()");
        return errno;
    }
```

Mutual Exclusion

You use mutual exclusion facilities whenever data is shared by multiple, independent processes or threads. Using such objects as *locks* (also called *mutexes*) and *semaphores*, you can:

- Ensure that only one process or thread uses a particular data structure at any time.

- Synchronize activities, so that processes or threads can wait for the completion of events or actions by other processes or threads.

- Coordinate the use of a shared collection such as a ring buffer or queue.

In order to share data between processes, you share memory between them. Memory sharing is covered in Chapter 3, "Sharing Memory Between Processes." When independent processes share access to data in disk files, they can ensure mutual exclusion using file locks, which are covered in Chapter 7, "File and Record Locking."

This chapter covers the following major topics:

- "Overview of Mutual Exclusion" on page 78 defines such terms as *lock*, *mutex*, *semaphore*, and *barrier*.

- "POSIX Facilities for Mutual Exclusion" on page 82 covers the POSIX functions for semaphores and mutexes.

- "IRIX Facilities for Mutual Exclusion" on page 87 covers IRIX locks, barriers, and semaphores, and the test-and-set facility.

- "Using Compiler Intrinsics for Test-and-Set" on page 96 covers System V semaphores.

Overview of Mutual Exclusion

IRIX offers five kinds of mutual exclusion, each kind with its limits and advantages:

- Test-and-set instructions use special instructions in the MIPS CPU to update a memory location in a predictable way.

- The lock (or mutex) enables processes to enforce serial use of data or code.

- The condition variable lets a thread give up a lock and sleep until an event happens, then reclaim the lock and continue, all in a single operation.

- The semaphore lets independent processes manage a countable resource in an orderly way.

- The barrier lets processes coordinate their initialization.

There is a hierarchy of complexity. Test-and-set instructions are a primitive facility that could be used to implement the others. The lock is a simple object that could be used to implement semaphores and barriers. The semaphore is the most flexible and general facility.

Test-and-Set Instructions

The MIPS instruction architecture includes two instructions designed to let programs update memory from independent processes running concurrently in a multiprocessor.

- The Load Linked (LL) instruction loads a 32- or 64-bit word from memory and also tags that cache line so that the hardware can recognize any change to memory from any CPU in a multiprocessor.

- The Store Conditional (SC) instruction stores a 32- or 64-bit word into memory provided that the destination cache line has not been modified. If the cache line has been altered since the LL instruction was used, SC does not update memory and sets a branch condition.

The combination of LL and SC can be used to guarantee that a change to a memory location is effective, even when multiple concurrent CPUs are trying to update the same location. You can use LL and SC only from an assembly language module. However, the IRIX kernel contains a family of services that are implemented using LL/SC, and you can call them from C or C++. These calls are discussed under "Using Test-and-Set Functions" on page 92.

Locks

A *lock* is a small software object that stands for the exclusive right to use some resource. The resource could be the right to execute a section of code, or the right to modify a variable in memory, or the right to read or write in a file, or any other software operation that must by performed serially, by one process at a time. Before using a serial resource, the program *claims* the lock, and *releases* the lock when it is done with the resource.

The POSIX standard refers to an object of this kind as a *mutex*, a contraction of "mutual exclusion" that is a conventional term in computer science. This book uses the simpler word "lock" when discussing locks in general and IRIX locks in particular, and uses "mutex" when discussing POSIX mutexes.

You can use IRIX locks to coordinate between unrelated processes or lightweight processes through an IRIX shared memory arena. You can use POSIX mutexes to coordinate between POSIX threads in a threaded program only (not IRIX processes).

You define the meaning of a lock in terms that are relevant to your program's design. You decide what resources can be used freely at any time, and you decide what resources must be used serially, by one process at a time. You create and initialize a lock for each serial resource.

It is also your job to ensure that locks are used consistently in all parts of the program. Two errors are easy to make. You can forget to claim a lock, so that some part of the program uses a resource freely instead of serializing. Or you can forget to release a lock, so that other processes trying to claim the lock "hang," or wait forever.

Both of these errors can be hard to find because the symptoms can be intermittent. Most of the time, there is no contention for the use of a shared variable. For example, if one process sometimes fails to claim a lock before updating memory, the program can seem to run correctly for hours (or months) before it suffers precisely the right combination of coincidences that cause two processes to update the variable at the same time.

Semaphores

A *semaphore* is an integer count that is accessed atomically using two operations that are conventionally called P and V:

- The P operation (mnemonic de*p*lete) decrements the count. If the result is not negative, the operation succeeds and returns. If the result is negative, the P operation suspends the calling process until the count has been made nonnegative by another process doing a V operation.

- The V operation (mnemonic re*v*ive) increments the count. If this changes the value from negative to nonnegative, one process that is waiting in a P operation is unblocked.

You can use a semaphore in place of a lock, to enforce serial use of resource. You initialize the semaphore to a value of 1. The P operation claims the semaphore, leaving it at 0 so that the next process to do P will be suspended. The V operation releases the semaphore.

You can also use a semaphore to control access to a pool that contains a countable number for resources. For example, say that a buffer pool contains n buffers. A process can proceed if there is at least 1 buffer available in the pool, but if there are no buffers, the process should sleep until at least 1 buffer is returned.

A semaphore, initialized to n, represents the population of the buffer pool. The pool itself might be implemented as a LIFO queue. The right to update the queue anchor (either to remove a buffer or to return one) is a separate resource that is guarded by a lock. The procedure for obtaining a buffer from the pool is as follows:

1. Perform P on the pool semaphore. When the operation completes, you are assured there is at least one buffer in the pool; and you are also assured that the count representing the buffer you need has been decremented from the semaphore.

2. Claim the lock that guards the buffer queue anchor. This ensures that there will be no conflict with another process taking or returning a buffer at the same time.

3. Remove one buffer from the queue, updating the queue anchor. Step 1 assures that the queue is not empty.

4. Release the lock on the queue anchor.

The procedure for returning a buffer to the pool is as follows:

1. Claim the lock that guards the buffer queue anchor. This ensures that there will be no conflict with another process taking or returning a buffer at the same time.

2. Put the returned buffer back on the queue, updating the queue anchor. The queue could be empty at this time.

3. Release the lock on the queue anchor.

4. Perform V on the pool semaphore. This announces that at least one additional buffer is now free, and may unblock some process waiting for a buffer.

The same two basic procedures work to allocate any collection of objects. For example, the semaphore could represent the number of open slots in a ring buffer, and the lock could stand for the right to update the ring buffer pointers. (A LIFO queue can be managed without a lock; see "Using Compare-and-Swap" on page 93.)

Semaphores created using POSIX functions, and semaphores created by the SVR4 IPC facility, can be used to coordinate IRIX processes or POSIX threads. Semaphores supported by the IRIX IPC facility can be used to coordinate IRIX processes only.

Condition Variables

A condition variable is a software object that represents the occurrence of an event. Typically the event is a software action such as "other thread has supplied needed data."

Condition variable support is included in the POSIX pthreads library, and can be used only to coordinate among POSIX threads, not between IRIX processes. (See Chapter 13, "Thread-Level Parallelism" for information on the pthread library.)

A thread that wants to wait for an event claims the condition variable, which causes the thread to wait. The thread that recognizes the event signals the condition variable, releasing one or all threads that are waiting for the event.

In the expected mode of use, there is a shared resource that can be depleted. Access to the resource is represented by a mutex. A thread claims the mutex, but then finds that the shared resource is depleted or unready. This thread needs to do three things:

1. Give up the mutex so that some other thread can renew the shared resource.

2. Wait for the event that "resource is now ready for use."

3. Re-claim the mutex for the shared resource.

These three actions are combined into one action using a condition variable. When a thread claims a condition variable, it must pass a mutex that it owns. The claim releases the mutex, waits, and reclaims the mutex in one operation.

Barriers

Barriers provide a convenient way of synchronizing parallel processes on multiprocessor systems. To understand barriers, think of a time when you planned to go to lunch with other people at your workplace. The group agrees to meet in the lobby of the building. Some of your coworkers reach the lobby early, and others arrive later. One comes running in last, apologizing. When all of you have gathered and you know that everyone is ready, you all leave the building in a group.

A barrier is the software equivalent of the lobby where you waited. A group of processes are going to work on a problem. None should start until all the data has been initialized. However, starting each process is part of the initialization, and they cannot all be started at the same time. Each process must be created; each must join an arena and perhaps open a file; and you cannot predict when they will all be ready. To coordinate them, you create a barrier. Each process, when it is ready to start the main operation, calls **barrier()**, passing the address of the barrier and the number of processes that will meet. When that many processes have called **barrier()**, all of them are released to begin execution.

Barriers are part of IRIX IPC and require the use of a shared arena. Barriers cannot be used to coordinate POSIX threads.

POSIX Facilities for Mutual Exclusion

The POSIX real-time extensions (detailed in IEEE standard 1003.1b) include named and unnamed semaphores. The POSIX threads library (detailed in IEEE standard 1003.1c) introduces mutexes and condition variables.

Managing Unnamed Semaphores

An unnamed semaphore is a semaphore object that exists in memory only. An unnamed semaphore can be identified only by its memory address, so it can be shared only by processes or threads that share that memory location.

The functions for creating and freeing unnamed semaphores are summarized in Table 4-1.

Table 4-1   POSIX Functions to Manage Unnamed Semaphores

Function Name	Purpose and Operation
sem_init(3)	Initialize a semaphore object, setting its value and preparing it for use.
sem_destroy(3)	Make a semaphore unusable.

The type of a POSIX semaphore is *sem_t*, which is declared in the header file *semaphore.h*. You create an unnamed semaphore by allocating memory for a *sem_t* variable, either dynamically or statically, and initializing it with **sem_init()**. The function in Example 4-1 allocates and initializes an unnamed semaphore and returns its address. It returns NULL if there is a failure of either **malloc()** or **sem_init()**.

Example 4-1   Dynamic Allocation of POSIX Unnamed Semaphore

```
sem_t * allocUnnSem(unsigned initVal)
{
   sem_t *psem = (sem_t*)malloc(sizeof(sem_t));
   if (psem) /* malloc worked */
   {
      if (sem_init(psem,0,initVal))
      {
         free(psem);
         psem = NULL;
      }
   }
   return psem;
}
```

The function in Example 4-1 passes the second argument of **sem_init()**, *pshared*, as 0, meaning the semaphore can only be used within the current process. A semaphore of this kind can be used to coordinate pthreads in a threaded program.

If you want to use a semaphore to coordinate between IRIX processes with separate address spaces, you must create the semaphore with a nonzero *pshared*, and place the semaphore in a memory segment that is shared among all processes. This feature is fully supported. However, you should specify *pshared* as 0 when possible, because nonshared semaphores have higher performance.

Managing Named Semaphores

A named semaphore is named in the filesystem, so it can be opened by any process (subject to access permissions), even when the process does not share address space with the creator of the semaphore. The functions used to create and manage named semaphores are summarized in Table 4-2.

Table 4-2   POSIX Functions to Manage Named Semaphores

Function Name	Purpose and Operation
sem_open(3)	Create or access a named semaphore, returning an address.
sem_close(3)	Give up access to a named semaphore, releasing a file descriptor.
sem_unlink(3)	Permanently remove a named semaphore.

The **sem_open()** function takes the following arguments:

name	Name of the semaphore in the form of a file pathname.
oflag	Either zero, or O_CREAT, or O_CREAT+O_EXCL.
mode	The access permissions to apply if the semaphore is created.
value	Initial value of the semaphore.

Creating a Named Semaphore

The POSIX standard leaves it to the implementation whether or not a named semaphore is represented by a disk file. The IRIX implementation does create a file to stand for each named semaphore (see "POSIX IPC Name Space" on page 48). The file that stands for a semaphore takes up no disk space other than the file node in a directory.

The *oflag* is used to handle the following cases:

- Specify 0 to receive an error if the semaphore does not exist; that is, to require that the semaphore must exist.

- Specify O_CREAT+O_EXCL to receive an error if the semaphore does exist; that is, to require that the semaphore not exist.

- Specify O_CREAT to have the semaphore created if necessary.

When **sem_open()** creates a semaphore, it sets the file permissions specified by *mode*. These permissions control access to a semaphore by UID and GID, just as for a file. (See the open(2) and chmod(2) reference pages.)

When **sem_open()** creates a semaphore, it sets the initial value to *value*, or to 0 if *value* is not specified. Otherwise the value depends on the history of the semaphore since it was created. The value of a semaphore is not preserved over a reboot (the POSIX standard says it is not valid to depend on the value of a semaphore over a reboot).

A named semaphore is opened as a file, and takes up one entry in the file descriptor table for the process. There is no way to convert between the address of the *sem_t* and the file descriptor number, or vice versa. As a result, you cannot directly pass the semaphore to a function such as **fcntl()** or **chmod()**.

Closing and Removing a Named Semaphore

When a process stops using a named semaphore, it can close the semaphore, releasing the associated file descriptor slot. This is done with **sem_close()**. The semaphore name persists in the filesystem, and as long as the system is up, the current semaphore value persists in a table in memory.

To permanently remove a semaphore, use **sem_unlink()**.

Using Semaphores

POSIX named and unnamed semaphores can be used to coordinate the actions of IRIX processes and POSIX threads. They are the only mutual-exclusion objects that can be freely used to coordinate between threaded and unthreaded programs alike. (Message queues can be used between threaded and unthreaded programs also; see Chapter 6, "Message Queues.")

The functions that operate on semaphores are summarized in Table 4-3.

Table 4-3   POSIX Functions to Operate on Semaphores

Function Name	Purpose and Operation
sem_getvalue(3)	Return a snapshot of the current value of a semaphore.
sem_post(3)	Perform the P operation, incrementing a semaphore and possibly unblocking a waiting process.
sem_trywait(3)	Perform the V operation only if the value of the semaphore is 1 or more.
sem_wait(3)	Perform the V operation, decrementing a semaphore and blocking if it becomes negative.

The abstract operation P is implemented as the **sem_wait()** function. Use this to decrement a semaphore's value and, if the result is negative, to suspend the calling function until the value is restored. The V operation is **sem_post()**.

You can sample a semaphore's value using **sem_getvalue()**. The **sem_trywait()** operation is useful when a process or thread cannot tolerate being suspended.

Using Mutexes and Condition Variables

Two additional types of mutual exclusion are available only within a threaded program, to coordinate the actions of POSIX threads. The mutex is comparable to a lock or to a semaphore initialized to a count of 1. The condition variable provides a convenient way for a thread to give up ownership of a mutex, wait for something to happen, and then reclaim the mutex.

Both of these facilities are covered in detail in Chapter 13, "Thread-Level Parallelism." See the headings "Mutexes" on page 283 and "Condition Variables" on page 286.

IRIX Facilities for Mutual Exclusion

IRIX supports a wide selection of mutual-exclusion facilities, all tuned for use between processes that run concurrently in a multiprocessor.

Using IRIX Semaphores

Two kinds of semaphores are supported in IRIX IPC: normal and polled. Both are allocated in a shared memory arena (see "IRIX Shared Memory Arenas" on page 61).

Creating Normal Semaphores

The functions for managing normal semaphores are summarized in Table 4-4.

Table 4-4   IRIX Functions to Manage Nonpolled Semaphores

Function Name	Purpose and Operation
usnewsema(3P)	Allocate a semaphore in an arena and give it an initial value.
usfreesema(3P)	Release arena memory used by a semaphore (does not release any process waiting on the semaphore).
usinitsema(3P)	Reset a semaphore value and its metering information (does not release any process waiting on the semaphore).
usctlsema(3P)	Set and reset semaphore metering information and other attributes.
usdumpsema(3P)	Dump semaphore metering information to a file.

To allocate a new shared-arena semaphore and set its initial value, call **usnewsema()**. Use **usctlsema()** to enable recursive use of the semaphore and to enable the collection of metering information. You can use the metering information to find out whether a semaphore is a bottleneck or not.

Tip: When reading the reference pages cited above, notice that **usnewsema()** returns the address of a *usema_t* object, and all the other functions take the address of a *usema_t*. That is, *usema_t* represents the type of the semaphore object itself, and you refer to a semaphore by its address. This is different from locks, which are discussed later in this chapter.

Creating Polled Semaphores

A polled semaphore differs from a normal semaphore in the P operation. When decrementing the semaphore value produces a negative number, the calling process is not blocked. Instead, it receives a return code. The process then has to include the address of the semaphore in the list of events passed to **poll()** (see the poll(2) reference page). The V operation, applied to a polled semaphore, does not release a block process but rather causes a **poll()** operation to end.

You can use polled semaphores to integrate semaphore handling with other events for which you wait with **poll()**, such as file operations. You cannot combine the use of normal semaphores with the use of polled devices, since a single process cannot wait in a **poll()** call and in a **uspsema()** call at the same time. The functions for creating and controlling polled semaphores are summarized in Table 4-5.

Table 4-5   IRIX IPC Functions for Managing Polled Semaphores

Function Name	Purpose and Operation
usnewpollsema(3P)	Allocate a polled semaphore in an arena and give it an initial value.
usopenpollsema(3P)	Assign a file descriptor to a polled semaphore. The file descriptor can be passed to **poll()** or **select()**. This must be done before the semaphore can be used.
usclosepollsema(3P)	Release a file descriptor assigned with **usopenpollsema()**.
usfreepollsema(3P)	Release arena memory used by a polled semaphore and invalidate any file descriptors assigned to it.

Operating on Semaphores

The functions for semaphore operations are summarized in Table 4-6.

Table 4-6   IRIX IPC Functions for Semaphore Operations

Function Name	Purpose and Operation
uspsema(3P)	Perform the P operation on either type of semaphore.
usvsema(3P)	Perform the V operation on either type of semaphore.
ustestsema(3P)	Return the current (instantaneous) value of a semaphore.
uscpsema(3P)	Perform the P operation only if the resulting count will be nonnegative.
usinitsema(3P)	Reset a semaphore value and its metering information (does not release any process waiting on the semaphore).
usctlsema(3P)	Set and reset semaphore metering information and other attributes.
usdumpsema(3P)	Dump semaphore metering information to a file.

To perform the P operation on a semaphore of either type, use **uspsema()**. When the decremented semaphore value is nonnegative, the function returns 1. The action when the decremented count would be negative differs between the polled and normal semaphores:

- When a normal semaphore count remains or becomes negative, the calling process is blocked; the function does not return until the count is nonnegative.

- When a polled semaphore count remains or becomes negative, the function returns 0 and the calling process must use **poll()** to find out when it becomes nonnegative.

To perform the V operation on a semaphore of either type, call **usvsema()**.

The **uscpsema()** function provides a conditional P operation: it performs a P operation on the semaphore only if it can do so without making the value negative. The **ustestsema()** function returns the current value of the semaphore—which of course is immediately out of date.

The **usinitsema()** function reinitializes the semaphore to a specified value. Note that if you reinitialize a semaphore on which processes are waiting, the processes continues to wait. You should reinitialize a semaphore only in unusual circumstances.

You can call **usctlsema()** to enable the keeping of either metering information—cumulative counts of usage—or a history trace. The metering information shows whether a semaphore is a bottleneck in the program's operations. The history trace can be used to analyze bugs.

Using Locks

IRIX locks are implemented differently depending on the hardware architecture of the computer using them. On a multiprocessor computer, locks are busy-wait locks, so the processor continually tries to acquire the lock until it succeeds. This implementation makes sense only on multiprocessor systems, where one processor can release the lock while another processor is "spinning," trying to acquire the lock. On a uniprocessor, a process waiting to claim a lock is suspended until the lock is released by another process.

Creating and Managing Locks

The functions for creating and controlling locks are summarized in Table 4-7.

Table 4-7   IRIX IPC Functions for Managing Locks

Function Name	Purpose and Operation
usnewlock(3P)	Allocate a lock in a specified arena.
usfreelock(3P)	Release lock memory (does not release any process waiting on the lock).
usinitlock(3P)	Reset a lock and its metering information (does not release any process waiting on the lock).
usctllock(3P)	Fetch and reset semaphore metering information or debugging information.
usdumplock(3P)	Dump lock metering information to a file.

You decide whether the locks in an arena will have metering information or not. You specify this before creating the arena, to **usconfig()** (see "Initializing Arena Attributes" on page 61). When lock metering is enabled, you can retrieve the information about a lock at any time to find out whether a lock is a bottleneck in a program.

Claiming and Releasing Locks

The functions for using locks are summarized in Table 4-8.

Table 4-8   IRIX IPC Functions for Using Locks

Function Name	Purpose and Operation
ussetlock(3P)	Seize a lock, suspending the caller if necessary, until the lock is available.
usunsetlock(3P)	Release a lock, making it available for other processes.
uscsetlock(3P)	Seize a lock if it is available; otherwise return a 1.
uswsetlock(3P)	Seize a lock, suspending the caller if necessary; takes a specified number of spins as an argument.
ustestlock(3P)	Test a lock, returning 0 if it is instantaneously available and 1 if it is not available.

Tip: When reading the reference pages cited above, notice that **usnewlock()** returns a *ulock_t* object, which is simply a pointer. All the functions that operate on locks take a *ulock_t* object—not a pointer to a *ulock_t*. That is, the *ulock_t* type represents a handle or reference to a lock, not a lock itself. This differs from the treatment of semaphores, which is described under "Creating Normal Semaphores" on page 87.

On uniprocessors, none of the functions **us[c,w]setlock()** spin; if the lock is available they return immediately, and if it is not, they suspend the calling process and give up the CPU. On multiprocessors, **ussettlock()** spins for a default number of times before it suspends the process. The function **uswsetlock()** is the same, but you can specify the number of spins to take before suspending.

A process can call **usunsetlock()** on a lock that is either not locked or locked by another process. In either case, the lock is unlocked. "Double tripping"—calling a set-lock function twice with the same lock—is also permissible. The caller blocks until another process unsets the lock.

Using Barriers

The functions to manage and use barriers are summarized in Table 4-9.

Table 4-9   IRIX IPC Functions for Barriers

Function Name	Purpose and Operation
new_barrier(3P)	Allocate and initialize a barrier in a specified arena.
free_barrier(3P)	Release the storage associated with a barrier.
barrier(3P)	Wait at a barrier until a specified number of processes have gathered.
init_barrier(3P)	Reinitialize a barrier (does not release any processes waiting).

The main process uses **new_barrier()** to allocate a barrier in some arena. To use the barrier, each process calls **barrier()**, passing the number of processes that are supposed to meet before proceeding.

Note: The **barrier()** function assumes that it is used on a multiprocessor. It always passes time by spinning in an empty loop. When used on a uniprocessor (or when used on a multiprocessor with fewer available CPUs than barrier processes), a call to **barrier(**n**)** can be quite inefficient. The waiting functions spin until each in turn uses up its time-slice. In general it is not a good idea to use **barrier()** except in a multiprocessor with a number of CPUs approximately equal to the number of coordinating processes.

Using Test-and-Set Functions

The C library includes a family of functions that apply the MIPS instructions Load Linked and Store Conditional to modify memory words in a reliable way in a multiprocessor. These functions are detailed in the test_and_set(3) and uscas(3) reference pages. In addition, the MIPSpro C and C++ compilers, version 7.0 and after, contain built-in support for these operations.

Using Test-and-Set

All test-and-set functions solve a similar problem: how to update the contents of a memory word reliably from two or more CPUs concurrently. Use a test-and-set function to avoid the traditional "race" condition. For example, suppose that two or more processes could execute code to increment a variable, as in the C expression *++shared*:

- Process A loads *shared* into a register and adds 1 to it.

- Process B loads *shared* into a register and adds 1 to it.

- Process A stores the value in memory.

- Process B stores the value in memory.

The result is to increment *shared* by 1 when it should be incremented by 2. However, if both processes use *test_then_add(&shared,1)* instead, they are assured that both increments will occur regardless of timing.

Using Compare-and-Swap

The test-and-set functions are not adequate to do race-free pointer manipulation; you need a compare-and-swap function for that. The C library includes the **uscas()** and **uscas32()** functions for this purpose. Use **uscas()** to work with pointer-sized values (which can be either 32 or 64 bits depending on compile options). Use **uscas32()** to work with words that should always be 32 bits in every program.

The compare-and-swap functions take four arguments:

destp   Address of the target memory field you want to update.

old   Expected current value of the memory field.

new   Desired new value, based on the expected old value.

u   Address of any IRIX shared memory arena.

The arena address *u* is not actually used by the functions. However, the functions cannot work until **usinit()** has been called at least once. Passing an arena address ensures that this has happened.

Use a compare-and-swap function in a loop like the following:

1. Copy the current value of the target memory field.

2. Calculate a new value based on that current value.

3. Use compare-and-swap to install the new value, provided that the current value has not changed during step 2.

4. If the compare failed so the swap was not done (**uscas()** returns 0), another process has changed the target: return to step 1 and repeat.

The code in Example 4-2 illustrates how this type of loop can be used to manage a simple LIFO queue.

Example 4-2   Using Compare-and-Swap on a LIFO Queue

```
#include <ulocks.h>
typedef struct item_s {
    struct item_s *next;
    /* ... other fields ... */
} item_t;
void push_item( item_t **lifo, item_t *new, usptr_t *u)
{
    item_t *old;
    do {
        new->next = old = *lifo;
    } while(0 == uscas(lifo,(ptrdiff_t)old,(ptrdiff_t)new,u));
}
item_t * pull_item( item_t **lifo, usptr_t *u)
{
    item_t *old, *new;
    do {
        old = *lifo;
        if (!old) break;
        new = old->next;
    } while(0 == uscas(lifo,(ptrdiff_t)old,(ptrdiff_t)new,u));
    return old;
}
#include <stdio.h>
main()
{
    usptr_t *arena = usinit("/var/tmp/cas.arena");
    item_t *lifo = NULL;
    item_t t1, t2;
    item_t *p1, *p2;
```

```
        push_item(&lifo, &t1, arena);
        push_item(&lifo, &t2, arena);
        p2 = pull_item(&lifo, arena);
        p1 = pull_item(&lifo, arena);
        printf("%x == %x ?\n", &t1, p1);
        printf("%x == %x ?\n", &t2, p2);
}
```

In Example 4-2, the **push_item()** function pushes an *item_t* onto a LIFO queue, and **pull_item()** removes and returns the first *item_t* from a queue. Both use **uscas()** to update the queue anchor. The **main()** function contains a unit-test of the functions, first pushing two items, then pulling them off, finally displaying the addresses to verify that what was pushed, could be pulled.

Using Compiler Intrinsics for Test-and-Set

The MIPSpro C and C++ compilers version 7.0 introduce the intrinsic functions summarized in Table 4-10.

Table 4-10   Compiler Intrinsics for Atomic Operations

Intrinsic Prototype	Purpose	Barrier
_ _*op*_and_fetch(p,v...)	Atomically execute {**p* *op*= **v**; **p*;}. The op can be **add**, **sub**, **or**, **and**, **xor**, and **nand**.	Full
_ _fetch_and_*op*(p,v...)	Atomically execute {**t** = **p*; **p* *op*= **v**; **t**;}. The op can be **add**, **sub**, **or**, **and**, **xor**, and **nand**.	Full
_ _lock_test_and_set(p,v...)	Atomically execute {**t** = **p*; **p* = **v**; **t**;}.	Backward
_ _lock_release(p...)	Atomically execute {**p* = **0**;}.	Forward
_ _compare_and_swap(p,w,v...)	Atomically execute **(w==*p) ?(*p=v, 1): 0**.	Full
_ _synchronize(...)	Issue the MIPS-3 instruction sync to synchronize the cache with memory.	Full

Each of the compiler intrinsics except **__synchronize()** causes the compiler to generate inline code using Load Linked and Store Conditional to update memory predictably. In this respect they are similar to the library functions documented in the test_and_set(3) and uscas(3) reference pages. For example, the statement

```
__add_and_fetch(&shared,1);
```

is functionally equivalent to the library call

```
test_then_add(&shared,1);
```

The compiler intrinsic **__compare_and_swap()** is simpler to use than **uscas()** since you do not have to create a shared memory arena first, and avoids the overhead of a system call.

The compiler intrinsics are different from the library functions, and different from an assembly language subroutine you might write, in one important way. The optimizer phases of the compiler recognize these intrinsics as barriers to code motion. The "Barrier" column in Table 4-10 shows this effect. For example, the compiler cannot move code in either direction across s use of **__compare_and_swap()**. However, it can move code backward (but not forward) across **__lock_test_and_set().**

You can make the code motion barrier explicit or general. If you invoke **__compare_and_swap()** passing only the pointer and two value arguments, the compiler can move no code across that source line. Alternatively, you can list specific variables as additional arguments to **__compare_and_swap()** (this is why the functions are shown as having a variable number of arguments). When you do so, the compiler cannot move assignments to the named variables across this point, but can move assignments to other variables, if the optimizer needs to.

System V Facilities for Mutual Exclusion

The System V Release 4 (SVR4) semaphore facility lets you create persistent semaphores that can be used to coordinate any processes or threads. The SVR4 facility differs from POSIX named semaphores in two ways:

- Each object is a set of from 1 to 25 independent semaphores, rather than a single semaphore. A process can operate on any selection of semaphores in a set in one system call.

- You can use SVR4 semaphores in ways that IRIX and POSIX do not support: incrementing or decrementing by more than 1, and waiting for a zero value.

- The name of a set is an integer in a kernel table, rather than a pathname in the filesystem (see "SVR4 IPC Name Space" on page 50).

The functions used to create and operate on semaphore sets are summarized in Table 4-11.

Table 4-11   SVR4 Semaphore Management Functions

Function Name	Purpose and Operation
semget(2)	Create a semaphore set, or return the ID of a semaphore set.
semctl(2)	Query or change semaphore values; query or change semaphore set attributes.
semop(2)	Perform operations on one or more semaphores in a set.

Semaphores are also discussed in the intro(2) reference page. You can display semaphore sets from the command line using *ipcs*, and remove them with *ipcrm* (see the ipcs(1) and ipcr(1) reference pages).

Creating or Finding a Semaphore Set

A process creates a semaphore set, or locates an existing set, using the **semget()** system function. The function creates a set only if the specified key is IPC_PRIVATE, or no set with that key exists, and the IPC_CREAT flag is used. When it creates a set, the arguments to the function establish

- the numeric key of the set

- the number of semaphores in the set, from 1 to 25

- the access permissions to the set

In addition, the effective user ID and group ID of the calling process become the creator and owner identification of the new semaphore set. (See "Example Uses of semget()" on page 102 for example code.)

When **semget()** locates an existing set, access is controlled by the access permissions of the set and by the user ID and group ID of the calling process.

The value returned by **semget()** is the ID number of the semaphore set. It is used to identify the segment to other functions.

Managing Semaphore Sets

The **semctl()** function gives you the ability to get information about a semaphore set, or to modify its attributes. These operations are summarized in Table 4-12.

Table 4-12   SVR4 Semaphore Set Management Operations

Keyword	Operation	Can Be Used By
IPC_STAT	Get information about the set.	Any process having read access.
IPC_SET	Set owner UID, owner GID, or access permissions.	Creator UID, owner UID, or superuser.
IPC_RMID	Remove the set from the IPC name space.	Creator UID, owner UID, or superuser.
GETALL	Copy current values of all semaphores to an array.	Any process having read access.
SETALL	Set current values of all semaphores from an array of integers.	Any process having write access.

Examples of some of these uses can be found under "Example Uses of semctl() for Management" on page 104.

In addition, **semctl()** allows you to query or set information about individual semaphores within the set, as summarized in Table 4-13.

Table 4-13   SVR4 Semaphore Management Operations

Keyword	Operation	Can Be Used By
GETVAL	Return value of one semaphore.	Any process having read access.
GETPID	Return process ID of the process that last operated on a semaphore.	Any process having read access.
GETNCNT	Return number of processes waiting for one semaphore to exceed zero	Any process having read access.
GETZCNT	Return number of processes waiting for one semaphore to equal zero.	Any process having read access.
SETVAL	Set current value of one semaphores.	Any process having write access.

Examples of some of these uses can be seen under "Example Uses of semctl() for Query" on page 106.

Caution: Some operations of the **semctl()** function use only three arguments, but some operations require a fourth argument (see reference page semctl(2) for details). When passing a fourth argument to **semctl()**, it is *extremely* important that you pass a *union semun*, as specified in the reference page. You might look at the contents of the union and think that, since all its fields are addresses, there is no effective difference between passing a union and passing a plain address of a buffer or array. However, if your program is compiled with the -n32 or -64 options, the alignment of the two kinds of arguments is different. Always pass an address as shown in the example programs in this chapter:

```
union semun arg4;
...
arg4.buffer = &ds_buffer;
semctl(a,b,c,arg4);
```

If your program passes only the address, as in

```
semctl(a,b,c,&ds_buffer);
```

the code will not work correctly when compiled -n32 or -64.

Using Semaphore Sets

You perform operations on the semaphores in a set by calling **semop()**. This function takes a semaphore set ID, and an array of one or more semaphore operation structures. Each of the operation structures specifies the following:

- The index of a semaphore in the set, numbering the semaphores from 0

- A number specifying one of three operations:

  - Zero, meaning to test the semaphore for equality to 0.

  - A positive number such as 1, meaning to increment the semaphore value, possibly releasing waiting processes or threads (the V operation).

  - A negative number such as -1, meaning to decrement the semaphore value when that can be done without making it negative (the P operation).

- A flag word that can specify these flags:

  - IPC_NOWAIT, do not suspend but return an error if the Zero test fails or the P operation cannot be done.

  - SEM_UNDO, undo this operation if it succeeds but an operation later in the array should fail.

In the simplest case, you pass an array containing just one operation, to increment or decrement one semaphore by 1 (the traditional V or P operation). Used this way, a semaphore in a set is functionally the same as an IRIX or POSIX semaphore.

SVR4 semaphores permit additional operations not available with IRIX or POSIX semaphores. The negative or positive value in the operation structure is not required to be 1, so you can increment or decrement a semaphore by more than 1 in an operation. The wait-for-zero operation allows one process or thread to monitor the state of a semaphore, independent of the P and V operations performed on the semaphore by other processes or threads.

You can also perform a sequence of operations—a sequence of P, or V, or zero-wait operations, or a mix of operation types—on multiple semaphores in a single call. To do this, you specify an array containing more than one operation structure. The **semop()** function performs each operation in sequence.

You can use this feature, for example, to claim multiple resources, each represented by a different semaphore. Your array would specify the P operation on each of the semaphores in sequence. When **semop()** returns successfully, you own all the resources. A similar, multiple V operation returns all the resources at once.

The IPC_NOWAIT and SEM_UNDO flags are important when claiming multiple resources at once. Specify SEM_UNDO on all operations; and specify IPC_NOWAIT on all but the first one. If the second or later resource is unavailable, **semop()** restores all preceding claims and returns an error code. As long as all processes or threads operate on semaphores in the same order, this logic prevents deadlocks, and it avoids long, fruitless suspensions.

Example Programs

The programs in this section allow you to experiment with semaphore sets from the command line:

- Example 4-3 on page 102 can be used to experiment with **semget()**, creating semaphore sets with different sizes and permissions.

- Example 4-4 on page 104 can be used to test **semctl()** for displaying and changing owner IDs and permissions.

- Example 4-5 on page 106 can be used to test **semctl()** for sampling the values of semaphores, or to display the state of a semaphore set.

- Example 4-6 on page 108 can be used to test **semop()** for single or multiple operations.

Example Uses of semget()

The program in Example 4-3, *semget*, invokes **semget()** with arguments you specify on the command line:

-k *key*	Numeric key to identify the semaphore set, required; for example *-k 99*. Default is IPC_PRIVATE.
-p *perms*	Access permissions to apply to a created set; for example, *-p 0664*. Default is octal 0600.
-s *setsize*	Number of semaphores in a created set; for example *-s 8*. The limit is 25, but feel free to experiment with larger numbers to see the return code.
-c	Use IPC_CREAT. No set is created unless this is specified.
-x	Use IPC_EXCL. Use with *-c* to require that a set not exist.

Example 4-3   Program to Demonstrate semget()

```
/*
|| semget: program to test semget(2) for creating semaphores.
||     semget [-k <key>] [-c] [-x] [-p <perms>] [-s <setsize>]
||          -k <key>        the key to use, default == 0 == IPC_PRIVATE
||          -p <perms>      permissions to use, default is 0666
||          -s <setsize>    size to use, default is 1
||          -c              use IPC_CREAT
||          -x              use IPC_EXCL
*/
#include <unistd.h> /* for getopt() */
#include <sys/sem.h>    /* for shmget etc */
#include <errno.h>  /* errno and perror */
#include <stdio.h>
int main(int argc, char **argv)
{
    key_t key = IPC_PRIVATE;/* key */
    int nsems = 1;          /* setsize */
    int perms = 0600;       /* permissions */
    int semflg = 0;         /* flag values */
    struct semid_ds ds;     /* info struct */
    union semun arg4;       /* way to pass &ds properly aligned */
    int c, semid;
    while ( -1 != (c = getopt(argc,argv,"k:p:s:xc")) )
    {
        switch (c)
        {
```

```
    case 'k': /* key */
        key = (key_t) strtoul(optarg, NULL, 0);
        break;
    case 's': /* setsize */
        nsems = (int) strtoul(optarg, NULL, 0);
        break;
    case 'p': /* permissions */
        perms = (int) strtoul(optarg, NULL, 0);
        break;
    case 'c':
        semflg |= IPC_CREAT;
        break;
    case 'x':
        semflg |= IPC_EXCL;
        break;
    default: /* unknown or missing argument */
        return -1;
    }
}
semid = semget(key,nsems,semflg+perms);
if (-1 != semid)
{
    printf("semid = %d\n",semid);
    arg4.buf = &ds;
    if (-1 != semctl(semid,0,IPC_STAT,arg4))
    {
        printf(
"owner uid.gid: %d.%d  creator uid.gid: %d.%d  mode: 0%o nsems:%d\n",
        ds.sem_perm.uid,ds.sem_perm.gid,
        ds.sem_perm.cuid,ds.sem_perm.cgid,
        ds.sem_perm.mode, ds.sem_nsems);
    }
    else
        perror("semctl(IPC_STAT)");
}
else
    perror("semget()");
return errno;
}
```

Example Uses of semctl() for Management

The program in Example 4-4, *semmod*, allows you to call **semctl()** from the command line to display the size, permissions, and owner and creator IDs of a semaphore set, and to change the permissions and owner. It takes the following arguments on the command line:

-k *key*   Numeric key to identify the semaphore set; for example *-k 99*.

-i *id*   Semaphore ID number, alternative to specifying the key.

-p *perms*   Access permissions to apply to the selected set; for example, *-p 0664*.

-u *uid*   New user ID for the semaphore owner.

-g *gid*   New group ID for the semaphore owner.

If only the key or ID is given, the program only displays the state of the set. When you specify permissions, owner, or group, the program first queries the current information to initialize an information structure. Then it inserts the new items you specified, and calls **semctl()** with IPC_SET to change the information.

Example 4-4   Program to Demonstrate semctl() for Management

```
/*
|| semmod: program to test semctl(2) for status, ownership and permissions.
||      semmod {-k <key> | -i <semid>} [-p <perms>] [-u <user>] [-g <group>]
||          -k <key>              the key to use, or..
||          -i <semid>            ..the semid to use
||          -p <perms>            permissions to set with IPC_SET
||          -u <uid>              uid to set as owner with IPC_SET
||          -g <gid>              gid to set as owner with IPC_SET
*/
#include <unistd.h> /* for getopt() */
#include <sys/sem.h> /* for shmget etc */
#include <errno.h> /* errno and perror */
#include <stdio.h>
int main(int argc, char **argv)
{
    key_t key;                /* key */
    int semid = -1;           /* object ID */
    int perms, popt = 0;      /* perms to set, if given */
    int uid, uopt = 0;        /* uid to set, if given */
    int gid, gopt = 0;        /* gid to set, if given */
    int val, vopt = 0;        /* setall value if given */
    struct semid_ds ds;
```

104

```
union semun arg4;        /* way to pass semctl 4th arg, properly aligned */
int c;
while ( -1 != (c = getopt(argc,argv,"k:i:p:u:g:")) )
{
    switch (c)
    {
    case 'k': /* key */
        key = (key_t) strtoul(optarg, NULL, 0);
        break;
    case 'i': /* semid */
        semid = (int) strtoul(optarg, NULL, 0);
        break;
    case 'p': /* permissions */
        perms = (int) strtoul(optarg, NULL, 0);
        popt = 1;
        break;
    case 'u': /* uid */
        uid = (int) strtoul(optarg, NULL, 0);
        uopt = 1;
        break;
    case 'g': /* gid */
        gid = (int) strtoul(optarg, NULL, 0);
        gopt = 1;
        break;
    default: /* unknown or missing argument */
        return -1;
    }
}
if (-1 == semid) /* -i not given, must have -k */
    semid = semget(key,0,0);
if (-1 != semid)
{
    arg4.buf = &ds;
    if (0 == semctl(semid,0,IPC_STAT,arg4))
    {
        if ((popt)||(uopt)||(gopt))
        {
            if (popt) ds.sem_perm.mode = perms;
            if (uopt) ds.sem_perm.uid = uid;
            if (gopt) ds.sem_perm.gid = gid;
            if (0 == semctl(semid,0,IPC_SET,arg4) )
                semctl(semid,0,IPC_STAT,arg4); /* refresh info */
            else
                perror("semctl(IPC_SET)");
        }
```

```
        printf(
"owner uid.gid: %d.%d   creator uid.gid: %d.%d   mode: 0%o nsems:%d\n",
        ds.sem_perm.uid,ds.sem_perm.gid,
        ds.sem_perm.cuid,ds.sem_perm.cgid,
        ds.sem_perm.mode, ds.sem_nsems);
    }
    else
        perror("semctl(IPC_STAT)");
}
else
    perror("semget()");
}
```

Example Uses of semctl() for Query

The program in Example 4-5, *semsnap*, displays a snapshot of the current values of all semaphores in a set you specify. The value of each semaphore is displayed in the first row (GETVAL), followed by the count of processes waiting in a P operation (GETNCNT) and the count of processes waiting for zero (GETZCNT). The arguments are as follows:

-k *key*   Numeric key to identify the semaphore set; for example *-k 99*.

-i *id*   Semaphore ID number, alternative to specifying the key.

Example 4-5   Program to Demonstrate semctl() for Sampling

```
/*
|| semsnap: program to test semctl(2) for semaphore status commands
||      semsnap {-k <key> | -i <semid>}
||          -k <key>              the key to use, or..
||          -i <semid>        ..the semid to use
*/
#include <unistd.h> /* for getopt() */
#include <sys/sem.h>    /* for shmget etc */
#include <errno.h>  /* errno and perror */
#include <stdio.h>
int main(int argc, char **argv)
{
    key_t key;                    /* key */
    int semid = -1;        /* object ID */
    int nsems, j;                 /* setsize, and loop variable */
    ushort_t semvals[25];  /* snapshot of values */
    ushort_t semns[25];     /* snapshot of P-waiting */
    ushort_t semzs[25];     /* snapshot of zero-waiting */
    struct semid_ds ds;
```

```
    union semun arg4;        /* semctl 4th argument, properly aligned */
    int c;
    while ( -1 != (c = getopt(argc,argv,"k:i:")) )
    {
        switch (c)
        {
        case 'k': /* key */
            key = (key_t) strtoul(optarg, NULL, 0);
            break;
        case 'i': /* semid */
            semid = (int) strtoul(optarg, NULL, 0);
            break;
        default: /* unknown or missing argument */
            return -1;
        }
    }
    if (-1 == semid) /* -i not given, must have -k */
        semid = semget(key,0,0);
    if (-1 != semid)
    {
        arg4.buf = &ds;
        if (0 == semctl(semid,0,IPC_STAT,arg4))
        {
            nsems = ds.sem_nsems;
            arg4.array = semvals;
            semctl(semid,0,GETALL,arg4);
            for (j=0; j<nsems; ++j)
            {
                semns[j] = semctl(semid,j,GETNCNT);
                semzs[j] = semctl(semid,j,GETZCNT);
            }
            printf("vals:");
            for (j=0; j<nsems; ++j) printf(" %2d",semvals[j]);
            printf("\nncnt:");
            for (j=0; j<nsems; ++j) printf(" %2d",semns[j]);
            printf("\nzcnt:");
            for (j=0; j<nsems; ++j) printf(" %2d",semzs[j]);
            putc('\n',stdout);
        }
        else
            perror("semctl(IPC_STAT)");
    }
    else
        perror("semget()");
}
```

Example Uses of semop()

The program in Example 4-6, *semop*, performs one or more semaphore operations on a set you specify. You can use it to specify any sequence of operations (including nonsensical sequences) from the command line. The command arguments are:

-k *key*   Numeric key to identify the semaphore set; for example *-k 99*.

-i *id*   Semaphore ID number, alternative to specifying the key.

-n   Apply IPC_NOWAIT to all following operations.

-u   Apply SEM_UNDO to all following operations.

-p *sem*   Apply the P (decrement by 1) operation to *sem*; for example, *-p 1*.

-v *sem*   Apply the V (increment by 1) operation to *sem*; for example, *-v 1*.

-z *sem*   Wait for *sem* to contain 0; for example, *-z 4*.

You can give a sequence of operations. For example, consider the following sequence:

1.   Wait for zero in semaphore 4.

2.   Increment semaphore 0, with undo if a following operation fails.

3.   Decrement semaphore 2, not waiting and with undo.

4.   Decrement semaphore 3, not waiting and with undo.

The sequence above can be specified as follows:

```
semop -k 0x101 -z 4 -u -v 0 -n -p 2 -p 3
```

The program does not support incrementing or decrementing by other than 1, and there is no way to turn off IPC_NOWAIT or SEM_UNDO once it is on.

Example 4-6   Program to Demonstrate semop()

```
/*
|| semop: program to test semop(2) for all functions.
||      semop {-k <key> | -i <semid>} [-n] [-u] {-p <n> | -v <n> | -z <n>}...
||          -k <key>         the key to use, or..
||          -i <semid>       ..the semid to use
||          -n               use the IPC_NOWAIT flag on following ops
||          -u               use the SEM_UNDO flag on following ops
||          -p <n>           do the P operation (+1) on semaphore <n>
||          -v <n>           do the V operation (-1) on semaphore <n>
||          -z <n>           wait for <n> to become zero
```

```
*/
#include <unistd.h>      /* for getopt() */
#include <sys/sem.h>     /* for shmget etc */
#include <errno.h>       /* errno and perror */
#include <stdio.h>
int main(int argc, char **argv)
{
    key_t key;                  /* key */
    int semid = -1;             /* object ID */
    int nsops = 0;              /* setsize, and loop variable */
    short flg = 0;              /* flag to use on all ops */
    struct semid_ds ds;
    int c, s;
    struct sembuf sops[25];
    while ( -1 != (c = getopt(argc,argv,"k:i:p:v:z:nu")) )
    {
        switch (c)
        {
        case 'k': /* key */
            key = (key_t) strtoul(optarg, NULL, 0);
            break;
        case 'i': /* semid */
            semid = (int) strtoul(optarg, NULL, 0);
            break;
        case 'n': /* use nowait */
            flg |= IPC_NOWAIT;
            break;
        case 'u': /* use undo */
            flg |= SEM_UNDO;
            break;
        case 'p': /* do the P() */
            sops[nsops].sem_num = (ushort_t) strtoul(optarg, NULL, 0);
            sops[nsops].sem_op = -1;
            sops[nsops++].sem_flg = flg;
            break;
        case 'v': /* do the V() */
            sops[nsops].sem_num = (ushort_t) strtoul(optarg, NULL, 0);
            sops[nsops].sem_op = +1;
            sops[nsops++].sem_flg = flg;
            break;
        case 'z': /* do the wait-for-zero */
            sops[nsops].sem_num = (ushort_t) strtoul(optarg, NULL, 0);
            sops[nsops].sem_op = 0;
            sops[nsops++].sem_flg = flg;
            break;
```

```
        default: /* unknown or missing argument */
            return -1;
        }
    }
    if (-1 == semid) /* -i not given, must have -k */
        semid = semget(key,0,0);
    if (-1 != semid)
    {
        if (0 != semop(semid,sops,nsops) )
            perror("semop()");
    }
    else
        perror("semget()");
}
```

Using the Examples

The following commands demonstrate the use of the example programs. First, a
semaphore set is created by *semget* and its existence verified with *ipcs*:

```
$ ipcs -s
IPC status from /dev/kmem as of Wed Jun 19 11:19:37 1996
T    ID    KEY        MODE       OWNER    GROUP
Semaphores:
$ semget -k 0xfab -c -x -p 0666 -s 4
semid = 130
owner uid.gid: 1110.20  creator uid.gid: 1110.20  mode: 0100666 nsems:4
$ ipcs -s
IPC status from /dev/kmem as of Wed Jun 19 11:19:59 1996
T    ID    KEY        MODE       OWNER    GROUP
Semaphores:
s   130 0x00000fab --ra-ra-ra-  cortesi     user
```

The effect of the IPC_EXCL flag is tested:

```
$ semget -k 0xfab -c -x
semget(): File exists
```

The permissions are changed using *semmod*:

```
$ semmod -i 130 -p 0640
owner uid.gid: 1110.20  creator uid.gid: 1110.20  mode: 0100640 nsems:4
$ ipcs -s
IPC status from /dev/kmem as of Wed Jun 19 11:20:09 1996
T     ID    KEY          MODE        OWNER     GROUP
Semaphores:
s    130 0x00000fab --ra-r-----  cortesi     user
```

The present state of the four semaphores in the set is displayed, then *semop* is used to increment the first two.

```
$ semsnap -i 130
vals:  0  0  0  0
ncnt:  0  0  0  0
zcnt:  0  0  0  0
$ semop -i 130 -v 0 -v 1
$ semsnap -i 130
vals:  1  1  0  0
ncnt:  0  0  0  0
zcnt:  0  0  0  0
```

One instance of *semop* is started in the background to wait on a sequence of operations. The *semsnap* display verifies that one process is waiting on zero in semaphore 0:

```
$ semop -i 130 -z 0 -p 1 -p 2 &
9956
$ semsnap -i 130
vals:  1  1  0  0
ncnt:  0  0  0  0
zcnt:  1  0  0  0
```

Semaphore 0 is decremented, and *semsnap* reveals that there is no longer a process waiting for zero in that semaphore, but that now a process is waiting for semaphore 2 to be incremented:

```
$ semop -i 130 -p 0
$ semsnap -i 130
vals:  0  1  0  0
ncnt:  0  0  1  0
zcnt:  0  0  0  0
```

Semaphore 2 is incremented and now there are no processes waiting:

```
$ semop -i 130 -v 2
$ semsnap -i 130
vals:  0  0  0  0
ncnt:  0  0  0  0
zcnt:  0  0  0  0
```

Another process is put in the background waiting on semaphore 0. Then the semaphore set is removed with *ipcrm*. The waiting instance of *semop* ends, displaying the error code from **semop()**:

```
$ semop -i 130 -p 0 &
9962
$ ipcrm -s 130
$ semop(): Identifier removed
```

Processes can receive *signals* in order to respond to asynchronous requests from software

Signalling Events

Processes can receive *signals* in order to respond to asynchronous requests from software or to unexpected hardware events. There are three different programming interfaces for receiving signals; you must select one and use it consistently throughout a program.

Many programs need access to time data for one of two purposes: to produce *timestamps* so that data can be ordered by its time of origin, and to define *intervals* so the program can take action at regular times. (Intervals are presented to the program as signals.)

These two issues are covered in the following topics:

- "Signals" on page 113 describes signal facilities in general and details the differences between the POSIX, SVR4, and BSD interfaces.

- "Timer Facilities" on page 127 describes POSIX and IRIX methods of defining timestamps and intervals.

Signals

A signal is a notification of an event, sent asynchronously to a process. Some signals originate from the kernel in response to hardware traps; for example, the SIGFPE signal that notifies of an arithmetic overflow, or the SIGALRM that notifies of the expiration of a timer interval. Other signals are issued by software. For a detailed, formal discussion of signals, read the signal(5) reference page.

A process can block all signals or selected signals, ignore some signals, or request a default system handling for some signals. When a signal that has been sent to a process is blocked by the process, the signal remains pending. When a signal is not blocked, the process receives the signal. In a multithreaded process, signals can be blocked or received by individual threads.

When receiving a signal, a process or thread can handle the signal by an asynchronous call into a signal-handling function. Alternatively, using the POSIX interface, a process or thread can handle signals synchronously, as a stream of event objects.

Signal Numbers

IRIX supports the following 64 signal numbers:

1-31   Same meanings as SVR4 and BSD; see Table 5-1.

32   Reserved by IRIX kernel.

33-48   Reserved by the POSIX standard for system use.

49-64   Reserved by POSIX for real-time programming.

Signals with smaller numbers have priority for delivery. The low-numbered BSD-compatible signals, which include all kernel-produced signals, are delivered ahead of real-time signals, and signal 49 takes precedence over signal 64.

Table 5-1 is reproduced from the signal(5) reference page for convenience.

Table 5-1   Signal Numbers and Default Actions

Symbolic Name	Numeric Value	Default Action	Normal Meaning
SIGHUP	1	Terminate	Controlling terminal disconnect; see termio(7).
SIGINT	2	Terminate	Interrupt key signal from controlling terminal; see termio(7).
SIGQUIT	3	Terminate and dump	Quit key signal from controlling terminal; see termio(7).
SIGILL	4	Terminate and dump	Attempt to execute illegal instruction.
SIGTRAP	5	Terminate and dump	Trace/breakpoint reached; see proc(4).
SIGABRT	6	Terminate and dump	Abort.
SIGEMT	7	Terminate and dump	Emulation trap.
SIGFPE	8	Terminate and dump	Arithmetic exception; see math(3M), sigfpe(3C), and matherr(3M).
SIGKILL	9	Terminate	Kill request from software or user.
SIGBUS	10	Terminate and dump	Bus error (hardware exception).
SIGSEGV	11	Terminate and dump	Segmentation fault (illegal address).

Table 5-1 (continued)   Signal Numbers and Default Actions

Symbolic Name	Numeric Value	Default Action	Normal Meaning
SIGSYS	12	Terminate and dump	Invalid system call.
SIGPIPE	13	Terminate	Read or write to broken pipe; see pipe(2), read(2), write(2).
SIGALRM	14	Terminate	Interval timer elapsed; see "Timer Facilities" on page 127.
SIGTERM	15	Terminate	Process terminated.
SIGUSR1	16	Terminate	Programmer-defined; see also text below.
SIGUSR2	17	Terminate	Programmer-defined.
SIGCHLD or SIGCLD	18	Terminate	Child process status change; see wait(2) and "Process "Reaping"" on page 259.
SIGPWR	19	Ignore	Power fail/restart.
SIGWINCH	20	Ignore	Change in size of window; see xterm(1).
SIGURG	21	Ignore	Urgent socket condition; see socket(2).
SIGPOLL	22	Terminate	Pollable event from a STREAMS device, see streamio(7).
SIGIO	22	Terminate	Input/output possible.
SIGSTOP	23	Suspend	Stopped.
SIGTSTP	24	Suspend	Stop key signal from controlling terminal; see termio(7).
SIGCONT	25	Ignore	Continued.
SIGTTIN	26	Suspend	Attempt to read terminal from background process; see termio(7).
SIGTTOU	27	Suspend	Attempt to write terminal from background process; see termio(7).
SIGVTALRM	28	Terminate	Virtual timer expired; see getitimer(2).
SIGPROF	29	Terminate	Profiling timer expired; see getitimer(2).

Table 5-1 (continued)   Signal Numbers and Default Actions

Symbolic Name	Numeric Value	Default Action	Normal Meaning
SIGXCPU	30	Terminate and dump	CPU time limit exceeded; see getrlimit(2).
SIGXFSZ	31	Terminate and dump	File size limit exceeded; see getrlimit(2) and write(2).
(no symbol)	32-48	Terminate	Unassigned; do not use.
SIGRTMIN - SIGRTMAX	49-64	Terminate	POSIX real-time signal range.

Although SIGUSR1 and SIGUSR2 are nominally defined by the you for your program's purposes, they are also used by different application packages for special signals. For example, if you set a file lock on an NFS mounted file, the NFS lock daemon may send SIGUSR1—see "NFS File Locking" on page 188.

Signal Implementations

There are three UNIX traditions for signals, and IRIX supports all three. They differ in the library calls used, in the range of signals allowed, and in the details of signal delivery. The basic signal operations and the implementing functions are summarized in Table 5-2.

Table 5-2   Signal Handling Interfaces

Function	POSIX Functions	SVR4 Functions	BSD 4.2 Functions
Set and query signal handler	sigaction(2) sigsetops(3) sigaltstack(2)	sigset(2) signal(2)	sigvec(3) signal(3)
Send a signal	sigqueue(2) kill(2) pthread_kill(3P)	sigsend(2) kill(2)	kill(3) killpg(3)
Temporarily block specified signals	sigprocmask(2) pthread_sigmask(3P)	sighold(2) sigrelse(2)	sigblock(3) sigsetmask(3)
Query pending signals	sigpending(2)	n.a.	n.a.

Table 5-2 (continued)   Signal Handling Interfaces

Function	POSIX Functions	SVR4 Functions	BSD 4.2 Functions
Wait for a signal handler to be invoked.	sigsuspend(2)	sigpause(2)	sigpause(3)
Wait for a signal and receive synchronously	sigwait(2) sigwaitinfo(2) sigtimedwait(2)	n.a.	n.a.

It is important to not mix these signal facilities. Your program should use functions from only one column of Table 5-2; otherwise unexpected results can occur.

Signal Blocking and Signal Masks

Certain ideas are basic to the use of signals. One basic idea is that a program can block the delivery of any signal. When a signal that is sent to a program is blocked, the signal is queued and remains pending until the program unblocks the signal, or terminates. Certain urgent signals—SIGKILL, SIGSTOP, SIGCONT—cannot be blocked.

You specify which signals are blocked using a signal mask, a set of bits in which each bit corresponds to one signal number. When a bit in the mask is set on, the signal is blocked (if it is a signal that can be blocked).

Each process has a signal mask, inherited from its parent process. All three interfaces provide ways to set and clear bits in the current signal mask. The BSD interface, however, only lets you mask the first 32 signal numbers listed in Table 5-1.

Each POSIX thread has a signal mask also (see "Setting Signal Masks" on page 279). A multithreaded program (defined as a program that is linked with *libpthread*, so it uses the pthreads version of the standard library) should use the POSIX interface for signal handling.

Multiple Signals

In most cases, if a signal of a certain number is pending for a process, and another signal of the same number arrives, the second signal is discarded. In other words, at most one signal of a given number can normally be pending for a process.

In the POSIX interface you can use one particular function, **sigqueue()**, to send a signal that is queued regardless of how many signals of the same number are already pending.

Signal Handling Policies

You can specify one of three policies for handling an unblocked signal. You set the policy for each signal number individually.

Default Handling

Initially, all signals receive default handling. This means that when a signal arrives and is not blocked, it causes the default action listed in Table 5-1. In many cases the default action is to ignore the signal, that is, to silently discard it. In other cases, the default action is to terminate the program, or to terminate it with a dump.

Each signal interface gives you a way to specify non-default handling or a specified signal, or to return a signal to default handling.

Ignoring Signals

You can request that a specified signal be ignored. You would do this when the signal is not meaningful to your program and the default action is not what you wish. For example, in a noninteractive program, you might set Ignore handling for SIGHUP (the default action is to terminate).

Catching Signals

You can request that a signal be caught and handled asynchronously, at the moment it arrives. You specify that a signal should be caught by specifying the address of a function to be called when the signal is received.

The signal-handling function is entered asynchronously, without regard for what the process was doing at the time the signal was delivered. You cannot be sure what code was executing when the signal handler is called; it could have been any function in your own code, or it could have been code in the C library or in any layer of the X-Windows or Motif support libraries.

All three interfaces provide for passing the signal number as the first argument of the signal-handling function. Other arguments to the handler function depend on the interface used and the options you specify when establishing the handler.

You can create an alternate memory area to be used as a stack when executing the signal handler. Typically a signal handler does not require a great deal of stack space. On the other hand, each POSIX thread has limited stack space, and when you provide an alternate signal-handling stack, you do not have to allow for possible signals in allocating thread stack space (see "Setting Signal Actions" on page 279).

Synchronous Signal Handling

Using the POSIX signal interface you can process signals in a synchronous way, as a stream of input items to your program. This allows you to design your program so that signals are received when the process is in a known state, without the uncertainties of asynchronous delivery.

Signal Latency

The time that elapses from the moment a signal is generated until a signal handler begins to execute is the *signal latency*. Signal latency can be long (as real-time programs measure time) and signal latency has a high variability.

The IRIX kernel normally delivers a pending, unblocked signal the next time the process returns to user code from the kernel domain. In most cases, this occurs

- when the process is dispatched after a wait or preemption

- upon return from a system function

- upon return from the kernel's usual 10-millisecond "tick" (dispatch) interrupt

SIGALRM, which signals the expiration of a real-time timer (see "Timer Facilities" on page 127), is given special treatment. It is delivered as soon as the kernel is ready to return to a user process after the timer interrupt, in order to preserve timer accuracy.

When a process is ready to run and is not preempted by a process of higher priority, and is executing in user code, not calling a system function, the latency for other than SIGALRM can be as much as 10 milliseconds. However, when the process is suspended (for example, waiting on a semaphore), or when there are competing processes having higher priorities, the delivery of a signal is delayed until the next time the receiving process is scheduled. This can be many milliseconds.

In general, you should use signals to deliver infrequent messages of high priority. You should not use the exchange of signals as the basis for real-time scheduling.

Signals Under X-Windows

If you plan to handle signals asynchronously in a program that uses X intrinsics, you must take special steps. Before establishing a signal handler with the operating system, you establish one or more signal callback procedures using **XtAppAddSignal()**. Then, in the asynchronous signal handling function, you call **XtNoticeSignal()**. This function ensures that the established signal callback will be invoked like other callback functions, when it is safe to do so. This process is documented in the XtAppAddSignal(3Xt) reference page.

The only X-windows function that can safely be called from a signal handler is **XtNoticeSignal()**.

POSIX Signal Facility

The POSIX interface to signals is the most functionally complete and robust of the three. It is the recommended interface for all new programs. The functions used in POSIX style signal handling are summarized in Table 5-3.

Table 5-3   Functions for POSIX Signal Handling

Function	Purpose
kill(2)	Send a signal to a process or process group. (Discards multiple signals of the same number.)
sigqueue(3)	Queue a signal to a specified process, including a *sigval* for added information about the signal. (Queues multiple signals of the same number.)
pthread_kill(3P)	Send a signal to a specified thread.
sigprocmask(2) pthread_sigmask(3P)	Examine or change the mask of signals allowed and blocked. You must use **pthread_sigmask()** in a program that is linked with *libpthread*.
sigaction(2)	Specify or query the signal handling policy for a specified signal.
sigaltstack(2)	Specify or query an alternate stack area to be used by a signal handler.
sigpending(2)	Return the set of signals pending for the calling process or thread.
sigsetops(3)	Manipulate signal mask objects in memory.
sigsuspend(2)	Unblock selected signals for the calling process or thread, and wait for a signal to be received asynchronously.
sigwait(3) sigtimedwait(3) sigwaitinfo(3)	Wait for and receive specified signals in a synchronous manner.

In addition to the reference pages listed in Table 5-3, the following have important information about signal handling:

signal(5)	Detailed overview of signals and signal handling.
siginfo(5)	Description of the information structure passed to a POSIX signal handler.
ucontext(5)	Description of machine context structure passed to a POSIX signal handler.

Signal Masking

Each process and thread has an active signal mask. A single-thread program sets or queries its signal mask using **sigprocmask()**. A multithreaded program (any program that linked *libpthread*, which provides the pthread version of the standard library) should use **pthread_sigmask()**.

Besides the active signal mask, you may have other signal mask objects (type *sigset_t*) in memory. The sigsetops(3) reference page documents a number of utility functions for setting, clearing, and testing the bits in a signal mask object. Several POSIX signal functions take a signal mask as an argument. For example, **sigsuspend()** takes a new signal mask and swaps it for the current signal mask, establishing which pending signals will be accepted while the process is suspended.

Using Synchronous Handling

You can design your program so that it treats arriving signals as a stream of event records to be processed in sequence. For example, you could use one or more signal numbers in the POSIX real-time range to signify events that are meaningful to your application. Your application, or one thread in your application, can receive each signal in turn and act upon it.

To implement this design approach, follow these steps:

1. Block the expected signal numbers in all processes or threads using **sigprocmask()** or **pthread_sigmask()**.

2. Send the signals using **sigqueue()**. This function permits you to augment the signal number with a *union sigval* (in effect creating an open-ended set of sub-signals), and also assures that multiple signals will be retained until you process them.

3. In the signal-processing loop, wait for the next signal with **sigwaitinfo()** or **sigtimedwait()**. When the signal arrives, act accordingly and wait again.

The **sigwaitinfo()** and **sigtimedwait()** functions accept a new signal mask. They unblock the specified signal or signals and suspend until one such signal arrives. They accept that signal, restore the original signal mask, and return the signal information.

You could construct a very similar work-handling application using a message queue (see Chapter 6, "Message Queues"). However, this design approach allows you to integrate the handling of unplanned signals such as SIGPIPE, and interval-timer signals such as SIGALRM, into the same scheme as planned application events.

Using Asynchronous Handling

Using **sigaction()**, you specify a function to be called when a particular signal is received. You have a choice of function prototypes. In each case the signal handler is passed the signal number, additional information about the signal, and information about the machine context at the time the signal was delivered.

Your signal handler can have the POSIX prototype, as follows:

```
void name(int sig, siginfo_t *sip, ucontext_t *up)
```

The second argument, a POSIX information structure *siginfo_t*, contains these fields:

si_signo   The signal number (again).

si_errno   Either 0 or an error code from *errno.h*.

si_code   An indication of the source of the signal.

si_value   When *si_code* is SI_QUEUE, the *union sigval* passed to **sigqueue()**.

si_pid   When *si_code* is SI_USER, the process ID that called **kill()**.

When the signal is an error reported by the kernel or hardware, *si_code* is an explanatory number. These values are spelled out in detail in the siginfo(5) reference page. The third argument, a pointer to a *ucontext_t* object, gives the machine state at the time the signal was delivered. The *ucontext_t* is detailed in the ucontext(5) reference page.

Alternatively, your signal handler can have this prototype:

```
void name(int sig, int code, struct sigcontext *sc);
```

The second argument gives some added information about the signal (see signal(5) for a list of codes). The third argument, a pointer to a *sigcontext_t* object, gives the machine state at the time the signal was delivered (in slightly different form from the *ucontext_t*).

When you use **sigaction()** to set up a signal handler, you pass an argument structure containing option flags that affect the treatment of the signal:

SA_SIGINFO   When set, you are specifying asynchronous handling and your handler uses the POSIX prototype. Its address is passed in the *sa_sigaction* structure field. When not set, a handler uses the older prototype and its address is passed in *sa_handler*.

SA_ONSTACK   When set, your handler is called using alternate stack memory you have previously assigned with **sigaltstack()**. Otherwise the handler uses the stack of the process or thread stack executing at the time of the signal.

SA_RESETHAND   When set, the policy for this signal is reset to the default when your handler is called. Your handler is expected to reestablish the action if that is desired.

SA_NODEFER   When not set, the signal is automatically blocked while your handler executes, and unblocked when your handler returns. When set, the same signal could be taken while your handler executes, resulting in multiple entries to the handler.

SA_RESTART   When not set, if this signal interrupts a blocked system function the system function returns EINTR. When set, the system function is restarted.

System V Signal Facility

The System V signal interface is compatible with code ported from UNIX System V. It includes compatibility for release 3 (SVR3) and release 4 (SVR4). Table 5-4 summarizes the functions you use to manage signals through this interface.

Table 5-4   Functions for SVR4 Signal Handling

Function	Purpose
kill(2)	Send a signal to a process or process group. (A duplicate of a pending signal is discarded.)
sigsend(2)	Send a signal to a set of processes or process groups, specified in a variety of ways, for example by user ID.
signal(2)	SVR3 call to establish handling policy of default, ignore, or catch for a specified signal.
sigset(2)	SVR4 call to establish handling policy of default, ignore, or catch for a specified signal.
sighold(2)	Hold (block) a specified signal.
sigignore(2)	Set the handling for a specified signal to Ignore.
sigrelse(2)	Release (unblock) a specified signal.
sigpause(2)	Suspend the calling process until a specified signal arrives.

Only asynchronous signal handling is supported by the System V interface. Also, you must block and unblock signals individually; there is no support for setting the entire signal mask in one operation.

The semantics of SVR3-compatible signal established with **signal()** are not desirable for most programs. When control enters a signal handler you established using **signal()**, the handling of that same signal is set to default, and that signal remains unblocked. Your signal handler can use **signal()** to reestablish itself as the handler, or it can use **sighold()** to block the signal. However, even if these actions are the first statements of the handler function, there is a period of time at the beginning of the handler during which a second signal of the same type could be received. If this occurs, the second signal receives default handling and is not seen by your handler.

You can avoid this problem by using the SVR4 function **sigset()** instead of **signal()** to establish a handler. Before a handler established by **sigset()** is called, that signal is blocked until the handler returns, and the signal disposition is not reset to default.

BSD Signal Facility

The BSD signal facility is compatible with code ported from the BSD 4.2 distribution. Table 5-5 summarizes the functions you use to manage signals with this interface.

Note: In order to use any of the functions in Table 5-5 you must define one of the compiler variables _BSD_SIGNALS or _BSD_COMPAT prior to the inclusion of the header file *signal.h*. You can do this directly in the source file with #define. More commonly you will include -D_BSD_COMPAT as one of the compiler flags you define in your Makefile.

Table 5-5   Functions for BSD Signal Handling

Function Name	Purpose and Operation
kill(3B)	Send a signal to a specified process, or broadcast a signal to a process group or to all processes with the same effective user ID. (A duplicate of a pending signal is discarded.)
killpg(3B)	Send a signal to all members of a process group. (A duplicate of a pending signal is discarded.)
sigvec(3)	Establish a policy of default, ignore, or catch for a specified signal.
signal(3B)	Simplified interface to **sigvec()**.
sigstack(2B)	Establish an alternate stack for the use of signal-handling functions.
sigsetmask(3)	Set the active signal mask.
sigblock(3)	Add blocked signals to the active signal mask.
sigpause(3B)	Wait for specified signals to arrive.

Only asynchronous signal handling is supported by the BSD interface. It is possible to set and interrogate the signal mask in a single operation; however, the signal mask type is the integer, so only signal numbers 1-32 can be blocked. The BSD interface does not recognize higher-numbered signals.

Timer Facilities

You use timer facilities for a number of purposes: to get information about program performance; to make a program pause for a certain time; to program an interval of time; and to create a timestamp value to store with other data.

Timed Pauses and Schedule Cession

In many instances a program, or a process within a multiprocess program, needs to suspend execution for a period of time. IRIX contains a variety of functions that provide this capability. The functions differ in their precision and in their portability. Table 5-6 contains a summary.

Table 5-6   Functions for Timed Suspensions

Reference Page	Precision	Compatibility	Operation
sched_yield(2)	n.a.	POSIX	Defer to any processes eligible to run.
sginap(2)	dispatching interval (10ms)	IRIX	Defer to other processes for the specified number of dispatching cycles.
sleep(3C)	second	POSIX	Suspend for a number of seconds or until a signal arrives.
usleep(3C)	microsecond	IRIX	Suspend for a number of microseconds or until a signal arrives.
nanosleep(2)	nanosecond	POSIX	Suspend for a number of seconds and nanoseconds or until a signal arrives.

Sometimes you do not want to suspend for any particular amount of time, but simply want to make the current process defer to other processes, so that any waiting processes receive a chance to run. You can achieve this in two ways. The IRIX unique function **sginap()** accepts an argument of 0, meaning to defer for the minimum amount of time. However, **sched_yield()** is a POSIX compliant function for this purpose.

Time Data Structures

The include files *time.h* and *sys/time.h* define several data types and data structures related to time. Some of these are used in POSIX time functions and others in BSD-based functions; and there are somewhat confusing similarities between them. Features of these structures are summarized in Table 5-7.

Table 5-7   Time Data Structures and Usage

Data Type	Declared In	Contains	Some Functions Using This Type
time_t	*time.h*	long int with time in seconds since 00:00:00 UTC, January 1, 1970	time(2), ctime(3C), cftime(3C), difftime(3C)
timeval	*sys/time.h*	structure of *time_t* giving seconds and a long int giving microseconds	adjtime(2), getitimer(2), getrusage(3C), gettimeofday(3C), select(2), utimes(3B)
itimerval	*sys/time.h*	structure of two *timeval* fields for first interval and repeat interval	getitimer(2) and setitimer(2)
timespec_t	*time.h*	structure of *time_t* giving seconds and a long int giving nanoseconds	clock_gettime(2), nanosleep(2), aio_suspend(3), sigtimedwait(3C)
itimerspec	*time.h*	structure of two *timespec_t* fields for first interval and repeat interval	timer_settime(3C), timer_gettime(3C)
tm	*time.h*	structure of int fields for seconds, minutes, hours, day, month, etc.	localtime(2), gmtime(2), strftime(3C)

Time Signal Latency

It takes time for the kernel to deliver the SIGALRM that notifies your program at the end of an interval. (The issue of signal latency in general is discussed under "Signal Latency" on page 119.) The signal latency is less for SIGALRM than for other signals, since the kernel initiates a scheduling cycle immediately after the timer interrupt, without waiting for the end of a fixed time slice. When the receiving process or thread is running or ready to run, the latency is fairly short and consistent from one signal to the next. (Even so, it is not advisable to use a repeating itimer as the time base for a real-time program). Under less favorable conditions, signal latency can be variable and sometimes lengthy (tens of milliseconds) relative to a fast timer frequency.

How Timers Are Managed

The IRIX kernel can be asked to implement itimers for many processes at once, each interval having a different length and starting at a different time. The kernel's method differs depending on the hardware architecture (this issue is discussed at length in the timers(5) reference page).

- Some obsolete Silicon Graphics systems have no hardware support for interval timers, so the kernel had to rely on frequent, periodic interrupts as a time base.

  In those systems, the precision of timer interrupts was controlled by a kernel tuning variable, *fasthz*, which determined the rate at which the kernel was interrupted to poll for an expired timer.

- In all current architectures, each CPU has a clock comparator that the kernel can program to cause an interrupt after a specific interval has elapsed.

  In these systems, timer interrupts have sub-microsecond precision and do not impose overhead for timer-polling interrupts.

In earlier versions of IRIX, in order to minimize the overhead of polling for elapsed timers, the kernel did not allow normal processes to ask for timer intervals with fine granularity (sub-millisecond precision). Only processes that executed under real-time scheduling priority could ask for precise timer intervals.

Starting with IRIX 6.2, any process can request a timer interval with any precision. If this support is misused, it is possible to cause performance problems. For example, a process can set up a repeating timer at an interval so short that one CPU is monopolized by setting and handling that timer.

POSIX Timers

IRIX supports the time and timer facilities specified by IEEE standard 1003.1b-1993, commonly called POSIX timers. This timer interface is the most complete, robust, and portable, and is recommended for all new applications. The functions it includes for time measurement are summarized in Table 5-8.

Table 5-8   POSIX Time Management Functions

Function Name	Purpose and Operation
time(2)	Return a *time_t* value containing the count of seconds elapsed since 00:00:00 UTC, January 1, 1970.
times(2)	Return user and system execution time consumption for the calling process and its terminated child processes.
clock_gettime(2)	Return the instantaneous reading of one of two clocks: the system time (CLOCK_REALTIME), or the hardware cycle counter (CLOCK_SGI_CYCLE).
clock_getres(2)	Return the precision of the system time (CLOCK_REALTIME), the hardware cycle counter in this system (CLOCK_SGI_CYCLE) or the high-resolution timer base (CLOCK_SGI_FAST).

The POSIX functions for interval timers are summarized in Table 5-9.

Table 5-9   POSIX Time Management Functions

Function Name	Purpose and Operation
alarm(2)	Cause a SIGALRM signal after a specified number of whole seconds.
timer_create(3C)	Create a POSIX timer and specify its time base (CLOCK_REALTIME or CLOCK_SGI_FAST) and the signal number it can generate.
timer_delete(3C)	Remove a timer created with **timer_create()**.
timer_settime(3C)	Set expiration and reload times of a timer, or disarm it.
timer_gettime(3C)	Query the time remaining in a timer.
timer_getoverrun(3C)	Query the number of overrun events generated by a timer.

Getting Program Execution Time

The **times()** function returns counts of accumulated user-process and system execution time. These counts have a resolution of the system dispatching interval, 10 milliseconds.

Creating Timestamps

The **time()** function returns a timestamp with a resolution of 1 second. A timestamp with a resolution this coarse can be used only for infrequent events.

You can use the **clock_gettime()** function to sample the system time with a resolution of 0.01 second, or you can use it to read the hardware cycle counter—a free-running binary counter with an update frequency near the machine clock rate. The **clock_getres()** function returns the resolution of either of these clocks.

The program in Example 5-1 demonstrates the use of **clock_gettime()** and **clock_getres()**. The following is an example of the output of this program, *ptime*, as executed on an Indy workstation:

```
$ ptime
CLOCK_REALTIME value: sec 835660711, ns 465330000 [8.35661e+08 sec]
CLOCK_REALTIME units: sec 0, ns 10000000 [0.01 sec]
CLOCK_SGI_CYCLE value: sec 83, ns 449744360 [83.4497 sec]
CLOCK_SGI_CYCLE units: sec 0, ns 40 [4e-08 sec]
CLOCK_SGI_FAST units: sec 0, ns 1000000 [0.001 sec]
```

Example 5-1   Example of POSIX Time Functions

```
/*
|| Program to exercise POSIX clock_gettime() and clock_getres() functions.
||
||   ptime [-r -c -R -C -F]
||      -r  display CLOCK_REALTIME value
||      -R  display CLOCK_REALTIME resolution
||      -c  display CLOCK_SGI_CYCLE value
||      -C  display CLOCK_SGI_CYCLE resolution
||      -F  display CLOCK_SGI_FAST resolution (cannot get time from this)
|| Default is display everything (-rRcC).
*/
#include <time.h>
#include <unistd.h>      /* for getopt() */
#include <errno.h>       /* errno and perror */
#include <stdio.h>
void showtime(const timespec_t tm, const char *caption)
{
    printf("%s: sec %ld, ns %ld [%g sec]\n",
        caption, tm.tv_sec, tm.tv_nsec,
        ((double)tm.tv_sec) + ((double)tm.tv_nsec / 1e9));
}
```

```
main(int argc, char **argv)
{
    int opta = 1;
    int optr = 0;
    int optR = 0;
    int optc = 0;
    int optC = 0;
    int optF = 0;
    timespec_t sample, res;
    int c;
    while ( -1 != (c = getopt(argc,argv,"arRcCF")) )
    {
        switch (c)
        {
        case 'a': opta=1; break;
        case 'r': optr=1; opta=0; break;
        case 'R': optR=1; opta=0; break;
        case 'c': optc=1; opta=0; break;
        case 'C': optC=1; opta=0; break;
        case 'F': optF=1; opta=0; break;
        default: return -1;
        }
    }
    if (opta || optr)
    {
        if (!clock_gettime(CLOCK_REALTIME,&sample))
            showtime(sample,"CLOCK_REALTIME value");
        else
            perror("clock_gettime(CLOCK_REALTIME)");
    }
    if (opta || optR)
    {
        if (!clock_getres(CLOCK_REALTIME,&res))
            showtime(res,"CLOCK_REALTIME units");
        else
            perror("clock_getres(CLOCK_REALTIME)");
    }
    if (opta || optc)
    {
        if (!clock_gettime(CLOCK_SGI_CYCLE,&sample))
            showtime(sample,"CLOCK_SGI_CYCLE value");
        else
            perror("clock_gettime(CLOCK_SGI_CYCLE)");
    }
    if (opta || optC)
```

```
    {
        if (!clock_getres(CLOCK_SGI_CYCLE,&res))
            showtime(res,"CLOCK_SGI_CYCLE units");
        else
            perror("clock_getres(CLOCK_SGI_CYCLE)");
    }
    if (opta || optF)
    {
        if (!clock_getres(CLOCK_SGI_FAST,&res))
            showtime(res,"CLOCK_SGI_FAST units");
        else
            perror("clock_getres(CLOCK_SGI_FAST)");
    }
}
```

The real-time clock (CLOCK_REALTIME) can shift backward or jump forward under the influence of adjustments to the system time by a time daemon. The Silicon Graphics hardware cycle counter always increases at a steady rate. However, the cycle counter has a limited precision that depends on the hardware. You can use the **syssgi()** system function to find out the precision of the cycle counter (see syssgi(2) and look for the SGI_CYCLECNTR_SIZE option).

Using Interval Timers

You create an interval timer object by calling **timer_create()**. To this function you pass codes that specify the time base to use and the signal to send upon timer expiration. It returns an ID value to identify the timer to other functions.

The time base for a timer is either CLOCK_REALTIME or CLOCK_SGI_FAST (the latter is a nonportable request). Typically CLOCK_SGI_FAST has finer resolution, but you can verify that using the **clock_getres()** function, as shown in Example 5-1.

You also pass a *sigevent_t* object to **timer_create()**. In it you would normally set the following values:

sigev_notify	SIGEV_SIGNAL to have the timer generate a signal on expiration.
sigev_signo	The signal number you want sent, possibly selected from the POSIX real-time range, for example, SIGRTMIN+1.
sigev_value.sival_int *sigev_value.sival_ptr*	An extra value to be passed to the signal-handling function or to **sigwait()** when the signal is delivered.

133

You can pass a NULL instead of the address of a *sigevent_t*. In that case, the timer signals with a SIGALRM.

Initially, a timer is disarmed (inactive). You start a timer by calling **timer_settime()**. The principal argument to this function is an *itimerspec_t* object, which contains two times. One, *it_value*, specifies when the timer next expires. The other, *it_interval*, is the value to be loaded into the timer when it expires. You can call **timer_settime()** to accomplish any of three different operations:

• With *it_value* nonzero and *it_interval* zero, arm the timer and initiate a one-time interval.

• With *it_value* nonzero and *it_interval* nonzero, arm and initiate a repeating timer.

• With *it_value* zero, disarm the timer, preventing it from expiring (if it has not expired already).

You can also use **timer_settime()** to reprogram the intervals in a timer while it runs.

A timer can be programmed in terms of relative time (you pass an *it_value* that represents increments past the present time) or absolute time (you pass an *it_value* that represents actual future times when the timer should expire).

You can interrogate the time remaining in a timer by calling **timer_gettime()**. After a timer has expired—for example, in the signal handling function—you can call **timer_getoverrun()** to find out how many additional intervals it would have signalled, but could not signal because the first signal was pending.

BSD Timers

IRIX supports the BSD UNIX feature of interval timers or "itimers." Table 5-10 summarizes the functions you use to manage itimers.

Table 5-10   BSD Functions for Interval Timers

Function Name	Purpose and Operation
setitimer(2)	Set the expiration and repeat interval of a timer.
getitimer(2)	Return the current value of a timer.

Each process has three itimers available to it, as summarized in Table 5-11.

Table 5-11   Types of itimer

Kind of itimer	Interval Measured	Resolution	Signal Sent
ITIMER_REAL	Elapsed clock time	1 millisecond or less	SIGALRM
ITIMER_VIRTUAL	User time (process execution time)	1 second	SIGVTALRM
ITIMER_PROF	User+system time	1 second	SIGPROF

The ITIMER_VIRTUAL and ITIMER_PROF have a relatively coarse precision. Their intervals vary depending on when and how often the process is dispatched. The ITIMER_REAL timer is comparable to the POSIX time base CLOCK_SGI_FAST.

In order to use an itimer, you establish a signal handler for the appropriate signal as shown in Table 5-11, then issue the **setitimer()** call. The principal argument to this function is a *struct itimerval*, an object containing two incremental time values. The *it_value* field specifies the time until the timer should expire. The *it_interval* field, when nonzero, gives the time that should be loaded into the timer after it expires.

Tip: One excellent reason not to mix BSD and POSIX timer support in the same program is that the POSIX *struct itimerspec*, used to set a POSIX timer, and the BSD *struct itermval*, used to set a BSD itimer, have fields with identical names, but these fields have different data types and precisions.

You can use **setitimer()** for any of three operations:

- With *it_value* nonzero and *it_interval* zero, initiate a one-time interval.

- With *it_value* nonzero and *it_interval* nonzero, initiate a repeating timer.

- With *it_value* zero, disarm the timer, preventing it from expiring (if it has not expired already).

Hardware Cycle Counter

All current Silicon Graphics systems have a hardware "cycle counter," a free-running binary counter that is incremented at a high, regular frequency. You can use the cycle counter as a high-precision timestamp.

The precision of the cycle counter is different in different system types; for example, it is a 24-bit counter in the Indy workstation, but a 64-bit counter in CHALLENGE and Onyx systems. The rate at which the timer increments is its resolution, and this also varies with the hardware type.

The cycle counter is an addressable hardware device that you can map into the address space of your process (see "Mapping Physical Memory" on page 20). When this is done you can sample the cycle counter as if it were a program variable. The code to do this mapping is discussed in the syssgi(2) reference page under SGI_QUERY_CYCLECNTR.

However, the use of the hardware cycle counter has been integrated into the POSIX timer support beginning in IRIX 6.2, and this makes access to the cycle counter much simpler than before:

- In order to sample the cycle counter, call **clock_gettime()** passing CLOCK_SGI_CYCLE.

- In order to find out the resolution (update frequency) of the cycle counter, call **clock_getres()** passing CLOCK_SGI_CYCLE.

- In order to find out the precision of the cycle counter, call **syssgi()** passing SGI_CYCLECNTR_SIZE. The returned value is the number of bits in the counter.

The first two operations are illustrated in Example 5-1 on page 131.

Message Queues

You use a message queue to pass blocks of data between processes or threads without having to share any memory between the processes. One process or thread puts a message into the queue. The message is held in the queue until another process or thread asks for the message.

IRIX supports two implementations of message queues: a POSIX implementation as specified by IEEE standard 1003.1b-1993, and an SVR4 implementation compatible with System V Release 4. Both implementations can be used to coordinate POSIX threads or IRIX processes. This chapter discusses message queues under these headings:

- "Overview of Message Queues" on page 138 describes message queues and the differences between the two implementations.

- "POSIX Message Queues" on page 140 documents the use of the POSIX implementation.

- "System V Message Queues" on page 153 documents the use of the SVR4 implementation.

Overview of Message Queues

A message queue is a software object maintained by the IRIX kernel, logically apart from the address space of any process. When you create a message queue, the queue has a public identifier. (The identifier is a file pathname for POSIX, or an integer for SVR4.) A process uses the identifier to open the queue. When the queue is open, the process can send messages to the queue or receive messages from the queue.

A message queue has an access mode similar to a file access mode, specifying read and write access for its owner, its owner's group, or all users. A process with an effective user ID giving only read access can only receive messages from the queue. A process with an effective user ID lacking access cannot open the queue.

When a process requests a message from a queue and no message is available, the process can be notified immediately with an error code, or it can be suspended until a message is sent.

A message queue has a limit on the amount of data that can be queued. (POSIX limits the number of messages; SVR4 limits the total size of queued messages.) When a process sends a message that would exceed the queue's limit, the process can be notified immediately with an error code, or it can be suspended until there is room in the queue.

Implementation Differences

The abstract operations that a message queue supports are summarized in Table 6-1 with the names of the POSIX and SVR4 functions that implement them.

Table 6-1   Abstract Operations on a Message Queue

Operation	POSIX Function	SVR4 Function
Gain access to a queue, creating it if it does not exist.	mq_open(3)	msgget(2)
Query attributes of a queue and number of pending messages.	mq_getattr(3)	msgctl(2)
Change attributes of a queue.	mq_setattr(3)	msgctl(2)
Give up access to a queue.	mq_close(3)	n.a.
Remove a queue from the system.	mq_unlink(3), rm(1)	msgctl(2), ipcrm(1)

Table 6-1 (continued)   Abstract Operations on a Message Queue

Operation	POSIX Function	SVR4 Function
Send a message to a queue.	mq_send(3)	msgsnd(2)
Receive a message from a queue.	mq_receive(3)	msgrcv(2)
Request asynchronous notification of a message arriving at a queue.	mq_notify(3)	n.a.

Both implementations can be used to communicate between POSIX threads and between IRIX processes in any combination. Besides obvious features of syntax, the principal differences between the two implementations are as follows:

- POSIX functions are implemented as library functions in the *libc* library and operate primarily in the user process address space. SVR4 functions are implemented in the kernel, and every operation requires a context switch. This generally results in lower overhead for the POSIX functions.

- The identity of a POSIX or an SVR4 queue is retained over a reboot. The contents of a POSIX queue might or might not survive a reboot, but you should not depend on either type of queue to retain its state after the last program closes it.

- POSIX allows you to set a limit on the number of messages and the size of one message. SVR4 allows you to set a limit on the aggregate size of queued messages, but not on their number or their individual sizes.

- With a POSIX queue, the choice of whether or not operations should block on a full or empty queue is an attribute of the queue descriptor. With SVR4, you specify blocking or nonblocking operation on each send or receive operation.

- POSIX supports asynchronous notification of a message arrival. SVR4 does not.

- SVR4 allows a receiver to request a message from a particular priority class, in effect creating sub-queues within a queue. POSIX supports a priority class on each message, but it always returns the first message of the highest priority class.

Uses of Message Queues

You can use message queues in a variety of ways. For example, you can use a message queue to implement the "producer-consumer" model of cooperating processes or threads. The "producer" sends its output to the queue; the "consumer" receives the data from the queue. When one process gets ahead of the other, it is automatically suspended on the queue until the other process catches up.

Another design model, common in real-time programming, is to use message queues to dispatch units of work to waiting processes or threads. A process or thread dedicated to one type of work waits on a message queue. Whenever another process or thread needs a unit of work of that type, it sends the unit to that queue as a message.

Another use of a message queue is to regulate the use of a scarce resource, such as the buffers in a pool of buffers. Each resource unit is represented by a message. In order to obtain a unit, you receive one message from the queue. To release a unit for other processes to use, you send the unit message back to the queue.

The latter scheme can be used to compensate for a performance problem. The speed of communication through a queue is limited by the fact that every message is copied twice: when a message is sent, it is copied from the sender's buffer to some reserved memory space; when the message is received, it is copied into the buffer supplied by the receiving process or thread. When messages are small (or few in number), copying is not a serious problem.

When messages are large, copying can be avoided as follows. Allocate a pool of message buffers. Set up a queue of small messages, each message representing a "ticket" to use a particular buffer. In order to obtain a buffer, a process receives a message from this queue. The process fills the buffer, then it sends the buffer without copying, by sending only the "ticket" on another message queue. The process that receives the "ticket" uses the data in the buffer without needing to copy it, and releases the buffer by sending the "ticket" to the original queue.

POSIX Message Queues

The POSIX real-time extensions (detailed in IEEE standard 1003.1b) include support for messages queues. These functions are discussed in the following topics and demonstrated in example programs.

Managing Message Queues

The POSIX functions for creating, controlling, closing, and removing message queues are summarized in Table 6-2.

Table 6-2   POSIX Functions for Managing Message Queues

Function Name	Purpose and Operation
mq_open(3)	Create a queue if it does not exist, and gain access to it.
mq_getattr(3)	Get information about an open message queue.
mq_setattr(3)	Change the blocking/nonblocking attribute of an open message queue.
mq_close(3)	Give up access to a queue.
mq_unlink(3)	Remove a message queue from the system when the last process to have it open, closes it.

Creating a Message Queue

The **mq_open()** function has two purposes. It is used to gain access to a queue that exists, and it can create a queue that does not exist. To create a new queue, call **mq_open()** with four arguments as follows (using the names given in the reference page):

mq_name	The pathname that the queue will have.
oflag	A set of flags that includes O_CREAT and may include O_EXCL.
mode	The access permissions the queue will have.
mq_attr	Either NULL or the address of an *mq_attr* structure specifying the queue attributes of maximum message size and maximum messages.

The name of a queue has the same form as a disk filename, and in fact a queue is implemented as a file. This implementation is permitted, but not required, by the POSIX standard. Other implementations might not use it.

Once created, a queue is a persistent object that survives until removed. If you want the program to create a queue, use it, and then remove it during termination, you can call **mq_unlink()** to remove the queue.

The file can retain some queued messages when the queue is not open, so that some queued data can persist beyond the termination of the programs that use the queue. The queued data cannot be trusted after a reboot, because the data might not have been written to disk before the system came down. You should not depend on the state of the message queue after a reboot.

Opening an Existing Queue

It is more common to open an existing queue. When the program expects the queue to exist, it omits the O_CREAT flag bit. An error is returned if the queue does not exist, or if the queue exists but the effective user ID or group ID of the program does not allow access to it.

The program can specify the O_RDONLY, O_WRONLY, or O_RDWR flag to show its intended use of the queue. Access is controlled by the access permissions of the queue, just as for a file.

Specifying Blocking or Nonblocking Access

An important flag when opening a queue is the O_NONBLOCK flag. When the program specifies O_NONBLOCK, it wants an immediate return with an error code (EAGAIN) when it sends a message to a full queue or requests a message from an empty queue. When the program omits O_NONBLOCK, it specifies that it is willing to be suspended in these situations.

The O_NONBLOCK flag applies to all operations using the queue descriptor returned by **mq_open()**. (The same queue, opened under a different descriptor, can have different blocking behavior.) The blocking behavior can be changed by applying **mq_setattr()** to the queue descriptor. If the program normally wants to allow suspension, but in a particular situation wants to avoid suspension, it can apply **mq_setattr()** to change the blocking state, and then set it back again.

Using Message Queues

The POSIX functions for using an open queue are summarized in Table 6-3.

Table 6-3   POSIX Functions for Using Message Queues

Function Name	Purpose and Operation
mq_send(3)	Send a message to a queue.
mq_receive(3)	Receive a message from a queue.
mq_notify(3)	Request asynchronous notification of a message on a queue.

Sending a Message

To send a message to a queue, call **mq_send()** specifying the queue, the address and length of the message data, and an integer specifying the priority class of the message. Messages on the queue are retained in arrival sequence within priority classes.

The message is copied out of the caller's buffer, so the buffer can be reused immediately after a successful send. The **mq_send()** function blocks if the queue is full, unless the O_NONBLOCK attribute is in effect for the queue.

Receiving a Message

To receive a message, call **mq_receive()** specifying the queue, the address and size of a buffer, and the address of an integer to receive the message's priority. The size of the buffer must be at least as large as the maximum size allowed by that queue. You can learn this size using **mq_getattr()** (see Example 6-4 for an example of this).

The **mq_receive()** function blocks if the queue is empty, unless O_NONBLOCK is in effect for the queue. The message returned is always the oldest message in the highest priority class.

Using Asynchronous Notification

Some applications are designed so that each process or thread does nothing but process messages. In a design of this kind, it makes sense for a process or thread to suspend itself when no messages are available on its queue.

Other applications are designed so that one process or thread performs multiple tasks besides handling messages, or handles messages from multiple queues. In this kind of program, a process cannot suspend itself on a single message queue. Instead, it needs to do other work and only request a message when a message is available. One way to do this is to set the O_NONBLOCK flag, and to periodically poll for a message by calling **mq_receive()** and testing its return code. However, this is inefficient.

The POSIX message facility offers the ability to receive an asynchronous notification in the event that a message is posted to an empty queue and no process or thread is suspended waiting for that message. You do this by calling **mq_notify()** passing a queue and a *sigevent_t* structure. (The *sigevent_t* is declared in *sys/signal.h*, which is included by *mqueue.h*.)

The *sigevent_t* structure allows you to specify either a signal or a callback function. However, only the signal notification (SIGEV_SIGNAL) request is supported by the POSIX message queue implementation.

Example Programs

The following programs demonstrate the use of POSIX message queues:

- Example 6-1 on page 146 demonstrates the use of **mq_getattr()** to query the attributes of a queue.

- Example 6-2 on page 147 demonstrates the use of **mq_open()** to create or access a message queue.

- Example 6-3 on page 149 demonstrates the use of **mq_send()** to put messages onto a message queue.

- Example 6-4 on page 151 demonstrates the use of **mq_receive()** to take messages from a message queue.

The four example programs have a consistent design and use consistent command-line arguments. Each accepts optional arguments that allow you to exercise most features of each function, including most error return codes. The following is a simple example of use. First, a queue is created:

```
$ mq_open -p 0664 -b 128 -m 32 -c -x /var/tmp/Q32x128
flags: 0x0  maxmsg: 32  msgsize: 128  curmsgs: 0
```

An attempt is made to send a message that is larger than the queue maximum size:

```
$ mq_send -b 129 /var/tmp/Q32x128
mq_send(): Inappropriate message buffer length
```

A message of appropriate size is sent. Its presence on the queue is verified using **mq_getattr()**:

```
$ mq_send -b 128 -p 7 /var/tmp/Q32x128
$ mq_attr /var/tmp/Q32x128
flags: 0x0  maxmsg: 32  msgsize: 128  curmsgs: 1
```

An attempt is made to send a message with an illegal priority (32 is the highest allowed):

```
$ mq_send -p 99 /var/tmp/Q32x128
mq_send(): Invalid argument
```

A message is sent with a valid priority:

```
$ mq_send -p 19 /var/tmp/Q32x128
$ mq_attr /var/tmp/Q32x128
flags: 0x0  maxmsg: 32  msgsize: 128  curmsgs: 2
```

The two messages are received. The one with higher priority is received first:

```
$ mq_receive -c 2 /var/tmp/Q32x128
1: priority 19  len 63 text 00001 Fri Jun 14 09:19:12 1996
2: priority 7  len 128 text 00001 Fri Jun 14 09:17:15 1996
```

Another message is requested. Since the O_NONBLOCK flag is used, the absence of any message is reported as an error code, rather than suspending the process:

```
$ mq_receive -n /var/tmp/Q32x128
mq_receive(): Resource temporarily unavailable
```

Example of mq_getattr()

The program *mq_attr* in Example 6-1 uses **mq_getattr()** to get and display the queue attributes. Only one command-line argument is accepted:

path   The file pathname of the queue must be given following all options.

Example 6-1   Program to Demonstrate mq_getattr() and mq_setattr()

```
/*
|| Program to test mq_getattr(3), displaying queue information.
||    mq_attr <path>
||    <path>   pathname of the queue, which must exist
*/
#include <mqueue.h>      /* message queue stuff */
#include <errno.h>       /* errno and perror */
#include <fcntl.h>       /* O_RDONLY */
#include <stdio.h>
int main(int argc, char **argv)
{
   mqd_t mqd;              /* queue descriptor */
   struct mq_attr obuf; /* output attr struct for getattr */
   if (argc < 2)
   {
      printf("A pathname of a message queue is required\n");
      return -1;
   }
   mqd = mq_open(argv[1],O_RDONLY);
   if (-1 != mqd)
   {
      if ( ! mq_getattr(mqd,&obuf) )
      {
         printf("flags: 0x%x  maxmsg: %d  msgsize: %d  curmsgs: %d\n",
         obuf.mq_flags, obuf.mq_maxmsg, obuf.mq_msgsize, obuf.mq_curmsgs);
      }
      else
         perror("mq_getattr()");
   }
   else
      perror("mq_open()");
}
```

Example of mq_open()

The program *mq_open* in Example 6-2 allows you to create a message queue from the command line. The following command-line arguments are supported:

path   The file pathname of the queue must be given, following all options.

-p *perms*   Access permissions to set, for example, *-p 0664*.

-b *bytes*   The maximum message size this queue allows, for example, *-b 256*.

-m *msgs*   The maximum number of messages that can be pending on this queue, for example, *-m 64*.

-c   Use the O_CREAT flag to create the queue if it doesn't exist.

-x   Use the O_EXCL flag to require that the queue not exist.

Example 6-2   Program to Demonstrate mq_open()

```
/*
|| Program to test mq_open(3).
||    mq_open  [-p <perms>] [-b <bytes>] [-m <msgs>] [-c] [-x] <path>
||    -p <perms>  access mode to use when creating, default 0600
||    -b <bytes>  maximum message size to set, default MQ_DEF_MSGSIZE
||    -m <msgs>   maximum messages on the queue, default MQ_DEF_MAXMSG
||    -f <flags>  flags to use with mq_open, including:
||       c         use O_CREAT
||       x         use O_EXCL
||    <path>       the pathname of the queue, required
|| Numeric arguments can be given in any form supported by strtoul(3).
*/
#include <mqueue.h>     /* message queue stuff */
#define MQ_DEF_MSGSIZE 1024
#define MQ_DEF_MAXMSG 16
#include <unistd.h>     /* for getopt() */
#include <errno.h>      /* errno and perror */
#include <fcntl.h>      /* O_flags */
#include <stdio.h>
int main(int argc, char **argv)
{
   int perms = 0600;           /* permissions */
   int oflags = O_RDWR;        /* flags: O_CREAT + O_EXCL */
   int rd=0, wr=0;             /* -r and -w options */
   mqd_t mqd;                  /* returned msg queue descriptor */
   int c;
   char *path;                 /* ->first non-option argument */
```

```
struct mq_attr buf;         /* buffer for stat info */
buf.mq_msgsize = MQ_DEF_MSGSIZE;
buf.mq_maxmsg = MQ_DEF_MAXMSG;
while ( -1 != (c = getopt(argc,argv,"p:b:m:cx")) )
{
   switch (c)
   {
   case 'p': /* permissions */
      perms = (int) strtoul(optarg, NULL, 0);
      break;
   case 'b': /* message size */
      buf.mq_msgsize = (int) strtoul(optarg, NULL, 0);
      break;
   case 'm': /* max messages */
      buf.mq_maxmsg = (int) strtoul(optarg, NULL, 0);
      break;
   case 'c': /* use O_CREAT */
      oflags |= O_CREAT;
      break;
   case 'x': /* use O_EXCL */
      oflags |= O_EXCL;
      break;
   default: /* unknown or missing argument */
      return -1;
   } /* switch */
} /* while */
if (optind < argc)
   path = argv[optind]; /* first non-option argument */
else
   { printf("Queue pathname required\n"); return -1; }
mqd = mq_open(path,oflags,perms,&buf);
if (-1 != mqd)
{
   if ( ! mq_getattr(mqd,&buf) )
   {
      printf("flags: 0x%x  maxmsg: %d  msgsize: %d  curmsgs: %d\n",
      buf.mq_flags, buf.mq_maxmsg, buf.mq_msgsize, buf.mq_curmsgs);
   }
   else
      perror("mq_getattr()");
}
else
   perror("mq_open()");
}
```

148

Example of mq_send()

The *mq_send* program in Example 6-3 allows you to send from 1 to 9999 messages to a queue from the command line. The following command line arguments are accepted:

path　　　　The file pathname of the queue must be given following all options.

-b *bytes*　　Size of each message, for example *-b 0x200*.

-c *count*　　Number of messages to send. The default is 1.

-p *priority*　Numeric priority of message to send. Numbers from 0 to 32 are allowed by **mq_send()**.

-n　　　　　Use the O_NONBLOCK flag with **mq_open()**.

The *count* argument is limited to 99,999 so that the message text will not exceed 32 bytes, the (arbitrary) minimum message size the program defines.

Example 6-3　　　Program to Demonstrate mq_send()

```
/*
|| Program to test mq_send(3)
||    mq_send [-p <priority>] [-b <bytes>] [-c <count>] [-n] <path>
||       -p <priority>  priority code to use, default 0
||       -b <bytes>     size of the message, default 64, min 32
||       -c <count>     number of messages to send, default 1, max 9999
||       -n             use O_NONBLOCK flag in open
||       <path>         path to queue, required
|| The program sends <count> messages of <bytes> each at <priority>.
|| Each message is an ASCII string containing the time and date and
|| a serial number 1..<count>. The minimum message is 32 bytes.
*/
#include <mqueue.h>          /* message queue stuff */
#include <unistd.h>          /* for getopt() */
#include <errno.h>           /* errno and perror */
#include <time.h>            /* time(2) and ctime_r(3) */
#include <fcntl.h>           /* O_WRONLY */
#include <stdlib.h>          /* calloc(3) */
#include <stdio.h>
int main(int argc, char **argv)
```

```
{
   char *path;              /* -> first non-option argument */
   int oflags = O_WRONLY;  /* open flags, O_NONBLOCK may be added */
   mqd_t mqd;              /* queue descriptor from mq_open */
   unsigned int msg_prio = 0; /* message priority to use */
   size_t msglen = 64;     /* message size */
   int count = 1;          /* number of messages to send */
   char *msgptr;           /* -> allocated message space */
   int c;
   while ( -1 != (c = getopt(argc,argv,"p:b:c:n")) )
   {
      switch (c)
      {
      case 'p': /* priority */
         msg_prio = strtoul(optarg, NULL, 0);
         break;
      case 'b': /* bytes */
         msglen = strtoul(optarg, NULL, 0);
         if (msglen<32) msglen = 32;
         break;
      case 'c': /* count */
         count = strtoul(optarg, NULL, 0);
         if (count > 99999) count = 99999;
         break;
      case 'n': /* use nonblock */
         oflags |= O_NONBLOCK;
         break;
      default: /* unknown or missing argument */
         return -1;
      }
   }
   if (optind < argc)
      path = argv[optind]; /* first non-option argument */
   else
      { printf("Queue pathname required\n"); return -1; }
   msgptr = calloc(1,msglen);
   mqd = mq_open(path,oflags);
   if (-1 != mqd)
   {
      char stime[26];
      const time_t tm = time(NULL); /* current time value */
      (void)ctime_r (&tm,stime);    /* formatted time string */
      stime[24] = '\0' ;            /* drop annoying \n */
      for( c=1; c<=count; ++c)
```

```
        {
            sprintf(msgptr,"%05d %s",c,stime);
            if ( mq_send(mqd,msgptr,msglen,msg_prio) )
            {
                perror("mq_send()");
                break;
            }
        }
    }
    else
        perror("mq_open(O_WRONLY)");
}
```

Example of mq_receive()

The *mq_receive* program in Example 6-4 allows you to receive and display messages from a queue. These command-line arguments are accepted:

path   The file pathname of the queue must be given following all options.

-c *count*   Number of messages to send. The default is 1.

-q   Tells program not to display a line for each message received.

-n   Use the O_NONBLOCK flag with **mq_open()**.

You can use the *-q* option to keep the program from displaying messages. Do this when receiving a large number of messages, for example, to test performance.

Example 6-4   Program to Demonstrate mq_receive()

```
/*
|| Program to test mq_receive(3)
||    mq_receive [-c <count>] [-n] [-q] <path>
||       -c <count>     number of messages to request, default 1
||       -n             use O_NONBLOCK flag on open
||       -q             quiet, do not display messages
||       <path>         path to message queue, required
|| The program calls mq_receive <count> times or until an error occurs.
*/
#include <mqueue.h>         /* message queue stuff */
#include <unistd.h>         /* for getopt() */
#include <errno.h>          /* errno and perror */
#include <fcntl.h>          /* O_RDONLY */
#include <stdlib.h>         /* calloc(3) */
#include <stdio.h>
```

```
int main(int argc, char **argv)
{
   char *path;              /* -> first non-option argument */
   int oflags = O_RDONLY;   /* open flags, O_NONBLOCK may be added */
   int quiet = 0;           /* -q option */
   int count = 1;           /* number of messages to request */
   mqd_t mqd;               /* queue descriptor from mq_open */
   char *msgptr;            /* -> allocated message space */
   unsigned int msg_prio;   /* received message priority */
   int c, ret;
   struct mq_attr obuf;     /* output of mq_getattr(): mq_msgsize */
   while ( -1 != (c = getopt(argc,argv,"c:nq")) )
   {
      switch (c)
      {
      case 'c': /* count */
         count = strtoul(optarg, NULL, 0);
         break;
      case 'q': /* quiet */
         quiet = 1;
         break;
      case 'n': /* nonblock */
         oflags |= O_NONBLOCK;
         break;
      default: /* unknown or missing argument */
         return -1;
      }
   }
   if (optind < argc)
      path = argv[optind]; /* first non-option argument */
   else
      { printf("Queue pathname required\n"); return -1; }
   mqd = mq_open(path,oflags);
   if (-1 != mqd)
   {
      if (! (mq_getattr(mqd,&obuf)) ) /* get max message size */
      {
         msgptr = calloc(1,obuf.mq_msgsize);
         for( c=1; c<=count; ++c)
         {
            ret = mq_receive(mqd,msgptr,obuf.mq_msgsize,&msg_prio);
            if (ret >= 0) /* got a message */
            {
               if (!quiet)
```

```
                    {
                        if ( isascii(*msgptr) )
                            printf("%d: priority %ld  len %d text %-32.32s\n",
                                    c, msg_prio,       ret,       msgptr);
                        else
                            printf("%d: priority %ld  len %d (nonascii)\n",
                                    c,  msg_prio,        ret);
                    }
                }
                else /* an error on receive, stop */
                {
                    perror("mq_receive()");
                    break;
                }
            } /* for c <= count */
        } /* if getattr */
        else
        {
            perror("mq_getattr()");
            return -1;
        }
    } /* if open */
    else
        perror("mq_open(O_WRONLY)");
}
```

System V Message Queues

IRIX contains an implementation of message queues compatible with UNIX System V
Release 4 (SVR4). These message queue functions are demonstrated in example
programs in this section.

Managing SVR4 Message Queues

The functions used to create and control SVR4 message queues are summarized in Table 6-4.

Table 6-4   SVR4 Functions for Managing Message Queues

Function Name	Purpose and Operation
msgget(2)	Create a message queue if it does not exist, and gain access to it.
msgctl(2)	Query the status of a queue, change its owner ID or access permissions, or remove it from the system.

Unlike a POSIX message queue, whose name is also a filename, the external name of an SVR4 message queue is an integer held in an IPC name table (see "SVR4 IPC Name Space" on page 50). You specify this key when creating the message queue, and again whenever you access it for use.

Creating a Message Queue

The **msgget()** function has two purposes. It is used to gain access to a queue that exists, and it can create a queue that does not exist. To create a new queue, call **msgget()** with the following arguments:

key   An integer key that is not defined at this time.

msgflag   A set of flags that includes IPC_CREAT and may include IPC_EXCL. This value also contains the access permission bits.

For example, a call to create a queue might be written as follows:

```
ret = msgget(PROJ_KEY,IPC_CREAT+IPC_EXCL+0660);
```

This example relies on a constant PROJ_KEY to supply the key. Another option is to use the **ftok()** library function (see the ftok(3C) reference page).

Accessing an Existing Queue

When the program expects the queue to exist, it calls **msgget()** passing the expected key value and omitting the IPC_CREAT flag. If the queue does not exist, or if the effective user and group ID of the process are not allowed access to the queue, an error is returned. The program receives read-only or read-write access depending on the access permissions of the queue, just as with a file.

Modifying a Message Queue

You can use **msgctl()** to modify four attributes of a queue after creating or accessing it:

- the user ID and group ID that owns the queue

- the access permissions

- the limit on the total size of all queued messages

The size limit on a new queue is set to the system limit (32,768 bytes as of IRIX 6.2). This determines how many messages can be waiting, unreceived, on the queue. That in turn determines how far the message-sending process can get ahead of the message-reading process. You can lower the limit to limit the sending process or thread more closely to the speed of the receiving process or thread.

Removing a Message Queue

You can remove a message queue using the *ipcrm* command (see the ipcrm(1) reference page), or by calling **msgctl()** and passing the IPC_RMID command code. In many cases, a message queue is meant for use within the scope of one program only, and you do not want the queue to persist after the termination of that program. Call **msgctl()** to remove the queue as part of termination.

Using SVR4 Message Queues

The SVR4 functions for using message queues are summarized in Table 6-5.

Table 6-5   SVR4 Functions for Using Message Queues

Function Name	Purpose and Operation
msgsnd(2)	Send a message to a queue.
msgrcv(2)	Receive a message from a queue.

Sending a Message

To send a message to a queue, call **msgsnd()** and specify the queue, the address and
length of the message data, and a flag number that can contain IPC_NOWAIT. The
message buffer contains an integer specifying the "type" of the message. Messages on the
queue are retained in arrival sequence within types.

The message is copied out of the caller's buffer, so the buffer can be reused immediately
after a successful send. If the queue is full, the **msgsnd()** function blocks unless the
IPC_NOWAIT flag is passed.

Receiving a Message

To receive a message, call **msgrcv()** and specify the queue, the address and size of a
buffer, a number for the desired message type, and a flag value. If the queue is empty, the
msgrcv() function blocks unless the IPC_NOWAIT flag is passed. If the message buffer is
not as large as the message, an error is returned unless the IPC_NOERROR flag is passed.
Then the message is simply truncated to fit the buffer.

The type value can be 0, to specify "any type," or it can be a specific (positive) type
number to select the first number of that type. Finally, it can be a negative value to specify
"any type less than or equal."

Example Programs

The following programs demonstrate the use of SVR4 message queues:

- Example 6-5 on page 159 demonstrates the use of **msgget()** to create or access a queue.

- Example 6-6 on page 161 demonstrates the use of **msgctl()** to query or modify a queue.

- Example 6-7 on page 163 demonstrates the use of **msgsnd()** to put messages onto a queue.

- Example 6-8 on page 166 demonstrates the use of **msgrcv()** to take messages from a queue.

The four example programs have a consistent design and use consistent command-line argument letters. Each accepts optional arguments that allow you to exercise all the features of one function, including most error return codes. The following is a simple example of use. First, *ipcs* is used to show no queues exist.

```
$ ipcs -q
IPC status from /dev/kmem as of Wed Jun 12 10:36:38 1996
T    ID    KEY        MODE      OWNER    GROUP
Message Queues:
```

Then a queue is created with key 9 and *ipcs* used to verify the operation.

```
$ msgget -k 9 -c
msqid = 0x0032. owner = 1110.20, perms = 100600, max bytes = 32768
0 msgs = 0 bytes on queue
$ ipcs -q
IPC status from /dev/kmem as of Thu Jun 20 09:32:25 1996
T    ID    KEY        MODE      OWNER    GROUP
Message Queues:
q    50 0x00000009 --rw-------  cortesi    user
```

The use of the IPC_EXCL flag is tested:

```
$ msgget -k 9 -c -x
msgget(): File exists
```

A message is sent to the queue, addressing the queue by its ID.

```
$ msgsnd -i 50 -t 17
$ msgctl -i 50
owner = 1110.20, perms = 100600, max bytes = 32768
1 msgs = 64 bytes on queue
```

The maximum queue size is changed, this time addressing the queue by its key.

```
$ msgctl -k 9 -b 1024
owner = 1110.20, perms = 100600, max bytes = 1024
1 msgs = 64 bytes on queue
```

A second message is sent:

```
$ msgsnd -i 50 -t 18
$ msgctl -i 50
owner = 1110.20, perms = 100600, max bytes = 1024
2 msgs = 128 bytes on queue
```

The first and second messages are received:

```
$ msgrcv -k 9
1: type 17  len 64 text 00001 Thu Jun 20 09:32:55 1996
$ msgrcv -i 50
1: type 18  len 64 text 00001 Thu Jun 20 09:33:18 1996
```

Another message receipt is attempted, first with IPC_NOWAIT:
```
$ msgrcv -i 50 -n
msgrcv(): No message of desired type
```

Another message is attempted without IPC_NOWAIT. While *msgrcv* is suspended, the message queue is removed.

```
$ msgrcv -k 9 &
12477
$ ipcrm -q 50
$ msgrcv(): Identifier removed
```

Example of msgget

The program *msgget* in Example 6-5 allows you to create a message queue from the command line. The following command-line arguments are supported:

-k *key*   Numeric identifier of a message queue, for example *-k 99*.

-p *perms*   Access permissions to set, for example *-p 0664*.

-x   Use the IPC_EXCL flag with **msgget()**.

-c   Use the IPC_CREAT flag with **msgget()**.

If the *-k* argument is omitted, the program uses a private key and thus creates a message queue that can be used from this program only. (This is not useful, since the program does nothing with the queue before it terminates.)

Example 6-5   Program to Demonstrate msgget()

```
/*
|| Program to test msgget(2).
||    msgget [-k <key>] [-p <perms>] [-x] [-c]
||       -k <key>    the key to use, default == 0 == IPC_PRIVATE
||       -p <perms>  permissions to use, default 600
||       -x          use IPC_EXCL
||       -c          use IPC_CREAT
*/
#include <sys/msg.h>    /* msg queue stuff, ipc.h, types.h */
#include <unistd.h>     /* for getopt() */
#include <errno.h>      /* errno and perror */
#include <stdio.h>
int main(int argc, char **argv)
{
   key_t key = IPC_PRIVATE;   /* key */
   int perms = 0600;          /* permissions */
   int msgflg = 0;            /* flags: CREAT + EXCL */
   int msqid;                 /* returned msg queue id */
   struct msqid_ds buf;       /* buffer for stat info */
   int c;
   while ( -1 != (c = getopt(argc,argv,"k:p:xc")) )
   {
      switch (c)
      {
      case 'k': /* key */
         key = (key_t) strtoul(optarg, NULL, 0);
         break;
```

```
            case 'p': /* permissions */
               perms = (int) strtoul(optarg, NULL, 0);
               break;
            case 'c':
               msgflg |= IPC_CREAT;
               break;
            case 'x':
               msgflg |= IPC_EXCL;
               break;
            default: /* unknown or missing argument */
               return -1;
         }
      }
      msqid = msgget (key, msgflg|perms);
      if (-1 != msqid)
      {
         printf("msqid = 0x%04x. ",msqid);
         if (-1 != msgctl(msqid,IPC_STAT,&buf))
         {
            printf("owner = %d.%d, perms = %04o, max bytes = %d\n",
                      buf.msg_perm.uid,
                      buf.msg_perm.gid,
                      buf.msg_perm.mode,
                      buf.msg_qbytes);
            printf("%d msgs = %d bytes on queue\n",
                      buf.msg_qnum, buf.msg_cbytes);
         }
         else
            perror("\nmsgctl()");
      }
      else
         perror("msgget()");
   }
```

Example of msgctl

The program *msgctl* in Example 6-6 allows you to display the state of a queue, or to change the permissions, owner ID, group ID, or maximum size of a queue. The following command-line arguments are supported:

-k *key*   Numeric identifier of a message queue, for example, *-k 99*.

-i *id*   Message queue ID, alternative to specifying the key; for example, *-i 80*.

-p *perms*   Access permissions to set, for example, *-p 0664*.

-b *bytes*   Maximum size of the message queue, for example, *-b 0x1000*.

-u *uid*   Numeric user ID to set as owner.

-g *gid*   Numeric group ID to set as owner.

Example 6-6   Program to Demonstrate msgctl()

```
/*
|| Program to test msgctl(2).
||    msgctl {-k <key> -i <id>} [-b <bytes>] [-p <perms>] [-u <uid>] [-g <gid>]
||       -k <key>    the key to use, or..
||       -i <id>      ..the mq id
||       -b <bytes>  new max number of bytes to set in msg_qbytes
||       -p <perms>  new permissions to assign in msg_perm.mode
||       -u <uid>    new user id (numeric) for msg_perm.uid
||       -g <gid>     new group id (numeric) for msg_perm.gid
*/
#include <sys/msg.h>    /* msg queue stuff, ipc.h, types.h */
#include <unistd.h>      /* for getopt() */
#include <errno.h>       /* errno and perror */
#include <stdio.h>
int main(int argc, char **argv)
{
   key_t key;           /* key for msgget.. */
   int msqid = -1;      /* ..specified or received msg queue id */
   long perms = -1L;    /* -1L is not valid for any of these */
   long bytes = -1L;
   long uid = -1L;
   long gid = -1L;
   struct msqid_ds buf;
   int c;
   while ( -1 != (c = getopt(argc,argv,"k:i:b:p:u:g:")) )
   {
      switch (c)
```

```
                  {
                  case 'k': /* key */
                     key = (key_t) strtoul(optarg, NULL, 0);
                     break;
                  case 'i': /* id */
                     msqid = (int) strtoul(optarg, NULL, 0);
                     break;
                  case 'p': /* permissions */
                     perms = strtoul(optarg, NULL, 0);
                     break;
                  case 'b': /* bytes */
                     bytes = strtoul(optarg, NULL, 0);
                     break;
                  case 'u': /* uid */
                     uid = strtoul(optarg, NULL, 0);
                     break;
                  case 'g': /* gid */
                     gid = strtoul(optarg, NULL, 0);
                     break;
                  default: /* unknown or missing argument */
                     return -1;
                  }
               }
               if (-1 == msqid) /* no id given, try key */
                  msqid = msgget (key, 0);
               if (-1 != msqid)
               {
                  if (-1 != msgctl(msqid,IPC_STAT,&buf))
                  {
                     if ((perms!=-1L)||(bytes!=-1L)||(uid!=-1L)||(gid!=-1L))
                     {
                        /* put new values in buf fields as requested */
                        if (perms != -1L) buf.msg_perm.mode = (mode_t)perms;
                        if (uid   != -1L) buf.msg_perm.uid  = (uid_t)uid;
                        if (gid   != -1L) buf.msg_perm.gid  = (gid_t)gid;
                        if (bytes != -1L) buf.msg_qbytes = (ulong_t)bytes;
                        if (-1 == msgctl(msqid,IPC_SET,&buf))
                           perror("\nmsgctl(IPC_SET)");
                     }
                     printf("owner = %d.%d, perms = %04o, max bytes = %d\n",
                           buf.msg_perm.uid,
                           buf.msg_perm.gid,
                           buf.msg_perm.mode,
                           buf.msg_qbytes);
                     printf("%d msgs = %d bytes on queue\n",
```

```
                          buf.msg_qnum, buf.msg_cbytes);
        }
        else
            perror("\nmsgctl(IPC_STAT)");
    }
    else
        perror("msgget()");
}
```

Example of msgsnd

The *msgsnd* program in Example 6-7 allows you to send one or more messages of
specified length and type to a message queue. The following command-line arguments
are supported:

-k *key*   Numeric identifier of a message queue, for example, *-k 99*.

-i *id*   Message queue ID, alternative to specifying the key; for example, *-i 80*.

-c *count*   Number of messages to send. The default is 1.

-t *type*   Numeric type of message to send. Types less than 1 are rejected by
                msgsnd().

-b *bytes*   Size of each message, for example, *-b 0x200*.

-n   Use the IPC_NOWAIT flag with **msgsnd()**.

The program sends as many messages as you specify, each with the specified type and
size. The first 32 bytes of each message is a printable string containing a sequence number
and the date and time. The message is padded out to the specified size with binary 0.

Example 6-7   Program to Demonstrate msgsnd()

```
/*
|| Program to test msgsnd(2)
||    msgsnd {-k <key> -i <id>} [-t <type>] [-b <bytes>] [-c <count>] [-n]
||        -k <key>    the key to use, or..
||        -i <id>     ..the mq id
||        -t <type>   the type of each message, default = 1
||        -b <bytes>  the size of each message, default = 64, min 32
||        -c <count>  the number of messages to send, default = 1, max 99999
||        -n          use IPC_NOWAIT flag
|| The program sends <count> messages of <type>, <bytes> each on the queue.
|| Each message is an ASCII string containing the time and date, and
|| a serial number 1..<count>, minimum message is 32 bytes.
```

```
*/
#include <sys/msg.h>    /* msg queue stuff, ipc.h, types.h */
#include <unistd.h>     /* for getopt() */
#include <errno.h>      /* errno and perror */
#include <time.h>       /* time(2) and ctime_r(3) */
#include <stdio.h>
int main(int argc, char **argv)
{
   key_t key;              /* key for msgget.. */
   int msqid = -1;         /* ..specified or received msg queue id */
   int msgflg = 0;         /* flag, 0 or IPC_NOWAIT */
   long type = 1;          /* message type -- 0 is not valid to msgsnd() */
   size_t bytes = 64;      /* message text size */
   int count = 1;          /* number to send */
   int c;
   struct msgspace { long type; char text[32]; } *msg;
   while ( -1 != (c = getopt(argc,argv,"k:i:t:b:c:n")) )
   {
      switch (c)
      {
      case 'k': /* key */
         key = (key_t) strtoul(optarg, NULL, 0);
         break;
      case 'i': /* id */
         msqid = (int) strtoul(optarg, NULL, 0);
         break;
      case 't': /* type */
         type = strtoul(optarg, NULL, 0);
         break;
      case 'b': /* bytes */
         bytes = strtoul(optarg, NULL, 0);
         if (bytes<32) bytes = 32;
         break;
      case 'c': /* count */
         count = strtoul(optarg, NULL, 0);
         if (count > 99999) count = 99999;
         break;
      case 'n': /* nowait */
         msgflg |= IPC_NOWAIT;
         break;
      default: /* unknown or missing argument */
         return -1;
      }
   }
   msg = (struct msgspace *)calloc(1,sizeof(long)+bytes);
```

164

```
       if (-1 == msqid) /* no id given, try key */
          msqid = msgget (key, 0);
       if (-1 != msqid)
       {
          const time_t tm = time(NULL);
          char stime[26];
          (void)ctime_r (&tm,stime); /* format timestamp for msg */
          stime[24] = '\0';          /* drop annoying \n */
          for( c=1; c<=count; ++c)
          {
             msg->type = type;
             sprintf(msg->text,"%05d %s",c,stime);
             if (-1 == msgsnd(msqid,msg,bytes,msgflg))
             {
                perror("msgsnd()");
                break;
             }
          }
       }
       else
          perror("msgget()");
    }
```

Example of msgrcv

The program *msgrcv* in Example 6-8 allows you to receive messages from a specified
queue. The following arguments are used in more than one program:

-k *key*   Numeric identifier of a message queue, for example *-k 99*.

-i *id*   Message queue ID, alternative to specifying the key; for example, *-i 80*.

-c *count*   Number of messages to attempt to receive.

-b *bytes*   Maximum size of a message, for example, *-b 0x200*.

-n   Use the IPC_NOWAIT flag with **msgrcv()**.

-e   Use the MSG_NOERROR flag with **msgrcv()**, to truncate messages longer
than *bytes*.

-q   Be quiet, do not display the received message. Use for performance testing.

As each message is received, it is displayed. A sequence number and the message type
are always displayed; the first 32 bytes of the text are displayed if it begins with ASCII.

Example 6-8   Program to Demonstrate msgrcv()

```
/*
|| Program to test msgrcv(2)
||    msgrcv {-k <key> -i <id>} [-t <type>] [-b <bytes>] [-c <count>]
||                                          [-n] [-e] [-q]
||       -k <key>    the key to use, or..
||       -i <id>     ..the mq id
||       -t <type>   the type of message, default = 0 (any msg)
||       -b <bytes>  the max size to receive, default = 64
||       -c <count>  the number of messages to receive, default = 1
||       -n          use IPC_NOWAIT flag
||       -e          use MSG_NOERROR flag (truncate long msg)
||       -q          quiet, do not display received message
|| The program calls msgrcv <count> times or until an error occurs,
|| each time requesting a message of type <type> and max size <bytes>.
*/
#include <sys/msg.h>    /* msg queue stuff, ipc.h, types.h */
#include <unistd.h>     /* for getopt() */
#include <errno.h>      /* errno and perror */
#include <ctype.h>      /* isascii() */
#include <stdio.h>
int main(int argc, char **argv)
```

```
{
   key_t key;              /* key for msgget.. */
   int msqid = -1;         /* ..specified or received msg queue id */
   int msgflg = 0;         /* flag, 0, IPC_NOWAIT, MSG_NOERROR */
   long type = 0;          /* message type */
   size_t bytes = 64;      /* message size limit */
   int count = 1;          /* number to receive */
   int quiet = 0;          /* quiet flag */
   int c;
   struct msgspace { long type; char text[32]; } *msg;
   while ( -1 != (c = getopt(argc,argv,"k:i:t:b:c:enq")) )
   {
      switch (c)
      {
      case 'k': /* key */
         key = (key_t) strtoul(optarg, NULL, 0);
         break;
      case 'i': /* id */
         msqid = (int) strtoul(optarg, NULL, 0);
         break;
      case 't': /* type -- can be negative */
         type = strtol(optarg, NULL, 0);
         break;
      case 'b': /* bytes -- no minimum */
         bytes = strtoul(optarg, NULL, 0);
         break;
      case 'c': /* count - no maximum */
         count = strtoul(optarg, NULL, 0);
         break;
      case 'n': /* nowait */
         msgflg |= IPC_NOWAIT;
         break;
      case 'e': /* noerror -- allow truncation of msgs */
         msgflg |= MSG_NOERROR;
         break;
      case 'q': /* quiet */
         quiet = 1;
         break;
      default: /* unknown or missing argument */
         return -1;
      }
   }
```

```
                 if (-1 == msqid) /* no id given, try key */
                    msqid = msgget (key, 0);
                 msg = (struct msgspace *)calloc(1,sizeof(long)+bytes);
                 if (-1 != msqid)
                 {
                    for( c=1; c<=count; ++c)
                    {
                       int ret = msgrcv(msqid,msg,bytes,type,msgflg);
                       if (ret >= 0) /* got a message */
                       {
                          if (!quiet)
                          {
                             if (isascii(msg->text[0]))
                                printf("%d: type %ld  len %d text %-32.32s\n",
                                          c,   msg->type, ret,      msg->text);
                             else
                                printf("%d: type %ld len %d (nonascii)\n",
                                          c,   msg->type,  ret);
                          }
                       }
                       else /* an error, end loop */
                       {
                          perror("msgrcv()");
                          break;
                       }
                    } /* for c<=count */
                 } /* good msgget */
                 else
                    perror("msgget()");
              }
```

Advanced File Control

Chapter 7, "File and Record Locking"

Describes the different ways of locking files or records within files for exclusive use between processes and systems.

Chapter 8, "Using Asynchronous I/O"

Describes how to schedule file I/O asynchronously, in a parallel thread.

Chapter 9, "High-Performance File I/O"

Describes the use of direct-to-disk file I/O, and guaranteed-rate I/O.

File and Record Locking

IRIX supports the ability to place a lock upon an entire file or upon a range of bytes within a file. Programs must cooperate in respecting record locks. A file lock can be made mandatory but only at a cost in performance. For these reasons, file and record locking should normally be seen as a synchronization mechanism, not a security mechanism.

The chapter includes these topics:

- "Overview of File and Record Locking" presents an introduction to locking mechanisms.

- "Controlling File Access With File Permissions" discusses the relationship of file permissions to exclusive file access.

- "Using Record Locking" discusses the use of file and record locks to get exclusive data access.

- "Enforcing Mandatory Locking" describes how file locks can be made mandatory on programs that do not use locking.

- "Record Locking Across Multiple Systems" discusses how file locking can be extended to NFS-mounted files.

Overview of File and Record Locking

Simultaneous access to file data is characteristic of many multiprocess, multithreaded, or real-time applications.The purpose of the file and record locking facility is to provide a way for programs to synchronize their use of common file data.

Advisory file and record locking can be used to coordinate independent, unrelated processes. In mandatory locking, on the other hand, the standard I/O subroutines and I/O system calls enforce the locking protocol. Mandatory locking keeps unrelated programs from accessing data out of sequence, at some cost of access speed.

The system functions used in file and record locking are summarized in Table 7-1.

Table 7-1   Functions for File and Record Locking

Function Name	Purpose and Operation
fcntl(2), fcntl(5)	General function for modifying an open file descriptor; can be used to set file and record locks.
lockf(3C), lockf(3F)	Library function to set and remove file and record locks on open files (SVR4 compatible).
flock(3B)	Library function to set and remove file and record locks on open files (BSD compatible).
chmod(1), chmod(2)	Command and system function that can enable mandatory file locking on a specified file.

Terminology

The discussion of file and record locking depends on the terms defined in this section.

Record

A record is any contiguous sequence of bytes in a file. The UNIX operating system does not impose any record structure on files. The boundaries of records are defined by the programs that use the files. Within a single file, a record as defined by one process can overlap partially or completely on a record as defined by some other process.

Read (Shared) Lock

A read lock keeps a record from changing while one or more processes read the data. If a process holds a read lock, it may assume that no other process can alter that record at the same time. A read lock is also a shared lock because more than one process can place a read lock on the same record or on a record that overlaps a read-locked record. No process, however, can have a write lock that overlaps a read lock.

Write (Exclusive) Lock

A write lock is used to gain complete control over a record. A write lock is an exclusive lock because, when a write lock is in place on a record, no other process may read- or write-lock that record or any data that overlaps it. If a process holds a write lock it can assume that no other process will read or write that record at the same time.

Advisory Locking

An advisory lock is visible only when a program explicitly tries to place a conflicting lock. An advisory lock is not visible to the file I/O system functions such as **read()** and **write()**. A process that does not test for an advisory lock can violate the terms of the lock, for example, by writing into a locked record.

Advisory locks are useful when all processes make an appropriate record lock request before performing any I/O operation. When all processes use advisory locking, access to the locked data is controlled by the advisory lock requests. The success of advisory locking depends on the cooperation of all processes in enforcing the locking protocol; it is not enforced by the file I/O subsystem.

Mandatory Locking

Mandatory record locking is enforced by the file I/O system functions, and so is effective on unrelated processes that are not part of a cooperating group. Respect for locked records is enforced by the **creat()**, **open()**, **read()**, and **write()** system calls. When a record is locked, access to that record by any other process is restricted according to the type of lock on the record. Cooperating processes should still request an appropriate record lock before an I/O operation, but an additional check is made by IRIX before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers security against unplanned file use by unrelated programs, but it imposes additional system overhead on access to the controlled files.

173

Lock Promotion and Demotion

A read lock can be promoted to write-lock status if no other process is holding a read lock in the same record. If processes with pending write locks are waiting for the same record, the lock promotion succeeds and the other (sleeping) processes wait. Demoting a write lock to a read lock can be done at any time.

Because the **lockf()** function does not support read locks, lock promotion is not applicable to locks set with that call. >

Controlling File Access With File Permissions

The access permissions for each UNIX file control which users can read, write, or execute the file. These access permissions may be set only by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the access permissions for a file. Note that if the permissions for a directory allow anyone to write in the directory, and the "sticky bit" is not included in the permissions, files within that directory can be removed even by a user who does not have read, write, or execute permission for those files.

If your application warrants the use of record locking, make sure that the permissions on your files and directories are also set properly. A record lock, even a mandatory record lock, protects only the records that are locked, while they are locked. Unlocked parts of the files can be corrupted if proper precautions are not taken.

Only a known set of programs or users should be able to read or write a database. This can be enforced through file permissions as follows:

1. Using the *chown* facility (see the chown(1) and chown(2) reference pages), set the ownership of the critical directories and files to reflect the authorized group ID.

2. Using the *chmod* facility (see also the chmod(1) and chmod(2) reference pages), set the file permissions of the critical directories and files so that only members of the authorized group have write access ("775" permissions).

3. Using the *chown* facility, set the accessing program executable files to be owned by the authorized group.

4. Using the *chmod* facility, set the set-GID bit for each accessing program executable file and to permit execution by anyone ("2755" permissions).

Users who are not members of the authorized group cannot modify the critical
directories and files. However, when an ordinary user executes one of the accessing
programs, the program automatically adopts the group ID of its owner. The accessing
program can create and modify files in the critical directory, but other programs started
by an ordinary user cannot.

Using Record Locking

This section covers the following topics:

- "Opening a File for Record Locking"

- "Setting a File Lock"

- "Setting and Removing Record Locks"

- "Getting Lock Information"

- "Deadlock Handling"

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file
descriptor. If read locks are to be used, then the file must be opened with at least read
access; likewise for write locks and write access.

Example 7-1 opens a file for both read and write access.

Example 7-1   Opening a File for Locked Use

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
int fd;    /* file descriptor */
char *filename;
main(argc, argv)
int argc;
char *argv[];
```

```
{
    extern void exit(), perror();
    /* get database file name from command line and open the
     * file for read and write access.
     */
    if (argc < 2) {
        (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
        exit(2);
    }
    filename = argv[1];
    fd = open(filename, O_RDWR);
    if (fd < 0) {
        perror(filename);
        exit(2);
    }
}
```

The file is now open to perform both locking and I/O functions. The next step is to set a
lock.

Setting a File Lock

Several ways exist to set a lock on a file. These methods depend upon how the lock
interacts with the rest of the program. Issues of portability and performance need to be
considered. Three methods for setting a lock are given here: using the **fcntl()** system call;
using the */usr/group* standards-compatible **lockf()** library function; and using the BSD
compatible **flock()** library function.

Locking an entire file is just a special case of record locking—one record is locked, which
has the size of the entire file. The file is locked starting at a byte offset of zero and size of
the maximum file size. This size is beyond any real end-of-file so that no other lock can
be placed on the file.

You have a choice of three functions for this operation: the basic **fcntl()**, the library
function **lockf()**, and the BSD compatible library function **flock()**. All three functions can
interoperate. That is, a lock placed by one is respected by the other two.

Whole-File Lock With fcntl()

The **fcntl()** function treats a lock length of 0 as meaning "size of file." The function **lockWholeFile()** in Example 7-2 attempts a specified number of times to obtain a whole-file lock using **fcntl()**. When the lock is placed, it returns 0; otherwise it returns the error code for the failure.

Example 7-2   Setting a Whole-File Lock With fcntl()

```
#include <fcntl.h>
#include <errno.h>
#define MAX_TRY 10

int
lockWholeFile(int fd, int tries)
{
    int limit = (tries)?tries:MAX_TRY;
    int try;
    struct flock lck;
    lck.l_type = F_WRLCK;        /* write (exclusive) lock */
    lck.l_whence = 0;            /* 0 offset for l_start */
    lck.l_start = 0L;            /* lock starts at BOF */
    lck.l_len = 0L;              /* extent is entire file */
    for (try = 0; try < limit; ++try)
    {
        if ( 0 == fcntl(fd, F_SETLK, &lck) )
            break; /* mission accomplished */
        if ((errno != EAGAIN) && (errno != EACCES))
            break; /* mission impossible */
        sginap(1); /* let lock holder run */
    }
    return errno;
}
```

The following points should be noted in Example 7-2:

- Because **fcntl()** supports both read and write locks, the type of the lock (F_WRLCK) is specified in the *l_type*.

- The operation code F_SETLK is used to request that the function return if it cannot place the lock. The code F_SETLKW would request that the function suspend until the lock can be placed.

- The starting location of the record is the sum of two fields, *l_whence* and *l_start*. Both must be set to 0 in order to get the starting point to the beginning of the file.

Whole-File Lock With lockf()

Example 7-3 shows a version of the **lockWholeFile()** function that uses **lockf()**. Like **fcntl()**, **lockf()** treats a record length of 0 as meaning "to end of file."

Example 7-3   Setting a Whole-File Lock With lockf()

```
#include <unistd.h> /* for F_TLOCK */
#include <fcntl.h>  /* for O_RDWR */
#include <errno.h>  /* for EAGAIN */
#define MAX_TRY 10

int
lockWholeFile(int fd, int tries)
{
    int limit = (tries)?tries:MAX_TRY;
    int try;
    lseek(fd,0L,SEEK_SET);  /* set start of lock range */
    for (try = 0; try < limit; ++try)
    {
        if (0 == lockf(fd, F_TLOCK, 0L) )
            break; /* mission accomplished */
        if (errno != EAGAIN)
            break; /* mission impossible */
        sginap(1); /* let lock holder run */
    }
    return errno;
}
```

The following points should be noted about Example 7-3:

- The type of lock is not specified, because **lockf()** only supports exclusive locks.

- The operation code F_TLOCK specifies that the function should return if the lock cannot be placed. The F_LOCK operation would request that the function suspend until the lock could be placed.

- The start of the record is set implicitly by the current file position. That is why **lseek()** is called, to ensure the correct file position before **lockf()** is called.

Whole-File Lock With flock()

Example 7-4 displays a third example of the lockWholeFile subroutine, this one using **flock()**.

Example 7-4   Setting a Whole-File Lock With flock()

```
#define _BSD_COMPAT
#include <sys/file.h> /* includes fcntl.h */
#include <errno.h>  /* for EAGAIN */
#define MAX_TRY 10
int
lockWholeFile(int fd, int tries)
{
    int limit = (tries)?tries:MAX_TRY;
    int try;
    for (try = 0; try < limit; ++try)
    {
        if ( 0 == flock(fd, LOCK_EX+LOCK_NB) )
            break; /* mission accomplished */
        if (errno != EWOULDBLOCK)
            break; /* mission impossible */
        sginap(1); /* let lock holder run */
    }
    return errno;
}
```

The following points should be noted about Example 7-4:

- The compiler variable _BSD_COMPAT is defined in order to get BSD-compatible definitions from standard header files.

- The only use of **flock()** is to lock an entire file, so there is no attempt to specify the start or length of a record.

- The LOCK_NB flag requests the function to return if the lock cannot be placed. Without this flag the function suspends until the lock can be placed.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file, except that the record does not encompass the entire file contents. This section examines an example problem of dealing with two records (which may be either in the same file or in different files) that must be updated simultaneously so that other processes get a consistent view of the information they contain. This type of problem occurs, for example, when updating the inter-record pointers in a doubly linked list.

To deal with multiple locks, consider the following questions:

- What do you want to lock?

- For multiple locks, in what order do you want to lock and unlock the records?

- What do you do if you succeed in getting all the required locks?

- What do you do if you fail to get one or more locks?

In managing record locks, you must plan a failure strategy for the case in which you cannot obtain all the required locks. It is because of contention for these records that you have decided to use record locking in the first place. Different programs might

- wait a certain amount of time, and try again

- end the procedure and warn the user

- let the process sleep until signaled that the lock has been freed

- a combination of the above

Look now at the example of inserting an entry into a doubly linked list. All the following examples assume that a record is declared as follows:

```
struct record {
.../* data portion of record */...
    long prev;    /* index to previous record in the list */
    long next;    /* index to next record in the list */
};
```

For the example, assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be promoted to a write lock so that the record may be edited. Example 7-5 shows a function that can be used for this.

Example 7-5   Record Locking With Promotion Using fcntl()

```
/*
|| This function is called with a file descriptor and the
|| offsets to three records in it: this, here, and next.
|| The caller is assumed to hold read locks on both here and next.
|| This function promotes these locks to write locks.
|| If write locks on "here" and "next" are obtained
||    Set a write lock on "this".
||    Return index to "this" record.
|| If any write lock is not obtained:
||    Restore read locks on "here" and "next".
||    Remove all other locks.
```

```
||    Return -1.
*/
long set3Locks(int fd, long this, long here, long next)
{
    struct flock lck;
    lck.l_type = F_WRLCK;    /* setting a write lock */
    lck.l_whence = 0;        /* offsets are absolute */
    lck.l_len = sizeof(struct record);
    /* Promote the lock on "here" to write lock */
    lck.l_start = here;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        return (-1);
    }
    /* Lock "this" with write lock */
    lck.l_start = this;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Failed to lock "this"; return "here" to read lock. */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLKW, &lck);
        return (-1);
    }
    /* Promote lock on "next" to write lock */
    lck.l_start = next;
    if (fcntl(fd, F_SETLKW, &lck) < 0) {
        /* Failed to promote "next"; return "here" to read lock... */
        lck.l_type = F_RDLCK;
        lck.l_start = here;
        (void) fcntl(fd, F_SETLK, &lck);
        /* ...and remove lock on "this".  */
        lck.l_type = F_UNLCK;
        lck.l_start = this;
        (void) fcntl(fd, F_SETLK, &lck);
        return (-1)
    }
    return (this);
}
```

Example 7-5 uses the F_SETLKW command to **fcntl()**, with the result that the calling process will sleep if there are conflicting locks at any of the three points. If the F_SETLK command was used instead, the **fcntl()** system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections (as in Example 7-2).

It is possible to unlock or change the type of lock on a subsection of a previously set lock; this may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Example 7-6 shows a similar example using the **lockf()** function. Since it does not support read locks, all (write) locks are referenced generically as locks.

Example 7-6   Record Locking Using lockf()

```
/*
|| This function is called with a file descriptor and the
|| offsets to three records in it: this, here, and next.
|| The caller is assumed to hold no locks on any of the records.
|| This function tries to lock "here" and "next" using lockf().
|| If locks on "here" and "next" are obtained
||    Set a lock on "this".
||    Return index to "this" record.
|| If any lock is not obtained:
||    Remove all other locks.
||    Return -1.
*/
long set3Locks(int fd, long this, long here, long next)
{
    /* Set a lock on "here" */
    (void) lseek(fd, here, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        return (-1);
    }
    /* Lock "this" */
    (void) lseek(fd, this, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Failed to lock "this"; clear "here" lock. */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1);
    }
    /* Lock "next" */
    (void) lseek(fd, next, 0);
    if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
        /* Failed to lock "next"; release "here"... */
        (void) lseek(fd, here, 0);
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        /* ...and remove lock on "this".  */
        (void) lseek(fd, this, 0);
```

```
        (void) lockf(fd, F_ULOCK, sizeof(struct record));
        return (-1)
    }
    return (this);
}
```

Locks are removed in the same manner as they are set; only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process. An unlock can affect only locks that were placed by the unlocking process.

Getting Lock Information

You can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. To find this information, set up a lock as in the previous examples and use the F_GETLK command in the **fcntl()** call. If the lock passed to **fcntl()** would be blocked, the first blocking lock is returned to the process through the structure passed to **fcntl()**. That is, the lock data passed to **fcntl()** is overwritten by blocking lock information.

The returned information includes two pieces of data, *l_pidf* and *l_sysid*, that are used only with F_GETLK. These fields uniquely identify the process holding the lock. (For systems that do not support a distributed architecture, the value in *l_sysid* can be ignored.)

If a lock passed to **fcntl()** using the F_GETLK command is not blocked by another lock, the *l_type* field is changed to F_UNLCK and the remaining fields in the structure are unaffected.

Example 7-7 shows how to use this capability to print all the records locked by other processes. Note that if several read locks occur over the same record, only one of these is found.

Example 7-7   Detecting Contending Locks Using fcntl()

```
/*
|| This function takes a file descriptor and prints a report showing
|| all locks currently set on that file. The loop variable is the
|| l_start field of the flock structure. The function asks fcntl()
|| for the first lock that would block a lock from l_start to the end
|| of the file (l_len==0). When no lock would block such a lock,
|| the returned l_type contains F_UNLCK and the loop ends.
|| Otherwise the contending lock is displayed, l_start is set to
|| the end-point of that lock, and the loop repeats.
*/
void printAllLocksOn(int fd)
{
    struct flock lck;
    /* Find and print "write lock" blocked segments of file. */
    (void) printf("sysid pid type start length\n");
    lck.l_whence = 0;
    lck.l_start = 0L;
    lck.l_len = 0L;
    for( lck.l_type = 0; lck.l_type != F_UNLCK; )
    {
        lck.l_type = F_WRLCK;
        (void) fcntl(fd, F_GETLK, &lck);
        if (lck.l_type != F_UNLCK)
        {
            (void) printf("%5d %5d %c %8d %8d\n",
                          lck.l_sysid,
                          lck.l_pid,
                          (lck.l_type == F_WRLCK) ? 'W' : 'R',
                          lck.l_start,
                          lck.l_len);
            if (lck.l_len == 0)
                break; /* this lock goes to end of file, stop */
            lck.l_start += lck.l_len;
        }
    }
}
```

fcntl() with the F_GETLK command always returns correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The **lockf()** function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. Example 7-8 shows a code fragment that uses **lockf()** to test for a lock on a file.

Example 7-8   Testing for Contending Lock Using lockf()

```
/* find a blocked record. */
/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero
 * to test until the end of the file address space.
 */
if (lockf(fd, F_TEST, 0L) < 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: unknown error <%d>\n", errno);
        break;
    }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent seeks to a point in the file, the child's file pointer is also set to that location. Similarly, when a share group of processes is created using **sproc()**, and the **sproc()** flag PR_SFDS is used to keep the open-file table synchronized for all processes (see the sproc(2) reference page), then there is a single file pointer for each file and it is shared by every process in the share group.

This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, in **lockf()** at all times and in **fcntl()** when using an *l_whence* value of 1. Since there is no way to perform the sequence *lseek(); fcntl();* as an atomic operation, there is an obvious potential for race conditions—a lock might be set using a file pointer that was just changed by another process.

The solution is to have the child process close and reopen the file. This creates a distinct file descriptor for the use of that process. Another solution is to always use the **fcntl()** function for locking with an *l_whence* value of 0 or 2. This makes the locking function independent of the file pointer (processes might still contend for the use of the file pointer for other purposes such as direct-access input).

Deadlock Handling

A certain level of deadlock detection and avoidance is built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard **lockf()** call. This deadlock detection is valid only for processes that are locking files or records on a single system.

Deadlocks can potentially occur only when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call fails and sets *errno* to the deadlock error number.

If a process wishes to avoid using the system's deadlock detection, it should set its locks using F_GETLK instead of F_GETLKW.

Enforcing Mandatory Locking

File locking is usually an in-memory service of the IRIX kernel. The kernel keeps a table of locks that have been placed. Processes anywhere in the system update the table by calling **fcntl()** or **lockf()** to request locks. When all processes that use a file do this, and respect the results, file integrity can be maintained.

It is possible to extend file locking by making it mandatory on all processes, whether or not they were designed to be part of the cooperating group. Mandatory locking is enforced by the file I/O function calls. As a result, an independent process that calls **write()** to update a locked record is blocked or receives an error code.

The **write()** and other system functions test for a contending lock on a file that has mandatory locking applied. The test is made for every operation on that file. When the caller is a process that is cooperating in the lock, and has already set an appropriate lock, the mandatory test is unnecessary overhead.

Mandatory locking is enforced on a file-by-file basis, triggered by a bit in the file inode that is set by *chmod* (see the chmod(1) and chmod(2) reference pages). In order to enforce mandatory locking on a particular file, turn on the set-group-ID bit along with a nonexecutable group permission, as in these examples, which are equivalent:

```
$ chmod 2644 target.file
$ chmod +l target.file
```

The bit must be set before the file is opened; a change has no effect on a file that is already open.

Example 7-9 shows a fragment of code that sets mandatory lock mode on a given filename.

Example 7-9   Setting Mandatory Locking Permission Bits

```
#include <sys/types.h>
#include <sys/stat.h>
int setMandatoryLocking(char *filename)
{
   int mode;
   struct stat buf;
   if (stat(filename, &buf) < 0)
   {
      perror("stat(2)");
      return error;
   }
   mode = buf.st_mode;
   /* ensure group execute permission 0010 bit is off */
   mode &= ~(S_IEXEC>>3);
   /* turn on 'set group id bit' in mode */
   mode |= S_ISGID;
   if (chmod(filename, mode) < 0)
   {
      perror("chmod(2)");
      return error;
   }
   return 0;
}
```

When IRIX opens a file, it checks to see whether both of two conditions are true:

- Set-group-ID bit is 1.

- Group execute permission is 0.

When both are true, the file is marked for mandatory locking, and each use of **creat()**, **open()**, **read()**, and **write()** tests for contending locks.

Some points to remember about mandatory locking:

- Mandatory locking does not protect against file truncation with the **truncate()** function (see the truncate(2) reference page), which does not look for locks on the truncated portion of the file.

- Mandatory locking protects only those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.

- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking Across Multiple Systems

Record locking is always effective within a single copy of the IRIX kernel. Locking is effective within a multiprocessor because processes running in different CPUs of the multiprocessor share a single copy of the IRIX kernel.

Record locking can be effective on processes that execute in different systems that access a filesystem mounted through NFS. However, there are these drawbacks:

- Deadlock detection is not possible between processes in different systems.

- You must make sure that the NFS locking daemon is running in both the NFS client (application) and server systems.

- Using record locking on NFS files has a strong impact on performance.

NFS File Locking

When a process running in an NFS client system requests a file or record lock, a complex sequence of events begins. (For details, consult the lockd(1M) reference page.)

First the kernel in the client system receives the lock request and determines that the file resides on a filesystem mounted using NFS. The kernel sends the lock request to a daemon called *rpc.lockd*. This daemon is responsible for communicating lock requests to other systems.

The *rpc.lockd* process sends the lock request to the *rpc.lockd* daemon running on the NFS server where the target file is physically mounted. On the server, that *rpc.lockd* issues the lock request locally. The server *rpc.lockd* sends the result, success or failure, back to the client *rpc.lockd*. The result is passed back to the calling process.

When the lock succeeds on the server side, *rpc.lockd* on the client system requests another daemon, *rpc.statd*, to monitor the NFS server that implements the lock. If the server fails and then recovers, *rpc.statd* will be informed. It then tries to reestablish all active locks. If the NFS server fails and recovers, and *rpc.lockd* is unable to reestablish a lock, it sends a signal (SIGUSR1) to the process that requested the lock.

When a process writes to a write-locked record, the data is sent directly to the NFS server, bypassing the local NFS buffer cache. This can have a significant impact on file performance.

Configuring NFS Locking

When *rpc.lockd* is not running in the NFS client system, or in the NFS server system, a cross-system lock cannot be established. In this case, locks are effective within the local system, but are not effective against contending file access from other systems.

To discover whether *rpc.lockd* is running, use the *chkconfig* command:

```
% /etc/chkconfig | grep lockd
```

If the returned value is off, *rpc.lockd* is not running and locks have local scope only.

To use *rpc.lockd*, the administrator must configure it on as follows:

```
% /etc/chkconfig lockd on
```

Then the system must be rebooted. This must be done on both the NFS file server and on all NFS clients where locks are requested.

Performance Impact

Normally, the NFS software uses a data cache to speed access to files. Data read or written to NFS mounted files is held in a memory cache for some time, and access requests to cached data is satisfied from memory instead of being read from the server. Data caching has a major effect on the speed of NFS file access.

189

As soon as any process places a file or record lock on an NFS mounted file, the file is marked as uncachable. All I/O requests for that file bypass the local memory cache and are sent to the NFS server. This ensures consistent results and data integrity. However, it means that every read or write to the file, at any offset, and from any process, incurs a network delay.

The file remains uncachable even when the lock is released. The file cannot use the cache again until it has been closed by all processes that have it open.

Using Asynchronous I/O

When you use asynchronous I/O, the work of buffering data and reading or writing a device is carried out in a parallel process or thread, while the process or thread that requested the I/O can continue doing other work. In a multiprocessor system, I/O can be fully overlapped with processing.

About Synchronous and Asynchronous I/O

Conventional I/O in UNIX is *synchronous*; that is, the process or thread that requests the I/O is blocked until the I/O has completed. The effects are different for input and for output.

About Synchronous Input

The normal sequence of operations for input is as follows:

1. A process invokes the system function **read()**, either directly or indirectly—for example, by accessing a new page of a memory-mapped file, or by calling a library function that calls **read()**.

2. The kernel, operating under the identity of the calling process, enters the read entry point of a device driver.

3. The device driver initiates an input operation and blocks the calling process, for example by waiting on a semaphore in the kernel address space.

4. The kernel schedules another process to use the CPU.

5. Later, the device completes the input operation and causes a hardware interrupt.

6. The kernel interrupt handler enters the device driver interrupt entry point.

7. The device driver, finding that the data has been received, unblocks the sleeping process, for example by posting a semaphore.

8. The kernel notes that the blocked process can now run.

9. Then or perhaps later, depending on scheduling priorities, the kernel schedules the process to run on some CPU.

10. The unblocked process exits the **read()** system call and returns to user code, the read being complete.

During steps 4-8, the process that requested input is blocked. The duration of the delay is unpredictable. For example, the delay can be negligible if the data is already in a buffer in memory. It can be as long as one rotation time of a disk, if the disk is positioned on the correct cylinder. It can be longer still, if the disk has to seek, or if the disk controller or bus adapter is busy with other transfers.

About Synchronous Output

For disk files, a process that calls **write()** is normally delayed only as long as it takes to copy the output data to a buffer in the kernel address space. The kernel asks the device driver to schedule the device write. The actual disk output is asynchronous. As a result, a process that requests output is usually blocked for only a short time. However, a number of disk write requests could be pending, so the true state of a file on disk is unknown until the file is closed.

In order to make sure that all data has been written to disk successfully, a program calls **fsync()** for a conventional file or **msync()** for a memory-mapped file (see the fsync(2) and msync(2) reference pages). The process that calls these functions is blocked until all buffered data has been written. (An alternative for disk output is to use direct output, discussed under "Using Direct I/O" on page 225.)

Devices other than disks may block the calling process until the output is complete. It is the device driver logic that determines whether a call to **write()** blocks the caller, and for how long.

About Asynchronous I/O

Some processes should never be blocked for the unpredictable times that I/O can require. One obvious solution can be summarized as "call **read()** or **write()** from a different process, and run that process in a different CPU." This is the essence of asynchronous I/O. You could implement an asynchronous I/O scheme of your own design, and you may wish to do so in order to integrate the I/O closely with your own design of processes and data structures. However, a standard solution is available.

IRIX supports asynchronous I/O library calls conforming to POSIX document 1003.1b-1993. You use relatively simple calls to initiate input or output. The library package handles the details of

- Creating asynchronous processes or threads to perform the I/O.

- Allocating a shared memory arena and the locks, semaphores, and other structures used to coordinate the I/O processes or threads.

- Queueing multiple input or output requests to each of multiple file descriptors.

- Reporting results back to your program, either on request, through signals, or through callback functions.

Asynchronous I/O Functions

Once you have opened the files and initialized asynchronous I/O, you perform asynchronous I/O by calling some of these functions:

aio_read(3)   Initiates asynchronous input from a file or device.

aio_write(3)   Initiates asynchronous output to a file or device.

lio_listio(3)   Initiates a list of operations to one or more files or devices.

aio_error(3)   Returns the status of an asynchronous operation.

aio_fsync(3)   Waits for all scheduled output for a file to complete.

aio_cancel(3)   Cancels pending, scheduled operations.

Each of these functions is described in detail in a reference page.

Asynchronous I/O Control Block

Each asynchronous I/O request is represented by an instance of *struct aiocb*, a data structure that your program must allocate. The important fields are as follows.

- The file descriptor that is the target of the operation.

  File descriptors are returned by **open()** (see the open(2) reference page). A file descriptor used for asynchronous I/O can represent any file or device—not only a disk file.

- The address and size of a buffer to supply or receive the data.

- The file position for the operation as it would be passed to **lseek()** (see the lseek(2) reference page)

  The use of this value is discussed under "Multiple Operations to One File" on page 203.

- A *sigevent* structure, whose contents indicate what, if anything, should be done to notify your program of the completion of the I/O.

  The use of the *sigevent* is discussed under "Checking for Completion" on page 199.

Note: The IRIX 5.2 implementation also accepted a request priority value. Request priorities are no longer supported. The request-priority field of *aiocb* exists for compatibility and for possible future use, but must currently contain zero.

Initializing Asynchronous I/O

You can initialize asynchronous I/O in either of two ways. One way is simple; the other gives you control over the initialization.

Implicit Initialization

You can initialize asynchronous I/O simply by starting an operation with **aio_read()**, **lio_listio()**, or **aio_write()**. The first such call causes default initialization. This is the only form of initialization described by the POSIX standard. However, you may need to control at least the timing of initialization.

Initializing with aio_sgi_init()

You can control initialization of asynchronous I/O by calling **aio_sgi_init()** (refer to the aio_sgi_init(3) reference page and to the declarations in */usr/include/aio.h*). The argument to this call can be a null pointer, indicating you want default values, or you can pass an *aioinit_t* structure. The principal fields of this structure specify

- the number of asynchronous processes or threads to execute I/O (*aio_threads*)

  The asynchronous I/O library creates asynchronous processes or threads to perform the I/O. It uses **sproc()** in normal programs, or **pthread_create()** in a pthread program.

  In either case, the default of asynchronous threads is 5 and the minimum is 2. Specify 1 more than the number of I/O operations that could reasonably be executed in parallel on the available hardware. For example if you will be doing asynchronous I/O to one disk file and one tape drive, there could be at most two concurrent I/O operations, so there is no need to have more than 3 (1 more than 2) asynchronous processes.

- the number of locks that the asynchronous I/O processes should preallocate (*aio_locks*)

  The default used by **aio_init()** is 3 locks; the minimum is 1. Specify the maximum number of simultaneous **lio_listio(**LIO_NOWAIT**)**, **aio_fsync()**, and **aio_suspend()** calls that your program could execute concurrently. If in doubt, specify the number of subprocesses your program contains.

- the number of processes or threads that will be sharing the use of asynchronous I/O (*aio_numusers*)

  The default is 5; the minimum is 2. Specify 1 more than the number of different processes or pthreads that will be requesting asynchronous I/O.

Other fields of the *aioinit_t* structure such as *aio_num* and *aio_usedba* are not used at this time and must be zero. Zero-valued fields are taken as a request for the default for that field. Example 8-1 shows a subroutine to initialize asynchronous I/O, given counts of devices and calling processes.

Example 8-1   Initializing Asynchronous I/O

```
int initAIO(int numDevs, int numSprocs, int maxOps)
{
    aioinit_t A = {0}; /* ensure zero'd fields */
    if (numDevs) /* we do know how many devices */
        A.aio_threads = 1+numDevs;
    if (numSprocs) /* we do know how many sprocs */
        A.aio_locks = A.aio_numusers = 1+numSprocs;
    if (maxOps) /* we do know max aiocbs at 1 time */
        A.aio_num = maxOps;
    return aioinit(&A);
}
```

When to Initialize

The time at which initialization occurs is important. If you initialize in a process that has been assigned to run on an isolated CPU, the asynchronous I/O processes will also run on that CPU. You probably want the I/O processes to run under normal dispatching on unrestricted CPUs. In that case, the proper sequence of initialization is:

- Open all file descriptors and verify that files and devices are ready.

- Initialize asynchronous I/O. The lightweight processes created by **aioinit()** inherit the attributes of the calling process, including its current priority and access to open file descriptors.

- Isolate any CPUs that are to be dedicated.

- Create child processes and assign them to their CPUs.

The asynchronous I/O processes created by **aioinit()** continue to be scheduled according to their priority in whatever CPUs remain available.

Scheduling Asynchronous I/O

You schedule an input or output operation by calling **aio_read()** or **aio_write()**, passing an *aiocb* structure to describe the operation (see the aio_read(3) and aio_write(3) reference pages). The operation is queued to that file descriptor. It will be executed when one of the asynchronous I/O processes or threads is available. The return code from the library call says nothing about the I/O operation itself; it merely indicates whether or not the *aiocb* could be queued.

Note: It is important to use a given *aiocb* for only one operation at a time, and to not modify an *aiocb* until its operation is complete.

You can schedule a list of operations using **lio_listio()** (see the lio_listio(3) reference page). The advantage of this function is that you can request a single notification (either a signal or a callback) when all of the operations in the list are complete. Alternatively, you can be notified of the completion of each one as it happens.

When an asynchronous I/O thread is free, it takes a queued *aiocb* and performs the equivalent function to **lseek()** (if a file position is specified), then the equivalent of **read()** or **write()**. The asynchronous process may be blocked for some time. That depends on the file or device and on the options that were specified when it was opened. When the operation is complete, the asynchronous process notifies the initiating process using the method requested in the *aiocb*.

You can cancel a started operation, or all pending operations for a given file descriptor, using **aio_cancel()** (see the aio_cancel(3) reference page).

Assuring Data Integrity

With sequential output, you call **fsync()** to ensure that all buffered data has been written. However, you cannot use **fsync()** with asynchronous I/O, since you are not sure when the **write()** calls will execute.

The **aio_fsync()** function queues the equivalent of an **fsync()** call for asynchronous execution (see the aio_fsync(3) reference page). This function takes an *aiocb*. The file descriptor in it specifies which file is to be synchronized. The **fsync()** operation is done following all other asynchronous operations that are pending when **aio_fsync()** is called. The synchronize operation can take considerable time, depending on how much output data has been buffered. Its completion is reported in the same ways as completion of a read or write (see the next topic).

Checking the Progress of Asynchronous Requests

You can test the progress and completion of an asynchronous operation by polling; or your program can be informed of the completion of an operation in a variety of ways. In the *aiocb*, the program can specify one of three things to be done when the operation is complete:

- Nothing; take no action.

- Send a signal of a specified number.

- Invoke a callback function directly from the asynchronous process.

In addition, the **aio_suspend()** function blocks its caller until one of a list of pending operations is complete.

Polling for Status

You can check the progress of any asynchronous operation (including **aio_fsync()**) by calling **aio_error()**, passing the *aiocb* for that operation.

While the operation is incomplete, **aio_error()** returns EIINPROGRESS. When the operation is complete, you can check the final return code from **read()**, **write()**, or **fsync()** using **aio_return()** (see the aio_error(3) and aio_return(3) reference pages).

To see in an example of polling for status, see function **inWait0()** under "Asynchronous I/O Example" on page 204. This function is used when the *aiocb* is initialized with SIGEV_NONE, meaning that no notification is to be returned at the completion of the operation. The function waits for an asynchronous operation to complete using a loop in the general form shown in Example 8-2.

Example 8-2   Polling for Asynchronous Completion

```
int waitForEndOfAsyncOp(aiocb *pab)
{
    while (EINPROGRESS == (ret = aio_error(pab)))
        sginap(0);
    return ret;
}
```

The function result is the final return code from the read, write, or sync operation that was started.

Checking for Completion

You have a wide variety of design options other than polling. Your program can:

- Use **aio_suspend()** to wait until one of a list of operations completes.

- Set up an empty signal handler function and use **sigsuspend()** or **sigwait()** to wait until a signal arrives (see the sigsuspend(2) and sigwait(3) reference pages).

- Use either a signal handler function or a callback function to report completion—for example, the function can post a semaphore.

Most of these methods are demonstrated in the example program under "Asynchronous I/O Example" on page 204.

Establishing a Completion Signal

You request a signal from an asynchronous operation by setting these values in the *aiocb* (refer to */usr/include/aio.h* and */usr/include/sys/signal.h*):

aio_sigevent.sigev_notify Set to SIGEV_SIGNAL.

aio_sigevent.sigev_signo The number of the signal. This should be one of the POSIX real-time signal numbers (see "Signal Numbers" on page 114).

aio_sigevent.sigev_value A value to be passed to the signal handler. This can be used to inform the signal handler of which I/O operation has completed; for example, it could be the address of the *aiocb*.

When you set up a signal handler for asynchronous completion, do so using **sigaction()** and specify the SA_SIGINFO flag (see the sigaction(2) reference page). This has two benefits: any new completion signal that arrives while the first is being handled is queued; and the *aio_sigev.sigev_value* word is passed to the handler in a *siginfo* structure.

Establishing a Callback Function

You request a callback at the end of an asynchronous operation by setting the following values in the *aiocb*:

aio_sigevent.sigev_notify	Set to SIGEV_CALLBACK.
aio_sigevent.sigev_func	The address of the callback function. Its prototype must be `void` *functionName*`(union sigval);`
aio_sigevent.sigev_value	A word to be passed to the callback function. This can be used to inform the function of which I/O operation has completed; for example, it could be the address of the *aiocb*.

The callback function is invoked from the asynchronous I/O thread when the **read()**, **write()** or **fsync()** operation finishes. This notification method has the lowest overhead and shortest latency, but it requires careful design to avoid race conditions in the use of shared variables.

The asynchronous I/O threads share the address space of the processes or threads that initialize asynchronous I/O. They may execute in a different CPU. Since the callback function could be entered at any time, it must coordinate its use of shared data structures. This is a good place to use a lock (see "Locks" on page 79). Locks have very low overhead in cases such as this, where there is likely to be little contention for the use of the lock.

Tip: You can call **aio_read()** or **aio_write()** from within a callback function or within a signal handler. This lets you start another operation with the least delay.

The code in Example 8-3 demonstrates a hypothetical set of subroutines to schedule asynchronous reads and writes using a single *aiocb*. The principle functions and global variables it uses are:

pendingIO	An array of records, each holding one request for an I/O operation.
dontTouchThatStuff	A lock used to gain exclusive use of *pendingIO*.
scheduleRead()	A function that accepts a request to read some amount of data, from a specified file descriptor, at a specified file offset. It places the request in *pendingIO* and then, if no asynchronous operation is under way, initiates it.

yeahWeFinishedOne() The callback function that is entered when an asynchronous operation completes. If any more operations are pending, it initiates one.

initiatePending() A function that initiates one selected pending operation. It prepares the *aiocb* structure, including the specification of **yeahWeFinishedOne()** as the callback function. The lock *dontTouchThatStuff* must be held before this function is called.

Note: The code in Example 8-3 is not intended to be realistic and is not recommended as a model. In order to demonstrate the use of callback functions and the *aiocb*, it essentially duplicates work that could be done by the **lio_listio()** feature of asynchronous I/O.

Example 8-3 Set of Functions to Schedule Asynchronous I/O

```
#define _ABI_SOURCE
#include <signal.h>
#include <aio.h>
#include <ulocks.h>
#define MAX_PENDING 10
#define STATUS_EMPTY 0
#define STATUS_ACTIVE 1
#define STATUS_PENDING 2
static struct onePendingIO {
    int status;
    int theFile;
    void *theData;
    off_t theSize;
    off_t theSeek;
    int readNotWrite;
    } pendingIO[MAX_PENDING];
static unsigned numPending;
static struct aiocb theAiocb;
static ulock_t dontTouchThatStuff;
static unsigned scanner;
static void initiatePending(int P);
static void
yeahWeFinishedOne(union sigval S)
{
    ussetlock(dontTouchThatStuff);
    pendingIO[S.sival_int].status = STATUS_EMPTY;
    if (numPending)
    {
        while (pendingIO[scanner].status != STATUS_PENDING)
```

```
            {
                if (++scanner >= MAX_PENDING)
                    scanner = 0;
            }
            initiatePending(scanner);
        }
    usunsetlock(dontTouchThatStuff);
}
static void
initiatePending(int P) /* lock must be held on entry */
{
    theAiocb.aio_fildes = pendingIO[P].theFile;
    theAiocb.aio_buf = pendingIO[P].theData;
    theAiocb.aio_nbytes = pendingIO[P].theSize;
    theAiocb.aio_offset = pendingIO[P].theSeek;
    theAiocb.aio_sigevent.sigev_notify = SIGEV_CALLBACK;
    theAiocb.aio_sigevent.sigev_func = yeahWeFinishedOne;
    theAiocb.aio_sigevent.sigev_value.sival_int = P;
    if (pendingIO[P].readNotWrite)
        aio_read(&theAiocb);
    else
        aio_write(&theAiocb);
    pendingIO[P].status = STATUS_ACTIVE;
    --numPending;
}
/*public*/ int
scheduleRead( int FD, void *pdata, off_t len, off_t pos )
{
    int j;
    if (numPending >= MAX_PENDING)
        likeTotallyFreakOut();
    ussetlock(dontTouchThatStuff);
    for(j=0; pendingIO[j].status != STATUS_EMPTY; ++j)
        ;
    pendingIO[j].theFile = FD;
    pendingIO[j].theData = pdata;
    pendingIO[j].theSize = len;
    pendingIO[j].theSeek = pos;
    pendingIO[j].readNotWrite = 1;
    pendingIO[j].status = STATUS_PENDING;
    if (1 == ++numPending)
        initiatePending(j);
    usunsetlock(dontTouchThatStuff);
}
```

Holding Callbacks Temporarily

You can temporarily prevent callback functions from being entered using the **aio_hold()** function. This function is not defined in the POSIX standard; it is added by the MIPS ABI standard. Use it as follows:

- Call **aio_hold**(AIO_HOLD_CALLBACK) to prevent any callback function from being invoked.

- Call **aio_hold**(AIO_RELEASE_CALLBACK) to allow callback functions to be invoked. Any that were held are now called.

- Call **aio_hold**(AIO_ISHELD_CALLBACK) returns 1 if callbacks are currently being held; otherwise it returns 0.

Multiple Operations to One File

When you queue multiple operations to a single file descriptor, the asynchronous I/O package does not always guarantee the order of their execution. There are three ways you can ensure the sequence of operations.

You can open any output file descriptor passing the flag O_APPEND (see the open(1) reference page). Asynchronous write requests to a file opened with O_APPEND are executed in the sequence of the calls to **aio_write()** or the sequence they are listed for **lio_listio()**. You can use this feature to ensure that a sequence of records is appended to a file in sequence.

For files that support **lseek()**, you can specify any order of operations by specifying the file offset in the *aiocb*. The asynchronous process executes an absolute seek to that offset as part of the operation. Even if the operations are not performed in the sequence they were requested, the data is transferred in sequence. You can use this feature to ensure that multiple requests for sequential disk input are stored in sequential locations.

For non-disk input operations, the only way you can be certain that operations are done in sequence is to schedule them one at a time, waiting for each one to complete.

Asynchronous I/O Example

The following source displays a highly artificial program whose purpose is to exercise most options of asynchronous I/O. The program syntax is:

```
aiocat [ -o outfile ] [-a {0|1|2|3} ] infilename...
```

The actual output of the program is the concatenation of all the one or more files *infilename...*, written to the file *outfile*. The default outfile is $TEMPDIR/*aiocat.out*. In effect, the program is an overcomplicated version of the standard *cat* command.

When you compile it with the variable DO_SPROCS defined as 1, the program creates one process for each *infilename*. Each of these processes uses asynchronous I/O requests to read its corresponding input file, and to write that data to the correct offset in *outfile*.

After all the files have been read and written, the program reports the CPU time charged for each file, and the effective data transfer rate in bytes per microsecond.

The -a parameter specifies which of four methods is used to wait for I/O completion:

-a 0   Poll for completion with **aio_error()**.

-a 1   Wait for completion with **aio_suspend()**.

-a 2   Wait on a semaphore posted from a signal handler.

-a 3   Wait on a semaphore posted from a callback routine.

Execution of *aiocat* can resemble the following (from an Origin2000 with 8 CPUs):

```
> ls -l incat?
-rwxr-xr-x    1 cortesi   nuucp      234964 Jun  4 10:17 incat1
-rwxr-xr-x    1 cortesi   nuucp      234964 Jun  4 10:17 incat2
-rwxr-xr-x    1 cortesi   nuucp      234964 Jun  4 10:18 incat3
-rwxr-xr-x    1 cortesi   nuucp      234964 Jun  4 10:19 incat4
> aiocat -o outcat -a 0 incat?
   procid   time      fsize      filename
 0: 920       440000   234964     incat1
 1: 939       480000   234964     incat2
 2: 940       510000   234964     incat3
 3: 936       530000   234964     incat4
total time 1960000 usec, total bytes 939856, 0.479518 bytes/usec
> aiocat -o outcat -a 1 incat?
   procid   time      fsize      filename
```

```
                   0: 942      350000   234964    incat1
                   1: 944      370000   234964    incat2
                   2: 949      370000   234964    incat3
                   3: 946      370000   234964    incat4
                   total time 1460000 usec, total bytes 939856, 0.643737 bytes/usec
                   > aiocat -o outcat -a 2 incat?
                       procid   time     fsize     filename
                   0: 962       90000    234964    incat1
                   1: 955       80000    234964    incat2
                   2: 967       90000    234964    incat3
                   3: 960       90000    234964    incat4
                   total time 350000 usec, total bytes 939856, 2.6853 bytes/usec
                   > aiocat -o outcat -a 3 incat?
                       procid   time     fsize     filename
                   0: 909       50000    234964    incat1
                   1: 969       50000    234964    incat2
                   2: 966       60000    234964    incat3
                   3: 972       60000    234964    incat4
                   total time 220000 usec, total bytes 939856, 4.27207 bytes/usec
```

Example 8-4   Source Code of aiocat

```c
/* ============================================================================
|| aiocat.c : This highly artificial example demonstrates asynchronous I/O.
||
|| The command syntax is:
||  aiocat [ -o outfile ] [-a {0|1|2|3} ] infilename...
||
|| The output file is given by -o, with $TMPDIR/aiocat.out by default.
|| The aio method of waiting for completion is given by -a as follows:
||  -a 0 poll for completion with aio_error() (default)
||  -a 1 wait for completion with aio_suspend()
||  -a 2 wait on a semaphore posted from a signal handler
||  -a 3 wait on a semaphore posted from a callback routine
||
|| Up to MAX_INFILES input files may be specified. Each input file is
|| read in BLOCKSIZE units. The output file contains the data from
|| the input files in the order they were specified. Thus the
|| output should be the same as "cat infilename... >outfile".
||
|| When DO_SPROCS is compiled true, all I/O is done asynchronously
|| and concurrently using one sproc'd process per file.  Thus in a
|| multiprocessor concurrent input can be done.
============================================================================ */
```

```
#define _SGI_MP_SOURCE  /* see the "Caveats" section of sproc(2) */
#include <sys/time.h>    /* for clock() */
#include <errno.h>       /* for perror() */
#include <stdio.h>       /* for printf() */
#include <stdlib.h>      /* for getenv(), malloc(3c) */
#include <ulocks.h>      /* usinit() & friends */
#include <bstring.h>     /* for bzero() */
#include <sys/resource.h> /* for prctl, get/setrlimit() */
#include <sys/prctl.h>   /* for prctl() */
#include <sys/types.h>   /* required by lseek(), prctl */
#include <unistd.h>      /* ditto */
#include <sys/types.h>   /* wanted by sproc() */
#include <sys/prctl.h>   /* ditto */
#include <signal.h>      /* for signals - gets sys/signal and sys/siginfo */
#include <aio.h>         /* async I/O */
#define BLOCKSIZE 2048   /* input units -- play with this number */
#define MAX_INFILES 10   /* max sprocs: anything from 4 to 20 or so */
#define DO_SPROCS 1      /* set 0 to do all I/O in a single process */
#define QUITIFNULL(PTR,MSG) if (NULL==PTR) {perror(MSG);return(errno);}
#define QUITIFMONE(INT,MSG) if (-1==INT) {perror(MSG);return(errno);}
/*****************************************************************************
|| The following structure contains the info needed by one child proc.
|| The main program builds an array of MAX_INFILES of these.
|| The reason for storing the actual filename here (not a pointer) is
|| to force the struct to >128 bytes.  Then, when the procs run in
|| different CPUs on a CHALLENGE, the info structs will be in different
|| cache lines, and a store by one proc will not invalidate a cache line
|| for its neighbor proc.
*/
typedef struct child
{
        /* read-only to child */
    char fname[100];        /* input filename from argv[n] */
    int         fd;         /* FD for this file */
    void*       buffer;     /* buffer for this file */
    int         procid;     /* process ID of child process */
    off_t       fsize;      /* size of this input file */
        /* read-write to child */
    usema_t*    sema;       /* semaphore used by methods 2 & 3 */
    off_t       outbase;    /* starting offset in output file */
    off_t       inbase;     /* current offset in input file */
    clock_t     etime;      /* sum of utime/stime to read file */
    aiocb_t     acb;        /* aiocb used for reading and writing */
} child_t;
/*****************************************************************************
```

```
|| Globals, accessible to all processes
*/
char*       ofName = NULL;  /* output file name string */
int         outFD;          /* output file descriptor */
usptr_t*    arena;          /* arena where everything is built */
barrier_t*  convene;        /* barrier used to sync up */
int         nprocs = 1;     /* 1 + number of child procs */
child_t*    array;          /* array of child_t structs in arena */
int         errors = 0;     /* always incremented on an error */
/*****************************************************************************
|| forward declaration of the child process functions
*/
void inProc0(void *arg, size_t stk);    /* polls with aio_error() */
void inProc1(void *arg, size_t stk);    /* uses aio_suspend() */
void inProc2(void *arg, size_t stk);    /* uses a signal and semaphore */
void inProc3(void *arg, size_t stk);    /* uses a callback and semaphore */
/*****************************************************************************
// The main()
*/
int main(int argc, char **argv)
{
    char*       tmpdir;         /* ->name string of temp dir */
    int         nfiles;         /* how many input files on cmd line */
    int         argno;          /* loop counter */
    child_t*    pc;             /* ->child_t of current file */
    void (*method)(void *,size_t) = inProc0; /* ->chosen input method */
    char        arenaPath[128]; /* build area for arena pathname */
    char        outPath[128];   /* build area for output pathname */
    /*
    || Ensure the name of a temporary directory.
    */
    tmpdir = getenv("TMPDIR");
    if (!tmpdir) tmpdir = "/var/tmp";
    /*
    || Build a name for the arena file.
    */
    strcpy(arenaPath,tmpdir);
    strcat(arenaPath,"/aiocat.wrk");
    /*
    || Create the arena. First, call usconfig() to establish the
    || minimum size (twice the buffer size per file, to allow for misc usage)
    || and the (maximum) number of processes that may later use
    || this arena.  For this program that is MAX_INFILES+10, allowing
    || for our sprocs plus those done by aio_sgi_init().
    || These values apply to any arenas made subsequently, until changed.
```

```
*/
{
    ptrdiff_t ret;
    ret = usconfig(CONF_INITSIZE,2*BLOCKSIZE*MAX_INFILES);
    QUITIFMONE(ret,"usconfig size")
    ret = usconfig(CONF_INITUSERS,MAX_INFILES+10);
    QUITIFMONE(ret,"usconfig users")
    arena = usinit(arenaPath);
    QUITIFNULL(arena,"usinit")
}
/*
|| Allocate the barrier.
*/
convene = new_barrier(arena);
QUITIFNULL(convene,"new_barrier")
/*
|| Allocate the array of child info structs and zero it.
*/
array = (child_t*)usmalloc(MAX_INFILES*sizeof(child_t),arena);
QUITIFNULL(array,"usmalloc")
bzero((void *)array,MAX_INFILES*sizeof(child_t));
/*
|| Loop over the arguments, setting up child structs and
|| counting input files.  Quit if a file won't open or seek,
|| or if we can't get a buffer or semaphore.
*/
for (nfiles=0, argno=1; argno < argc; ++argno )
{
    if (0 == strcmp(argv[argno],"-o"))
    { /* is the -o argument */
        ++argno;
        if (argno < argc)
            ofName = argv[argno];
        else
        {
            fprintf(stderr,"-o must have a filename after\n");
            return -1;
        }
    }
    else if (0 == strcmp(argv[argno],"-a"))
    { /* is the -a argument */
        char c = argv[++argno][0];
        switch(c)
```

```
            {
            case '0' : method = inProc0; break;
            case '1' : method = inProc1; break;
            case '2' : method = inProc2; break;
            case '3' : method = inProc3; break;
            default:
                {
                    fprintf(stderr,"unknown method -a %c\n",c);
                    return -1;
                }
            }
    }
    else if ('-' == argv[argno][0])
    { /* is unknown -option */
        fprintf(stderr,"aiocat [-o outfile] [-a 0|1|2|3] infiles...\n");
        return -1;
    }
    else
    { /* neither -o nor -a, assume input file */
        if (nfiles < MAX_INFILES)
        {
            /*
            || save the filename
            */
            pc = &array[nfiles];
            strcpy(pc->fname,argv[argno]);
            /*
            || allocate a buffer and a semaphore.  Not all
            || child procs use the semaphore but so what?
            */
            pc->buffer = usmalloc(BLOCKSIZE,arena);
            QUITIFNULL(pc->buffer,"usmalloc(buffer)")
            pc->sema = usnewsema(arena,0);
            QUITIFNULL(pc->sema,"usnewsema")
            /*
            || open the file
            */
            pc->fd = open(pc->fname,O_RDONLY);
            QUITIFMONE(pc->fd,"open")
            /*
            || get the size of the file. This leaves the file
            || positioned at-end, but there is no need to reposition
            || because all aio_read calls have an implied lseek.
            || NOTE: there is no check for zero-length file; that
            || is a valid (and interesting) test case.
```

```
              */
              pc->fsize = lseek(pc->fd,0,SEEK_END);
              QUITIFMONE(pc->fsize,"lseek")
              /*
              || set the starting base address of this input file
              || in the output file.  The first file starts at 0.
              || Each one after starts at prior base + prior size.
              */
              if (nfiles) /* not first */
                  pc->outbase =
                      array[nfiles-1].fsize + array[nfiles-1].outbase;
              ++nfiles;
          }
          else
          {
              printf("Too many files, %s ignored\n",argv[argno]);
          }
      }
} /* end for(argc) */
/*
|| If there was no -o argument, construct an output file name.
*/
if (!ofName)
{
    strcpy(outPath,tmpdir);
    strcat(outPath,"/aiocat.out");
    ofName = outPath;
}
/*
|| Open, creating or truncating, the output file.
|| Do not use O_APPEND, which would constrain aio to doing
|| operations in sequence.
*/
outFD = open(ofName, O_WRONLY+O_CREAT+O_TRUNC,0666);
QUITIFMONE(outFD,"open(output)")
/*
|| If there were no input files, just quit, leaving empty output
*/
if (!nfiles)
{
    return 0;
}
/*
|| Note the number of processes-to-be, for use in initializing
|| aio and for use by each child in a barrier() call.
```

```
    */
    nprocs = 1+nfiles;
    /*
    || Initialize async I/O using aio_sgi_init(), in order to specify
    || a number of locks at least equal to the number of child procs
    || and in order to specify extra sproc users.
    */
    {
        aioinit_t ainit = {0}; /* all fields initially zero */
        /*
        || Go with the default 5 for the number of aio-created procs,
        || as we have no way of knowing the number of unique devices.
        */
#define AIO_PROCS 5
        ainit.aio_threads = AIO_PROCS;
        /*
        || Set the number of locks aio needs to the number of procs
        || we will start, minimum 3.
        */
        ainit.aio_locks = (nprocs > 2)?nprocs:3;
        /*
        || Warn aio of the number of user procs that will be
        || using its arena.
        */
        ainit.aio_numusers = nprocs;
        aio_sgi_init(&ainit);
    }
    /*
    || Process each input file, either in a child process or in
    || a subroutine call, as specified by the DO_SPROCS variable.
    */
    for (argno = 0; argno < nfiles; ++argno)
    {
        pc = &array[argno];
#if DO_SPROCS
#define CHILD_STACK 64*1024
        /*
        || For each input file, start a child process as an instance
        || of the selected method (-a argument).
        || If an error occurs, quit. That will send a SIGHUP to any
        || already-started child, which will kill it, too.
```

```
        */
        pc->procid = sprocsp(method      /* function to start */
                            ,PR_SALL    /* share all, keep FDs sync'd */
                            ,(void *)pc /* argument to child func */
                            ,NULL       /* absolute stack seg */
                            ,CHILD_STACK);  /* max stack seg growth */
        QUITIFMONE(pc->procid,"sproc")
#else
        /*
        || For each input file, call the selected (-a) method as a
        || subroutine to copy its file.
        */
        fprintf(stderr,"file %s...",pc->fname);
        method((void*)pc,0);
        if (errors) break;
        fprintf(stderr,"done\n");
#endif
    }
#if DO_SPROCS
    /*
    || Wait for all the kiddies to get themselves initialized.
    || When all have started and reached barrier(), all continue.
    || If any errors occurred in initialization, quit.
    */
    barrier(convene,nprocs);
    /*
    || Child processes are executing now. Reunite the family round the
    || old hearth one last time, when their processing is complete.
    || Each child ensures that all its output is complete before it
    || invokes barrier().
    */
    barrier(convene,nprocs);
#endif
    /*
    || Close the output file and print some statistics.
    */
    close(outFD);
    {
        clock_t timesum;
        long    bytesum;
        double  bperus;
        printf("   procid   time     fsize     filename\n");
        for(argno = 0, timesum = bytesum = 0 ; argno < nfiles ; ++argno)
```

```
        {
            pc = &array[argno];
            timesum += pc->etime;
            bytesum += pc->fsize;
            printf("%2d: %-8d %-8d %-8d  %s\n"
                    ,argno,pc->procid,pc->etime,pc->fsize,pc->fname);
        }
        bperus = ((double)bytesum)/((double)timesum);
        printf("total time %d usec, total bytes %d, %g bytes/usec\n"
                    ,timesum              , bytesum , bperus);
    }
    /*
    || Unlink the arena file, so it won't exist when this progam runs
    || again. If it did exist, it would be used as the initial state of
    || the arena, which might or might not have any effect.
    */
    unlink(arenaPath);
    return 0;
}
/****************************************************************************
|| inProc0() alternates polling with aio_error() with sginap(). Under
|| the Frame Scheduler, it would use frs_yield() instead of sginap().
|| The general pattern of this function is repeated in the other three;
|| only the wait method varies from function to function.
*/
int inWait0(child_t *pch)
{
    int ret;
    aiocb_t* pab = &pch->acb;
    while (EINPROGRESS == (ret = aio_error(pab)))
    {
        sginap(0);
    }
    return ret;
}
void inProc0(void *arg, size_t stk)
{
    child_t *pch = arg;          /* starting arg is ->child_t for my file */
    aiocb_t *pab = &pch->acb;    /* base address of the aiocb_t in child_t */
    int ret;                     /* as long as this is 0, all is ok */
    int bytes;                   /* #bytes read on each input */
    /*
    || Initialize -- no signals or callbacks needed.
```

```
    */
    pab->aio_sigevent.sigev_notify = SIGEV_NONE;
    pab->aio_buf = pch->buffer; /* always the same */
#if DO_SPROCS
    /*
    || Wait for the starting gun...
    */
    barrier(convene,nprocs);
#endif
    pch->etime = clock();
    do /* read and write, read and write... */
    {
        /*
        || Set up the aiocb for a read, queue it, and wait for it.
        */
        pab->aio_fildes = pch->fd;
        pab->aio_offset = pch->inbase;
        pab->aio_nbytes = BLOCKSIZE;
        if (ret = aio_read(pab))
            break;  /* unable to schedule a read */
        ret = inWait0(pch);
        if (ret)
            break;  /* nonzero read completion status */
        /*
        || get the result of the read() call, the count of bytes read.
        || Since aio_error returned 0, the count is nonnegative.
        || It could be 0, or less than BLOCKSIZE, indicating EOF.
        */
        bytes = aio_return(pab); /* actual read result */
        if (!bytes)
            break;  /* no need to write a last block of 0 */
        pch->inbase += bytes;   /* where to read next time */
        /*
        || Set up the aiocb for a write, queue it, and wait for it.
        */
        pab->aio_fildes = outFD;
        pab->aio_nbytes = bytes;
        pab->aio_offset = pch->outbase;
        if (ret = aio_write(pab))
            break;
        ret = inWait0(pch);
        if (ret)
            break;
        pch->outbase += bytes;  /* where to write next time */
    } while ((!ret) && (bytes == BLOCKSIZE));
```

214

```
    /*
    || The loop is complete.  If no errors so far, use aio_fsync()
    || to ensure that output is complete.  This requires waiting
    || yet again.
    */
    if (!ret)
    {
        if (!(ret = aio_fsync(O_SYNC,pab)))
        ret = inWait0(pch);
    }
    /*
    || Flag any errors for the parent proc. If none, count elapsed time.
    */
    if (ret) ++errors;
    else pch->etime = (clock() - pch->etime);
#if DO_SPROCS
    /*
    || Rendezvous with the rest of the family, then quit.
    */
    barrier(convene,nprocs);
#endif
    return;
} /* end inProc1 */
/******************************************************************************
|| inProc1 uses aio_suspend() to await the completion of each operation.
|| Otherwise it is the same as inProc0, above.
*/

int inWait1(child_t *pch)
{
    int ret;
    aiocb_t* susplist[1]; /* list of 1 aiocb for aio_suspend() */
    susplist[0] = &pch->acb;
    /*
    || Note: aio.h declares the 1st argument of aio_suspend() as "const."
    || The C compiler requires the actual-parameter to match in type,
    || so the list we pass must either be declared "const aiocb_t*" or
    || must be cast to that -- else cc gives a warning.  The cast
    || in the following statement is only to avoid this warning.
    */
    ret = aio_suspend( (const aiocb_t **) susplist,1,NULL);
    return ret;
}
void inProc1(void *arg, size_t stk)
```

```
{
    child_t *pch = arg;          /* starting arg is ->child_t for my file */
    aiocb_t *pab = &pch->acb;    /* base address of the aiocb_t in child_t */
    int ret;                     /* as long as this is 0, all is ok */
    int bytes;                   /* #bytes read on each input */
    /*
    || Initialize -- no signals or callbacks needed.
    */
    pab->aio_sigevent.sigev_notify = SIGEV_NONE;
    pab->aio_buf = pch->buffer; /* always the same */
#if DO_SPROCS
    /*
    || Wait for the starting gun...
    */
    barrier(convene,nprocs);
#endif
    pch->etime = clock();
    do /* read and write, read and write... */
    {
        /*
        || Set up the aiocb for a read, queue it, and wait for it.
        */
        pab->aio_fildes = pch->fd;
        pab->aio_offset = pch->inbase;
        pab->aio_nbytes = BLOCKSIZE;
        if (ret = aio_read(pab))
            break;
        ret = inWait1(pch);
        /*
        || If the aio_suspend() return is nonzero, it means that the wait
        || did not end for i/o completion but because of a signal. Since we
        || expect no signals here, we take that as an error.
        */
        if (!ret) /* op is complete */
            ret = aio_error(pab);  /* read() status, should be 0 */
        if (ret)
            break;  /* signal, or nonzero read completion */
        /*
        || get the result of the read() call, the count of bytes read.
        || Since aio_error returned 0, the count is nonnegative.
        || It could be 0, or less than BLOCKSIZE, indicating EOF.
        */
        bytes = aio_return(pab); /* actual read result */
        if (!bytes)
            break;  /* no need to write a last block of 0 */
```

216

```
        pch->inbase += bytes;   /* where to read next time */
        /*
        || Set up the aiocb for a write, queue it, and wait for it.
        */
        pab->aio_fildes = outFD;
        pab->aio_nbytes = bytes;
        pab->aio_offset = pch->outbase;
        if (ret = aio_write(pab))
            break;
        ret = inWait1(pch);
        if (!ret) /* op is complete */
            ret = aio_error(pab);  /* should be 0 */
        if (ret)
            break;
        pch->outbase += bytes;  /* where to write next time */
    } while ((!ret) && (bytes == BLOCKSIZE));
    /*
    || The loop is complete.  If no errors so far, use aio_fsync()
    || to ensure that output is complete.  This requires waiting
    || yet again.
    */
    if (!ret)
    {
        if (!(ret = aio_fsync(O_SYNC,pab)))
            ret = inWait1(pch);
    }
    /*
    || Flag any errors for the parent proc. If none, count elapsed time.
    */
    if (ret) ++errors;
    else pch->etime = (clock() - pch->etime);
#if DO_SPROCS
    /*
    || Rendezvous with the rest of the family, then quit.
    */
    barrier(convene,nprocs);
#endif
} /* end inProc0 */
/*****************************************************************************
|| inProc2 requests a signal upon completion of an I/O. After starting
|| an operation, it P's a semaphore which is V'd from the signal handler.
*/
#define AIO_SIGNUM SIGRTMIN+1 /* arbitrary choice of signal number */
void sigHandler2(const int signo, const struct siginfo *sif )
```

```
{
    /*
    || In this minimal signal handler we pick up the address of the
    || child_t info structure -- which was put in aio_sigevent.sigev_value
    || field during initialization -- and use it to find the semaphore.
    */
    child_t *pch = sif->si_value.sival_ptr ;
    usvsema(pch->sema);
    return; /* stop here with dbx to print the above address */
}
int inWait2(child_t *pch)
{
    /*
    || Wait for any signal handler to post the semaphore.  The signal
    || handler could have been entered before this function is called,
    || or it could be entered afterward.
    */
    uspsema(pch->sema);
    /*
    || Since this process executes only one aio operation at a time,
    || we can return the status of that operation.  In a more complicated
    || design, if a signal could arrive from more than one pending
    || operation, this function could not return status.
    */
    return aio_error(&pch->acb);
}
void inProc2(void *arg, size_t stk)
{
    child_t *pch = arg;         /* starting arg is ->child_t for my file */
    aiocb_t *pab = &pch->acb;   /* base address of the aiocb_t in child_t */
    int ret;                    /* as long as this is 0, all is ok */
    int bytes;                  /* #bytes read on each input */
    /*
    || Initialize -- request a signal in aio_sigevent. The address of
    || the child_t struct is passed as the siginfo value, for use
    || in the signal handler.
    */
    pab->aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    pab->aio_sigevent.sigev_signo = AIO_SIGNUM;
    pab->aio_sigevent.sigev_value.sival_ptr = (void *)pch;
    pab->aio_buf = pch->buffer; /* always the same */
    /*
    || Initialize -- set up a signal handler for AIO_SIGNUM.
    */
```

```
    {
        struct sigaction sa = {SA_SIGINFO,sigHandler2};
        ret = sigaction(AIO_SIGNUM,&sa,NULL);
        if (ret) ++errors; /* parent will shut down ASAP */
    }
#if DO_SPROCS
    /*
    || Wait for the starting gun...
    */
    barrier(convene,nprocs);
#else
    if (ret) return;
#endif
    pch->etime = clock();
    do /* read and write, read and write... */
    {
        /*
        || Set up the aiocb for a read, queue it, and wait for it.
        */
        pab->aio_fildes = pch->fd;
        pab->aio_offset = pch->inbase;
        pab->aio_nbytes = BLOCKSIZE;
        if (!(ret = aio_read(pab)))
            ret = inWait2(pch);
        if (ret)
            break;  /* could not start read, or it ended badly */
        /*
        || get the result of the read() call, the count of bytes read.
        || Since aio_error returned 0, the count is nonnegative.
        || It could be 0, or less than BLOCKSIZE, indicating EOF.
        */
        bytes = aio_return(pab); /* actual read result */
        if (!bytes)
            break;  /* no need to write a last block of 0 */
        pch->inbase += bytes;   /* where to read next time */
        /*
        || Set up the aiocb for a write, queue it, and wait for it.
        */
        pab->aio_fildes = outFD;
        pab->aio_nbytes = bytes;
        pab->aio_offset = pch->outbase;
        if (!(ret = aio_write(pab)))
            ret = inWait2(pch);
        if (ret)
            break;
```

219

```
        pch->outbase += bytes;  /* where to write next time */
    } while ((!ret) && (bytes == BLOCKSIZE));
    /*
    || The loop is complete.  If no errors so far, use aio_fsync()
    || to ensure that output is complete.  This requires waiting
    || yet again.
    */
    if (!ret)
    {
        if (!(ret = aio_fsync(O_SYNC,pab)))
            ret = inWait2(pch);
    }
    /*
    || Flag any errors for the parent proc. If none, count elapsed time.
    */
    if (ret) ++errors;
    else pch->etime = (clock() - pch->etime);
#if DO_SPROCS
    /*
    || Rendezvous with the rest of the family, then quit.
    */
    barrier(convene,nprocs);
#endif
} /* end inProc2 */

/****************************************************************************
|| inProc3 uses a callback and a semaphore. It waits with a P operation.
|| The callback function executes a V operation.  This may come before or
|| after the P operation.
*/
void callBack3(union sigval usv)
{
    /*
    || The callback function receives the pointer to the child_t struct,
    || as prepared in aio_sigevent.sigev_value.sival_ptr.  Use this to
    || post the semaphore in the child_t struct.
    */
    child_t *pch = usv.sival_ptr;
    usvsema(pch->sema);
    return;
}
int inWait3(child_t *pch)
```

```
{
    /*
    || Suspend, if necessary, by polling the semaphore.  The callback
    || function might be entered before we reach this point, or after.
    */
    uspsema(pch->sema);
    /*
    || Return the status of the aio operation associated with the sema.
    */
    return aio_error(&pch->acb);
}
void inProc3(void *arg, size_t stk)
{
    child_t *pch = arg;         /* starting arg is ->child_t for my file */
    aiocb_t *pab = &pch->acb;   /* base address of the aiocb_t in child_t */
    int ret;                    /* as long as this is 0, all is ok */
    int bytes;                  /* #bytes read on each input */
    /*
    || Initialize -- request a callback in aio_sigevent. The address of
    || the child_t struct is passed as the siginfo value to be passed
    || into the callback.
    */
    pab->aio_sigevent.sigev_notify = SIGEV_CALLBACK;
    pab->aio_sigevent.sigev_func = callBack3;
    pab->aio_sigevent.sigev_value.sival_ptr = (void *)pch;
    pab->aio_buf = pch->buffer; /* always the same */
#if DO_SPROCS
    /*
    || Wait for the starting gun...
    */
    barrier(convene,nprocs);
#endif
    pch->etime = clock();
    do /* read and write, read and write... */
    {
        /*
        || Set up the aiocb for a read, queue it, and wait for it.
        */
        pab->aio_fildes = pch->fd;
        pab->aio_offset = pch->inbase;
        pab->aio_nbytes = BLOCKSIZE;
        if (!(ret = aio_read(pab)))
            ret = inWait3(pch);
        if (ret)
            break;  /* read error */
```

221

```
        /*
        || get the result of the read() call, the count of bytes read.
        || Since aio_error returned 0, the count is nonnegative.
        || It could be 0, or less than BLOCKSIZE, indicating EOF.
        */
        bytes = aio_return(pab); /* actual read result */
        if (!bytes)
            break;  /* no need to write a last block of 0 */
        pch->inbase += bytes;   /* where to read next time */
        /*
        || Set up the aiocb for a write, queue it, and wait for it.
        */
        pab->aio_fildes = outFD;
        pab->aio_nbytes = bytes;
        pab->aio_offset = pch->outbase;
        if (!(ret = aio_write(pab)))
            ret = inWait3(pch);
        if (ret)
            break;
        pch->outbase += bytes;  /* where to write next time */
    } while ((!ret) && (bytes == BLOCKSIZE));
    /*
    || The loop is complete.  If no errors so far, use aio_fsync()
    || to ensure that output is complete.  This requires waiting
    || yet again.
    */
    if (!ret)
    {
        if (!(ret = aio_fsync(O_SYNC,pab)))
            ret = inWait3(pch);
    }
    /*
    || Flag any errors for the parent proc. If none, count elapsed time.
    */
    if (ret) ++errors;
    else pch->etime = (clock() - pch->etime);
#if DO_SPROCS
    /*
    || Rendezvous with the rest of the family, then quit.
    */
    barrier(convene,nprocs);
#endif
} /* end inProc3 */
```

222

High-Performance File I/O

This chapter describes three special modes of disk I/O:

- "Using Synchronous Output" on page 223 describes the effect of the O_SYNC file option.

- "Using Direct I/O" on page 225 compares the use of direct-to-disk output with normal (buffered) output.

- "Using Guaranteed-Rate I/O" on page 230 describes a special mode of I/O used with real-time XFS volumes.

Using Synchronous Output

You use synchronous disk output to prevent the IRIX kernel scheme from deferring disk output.

About Buffered Output

When you open a disk file and do not specify the O_SYNC flag (see the open(2) reference page), a call to **write()** for that file descriptor returns as soon as the data has been copied to a buffer in the kernel address space.

The actual disk write may not take place until considerable time has passed. A common pool of disk buffers is used for all disk files. (The size of the pool is set by the *nbuf* system configuration variable, and defaults to approximately 2.5% of all physical memory.) Disk buffer management is integrated with the virtual memory paging mechanism. A daemon executes periodically and initiates output of buffered blocks according to the age of the data and the needs of the system.

The default management of disk output improves performance for the system in general but has three drawbacks:

- All output data must be copied from the buffer in process address space to a buffer in the kernel address space. For small or infrequent writes, the copy time is negligible, but for large quantities of data it adds up.

- You do not know when the written data is actually safe on disk. A system crash could prevent the output of a large amount of buffered data.

- When the system does decide to flush output buffers to disk, it can generate a large quantity of I/O that monopolizes the disk channel for a long time, delaying other I/O operations.

You can force the writing of all pending output for a file by calling **fsync()** (see the fsync(2) reference page). This gives you a way of creating a known checkpoint of a file. However, **fsync()** blocks until all buffered writes are complete, possibly a long time. When using asynchronous I/O, you can make file synchronization asynchronous also (see "Assuring Data Integrity" on page 197).

Requesting Synchronous Output

When you open a disk file specifying O_SYNC, each call to **write()** blocks until the data has been written to disk. This gives you a way of ensuring that all output is complete as it is created. If you combine O_SYNC access with asynchronous I/O, you can let the asynchronous process suffer the delay (see "About Asynchronous I/O" on page 192).

Synchronous output is still buffered output—data is copied to a kernel buffer before writing. The meaning of O_SYNC is that the file data is all present even if the system crashes. For this reason, each write to an O_SYNC file can cause a write of file metadata as well as the file data itself. These extra writes can make synchronous output quite slow.

The O_SYNC option takes effect even when the amount of data you write is less than the physical blocksize of the disk, or when the output does not align with the physical boundaries of disk blocks. In order to guarantee writing of misaligned data, the kernel has to read disk blocks, update them, and write them back. If you write using incomplete disk blocks (512 bytes) on block boundaries, synchronous output is slower.

Using Direct I/O

You can bypass the kernel's buffer cache completely by using the option O_DIRECT. Under this option, writes to the file take place directly from your program's buffer to the device—the data is not copied to a buffer in the kernel first. In order to use O_DIRECT you are required to transfer data in quantities that are multiples of the disk blocksize, aligned on blocksize boundaries. (The requirements for O_DIRECT use are documented in the open(2) and fcntl(2) reference pages.)

An O_DIRECT **read()** or **write()** is synchronous—control does not return until the disk operation is complete. Also, an O_DIRECT **read()** call always causes disk input—there is input cache. However, you can open a file O_DIRECT and use the file descriptor for asynchronous I/O, so that the delays are taken by an asynchronous thread (see "About Asynchronous I/O" on page 192).

Direct I/O is required when you use guaranteed-rate I/O (see "Using Guaranteed-Rate I/O" on page 230).

Direct I/O Example

The program in Example 9-1 allows you to experiment and compare buffered output, synchronized output, and direct output. An example of using it might resemble this:

```
> timex dirio -o /var/tmp/dout -m b -b 4096 -n 100
real        0.10
user        0.01
sys         0.02
> timex dirio -o /var/tmp/dout -m d -b 4096 -n 100
real        1.35
user        0.01
sys         0.06
> timex dirio -o /var/tmp/dout -m s -b 4096 -n 100
real        3.43
user        0.01
sys         0.09
```

Example 9-1   Source of Direct I/O Example

```
/*
|| dirio: program to test and demonstrate direct I/O.
||
|| dirio  [-o outfile] [ -m {b|s|d} ] [ -b bsize ] [ -n recs ] [ -i ]
||
||  -o outfile      output file pathname, default $TEMPDIR/dirio.out
||
||  -m {b|s|d}      file mode: buffered (default), synchronous, or direct
||
||  -b bsize        blocksize for each write, default 512
||
||  -n recs         how many writes to do, default 1000
||
||  -i              display info from fcntl(F_DIOINFO)
||
*/
#include <errno.h>      /* for perror() */
#include <stdio.h>      /* for printf() */
#include <stdlib.h>     /* for getenv(), malloc(3c) */
#include <sys/types.h>  /* required by open() */
#include <unistd.h>     /* getopt(), open(), write() */
#include <sys/stat.h>   /* ditto */
#include <fcntl.h>      /* open() and fcntl() */

int main(int argc, char **argv)
{
    char*       tmpdir;         /* ->name string of temp dir */
    char*       ofile = NULL;   /* argument name of file path */
    int         oflag = 0;      /* -m b/s/d result */
    size_t      bsize = 512;    /* blocksize */
    void*       buffer;         /* aligned buffer */
    int         nwrites = 1000; /* number of writes */
    int         ofd;            /* file descriptor from open() */
    int         info = 0;       /* -i option default 0 */
    int         c;              /* scratch var for getopt */
    char        outpath[128];   /* build area for output pathname */
    struct dioattr dio;

    /*
    || Get the options
    */
    while ( -1 != (c = getopt(argc,argv,"o:m:b:n:i")) )
    {
```

```
    switch (c)
    {
    case 'o': /* -o outfile */
    {
        ofile = optarg;
        break;
    }
    case 'm': /* -m mode */
    {
        switch (*optarg)
        {
        case 'b' : /* -m b buffered i.e. normal */
            oflag = 0;
            break;
        case 's' : /* -m s synchronous (but not direct) */
            oflag = O_SYNC;
            break;
        case 'd' : /* -m d direct */
            oflag = O_DIRECT;
            break;
        default:
            fprintf(stderr,"? -m %c\n",*optarg);
            return -1;
        }
        break;
    }
    case 'b' : /* blocksize */
    {
        bsize = strtol(optarg, NULL, 0);
        break;
    }
    case 'n' : /* number of writes */
    {
        nwrites = strtol(optarg, NULL, 0);
        break;
    }
    case 'i' : /* -i */
    {
        info = 1;
        break;
    }
    default:
        return -1;
    } /* end switch */
} /* end while */
```

```
    /*
    || Ensure a file path
    */
    if (ofile)
        strcpy(outpath,ofile);
    else
    {
        tmpdir = getenv("TMPDIR");
        if (!tmpdir)
            tmpdir = "/var/tmp";
        strcpy(outpath,tmpdir);
        strcat(outpath,"/dirio.out");
    }
    /*
    || Open the file for output, truncating or creating it
    */
    oflag |= O_WRONLY | O_CREAT | O_TRUNC;
    ofd = open(outpath,oflag,0644);
    if (-1 == ofd)
    {
        char msg[256];
        sprintf(msg,"open(%s,0x%x,0644)",outpath,oflag);
        perror(msg);
        return -1;
    }
    /*
    || If applicable (-m d) get the DIOINFO for the file and display.
    */
    if (oflag & O_DIRECT)
    {
        (void)fcntl(ofd,F_DIOINFO,&dio);
        if (info)
        {
        printf("dioattr.d_mem    : %8d (0x%08x)\n",dio.d_mem,dio.d_mem);
        printf("dioattr.d_miniosz: %8d (0x%08x)\n",dio.d_miniosz,dio.d_miniosz);
        printf("dioattr.d_maxiosz: %8d (0x%08x)\n",dio.d_maxiosz,dio.d_maxiosz);
        }
        if (bsize < dio.d_miniosz)
        {
            fprintf(stderr,"bsize %d too small\n",bsize);
            return -2;
        }
        if (bsize % dio.d_miniosz)
        {
            fprintf(stderr,"bsize %d is not a miniosz-multiple\n",bsize);
```

228

```
            return -3;
        }
        if (bsize > dio.d_maxiosz)
        {
            fprintf(stderr,"bsize %d too large\n",bsize);
            return -4;
        }
    }
    else
    { /* set a default alignment rule */
        dio.d_mem = 8;
    }
    /*
    || Get a buffer aligned the way we need it.
    */
    buffer = memalign(dio.d_mem,bsize);
    if (!buffer)
    {
        fprintf(stderr,"could not allocate buffer\n");
        return -5;
    }
    bzero(buffer,bsize);
    /*
    || Write the number of records requested as fast as we can.
    */
    for(c=0; c<nwrites; ++c)
    {
        if ( bsize != (write(ofd,buffer,bsize)) )
        {
            char msg[80];
            sprintf(msg,"%d th write(%d,buffer,%d)",c+1,ofd,bsize);
            perror(msg);
            break;
        }
    }
    /*
    || To level the playing field, sync the file if not sync'd already.
    */
    if (0==(oflag & (O_DIRECT|O_SYNC)))
        fdatasync(ofd);

    close(ofd);
    return 0;
}
```

Using a Delayed System Buffer Flush

When your application has both clearly defined times when all unplanned disk activity should be prevented, and clearly defined times when disk activity can be tolerated, you can use the **syssgi()** function to control the kernel's automatic disk writes.

Prior to a critical section of length s seconds that must not be interrupted by unplanned disk writes, use **syssgi()** as follows:

```
syssgi(SGI_BDFLUSHCNT,s);
```

The kernel will not initiate any deferred disk writes for s seconds. At the start of a period when disk activity can be tolerated, initiate a flush of the kernel's buffered writes with **syssgi()** as follows:

```
syssgi(SGI_SSYNC);
```

Note: This technique is meant for use in a uniprocessor. Code executing in an isolated CPU of a multiprocessor is not affected by kernel disk writes (unless a large buffer flush monopolizes a needed bus or disk controller).

Using Guaranteed-Rate I/O

Under specific conditions, your program can demand a guaranteed rate of data transfer. You would use this feature, for example, to ensure input of data for a real-time video display, or to ensure adequate disk bandwidth for high-speed telemetry capture.

About Guaranteed-Rate I/O

Guaranteed-rate I/O (GRIO) allows a program to request a specific data bandwidth to or from a filesystem. The GRIO subsystem grants the request if that much requested bandwidth is available from the hardware. For the duration of the grant, the application is assured of being able to move the requested amount of data per second. Assurance of this kind is essential to real-time data capture and digital media programming.

GRIO is a feature of the XFS filesystem support—EFS, the older IRIX file system, does not support GRIO. In addition, the optional subsystem *eoe.sw.xfsrt* must be installed. With IRIX 6.5, GRIO is supported on XLV volumes over disks or RAID systems.

GRIO is available only to programs that use direct I/O (see "Using Direct I/O" on page 225).

The concepts of GRIO are covered in sources you should examine:

IRIX Admin:Disks and Filesystems	Documents the administration of XFS and XLV in general, and GRIO volumes in particular.
grio(5)	Reference page giving an overview of GRIO use.
grio(1M)	Reference page for the administrator command for querying the state of the GRIO system.
ggd(1M)	Reference page for the GRIO grant daemon.
grio_disks(4)	Reference page for the configuration files prepared by the administrator to name GRIO devices.

About Types of Guarantees

GRIO offers two types of guarantee: a real-time (sometimes called "hard") guarantee, and a non-real-time (or "soft") guarantee. The real-time guarantee promises to subordinate every other consideration, including especially data integrity, to on-time delivery.

The two types of guarantee are effectively the same as long as no I/O read errors occur. When a read error occurs under a real-time guarantee, no error recovery is attempted— the **read()** function simply returns an error indication. Under a non-real-time guarantee, I/O error recovery is attempted, and this can cause a temporary failure to keep up to the guaranteed bandwidth.

You can qualify either type of guarantee as being Rotor scheduling, also known as Video On Demand (VOD). This indicates a particular, special use of a striped volume. These three types of guarantee, and several other options, are described in detail in *IRIX Admin:Disks and Filesystems* and in the grio(5) reference page.

About Device Configuration

GRIO is permitted on a device managed by XFS. A real-time guarantee can only be supported on the real-time subvolume of a logical volume created by XLV. The real-time subvolume differs from the more common data subvolume in that it contains only data, no file system management data such as directories or inodes. The real-time subvolume of an XLV volume can span multiple disk partitions, and can be striped.

In addition, the predictive failure analysis feature and the thermal recalibration feature of the drive firmware must be disabled, as these can make device access times unpredictable. For other requirements see *IRIX Admin:Disks and Filesystems* and the grio(5) reference page.

Creating a Real-time File

You can only request a hard guaranteed rate against a real-time disk file. A real-time disk file is identified by the fact that it is stored within the real-time subvolume of an XFS logical volume.

The file management information for all files in a volume (the directories as well as XFS management records) are stored in the data subvolume. A real-time subvolume contains only the data of real-time files. A real-time subvolume comprises an entire disk device or partition and uses a separate SCSI controller from the data subvolume. Because of these constraints, the GRIO facility can predict the data rate at which it can transfer the data of a real-time file.

You create a real-time file in the following steps, which are illustrated in Example 9-2.

1. Open the file with the options O_CREAT, O_EXCL, and O_DIRECT. That is, the file must not exist at this point, and must be opened for direct I/O (see "Using Direct I/O" on page 225).

2. Modify the file descriptor to set its extent size, which is the minimum amount by which the file will be extended when new space is allocated to it, and also to establish that the new file is a real-time file. This is done using **fcntl()** with the FS_FSSETXATTR command. Check the value returned by **fcntl()** as several errors can be detected at this point.

    The extent size must be chosen to match the characteristics of the disk; for example it might be the "stripe width" of a striped disk.

3. Write any amount of data to the new file. Space will be allocated in the real-time subvolume instead of the data subvolume because of step (2). Check the result of the first **write()** call carefully, since this is another point at which errors could be detected.

Once created, you can read and write a real-time file the same as any other file, except that it must always be opened with O_DIRECT. You can use a real-time file with asynchronous I/O, provided it is created with the PROC_SHARE_GUAR option.

Example 9-2   Function to Create a Real-time File

```
#include <sys/fcntl.h>
#include <sys/fs/xfs_itable.h>
int createRealTimeFile(char *path, __uint32_t esize)
{
   struct fsxattr attr;
   bzero((void*)&attr,sizeof(attr));
   attr.fsx_xflags = XFS_XFLAG_REALTIME;
   attr.fsx_extsize = esize;
   int rtfd = open(path, O_CREAT + O_EXCL + O_DIRECT );
   if (-1 == rtfd)
      {perror("open new file"); return -1; }
   if (-1 == fcntl(rtfd, F_FSSETXATTR, &attr) )
      {perror("fcntl set rt & extent"); return -1; }
   return rtfd; /* first write to it creates file*/
}
```

Requesting a Guarantee

To obtain a guaranteed rate, a program places a reservation for a specified part of the I/O capacity of a file or a filesystem. In the request, the program specifies

•   the file or filesystem to be used

•   the start time and duration of the reservation

•   the time unit of interest, typically 1 second

•   the amount of data required in any one unit of time

For example, a reservation might specify: starting now, for 90 minutes, 1 megabyte per second. A process places a reservation by calling either **grio_request_file()** or **grio_request_fs()** (refer to the grio_request_file(3X) and grio_request_fs(3X) reference pages).

The GRIO daemon *ggd* keeps information on the transfer capacity of all XFS volumes, as well as the capacity of the controllers and busses to which they are attached. When you request a reservation, XFS tests whether it is possible to transfer data at that rate, from that file, during that time period.

This test considers the capacity of the hardware as well as any other reservations that apply during the same time period to the same subvolume, drives, or controllers. Each reservation consumes some part of the total capacity.

When XFS predicts that the guaranteed rate can be met, it accepts the reservation. Over the reservation period, the available bandwidth from the disk is reduced by the promised rate. Other processes can place reservations against any capacity that remains.

If XFS predicts that the guaranteed rate cannot be met at some time in the reservation period, XFS returns the maximum data rate it could supply. The program can reissue the request for that available rate. However, this is a new request that is evaluated afresh.

During the reservation period, the process can use **read()** and **write()** to transfer up to the guaranteed number of bytes in each time unit. XFS raises the priority of requests as needed in order to ensure that the transfers take place. However, a request that would transfer more than the promised number of bytes within a 1-second unit is blocked until the start of the next time unit.

Releasing a Guarantee

A guarantee ends under three circumstances,

- when the process calls **grio_unreserve_bw()** (see the grio_unreserve_bw(3X) reference page)

- when the requested duration expires

- when all file descriptors held by the requesting process that refer to the guaranteed file are closed (an exception is discussed in the next topic)

When a guarantee ends, the guaranteed transfer capacity becomes available for other processes to reserve. When a guarantee expires but the file is not closed, the file remains usable for ordinary I/O, with no guarantee of rate.

Models of Parallel Computation

Chapter 10, "Models of Parallel Computation"

Provides an overview of the different models around which you can design a parallel or distributed application in Silicon Graphics systems.

Chapter 11, "Statement-Level Parallelism"

Gives an overview of the use of Power Fortran and Power C to execute do-loops across multiple CPUs.

Chapter 12, "Process-Level Parallelism"

Describes the use of IRIX processes to execute in parallel within one address space or in multiple address spaces.

Chapter 13, "Thread-Level Parallelism"

Describes the use of POSIX threads (IEEE 1003.1c) for parallel execution within a single address space.

Chapter 14, "Message-Passing Parallelism"

Describes two different facilities for distributing an application across multiple host computers: PVM and MPI.

Models of Parallel Computation

You design a program to perform computations in parallel in order to get higher performance, by bringing more hardware to bear on the problem concurrently. In order to succeed, you need to understand the hardware architecture of the target system, and also the software interfaces that are available.

The purpose of this chapter is to give a high-level overview of parallel programming models and of the hardware that they use. The parallel models are discussed in more detail in following chapters.

Parallel Hardware Models

Silicon Graphics makes a variety of systems:

- The O2, Indy, and Indigo workstations have single CPUs. Although they can perform I/O operations in parallel with computing, they can execute only one stream of instructions at a time, and time-share the CPU across all active processes.

- The CHALLENGE and Onyx systems (and their POWER versions) are symmetric multiprocessor (SMP) computers. In these systems at least 2, and as many as 36, identical microprocessors access a single, common memory and a common set of peripherals through a high-speed bus.

- The OCTANE workstation is a two-CPU SMP.

- The POWER CHALLENGEarray comprises 2 or more POWER CHALLENGE systems connected by a high-speed local HIPPI network. Each node in the array is an SMP with 2 to 36 CPUs. Nodes do not share a common memory; communication between programs in different nodes passes through sockets. However, the entire array can be administered and programmed as a single entity.

- An Origin2000 system provides nodes each containing two or four CPUs, connected in systems of 2 to 128 nodes by a high-speed connection fabric. All system memory is uniformly addressable, but there is a time penalty for the use of nonlocal memory (see "Using Origin2000 Nonuniform Memory" on page 29).

Most programs have a single thread of execution that runs as if it were in a uniprocessor, employing the facilities of a single CPU. The IRIX operating system applies CPUs to different programs in order to maximize system throughput.

You can write a program so that it makes use of more than one CPU at a time. The software interface that you use for this is the parallel programming model. The IRIX operating system gives you a variety of such interfaces. Each one is designed around a different set of assumptions about the hardware, especially the memory system.

Each model is implemented using a different library of code linked with your program. In some cases you can design a mixed-model program, but in general this is a recipe for confusion.

Parallel Programs on Uniprocessors

It might seem a contradiction, but it is possible to execute some parallel programs in uniprocessors. Obviously you would not do this expecting the best performance. However, it is easier to debug a parallel program by running it in the more predictable environment of a single CPU, on a multiprocessor or on a uniprocessor workstation. Also, you might deliberately restrict a parallel program to one CPU in order to establish a performance baseline.

Most parallel programming libraries adapt to the available hardware. They run concurrently on multiple CPUs when the CPUs are available (up to some programmer-defined limit). They run on a limited number, or even just one CPU, when necessary. For example, the Fortran programmer can control the number of CPUs used by a MIPSpro Fortran 77 program by setting environment variables before the program starts (see Chapter 11, "Statement-Level Parallelism").

Types of Memory Systems

The key memory issue for parallel execution is this: Can one process access data in memory that belongs to another concurrent process, and if so, what is the time penalty for doing so? The answer depends on the hardware architecture, and determines the optimal programming model.

Single Memory Systems

The CHALLENGE/Onyx system architecture uses a high speed system bus to connect all components of the system.

One component is the physical memory system, which plugs into the bus and is equally available to all other components. Other units that plug into the system bus are I/O adapters, such as the VME bus adapter. CPU modules containing MIPS R4000, R8000, or R10000 CPUs are also plugged into the system bus.

In the CHALLENGE/Onyx architecture, the single, common memory has these features:

- There is a single address map; that is, the same word of memory has the same address in every CPU.

- There is no time penalty for communication between processes because every memory word is accessible in the same amount of time from any CPU.

- All peripherals are equally accessible from any process.

The OCTANE workstation also uses a single, common memory that is accessible from either of its CPUs in the same amount of time.

The effect of a single, common memory is that processes running in different CPUs can share memory and can update the identical memory locations concurrently. For example, suppose there are four CPUs available to a Fortran program that processes a large array of data. You can divide a single DO-loop so that it executes concurrently on the four CPUs, each CPU working in one-fourth of the array in memory.

As another example, IRIX allows processes to map a single segment of memory into the virtual address spaces of two or more concurrent processes (see Chapter 3, "Sharing Memory Between Processes"). Two processes can transfer data at memory speeds, one putting the data into a mapped segment and the other process taking the data out. They can coordinate their access to the data using semaphores located in the shared segment (see Chapter 4, "Mutual Exclusion").

Multiple Memory Systems

In an Array system, such as a POWER CHALLENGEarray, each node is a computer built on the CHALLENGE/Onyx architecture. However, the only connection between nodes is the high-speed HIPPI bus between nodes. The system does not offer a single system memory; instead, there is a separate memory subsystem in each node. The effect is that:

- There is not a single address map. A word of memory in one node cannot be addressed at all from another node.

- There is a time penalty for some interprocess communication. When data passes between programs in different nodes, it passes over the HIPPI network, which takes longer than a memory-to-memory transfer.

- Peripherals are accessible only in the node to which they are physically attached.

Nevertheless, it is possible to design an application that executes concurrently in multiple nodes of an Array. The message-passing interface (MPI) is designed specifically for this.

Hierarchic, Nonuniform Memory Systems

The Origin2000 system uses a memory hierarchy. A certain amount of memory is a physical part of each node. The hardware creates the image of a single system memory. The memory installed in any node can be accessed from any other node as if all memory were local. However, the node number is part of the physical address of a word of memory. There is a multiple-level hierarchy of speed: memory in the same node as the CPU is accessed in the least amount of time, while memory in any other node takes an additional fraction of a microsecond to access. The time penalty depends on the relative location of the two nodes in the system.

These are the results of this design:

- There is a single address map. A word of memory can be addressed from any node.

- There is a time penalty for some accesses, depending on the node that requests the memory and the node that contains it. However, this time cost is far smaller than the cost of communicating over a socket and a network link.

- Peripherals are accessible from any node, but there is a time penalty for access to a peripheral from a node other than the one to which the peripheral is attached.

The implications of these features are explored at more length under "Using Origin2000 Nonuniform Memory" on page 29.

Parallel Execution Models

You can compare the available models for parallel programming on two features:

granularity | The relative size of the unit of computation that executes in parallel: a single statement, a function, or an entire process.

communication channel | The basic mechanism by which the independent, concurrent units of the program exchange data and synchronize their activity.

A summary comparison of the available models is shown in Table 10-1.

Table 10-1   Comparing Parallel Models

Model	Granularity	Communication
Power Fortran, IRIS POWER C	Looping statement (DO or *for* statement)	Shared variables in a single user address space.
Ada95 tasks	Ada Procedure	Shared variables in a single user address space.
POSIX threads	C function	Shared variables in a single user address space.
Lightweight UNIX processes (**sproc()**)	C function	Arena memory segment in a single user address space.
General UNIX processes (**fork()**, **exec()**)	Process	Arena segment mapped to multiple address spaces.
Shared Memory (SHMEM)	Process	Memory copy.
Parallel Virtual Machine (PVM)	Process	Memory copy within node; HIPPI network between nodes.
Message-Passing (MPI)	Process	Memory copy within node; special HIPPI Bypass interface between nodes.

Process-Level Parallelism

A UNIX process consists of an address space, a large set of process state values, and one thread of execution. The main task of the IRIX kernel is to create processes and to dispatch them to different CPUs to maximize the utilization of the system.

IRIX contains a variety of interprocess communication (IPC) mechanisms, which are discussed in Chapter 2, "Interprocess Communication." These mechanisms can be used to exchange data and to coordinate the activities of multiple, asynchronous processes within a single-memory system. (Processes running in different nodes of an Array must use one of the distributed models; see "Message-Passing Models" on page 245.)

In traditional UNIX practice, one process creates another with the system call **fork()**, which makes a duplicate of the calling process, after which the two copies execute in parallel. Typically the new process immediately uses the **exec()** function to load a new program. (The fork(2) reference page contains a complete list of the state values that are duplicated when a process is created. The exec(2) reference page details the process of creating a new program image for execution.)

IRIX also supports the system function **sproc()**, which creates a lightweight process. A process created with **sproc()** shares some of its process state values with its parent process (the sproc(2) reference page details how this sharing is specified).

In particular, a process made with **sproc()** does not have its own address space. It continues to execute in the address space of the original process. In this respect, a lightweight process is like a thread (see "Thread-Level Parallelism" on page 243). However, a lightweight process differs from a thread in two significant ways:

- A lightweight process still has a full set of UNIX state values. Some of these, for example the table of open file descriptors, can be shared with the parent process, but in general a lightweight process carries most of the state information of a process.

- Dispatch of lightweight processes is done in the kernel, and has the same overhead as dispatching any process.

The library support for statement-level parallelism is based on the use of lightweight processes (see "Statement-Level Parallelism" on page 245).

Thread-Level Parallelism

A thread is an independent execution state within the context of a larger program. The concept of a thread is well-known, but the most common formal definition of threads and their operation is provided by POSIX standard 1003.1c, "System Application Program Interface—Amendment 2: Threads Extension."

There are three key differences between a thread and a process:

- A UNIX process has its own set of UNIX state information, for example, its own effective user ID and set of open file descriptors.

    Threads exist within a process and do not have distinct copies of these UNIX state values. Threads share the single state belonging to their process.

- Normally, each UNIX process has a unique address space of memory segments that are accessible only to that process (lightweight processes created with **sproc()** share all or part of an address space).

    Threads within a process always share the single address space belonging to their process.

- Processes are scheduled by the IRIX kernel. A change of process requires two context changes, one to enter the kernel domain and one to return to the user domain of the next process. The change from the context of one process to the context of another can entail many instructions.

    In contrast, threads are scheduled by code that operates largely in the user address space, without kernel assistance. Thread scheduling can be faster than process scheduling.

The POSIX standard for multithreaded programs is supported by IRIX 6.2 with patches 1361, 1367, and 1389 installed, and in all subsequent releases of IRIX.

In addition, the Silicon Graphics implementation of the Ada95 language includes support for multitasking Ada programs—using what are essentially threads, although not implemented using the POSIX library. For a complete discussion of the Ada 95 task facility, refer to the *Ada 95 Reference Manual*, which installs with the Ada 95 compiler (GNAT) product.

Statement-Level Parallelism

The finest level of granularity is to run individual statements in parallel. This is provided using any of three language products:

- MIPSpro Fortran 77 supports compiler directives that command parallel execution of the bodies of DO-loops. The MIPSpro POWER Fortran 77 product is a preprocessor that automates the insertion of these directives in a serial program.

- MIPSpro Fortran 90 supports parallelizing directives similar to MIPSpro Fortran 77, and the MIPSpro POWER Fortran 90 product automates their placement.

- MIPSpro POWER C supports compiler pragmas that command parallel execution of segments of code. The IRIS POWER C analyzer automates the insertion of these pragmas in a serial program.

In all three languages, the run-time library—which provides the execution environment for the compiled program—contains support for parallel execution. The compiler generates library calls. The library functions create lightweight processes using **sproc()**, and distribute loop iterations among them.

The run-time support can adapt itself dynamically to the number of available CPUs. Alternatively, you can control it—either using program source statements, or using environment variables at execution time—to use a certain number of CPUs.

Statement-level parallel support is based on using common variables in memory, and so it can be used only within the bounds of a single-memory system, a CHALLENGE system or a single node in a POWER CHALLENGEarray system.

Message-Passing Models

One way to design a parallel program is to think of each thread of execution as operating in an independent address space, communicating with other threads by exchanging discrete units of data as messages through an abstract, formal interface for message exchange.

The threads of a program designed in this way can be distributed over different computers. Three message-passing execution models are supported by Silicon Graphics systems. Each defines and implements a formal, abstract model for data exchange. Two of the models allow a computation to be distributed across the nodes of a multiple-memory system, without having to reflect the system configuration in the source code. The programming models are:

- Shared Memory Model (SHMEM)

- Message-Passing Interface (MPI)

- Parallel Virtual Machine (PVM)

All three models are briefly summarized in the following topics, and are discussed in more detail in Chapter 14, "Message-Passing Parallelism." Support for all three is included in the Message-Passing Toolkit (MPT) product. For an overview of MPT, see this URL:

http://www.cray.com/products/software/mpt/mpt.html

Shared Memory (SHMEM) Model

The SHMEM library has been used for some time on Cray systems and is now available for all Silicon Graphics multiprocessors as part of the MPT. A program built on SHMEM is a process-level parallel program. Each process runs in a separate address space. The SHMEM library routines are used to exchange data, and to coordinate execution, between the processes.

SHMEM routines support remote data transfer through *put* operations, which transfer data to a different process, and *get* operations, which transfer data from a different process. Other operations supported include data broadcast and data reduction; barrier synchronization; as well as atomic memory updates such as a fetch-and-increment on remote or local data objects.

SHMEM operations are all memory-to-memory, and as a result have extremely high bandwidth and low latency. However, a SHMEM-based program cannot be distributed across multiple systems.

Message-Passing Interface (MPI) Model

MPI is a standard programming interface for the construction of a portable, parallel application in Fortran 77 or in C, especially when the application can be decomposed into a fixed number of processes operating in a fixed topology (for example, a pipeline, grid, or tree). MPI has wide use on many large computers.

A highly tuned, efficient implementation of MPI is part of the MPT. Within a single system, MPI messages are moved memory-to-memory. Between nodes of an Silicon Graphics Array system, MPI messages are passed over a HIPPI network. Latency and bandwidth are intermediate between memory-to-memory data exchange and socket-based network communication.

Parallel Virtual Machine (PVM) Model

PVM is an integrated set of software tools and libraries that emulates a general-purpose, flexible, heterogeneous, concurrent computing framework on interconnected computers of varied architecture. Using PVM, you can create a parallel application that executes as a set of concurrent processes on a set of computers that can include uniprocessors, multiprocessors, and nodes of Array systems.

Like MPI, PVM has wide use on many types of supercomputer, including Cray systems. An implementation of PVM is included in the MPT. PVM is discussed in more detail under Chapter 14, "Message-Passing Parallelism."

Statement-Level Parallelism

You can use statement-level parallelism in three language packages: Fortran 77, Fortran 90, and C. This execution model is unique in that you begin with a normal, serial program, and you can always return the program to serial execution by recompiling. Every other parallel model requires you to plan and write a parallel program from the start.

Products for Statement-Level Parallelism

Software support for statement-level parallelism is available from Silicon Graphics and from independent vendors.

Silicon Graphics Support

The parallel features of the three languages from Silicon Graphics are documented in detail in the manuals listed in Table 11-1.

Table 11-1   Documentation for Statement-Level Parallel Products

Manual	Document Number	Contents
C Language Reference Manual	007-0701-*nnn*	Covers all pragmas, including parallel ones.
IRIS Power C User's Guide	007-0702-*nnn*	Use of Power C source analyzer to place pragmas automatically.
MIPSpro Fortran 77 Programmer's Guide	007-2361-*nnn*	General use of Fortran 77, including parallelizing assertions and directives.
MIPSpro Power Fortran 77 Programmer's Guide	007-2363-*nnn*	Use of the Power Fortran source analyzer to place directives automatically.
MIPSpro Fortran 90 Programmer's Guide	007-2761-*nnn*	General use of Fortran 90, including parallelizing assertions and directives.
MIPSpro Power Fortran 90 Programmer's Guide	007-2760-*nnn*	Use of the Power Fortran 90 source analyzer to place directives automatically.

Products from Other Vendors

In addition to these products from Silicon Graphics, the High Performance Fortran (HPF) compiler from the Portland Group is a compiler for Fortran 90 augmented to the HPF standard. It supports automatic parallelization. (Refer to http://www.pgroup.com for more information).

The FORGE products from Applied Parallel Research (APRI) contain a Fortran 77 source analyzer that can insert parallelizing directives, although not the directives supported by MIPSpro Fortran 77. (Refer to http://www.apri.com for more information.)

Creating Parallel Programs

In each of the three languages, the language compiler supports explicit statements that command parallel execution (**#pragma** lines for C; directives and assertions for Fortran). However, placing these statements can be a demanding, error-prone task. It is easy to create a suboptimal program, or worse, a program that is incorrect in subtle ways. Furthermore, small changes in program logic can invalidate parallel directives in ways that are hard to foresee, so it is difficult to modify a program that has been manually made parallel.

For each language, there is a source-level program analyzer that is sold as a separate product (IRIS POWER C, MIPSpro Power Fortran 77, MIPSpro Power Fortran 90). The analyzer identifies sections of the program that can safely be executed in parallel, and automatically inserts the parallelizing directives. After any logic change, you can run the analysis again, so that maintenance is easier.

The source analyzer makes conservative assumptions about the way the program uses data. As a result, it often is unable to find all the potential parallelism. However, the analyzer produces a detailed listing of the program source, showing each segment that could or could not be parallelized, and why. Directed by this listing, you insert source assertions that give the analyzer more information about the program.

The method of creating an optimized parallel program is as follows:

1. Write a complete application that runs on a single processor.

2. Completely debug and verify the correctness of the program in serial execution.

3. Apply the source analyzer and study the listing it produces.

4. Add assertions to the source program. These are not explicit commands to parallelize, but high-level statements that describe the program's use of data.

5. Repeat steps 3 and 4 until the analyzer finds as much parallelism as possible.

6. Run the program on a single-memory multiprocessor.

When the program requires maintenance, you make the necessary logic changes and, simultaneously, remove any assertions about the changed code—unless you are certain that the assertions are still true of the modified logic. Then repeat the preceding procedure from step 2.

Managing Statement-Parallel Execution

The run-time library for all three languages is the same, *libmp*. It is documented in the mp(3) reference page. *libmp* uses IRIX lightweight processes to implement parallel execution (see Chapter 12, "Process-Level Parallelism").

When a parallel program starts, the run-time support creates a pool of lightweight processes using the **sproc()** function. Initially the extra processes are blocked, while one process executes the opening passage of the program. When execution reaches a parallel section, the run-time library code unblocks as many processes as necessary. Each process begins to execute the same block of statements. The processes share global variables, while each allocates its own copy of variables that are local to one iteration of a loop, such as a loop index.

When a process completes its portion of the work of that parallel section, it returns to the run-time library code, where it picks up another portion of work if any work remains, or suspends until the next time it is needed. At the end of the parallel section, all extra processes are suspended and the original process continues to execute the serial code following the parallel section.

Controlling the Degree of Parallelism

You can specify the number of lightweight processes that are started by a program. In IRIS POWER C, you can use #*pragma numthreads* to specify the exact number of processes to start, but it is not a good idea to embed this number in a source program. In all implementations, the run-time library by default starts enough processes so there is one for each CPU in the system. That default is often too high, since typically not all CPUs are available for one program.

252

The run-time library checks an environment variable, MP_SET_NUM_THREADS, for the number of processes to start. You can use this environment variable to choose the number of processes used by a particular run of the program, thereby tuning the program's requirements to the system load. You can even force a parallelized program to execute on a single CPU when necessary.

MIPSpro Fortran 77 and MIPSpro Fortran 90 also recognize additional environment variables that specify a range of process numbers, and use more or fewer processes within this range as system load varies. (See the *Programmer's Guide* for the language for details.)

At certain points the multiple processes must wait for one another before continuing. They do this by waiting in a busy loop for a certain length of time, then by blocking until they are signaled. You can specify the amount of time that a process should spend spinning before it blocks, using either source directives or an environment variable (see the *Programmer's Guide* for the language for system functions for this purpose).

Choosing the Loop Schedule Type

Most parallel sections are loops. The benefit of parallelization is that some iterations of the loop are executed in one CPU, concurrent with other iterations of the same loop in other CPUs. But how are the different iterations distributed across processes? The languages support four possible methods of scheduling loop iterations, as summarized in Table 11-2.

Table 11-2   Loop Scheduling Types

Schedule	Purpose
SIMPLE	Each process executes $\lfloor N/P \rfloor$ iterations starting at $Q*(\lfloor N/P \rfloor)$. First process to finish takes the remainder chunk, if any.
DYNAMIC	Each process executes C iterations of the loop, starting with the next undone chunk unit, returning for another chunk until none are left undone.
INTERLEAVE	Each process executes C iterations at $C*Q$, $C*2Q$, $C*3Q$...
GSS	Each process executes chunks of decreasing size, $(N/2P)$, $(N/4P)$,...

The variables used in Table 11-2 are as follows:

N Number of iterations in the loop, determined from the source or at run-time.

P Number of available processes, set by default or by environment variable (see "Controlling the Degree of Parallelism" on page 252).

Q Number of a process, from 0 to *N*-1.

C "Chunk" size, set by directive or by environment variable.

The effects of the scheduling types depend on the nature of the loops being parallelized. For example:

- The SIMPLE method works well when *N* is relatively small. However, unless *N* is evenly divided by *P*, there will be a time at the end of the loop when fewer than *P* processes are working, and possibly only one.

- The DYNAMIC and INTERLEAVE methods allow you to set the chunk size to control the span of an array referenced by each process. You can use this to reduce cache effects. When *N* is very large so that not all data fits in memory, INTERLEAVE may reduce the amount of paging compared to DYNAMIC.

- The guided self-scheduling (GSS) method is good for triangular matrices and other algorithms where loop iterations become faster toward the end.

You can use source directives or pragmas within the program to specify the scheduling type and chunk size for particular loops. Where you do not specify the scheduling, the run-time library uses a default method and chunk size. You can establish this default scheduling type and chunk size using environment variables.

Distributing Data

In any statement-level parallel program, memory cache contention can harm performance. This subject is covered under "Dealing With Cache Contention" on page 34.

When a statement-parallel program runs in an Origin2000 or Onyx2 system, the location of the program's data can affect performance. These issues are covered at length under "Using Origin2000 Nonuniform Memory" on page 29.

Process-Level Parallelism

The process is the traditional unit of UNIX execution. The concept of the process (and its relationship to the concept of a thread) are covered under "Process-Level Parallelism" on page 242. The purpose of this chapter is to review how you can use IRIX processes to perform parallel processing in a single program.

Using Multiple Processes

In general, you can create a new process for each unit of work that your program could do in parallel. The processes can share the address space of the original program, or each can have its own address space. You design the processes so that they coordinate work and share data using any and all of the interprocess communication (IPC) features discussed in Part II, "Interprocess Communication."

Software products from Silicon Graphics use process-level parallelism. For example, the IRIS Performer graphics library normally creates a separate lightweight process to manage the graphics pipe in parallel with rendering work. The run-time library for statement-level parallelism creates a pool of lightweight processes and dispatches them to execute parts of loop code in parallel (see "Managing Statement-Parallel Execution" on page 252).

Process Creation and Share Groups

The most important system functions you use to create and manage processes are summarized in Table 12-1.

Table 12-1   Commands and System Functions for Process Management

Function Name	Purpose and Operation
npri(1)	Command to run a process at a specified nondegrading priority.
runon(1)	Command to run a process on a specific CPU.
fork(2)	Create a new process with a private address space.
pcreate(3C)	Create a new process with a private address space running a designated program with specified arguments.
sproc(2)	Create a new process in the caller's address space using a private stack.
sprocsp(2)	Create a new process in the caller's address space using a preallocated stack area.
prctl(2)	Query and set assorted process attributes.
sysmp(2)	Query multiprocessor status and assign processes to CPUs.
syssgi(2)	Query process virtual and real memory use, and other operations.

You can initiate a program at a specified nondegrading priority (explained under "Process Scheduling" on page 260) using *npri*. You can initiate a program running on a specific CPU of a multiprocessor using *runon*. Both attributes—the assigned priority and the assigned CPU—are inherited by any child processes that the program creates.

Process Creation

The process that creates another is called the *parent* process. The processes it creates are *child* processes, or siblings. The parent and its children together are a *share group*. IRIX provides special services to share groups. For example, you can send a signal to all processes in a share group.

The **fork()** function is the traditional UNIX way of creating a process. The new process is a duplicate of the parent process, running in a duplicate of the parent's address space. Both execute the identical program text; that is, both processes "return" from the **fork()** call. Your code can distinguish them by the return code, which is 0 in the child process, but in the parent is the new process ID.

The **sproc()** and **sprocsp()** functions create a lightweight process. The difference between these calls is that **sproc()** allocates a new memory segment to serve as the stack for the new process. You use **sprocsp()** to specify a stack segment that you have already allocated—for example, a block of memory that you allocate and lock against paging using **mpin()**.

The **sproc()** calls take as an argument the address of the function that the new process should execute. The new process begins execution in that function, and when that function returns, the process is terminated. Read the sproc(2) reference page for details on the flags that specify which process attributes a child process shares with its parent, and for other comparisons between **fork()** and **sproc()**.

Note: The **sproc()** and **sprocsp()** functions are not available for use in a threaded program (see Chapter 13, "Thread-Level Parallelism"). The pthreads library uses lightweight processes to implement threading, and has to control the creation of processes. Also, when your program uses the MPI library (see Chapter 14, "Message-Passing Parallelism"), the use of **sproc()** and **sprocsp()** can cause problems.

257

Process Management

Certain system functions give you some control over the processes you create. The **prctl()** function offers a variety of operations. These are some of the most useful:

PR_MAXPROCS   Query the system limit on processes per user (also available from **sysconf(**_SC_CHILD_MAX**)**, see sysconf(2).

PR_MAXPPROCS   Query the maximum number of CPUs that are available to the calling process and its children. This reflects both the system hardware and reservations made on CPUs, but does not reflect system load.

PR_GETNSHARE   Query the number of processes in the share group with the calling process.

PR_GETSTACKSIZE   Query the maximum size of the stack segment of the calling process. For the parent process this reflects the system limit (also available from **getrlimit(**RLIMIT_STACK**)**, see getrlimit(2)). For a process started by **sprocsp()**, the size of the allocated stack.

PR_SETSTACKSIZE   Set an upper limit on stack growth for the calling process and for child processes it creates in the future.

PR_RESIDENT   Prevent the calling process from being swapped out. This has no connection to paging, but to swapping out an entire, inactive process under heavy system load.

The **sysmp()** function gives a privileged process information about and control over the use of a multiprocessor. Some of the operations it provides are as follows:

MP_NPROCS   Number of CPUs physically in the system.

MP_NAPROCS   Number of CPUs available to the scheduler; should be the same as **prctl(**PR_MAXPPROCS**)**.

MP_MUSTRUN   Assign the calling process to run on a specific CPU.

MP_MUSTRUN_PID   Assign a specified other process (typically a just-created child process) to run on a specific CPU.

MP_GETMUSTRUN
MP_GETMUSTRUN_PID   Query the must-run assignment of the calling process or of a specified process.

MP_RUNANYWHERE
MP_RUNANYWHERE_PID   Allow the calling process, or a specified process, to run on any CPU.

The *runon* command (see "Process Creation" on page 257 and runon(1)) initiates the parent process of a program running on a specific CPU. Any child processes also runs on that CPU unless the parent reassigns them to run anywhere, or to run on a different CPU, using **sysmp()**. The use of restricted CPUs and assigned CPUs to get predictable real-time performance is discussed at length in the *REACT Real-Time Programmer's Guide*.

The **syssgi()** function has a number of interesting uses but only one of interest for managing processes: **syssgi(**SGI_PROCSZ**)** returns the virtual and resident memory occupancy of the calling process.

Process "Reaping"

A parent process should not terminate while its child processes continue to run. When it does so, the parent process of each child becomes 1, the *init* process. This causes problems if a child process should loop or hang. The functions you use to collect (the technical term is to "reap") the status of child processes are summarized in Table 12-2.

Table 12-2   Functions for Child Process Management

Function Name	Purpose and Operation
wait(2)	Function to block until a child stops or terminates, and to receive the cause of its change of status.
waitpid(2)	POSIX extension of **wait()** which allows more selectivity and returns more information.
wait3(2)	BSD extension of **wait()** that allows you to poll for terminated children without suspending.
waitid(2)	Function to suspend until one of a selected set of status changes occurs in one or more child processes.

When the parent process has nothing to do after starting the child processes, it can loop on **wait()** until **wait()** reports no more children exist; then it can exit.

Sometimes it is necessary to handle child termination and other work, and the parent cannot suspend. In this case the parent can treat the termination of a child process as an asynchronous event, and trap it in a signal handler for SIGCLD (see "Catching Signals" on page 118). The wait(2) reference page has extensive discussion of the three methods (BSD, SVR4, and POSIX) for handling this situation, with example code for each.

Process Scheduling

There are two different approaches to setting the scheduling priorities of a process, one compatible with IRIX and BSD, the other POSIX compliant.

Controlling Scheduling With IRIX and BSD-Compatible Facilities

The IRIX compatible and BSD compatible scheduling operations are summarized in Table 12-3.

Table 12-3   Commands and Functions for Scheduling Control

Function Name	Purpose and Operation
schedctl(2)	Query and set IRIX process scheduling attributes.
getpriority(2)	Return the scheduling priority of a process or share group.
setpriority(2)	Set the priority of a process or process group.
nice(1)	Run a program at a positive or negative increment from normal priority.
renice(1)	Alter the priority of a running process by a positive or negative increment.

For BSD compatibility, use the *nice* and *renice* commands to alter priorities, and within a program use **getpriority()** and **setpriority()** to query and set priorities. These commands and functions use priority numbers ranging from -20 through 0 to +20, with lower arithmetic values having superior access to the CPU.

Only the IRIX **schedctl()** function gives you complete access to a variety of operations related to process scheduling. Some of the key operations are as follows:

NDPRI	Set a nondegrading priority for the calling process (see text).
GETNDPRI	Query the nondegrading priority of the calling process.
SETMASTER	Set the master process of a share group. By default the parent process is the master process, but it can transfer that honor.
SCHEDMODE, SGS_SINGLE	Cause all processes in the share group to be suspended except the master process (set with SETMASTER).

SCHEDMODE, SGS_GANG	Cause all processes in the share group to be scheduled as a "gang," with all running concurrently.
SCHEDMODE, SGS_FREE	Schedule the share group in the default fashion.

A program started interactively inherits a scheduling discipline based on degrading priorities. That is, the longer the process executes without voluntarily suspending, the lower its dispatching priority becomes. This strategy keeps a runaway process from monopolizing the hardware. However, you may have a CPU-intensive application that needs a predictable execution rate. This is the purpose of nondegrading priorities set with **schedctl(**NDPRI**)** or with the *npri* command (see the npri(1) reference page).

There are three bands of nondegrading priorities, designated by symbolic names declared in *sys/schedctl.h*:

- A real-time band from NDPHIMAX to NDPHIMIN. System daemons and real-time programs run in this band, which has higher priority than any interactive process.

- A normal band from NDPNORMMAX to NDPNORMMIN. These values have the same priority as interactive programs. Processes at these priorities compete with interactive processes, but their priorities do not degrade with time.

- A batch band from NDPLOMAX to NDPLOMIN. Processes at these priorities receive available CPU time and are scheduled from a batch queue.

Tip: The IRIX priority numbers are inverted, in the sense that numerically smaller values have superior priority. For example. NDPHIMAX is 30 and NDPHIMIN is 39. However, as long as you declare priority values using symbolic expressions, the numbers work out correctly. For example, the statement

```
#define NDPHIMIDDLE NDPHIMIN+((NDPHIMAX-NDPHIMIN)/2)
```

produces a "middle" value of 35, as it should.

When you create a cooperating group of processes, it is important that they all execute at the same time, provided there are enough CPUs to handle all the members of the group that are ready to run. This minimizes the time that members of the share group spend waiting for each other to release locks or semaphores.

Use **schedctl()** to initiate "gang" scheduling for the share group. IRIX attempts to schedule all processes to execute at the same time, when possible.

Note: Through IRIX 6.2, **schedctl()** also supported a scheduling mode called "deadline scheduling." This scheduling mode is being removed and will not be supported in the future. Do not design a program based on the use of deadline scheduling.

Controlling Scheduling With POSIX Functions

The POSIX compliant functions to control process scheduling are summarized in Table 12-4.

Table 12-4   POSIX Functions for Scheduling

Function Name	Purpose and Operation
sched_getparam(2) sched_setparam(2)	Query and change the POSIX scheduling priority of a process.
sched_getscheduler(2) sched_setscheduler(2)	Query and change the POSIX scheduling policy and priority of a process.
sched_get_priority_max(2) sched_get_priority_min(2)	Query the maximum (most use of CPU) and minimum (least use) priority numbers for use with **sched_getparam()**.
sched_get_rr_interval(2)	Query the timeslice interval of the round-robin scheduling policy.
sched_yield(2)	Let other processes of the same priority execute.

Use the functions **sched_get_priority_max()** and **sched_get_priority_min()** to get the ranges of priority numbers you can use. Use **sched_setparam()** to change priorities. POSIX dispatching priorities are nondegrading. (Note that in a program that links with the pthreads library, these same function names are library functions that return thread scheduling priority numbers unrelated to process scheduling.)

Tip: The POSIX scheduling priority values reported by these functions and declared in *sched.h* are not numerically the same as the bands supported by **schedctl()** and declared in *sys/schedctl.h*. The POSIX numbers are numerically higher for superior priority. However, the POSIX range is functionally (but not numerically) equivalent to the "normal" range supported by **schedctl()** (NDPNORMMAX to NDPNORMMIN).

POSIX scheduling uses one of two scheduling policies, strict FIFO and round-robin, which are described in detail in the sched_setscheduler(2) reference page. The round-robin scheduler, which rotates processes of equal priority on a time-slice basis, is the default. You can query the time-slice interval with **sched_get_rr_interval()**. You can change both the policy and the priority by using **sched_setscheduler()**.

Self-Dispatching Processes

Often, each child process has a particular role to play in the application, and the function that you name to **sproc()** represents that work. The child process stays in that function until it terminates.

Another design is possible. In some applications, you may have to manage a flow of many relatively short activities that should be done in parallel. However, the **sproc()** function has considerable overhead. It is inefficient to continually create and destroy child processes. You do not want to create a new child process for each small activity and destroy it afterward. Instead, you can create a pool containing a small number of processes. When a piece of work needs to be done, you can dispatch one process to do it. The fragmentary code in Example 12-1 shows the general approach.

Example 12-1   Partial Code to Manage a Pool of Processes

```
typedef void (*workFunc)(void *arg);
struct oneSproc {
    struct oneSproc *next;        /* -> next oneSproc ready to run */
    workFunc calledFunc;          /* -> function the sproc is to call */
    void *callArg;                /* argument to pass to the called func */
    usema_t *sprocDone;           /* optional sema to post on completion */
    usema_t *sprocWait;           /* sproc waits for work here */
} sprocList[NUMSPROCS];
usema_t *readySprocs;             /* count represents sprocs ready to work */
uslock_t sprocListLock;           /* mutex control of sprocList head */
struct oneSproc *sprocList;       /* -> first ready oneSproc */
/*
|| Put a oneSproc structure on the ready list and sleep on it.
|| Called by a child process when its work is done.
*/
void sprocSleep(struct oneSproc *theSproc)
{
    ussetlock(sprocListLock);        /* acquire exclusive rights to sprocList */
    theSproc->next = sprocList;  /* put self on the list */
    sprocList = theSproc;
```

```
        usunsetlock(sprocListLock);  /* release sprocList */
        usvsema(readySprocs);            /* notify master, at least 1 on the list */
        uspsema(theSproc->sprocWait);/* sleep until master posts me */
}
/*
|| Body of a general-purpose child process. The argument, which must
|| be declared void* to match the sproc() prototype, is the oneSproc
|| structure that represents this process.   The contents of that
|| struct, in particular sprocWait, are initialized by the parent.
*/
void childBody(void *theSprocAsVoid)
{
    struct oneSproc *mySproc = (struct oneSproc *)theSprocAsVoid;
    /* here one could establish signal handlers, etc. */
    for(;;)
    {
        sprocSleep(mySproc);       /* wait for work to do */
        mySproc->calledFunc(mySproc->callArg);  /* do the work */
        if (mySproc->sprocDone)   /* if a completion sema is given, */
            usvsema(mySproc->sprocDone); /* ..post it */
    }
}
/*
|| Acquire a oneSproc structure from the ready list, waiting if necessary.
|| Called by the master process as part of dispatching a sproc.
*/
struct oneSproc *getSproc()
{
    struct oneSproc *theSproc;
    uspsema(readySprocs);          /* wait until at least 1 sproc is free */
    ussetlock(sprocListLock);        /* acquire exclusive rights to sprocList */
    theSproc = sprocList;          /* get address of first free oneSproc */
    sprocList = theSproc->next;  /* make next in list, the head of list */
    usunsetlock(sprocListLock);  /* release sprocList */
    return theSproc;
}
/*
|| Start a function going asynchronously. Called by master process.
*/
void execFunc(workFunc toCall, void *callWith, usema_t *done)
```

```
{
    struct oneSproc *theSproc = getSproc();
    theSproc->calledFunc = toCall;      /* set address of func to exec */
    theSproc->callArg = callWith;       /* set argument to pass */
    theSproc->sprocDone = done;            /* set sema to post on completion */
    usvsema(theSproc->sprocWait);       /* wake up sleeping process */
}
```

Parallelism in Real-Time Applications

In real-time programs such as aircraft or vehicle simulators, separate processes are used to divide the work of the simulation and distribute it onto multiple CPUs. In these demanding applications, the programmer frequently uses IRIX facilities to

- reserve one or more CPUs of a multiprocessor for exclusive use by the application

- isolate the reserved CPUs from all interrupts

- assign specific processes to execute on specific, reserved CPUs

These facilities are described in detail in the *REACT Real-Time Programmer's Guide* (007-2499-*nnn*). Also covered in that book is the use of the Frame Scheduler, an alternate process scheduler. The normal process scheduling algorithm of the IRIX kernel attempts to keep all CPUs busy and to keep all processes advancing in a fair manner. This algorithm is in conflict with the stringent needs of a real-time program, which needs to dedicate predictable amounts of hardware capacity to its processes, without regard to fairness.

The Frame Scheduler seizes one or more CPUs of a multiprocessor, isolates them, and executes a specified set of processes on each CPU in strict rotation. The Frame Scheduler has much lower overhead than the normal IRIX scheduler, and it has features designed for real-time work, including detection of overrun (when a scheduled process does not complete its work in the necessary time) and underrun (when a scheduled process fails to execute in its turn).

At this writing there are no real-time applications that use multiple nodes of an Array system.

Thread-Level Parallelism

IRIX 6.5 conforms to ISO/IEC 9945-1:1996 and UNIX 98; that is, it supports POSIX threads, or pthreads.

This chapter contains the following main topics:

- "Overview of POSIX Threads" on page 268 summarizes the similarities and differences of pthreads and processes.

- "Compiling and Debugging a Pthread Application" on page 269 covers compiling and debugging tools.

- "Creating Pthreads" on page 271 covers the process of creating a pthread with the desired attributes.

- "Executing and Terminating Pthreads" on page 274 discusses how threads initialize themselves and how you synchronize on thread termination.

- "Using Thread-Unique Data" on page 277 tells how to define variables that have a unique value in each thread.

- "Pthreads and Signals" on page 278 discusses the pthread-specific details of signal handling (see "Signals" on page 113 for the general information).

- "Scheduling Pthreads" on page 280 covers scheduling priorities and policies.

- "Synchronizing Pthreads" on page 282 details the use of mutexes and condition variables.

Overview of POSIX Threads

A *thread* is an independent execution state; that is, a set of machine registers, a call stack, and the ability to execute code. When IRIX creates a process, it also creates one thread to execute that process. However, you can write a program that creates many more threads to execute in the same address space. For a comparison of pthreads to processes, see "Thread-Level Parallelism" on page 243.

POSIX threads are similar in some ways to IRIX lightweight processes made with **sproc()**. You use pthreads in preference to lightweight processes for two main reasons: portability and performance. A program based on pthreads is normally easier to port from another vendor's equipment than a program that depends on a unique facility such as **sproc()**. Table 13-1 summarizes some of the differences between pthreads and lightweight processes.

Table 13-1   Comparison of Pthreads and Processes

Attribute	POSIX Threads	Lightweight Processes	UNIX Processes
Source portability	Standard interface, portable between vendors	**sproc()** is unique to IRIX	**fork()** is a UNIX standard
Creation overhead	Relatively small	Moderately large	Quite large
Block/Unblock (Dispatch) Overhead	Few microseconds	Many microseconds	Many microseconds
Address space	Shared	Shared, or copy on write, or separate	Separate
Memory-mapped files and arenas	Shared	Shared, or copy on write, or separate	Explicit sharing only
Mutual exclusion objects	Mutexes, condition variables, and read-write locks; POSIX semaphores; IRIX semaphores and locks	IRIX semaphores and locks; POSIX semaphores	IRIX semaphores and locks; POSIX semaphores
Files, pipes, and I/O streams	Shared single-process file table	Shared or separate file table	Separate file table

Table 13-1 (continued)   Comparison of Pthreads and Processes

Attribute	POSIX Threads	Lightweight Processes	UNIX Processes
Signal masks and signal handlers	Each thread has a mask but handlers are shared	Each process has a mask and its own handlers	Each process has a mask and its own handlers
Resource limits	Single-process limits	Single-process limits	Limits apply to each process separately
Process ID	One PID applies to all threads	PID per process plus share-group PID	PID per process

It takes relatively little time to create or destroy a pthread, as compared to creating a lightweight process. Threads share all resources and attributes of a single process (except for the signal mask; see "Pthreads and Signals" on page 278). If you want each executing entity to have its own set of file descriptors, or if you want to make sure that one entity cannot modify data shared with another entity, you must use lightweight processes or normal processes.

Compiling and Debugging a Pthread Application

A pthread application is a C or a C++ program that uses some of the POSIX pthreads functions. In order to use these functions, and in order to access the thread-safe versions of the standard I/O macros, you must include the proper header files and link with the pthreads library. You can debug and analyze the compiled program using some of the tools available for IRIX.

Compiling Pthread Source

The header files related to pthreads functions are summarized in Table 13-2.

Table 13-2   Header Files Related to Pthreads

Header	Primary Contents
errno.h	System error codes returned by pthreads functions.
pthread.h	Pthread functions and special pthread data types.
sched.h	The *sched_param* structure and related functions used in setting thread priorities.
stdio.h	Standard stream I/O macros, including thread-safe versions.
sys/types.h	IRIX and standard data types.
limits.h	Some POSIX constants such as _POSIX_THREAD_THREADS_MAX.
unistd.h	Constants used when calling **sysconf()** to query POSIX limits (see the sysconf(3) reference page).

It is recommended that the thread-safe options be enabled at compile time using the feature test macro, _POSIX_C_SOURCE (see intro(3) for details). For example, to compile these options, use this command:

```
cc -D_POSIX_C_SOURCE=199506L app.c -llib0 -llib1 ... -lpthread
```

You can use pthreads with a program compiled to any of the supported execution models: *-32* for compatibility with older systems, *-n32* for 64-bit data and 32-bit addressing, or *-64* for 64-bit addressing.

The pthreads functions are defined in the library *libpthread.so*. Link with this library using the *-lpthread* compiler option, which should be the last library on the command line. The compiler chooses the correct library based on the execution model: */usr/lib/libpthread.so*, */usr/lib32/libpthread.so*, and */usr/lib64/libpthread.so*.

Note: A pthread program is a program that links with *libpthread*. Do not link with *libpthread* unless you intend to use the pthread interface, because *libpthread* replaces many standard library functions.

Debugging Pthread Programs

The *dbx* debugger and Workshop Debugger have been extended for use with threaded programs. See the dbx(1M) reference page and the documentation for Workshop Debugger for more details.

Creating Pthreads

You create a pthread by calling **pthread_create()**. One argument to this function is a thread attribute object of type *pthread_attr_t*. You pass a null address to request a thread having default attributes, or you prepare an attribute object to reflect the features you want the thread to have. You can use one attribute object to create many pthreads.

Functions related to attribute objects and pthread creation are summarized in Table 13-3 and described in the following sections:

- "Initial Detach State" on page 272
- "Initial Scheduling Scope, Priority, and Policy" on page 272
- "Thread Stack Allocation" on page 273

Table 13-3   Functions for Creating Pthreads

Function	Purpose
pthread_attr_init(3P)	Initialize a *pthread_attr_t* object to default settings.
pthread_attr_setdetachstate(3P)	Set the automatic-detach attribute.
pthread_attr_setinheritsched(3P)	Specify whether scheduling attributes come from the attribute object or are inherited from the creating thread.
pthread_attr_setschedparam(3P)	Set the starting thread priority.
pthread_attr_setschedpolicy(3P)	Set the scheduling policy.
pthread_attr_setscope(3P)	Set the scheduling scope.
pthread_attr_setstacksize(3P)	Set the stack size attribute.
pthread_attr_setguardsize(3P)	Set the stack guard size attribute.
pthread_attr_setstackaddr(3P)	Set the address of memory to use as a stack (when you allocate the stack for the new thread).

Table 13-3 (continued)   Functions for Creating Pthreads

Function	Purpose
pthread_attr_destroy(3P)	Uninitialize a *pthread_attr_t* object.
pthread_create(3P)	Create a new thread based on an attribute object, or with default attributes.

Initial Detach State

Detaching means that the pthreads library frees up resources held by the thread after it terminates (see "Joining and Detaching" on page 277). There are three ways to detach a thread:

- automatically when the thread terminates

- explicitly by calling **pthread_join()**

- explicitly by calling **pthread_detach()**

You can use **pthread_attr_setdetachstate()** to specify that a thread should be detached automatically when it terminates. Do this when you know that the thread will not be joined or detached by an explicit function call.

Initial Scheduling Scope, Priority, and Policy

You can specify an initial thread scheduling scope by calling **pthread_attr_setscope()** and passing one of the scope constants (PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS) in the *pthread_attr_t* object. By default, process scope is selected and scheduling is performed by the thread runtime, but thread scheduling by the kernel is provided with the system scope attribute. System scope threads run at real-time policy and priority and may be created only by privileged users.

You can specify an initial thread priority in a *struct sched_param* object in memory (the structure is declared in *sched.h*). Set the desired priority in the *sched_priority* field. Pass the structure to **pthread_attr_setschedparam()**.

You can specify an initial scheduling policy by calling **pthread_attr_setschedpolicy()**, passing one of the policy constants SCHED_FIFO or SCHED_RR.

The **pthread_attr_setinheritsched()** function is used to specify, in the attribute object, whether a new thread's scheduling policy and priority should be taken from the attribute object, or whether they should be inherited from the thread that creates the new thread. When you set an attribute object for inheritance, the scheduling policy and priority in the attribute object are ignored.

Scheduling scope, priorities, and policies are described in "Scheduling Pthreads" on page 280.

Thread Stack Allocation

Each pthread has an execution stack area in memory. By default, **pthread_create()** allocates stack space from dynamic memory, and automatically releases it when the thread terminates.

You use **pthread_attr_setstacksize()** to specify the size of this stack area. You cannot specify a stack size less than a minimum. A pthread process can find the minimum by calling **sysconf()** with _SC_THREAD_STACK_MIN (see the sysconf(3C) reference page).

Threads may overrun their stack area. By default, a thread's stack is created with guard protection, and extra memory is allocated at the overflow end of the stack as a buffer. If an application overflows into this buffer, an exception results (a SIGSEGV signal is delivered to the thread).

The *guardsize* attribute controls the size of the guard area for the created thread's stack and protects against overflow of the stack pointer. The *guardsize* attribute is set using **pthread_attr_setguardsize()**.

Note: Because thread stack space is taken from dynamic memory, the allocation is charged against the process virtual memory limit, not the process stack size limit as you might expect.

273

Executing and Terminating Pthreads

The functions for managing the progress of a thread are summarized in Table 13-4 and described in the following sections:

- "Getting the Thread ID" on page 275
- "Initializing Static Data" on page 275
- "Setting Event Handlers" on page 275
- "Terminating a Thread" on page 276
- "Joining and Detaching" on page 277

Table 13-4   Functions for Managing Thread Execution

Function	Purpose
pthread_atfork(3P)	Register functions to handle the event of a **fork()**.
pthread_cancel(3P)	Request cancellation of a specified thread.
pthread_cleanup_push(3P)	Register function to handle the event of thread termination.
pthread_cleanup_pop(3P)	Unregister and optionally call termination handler.
pthread_detach(3P)	Detach a terminated thread.
pthread_exit(3P)	Explicitly terminate the calling thread.
pthread_join(3P)	Wait for a thread to terminate and receive its return value.
pthread_once(3P)	Execute initialization function once only.
pthread_self(3P)	Return the calling thread's ID.
pthread_equal(3P)	Compare two thread IDs for equality.
pthread_setcancelstate(3P)	Permit or block cancellation of the calling thread.
pthread_setcanceltype(3P)	Specify deferred or asynchronous cancellation.
pthread_testcancel(3P)	Permit cancellation to take place, if it is pending.

Getting the Thread ID

Call **pthread_self()** to get the thread ID of the calling thread. A thread can use this thread ID when changing its own scheduling priority, for example (see "Scheduling Pthreads" on page 280).

Initializing Static Data

Your program may use static data that should be initialized exactly once. The code can be entered by multiple threads, and might be entered concurrently. How can you ensure that only one thread will perform the initialization?

One answer is to create a variable of type *pthread_once_t*, statically initialized to the value PTHREAD_ONCE_INIT. Call **pthread_once(),** passing the addresses of the variable and of an initialization function. The pthreads library ensures that the initialization function is called only once, and that any other threads calling **pthread_once()** for this variable wait until the first thread completes the initialization function. See Example 13-1.

Example 13-1   One-Time Initialization

```
pthread_once_t first_time_flag = PTHREAD_ONCE_INIT;
elaborate_struct_t uninitialized; /* thing to initialize */
void elaborate_initializer(void); /* function to do it */
int subroutine(...)
{
   ...
   pthread_once(&first_time_flag, elaborate_initializer);
   ...
}
```

Setting Event Handlers

A thread can establish functions that are called when it terminates and when the process forks.

Call **pthread_cleanup_push()** to register a function that is to be called in the event that the current thread terminates, either by exiting or by cancellation. Call **pthread_cleanup_pop()** to retract this registration and, optionally, to call the handler. These functions are often used in library code, with the push operation done on entry to the library and the pop done upon exit from the library. The push and pop operations are in fact implemented partly as macro code. For this reason, calls to them must be strictly balanced—a pop for each push—and each push/pop pair must appear in a single C lexical scope. A nonstructured jump such as a longjmp (see the setjmp(3) reference page) or goto can cause unexpected results.

Call **pthread_atfork()** to register three handlers related to a UNIX **fork()** call. The first handler executes just before the **fork()** takes place; the second executes just after the **fork()** in the parent process; the third executes just after the **fork()** in the child process.

The **fork()** operation creates a new process with a copy of the calling process's address space, including any locked mutexes or semaphores. Typically, the new process immediately calls **exec()** to replace the address space with a new program. When this is the case, there is no need for **pthread_atfork()** (see the exec(2) and fork(2) reference pages). However, if the new process continues to execute with the inherited address space, including perhaps calls to library code that uses pthreads, it may be necessary for the library code to reinitialize data in the address space of the child process. You can do this in the fork event handlers.

Terminating a Thread

A thread begins execution in the function that is named in the **pthread_create()** call. When it returns from that function, the thread terminates. A thread can terminate earlier by calling **pthread_exit()**. In either case, the thread returns a value of type *void**.

One thread can request early termination of another by calling **pthread_cancel()**, passing the thread ID of the target thread. A thread can protect itself against cancellation using two built-in switches:

- The **pthread_setcancelstate()** function lets you postpone cancellation indefinitely (PTHREAD_CANCEL_DISABLE) or permit cancellation (PTHREAD_CANCEL_ENABLE).

- The **pthread_setcanceltype()** function lets you decide when cancellation will take place, if it is allowed at all. Cancellation can happen whenever it is requested (PTHREAD_CANCEL_ASYNCHRONOUS) or only at defined points (PTHREAD_CANCEL_DEFERRED).

When you prevent cancellation by setting PTHREAD_CANCEL_DISABLE, a cancellation request is blocked but remains pending until the thread terminates or changes its cancellation state.

The initial cancellation state of a thread is PTHREAD_CANCEL_ENABLE and the type is PTHREAD_CANCEL_DEFERRED. In this state, a cancellation request is blocked until the thread calls a function that is a defined cancellation point. The functions that are cancellation points are listed in the pthread_setcanceltype(3P) reference page. A thread can explicitly permit cancellation by calling **pthread_testcancel()**.

Joining and Detaching

Sometimes you do not care when threads terminate—your program starts a set of threads, and they continue until the entire program terminates.

In other cases, threads are created and terminated as the program runs. One thread can wait for another to terminated by calling **pthread_join()**, specifying the thread ID. The function does not return until the specified thread terminates. The value the specified thread passed to **pthread_exit()** is returned. At this time, your program can release any resources that you associate with the thread, for example, stack space (see "Thread Stack Allocation" on page 273).

The **pthread_join()** function also detaches the terminated thread. If your program does not use **pthread_join()**, you must arrange for terminated threads to be detached in some other way. One way is by specifying automatic detachment when the threads are created (see "Initial Detach State" on page 272). Another is to call **pthread_detach()** at any time after creating the thread, including after it has terminated.

If your program creates threads and lets them terminate, but does not detach them, resources will be used up and eventually an error will occur when trying to create a thread.

Using Thread-Unique Data

In some designs, especially modules of library code, you need to store data that is both

- unique to the calling thread
- persistent from one function call to another

Normally, the only data that is unique to a thread is the contents of its local variables on the stack, and these do not persist between calls. However, the pthreads library provides a way to create persistent, thread-unique data. The functions for this are summarized in Table 13-5.

Table 13-5   Functions for Thread-Unique Data

Function	Purpose
pthread_key_create(3P)	Create a key.
pthread_key_delete(3P)	Delete a key.
pthread_getspecific(3P)	Retrieve this thread's value for a key.
pthread_setspecific(3P)	Set this thread's value for a key.

Your program calls **pthread_key_create()** to define a new storage *key*. Once created, a key may be used by all threads to identify a unique key *value*.

Any thread can use **pthread_getspecific()** to retrieve that thread's unique value stored under a key. A thread can fetch only its own value, which is the value stored by this same thread using **pthread_setspecific()**. The initial stored value is NULL.

When you create a key, you can specify a destructor function that is called automatically when a thread terminates. The destructor is called while the key is valid and the key value for the terminating thread is not NULL. The destructor receives the thread's key value as its argument.

Pthreads and Signals

For a general overview of signal concepts and numbers, see "Signals" on page 113 and the signal(5) reference page. IRIX supports three different signal facilities: BSD signals, SVR4 signals, and POSIX signals. When you are writing a pthreads program, you should use only the POSIX signal facilities (see "POSIX Signal Facility" on page 120).

Setting Signal Masks

Each thread has a signal mask that specifies the signals it is willing to receive (see "Signal Blocking and Signal Masks" on page 117). In a program that is linked with the pthreads library, this should be changed using **pthread_sigmask()**. Each thread inherits the signal mask of the thread that calls **pthread_create()**. Typically you set an initial mask in the first thread, so that it can be inherited by all other threads.

Note: In IRIX, you can use **sigprocmask()** instead of **pthread_sigmask()**, but it may not be portable to other systems.

When a signal is directed to a specific thread that is blocking the signal, the signal remains pending on the thread until that thread unblocks it. When a signal is directed to a process, it is delivered to the first thread that is not blocking that signal. If all threads block that signal, the signal remains pending on the process until some thread unblocks it or the process terminates.

A thread can find out which signals are pending by calling **sigpending()**. This function returns a mask showing the set of signals pending on the process as a whole or for the calling thread; that is, the signals that could be delivered to the calling thread if they were not blocked.

Setting Signal Actions

When a signal is delivered, some action is taken. You specify what that action should be using the **sigaction()** function. These actions are set on a process-wide basis, *not* individually for each thread. Although each thread has a private signal mask, signal actions are shared with all threads in the process. See "Signal Handling Policies" on page 118 for details.

Receiving Signals Synchronously

You can design a program to receive signals in a synchronous manner instead of asynchronously. To do this, set a mask that blocks all the signals that are to be received synchronously. Then call one of the following three functions:

sigwait(3)   Suspend until one of a specified set of signals is generated, then return the signal number.

sigwaitinfo(3)   Like **sigwait()**, but returns additional information about the signal.

sigtimedwait(3)   Like **sigwaitinfo()**, but also returns after a specified time has elapsed if no signal is received.

Using these functions you can write a thread that treats signals as a stream of events to be processed. This is generally the safest program model, much easier to work with than the asynchronous model of signal delivery.

Scheduling Pthreads

The pthreads scheduling algorithm is controlled by three variables: a scope, policy, and priority for each thread. These variables are set initially when the thread is created (see "Initial Scheduling Scope, Priority, and Policy" on page 272), but policy and priority can be modified while the thread is running.

Contention Scope

The scheduling contention scope of a pthread (see pthread_attr_setscope(3P)) determines the set of threads that it competes against for resources.

System scope threads compete with all other threads on the system and can be created only by privileged users. These threads are used in programs when some form of guaranteed (that is, real-time) response is required. Their scheduling parameters directly affect how the system treats them. In addition to the usual scheduling attributes, they can select a CPU on which to run using the **pthread_setrunon_np()** call.

Process scope threads compete within the process and their scheduling attributes are used by the pthread library to select which threads to run on a pool of kernel entities. The size of the pool is determined dynamically, but may be influenced using the **pthread_setconcurrency()** call.

Process scope threads generally require fewer resources than system scope threads because they can share kernel resources. The kernel entities themselves share a common set of scheduling attributes which privileged users can change using the process scheduling interfaces (see sched_setscheduler(2) and sched_setparam(2)). For further details, see the pthreads(5) reference page.

The functions used in scheduling are summarized in Table 13-6 and described in the following sections:

- "Scheduling Policy" on page 281
- "Scheduling Priority" on page 282

Table 13-6   Functions for Schedule Management

Function	Purpose
pthread_getschedparam(3P)	Get a thread's policy and priority.
pthread_setschedparam(3P)	Set a thread's policy and priority.
sched_get_priority_max(3C)	Return the maximum priority value.
sched_get_priority_min(3C)	Return the minimum priority value.
sched_yield(2)	Relinquish the processor.
pthread_setconcurrency(3P)	Modify concurrency level.
pthread_getconcurrency(3P)	Check the concurrency level.
pthread_setrunon_np(3P)	Select a CPU to run a system scope thread.
pthread_getrunon_np(3P)	Query a named CPU's affinity.

Scheduling Policy

There are two scheduling policies in this implementation: first-in-first-out (SCHED_FIFO) and the default round-robin (SCHED_RR). SCHED_FIFO and SCHED_RR are similar. The round-robin scheduler ensures that after a thread has used a certain maximum amount of time, it is moved to the end of the queue of threads of the same priority, and can be preempted by other threads.

The details of scheduling are discussed in the pthread_attr_setschedpolicy(3P) reference page.

Scheduling Priority

Threads are ordered by priority values, with a small number representing a low priority, and a larger number representing a higher priority. Threads with higher priorities are chosen to execute before threads with lower priorities.

The **sched_get_priority_max()** and **sched_get_priority_min()** functions return the highest and lowest priority numbers for a given policy. There are at least 32 priority values and the lowest is greater than or equal to 0.

A thread can set another's priority and scheduling policy, using **pthread_setschedparam()**. A simple function to set a specified priority on the current thread is shown in Example 13-2.

Example 13-2   Function to Set Own Priority

```
#include <sched.h> /* struct sched_param */
void setMyPriority(int newP)
{
   pthread_t myTid = pthread_self();
   int policy;
   struct sched_param sp;
   (void) pthread_getschedparam(myTID,&policy,&sp);
   sp.sched_priority = newP;
   (void) pthread_setschedparam(myTID,policy,&sp);
}
```

Synchronizing Pthreads

Threads using a common address space must cooperate and coordinate their use of shared variables. IRIX provides many mechanisms for coordinating threads, including:

- POSIX semaphores for general coordination and resource management (see "POSIX Facilities for Mutual Exclusion" on page 82).

- POSIX or SVR4 message queues (see Chapter 6, "Message Queues").

- POSIX mutex objects, which allow threads to gain exclusive use of a shared variable (see "Mutexes" on page 283).

- POSIX condition variables, which allow a thread to wait when a controlling predicate is false (see "Condition Variables" on page 286).

- POSIX read-write locks, which allow one thread exclusive access to locked data to write it or read access to locked data for several threads (see "Read-Write Locks" on page 292).

- IRIX semaphores and locks.

- SVR4 semaphores.

Tip: Synchronization between processes (such as POSIX process-shared mechanisms, IRIX IPC, and SVR4 IPC) is more costly than synchronization between threads (POSIX process-private mechanisms). So where possible, use the process-private mechanisms.

Mutexes

A mutex is a software object that arbitrates the right to modify some shared variable, or the right to execute a critical section of code. A mutex can be owned by only one thread at a time; other threads trying to acquire it wait. Mutexes are intended to be lightweight and owned only for a short time.

Preparing Mutex Objects

When a thread wants to modify a variable that it shares with other threads, or execute a critical section, the thread claims the associated mutex. This can cause the thread to wait until it can acquire the mutex. When the thread has finished using the shared variable or critical code, it releases the mutex. If two or more threads claim the mutex at once, one acquires the mutex and continues, while the others are blocked until the mutex is released.

A mutex has attributes that control its behavior. The pthreads library contains several functions used to prepare a mutex for use. These functions are summarized in Table 13-7.

Table 13-7   Functions for Preparing Mutex Objects

Function	Purpose
pthread_mutexattr_init(3P)	Initialize a *pthread_mutexattr_t* with default attributes.
pthread_mutexattr_destroy(3P)	Uninitialize a *pthread_mutexattr_t*.
pthread_mutexattr_getprotocol(3P)	Query the priority protocol.
pthread_mutexattr_setprotocol(3P)	Set the priority protocol choice.
pthread_mutexattr_getprioceiling(3P)	Query the minimum priority.
pthread_mutexattr_setprioceiling(3P)	Set the minimum priority.
pthread_mutexattr_getpshared(3P)	Query the process-shared attribute.
pthread_mutexattr_setpshared(3P)	Set the process-shared attribute.
pthread_mutexattr_gettype(3P)	Get the mutex type.
pthread_mutexattr_settype(3P)	Set the mutex type.
pthread_mutex_init(3P)	Initialize a mutex object.
pthread_mutex_destroy(3P)	Uninitialize a mutex object.

A mutex must be initialized before use. You can do this in one of three ways:

- Static assignment of the constant PTHREAD_MUTEX_INITIALIZER.

- Calling **pthread_mutex_init()** passing NULL instead of the address of a mutex attribute object.

- Calling **pthread_mutex_init()** passing a *pthread_mutexattr_t* object that you have set up with attribute values.

The first two methods initialize the mutex to default attributes.

Four attributes can be set in a *pthread_mutexattr_t*. You can set the priority inheritance protocol using **pthread_mutexattr_setprotocol()** to one of three values:

PTHREAD_PRIO_NONE   The mutex has no effect on the thread that acquires it. This is the default.

PTHREAD_PRIO_PROTECT   The thread holding the mutex runs at a priority at least as high as the highest priority of any mutex that it currently holds.

PTHREAD_PRIO_INHERIT   The thread holding the mutex runs at a priority at least as high as the highest priority of any thread blocked on that mutex.

If a thread acquires a mutex and then is suspended (for example, because its time slice is up), other threads can be blocked waiting for the mutex. The PTHREAD_PRIO_PROTECT protocol prevents this. Using **pthread_mutexattr_setprioceiling(),** you set a priority higher than normal for the mutex. A thread that acquires the mutex runs at this higher priority while it holds the mutex.

Another problem is that when a low-priority thread has acquired a mutex, and a thread with higher priority claims the mutex and is blocked, a "priority inversion" takes place— a higher-priority thread is forced to wait for one of lower priority. The PTHREAD_PRIO_INHERIT protocol prevents this—when a thread of higher priority blocks, the thread holding the mutex has its priority boosted during the time it holds the mutex.

Tip: PTHREAD_PRIO_NONE uses a faster code path than the other two priority options for mutexes.

By default, only threads within a process share a mutex. Using **pthread_mutexattr_setpshared()**, you can allow any thread (from any process) with access to the mutex memory location to use the mutex. Enable mutex sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Note: The PTHREAD_PRIO_INHERIT attribute is not available with **pthread_mutexattr_setpshared()**.

By default, no error checking is performed on threads that attempt to use a mutex. For example, a thread that attempts to lock a mutex that it already owns deadlocks. Using **pthread_mutexattr_settype()** with PTHREAD_MUTEX_ERRORCHECK allows you to have the lock call return an error instead. If recursive mutexes are required, PTHREAD_MUTEX_RECURSIVE enables recursive mutexes.

Using Mutexes

The functions for claiming, releasing, and using mutexes are summarized in Table 13-8.

Table 13-8   Functions for Using Mutexes

Function	Purpose
pthread_mutex_lock(3P)	Claim a mutex, blocking until it is available.
pthread_mutex_trylock(3P)	Test a mutex and acquire it if it is available, else return an error.
pthread_mutex_unlock(3P)	Release a mutex.
pthread_mutex_getprioceiling(3P)	Query the minimum priority of a mutex.
pthread_mutex_setprioceiling(3P)	Set the minimum priority of a mutex.

To determine where mutexes should be used, examine the memory variables and other objects (such as files) that can be accessed from multiple threads. Create a mutex for each set of shared objects that are used together. Ensure that the code acquires the proper mutex before it modifies the shared objects. You acquire a mutex by calling **pthread_mutex_lock()**, and release it with **pthread_mutex_unlock()**. When a thread must not be blocked, it can use **pthread_mutex_trylock()** to test the mutex and lock it only if it is available.

Condition Variables

A condition variable provides a way in which a thread can wait for an event (or condition) defined by the program, to be satisfied. Condition variables use mutexes to synchronize the wait and wakeup operations.

Preparing Condition Variables

Like mutexes and threads themselves, condition variables are supplied with a mechanism of attribute objects (*pthread_condattr_t* objects) and static and dynamic initializers. (Only the condition variable for the process-shared attribute can be initialized in this implementation.) The functions for initializing one are summarized in Table 13-9.

Table 13-9 Functions for Preparing Condition Variables

Function	Purpose
pthread_condattr_init(3P)	Initialize a *pthread_condattr_t* to default attributes.
pthread_condattr_destroy(3P)	Uninitialize a *pthread_condattr_t*.
pthread_condattr_getpshared(3P)	Get the process-shared attribute.
pthread_condattr_setpshared(3P)	Set the process-shared attribute.
pthread_cond_init(3P)	Initialize a condition variable based on an attribute object.
pthread_cond_destroy(3P)	Uninitialize a condition variable.

A condition variable must be initialized before use. You can do this in one of three ways:

- Static assignment of the constant PTHREAD_COND_INITIALIZER.

- Calling **pthread_cond_init()** passing NULL instead of the address of an attribute object.

- Calling **pthread_cond_init()** passing a *pthread_condattr_t* object that you have set up with attribute values.

The first two methods initialize the variable to default attributes.

By default, only threads within a process share a condition variable. Using **pthread_condattr_setpshared()**, you can allow any thread (from any process) with access to the condition variable memory location to use the condition variable. Enable condition variable sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Using Condition Variables

A condition variable is a software object that represents a test of a Boolean condition. Typically the condition changes because of a software event such as "other thread has supplied data." A thread establishes that it needs to wait by first evaluating the condition. The thread that satisfies the condition signals the condition variable, releasing one or all threads that are waiting.

For example, a thread might acquire a mutex that represents a shared resource. While holding the mutex, the thread finds that the shared resource is not complete. The thread does three things:

- Wait, giving up the mutex so that some other thread can renew the shared resource.

- Wait until the condition is signalled.

- Wake-up, re-acquiring the mutex for the shared resource and rechecking the condition.

These three actions are combined into one using a condition variable. The functions used with condition variables are summarized in Table 13-10.

Table 13-10   Functions for Using Condition Variables

Function	Purpose
pthread_cond_wait(3P)	Wait on a condition variable.
pthread_cond_timedwait(3P)	Wait on a condition variable, returning with an error after a time limit expires.
pthread_cond_signal(3P)	Signal that an awaited event has occurred, releasing at least one waiting thread.
pthread_cond_broadcast(3P)	Signal that an awaited event has occurred, releasing all waiting threads.

The **pthread_cond_wait()** and **pthread_cond_timedwait()** functions require both a condition variable and a mutex that is owned by the calling thread. The mutex is released and the wait begins. When the event is signalled (or the time limit expires), the mutex is reacquired, as if by a call to **pthread_mutex_lock()**.

288

The POSIX standard explicitly warns that it is possible in some cases for a conditional wait to return before the event has been signalled. For this reason, a conditional wait should always be coded in a loop that tests the shared resource for the needed status. These principles are suggested in the code in Example 13-3, which is modeled after an example in the POSIX 1003.1c standard.

Example 13-3   Use of Condition Variables

```c
#include <assert.h>
#include <pthread.h>
typedef int listKey_t;
typedef struct element_s { /* list element */
   listKey_t key;
   struct element_s *next;
   int busyFlag;
   pthread_cond_t notBusy; /* event of no-longer-in-use */
} element_t;
typedef struct listHead_s { /* list head and mutex */
   pthread_mutex_t mutList; /* right to modify the list */
   element_t *head;
} listHead_t;
/*
|| Internal function to find an element in a list, returning NULL
|| if the key is not in the list.
|| A returned element could be in use by another thread (busy).
|| The caller is assumed to hold the list mutex, otherwise
|| the returned value could be made invalid at any time.
*/
static element_t *scanList(listHead_t* lp, listKey_t key)
{
   element_t *ep;
   for (ep=lp->head; (ep) ; ep=ep->next)
   {
      if (ep->key == key) break;
   }
   return ep;
}
/*
|| Public function to find a key in a list, wait until the element
|| is no longer busy, mark it busy, and return it.
*/
element_t *getFromList(listHead_t* lp, listKey_t key)
{
   element_t *ep;
   pthread_mutex_lock(&lp->mutList); /* lock list against changes */
```

```
        while ((ep=scanList(lp,key)) && (ep->busyFlag))
        {
            pthread_cond_wait(&ep->notBusy, &lp->mutList); /* (A) */
        }
        if (ep) ep->busyFlag = 1;
        pthread_mutex_unlock(&lp->mutList);
        return ep;
}
/*
|| Public function to release an element returned by getFromList().
*/
void freeInList(listHead_t* lp, element_t *ep)
{
    assert(ep->busyFlag);
    pthread_mutex_lock(&lp->mutList); /* lock list to prevent races */
    ep->busyFlag = 0;
    pthread_cond_signal(&ep->notBusy);
    pthread_mutex_unlock(&lp->mutList);
}
/*
|| Public function to delete a list element returned by getFromList().
*/
void deleteInList(listHead_t* lp, element_t *ep)
{
    element_t **epp;
    assert(ep->busyFlag);
    pthread_mutex_lock(&lp->mutList);
    for (epp = &lp->head; ep != *epp; epp = &((*epp)->next))
    { /* finding anchor of *ep in list */ }
    *epp = ep->next; /* remove *ep from list */
    ep->busyFlag = 0;
    pthread_cond_broadcast(&ep->notBusy);
    pthread_mutex_unlock(&lp->mutList);
    pthread_cond_destroy(&ep->notBusy);
    free(ep);
}
```

The functions in Example 13-3 implement part of a simple library for managing lists. In a list head, *mutList* is a mutex object that represents the right to modify any part of the list. The elements of a list can be "busy," that is, in use by some thread. An element that is busy has a nonzero *busyFlag* field.

The **getFromList()** function looks up an element in a specified list, makes that element busy, and returns it. The function begins by acquiring the list mutex. This ensures that the list cannot change while the function is searching the list, and makes it legitimate for the function to change the busy flag in an element.

When it finds the element, the function might discover that the element is already busy. In this case, it must wait for the event "element is no longer busy," which is represented by the condition variable *notBusy* in the element. In order to wait for this event, **getFromList()** calls **pthread_cond_wait()** passing its list mutex and the condition variable (point "(A)" in the code). This releases the list mutex so that other threads can acquire the list and do their work on other elements.

When any thread wants to release the use of a list element, it calls **freeInList()**. After clearing the busy flag in the list element, **freeInList()** announces that the event "element is no longer busy" has occurred, by calling **pthread_cond_signal()**.

This call releases a thread that is waiting at point "(A)." If there is more than one thread waiting for the same element, the first in priority order is released. The released thread re-acquires the list mutex and resumes execution. The first thing it does is repeat its search of the list for the desired key and, on finding the element again, test it again for busyness. This repetition is needed because it is possible to get spurious returns from a condition variable.

When a thread wants to delete a list element, it gets the list element by calling **getFromList()**. This ensures that the element is busy, so no other thread is using it. Then the thread calls **deleteInList()**. This function changes the list, so it begins by acquiring the list mutex. Then it can safely modify the list pointers. It scans up the list looking for the pointer that points to the target element. It removes the target element from the list by copying its *next* field to replace the pointer to the target element.

With the element removed from the list, **deleteInList()** calls **pthread_cond_broadcast()** to wake up all threads—not just the first thread—that might be waiting for the element to become nonbusy. Each of these threads resumes execution at point "(A)" by attempting to re-acquire the list mutex. However, **deleteInList()** is still holding the list mutex. The mutex is released; then the other threads can resume execution following point "(A)," but this time when they search the list, the desired key is no longer found.

Meanwhile, **deleteInList()** uses **pthread_cond_destroy()** to release any memory that the pthreads library might have associated with the condition variable, before releasing the list element object itself.

Read-Write Locks

A read-write lock is a software object that gives one thread the right to modify some data, or multiple threads the right to read that data. A read-write lock can be owned for write or for read. If acquired for write, only one thread can own it and other threads must wait. If acquired for read, other threads wishing to acquire it for write must wait, but multiple readers can own the lock at the same time.

Preparing Read-Write Locks

When a thread wants to modify or read data shared by several threads, the thread claims the associated lock. This can cause the thread to wait until it can acquire the lock. When the thread has finished reading or writing the shared data, it releases the lock.

A read-write lock has attributes that control its behavior. The pthreads library contains several functions used to prepare a lock for use. These functions are summarized in Table 13-11.

Table 13-11   Functions for Preparing Read-Write Locks

Function	Purpose
pthread_rwlockattr_init(3P)	Initialize a *pthread_rwlockattr_t* with default attributes.
pthread_rwlockattr_destroy(3P)	Uninitialize a *pthread_rwlockattr_t*.
pthread_rwlockattr_getpshared(3P)	Query the process-shared attribute.
pthread_rwlockattr_setpshared(3P)	Set the process-shared attribute.
pthread_rwlock_init(3P)	Initialize a rwlock object based on a *pthread_rwlockattr_t*.
pthread_rwlock_destroy(3P)	Uninitialize a read-write lock object.

A read-write lock must be initialized before use. You can do this in one of three ways:

- Static assignment of the constant PTHREAD_RWLOCK_INITIALIZER.

- Calling **pthread_rwlock_init()** passing NULL instead of the address of a read-write lock attribute object.

- Calling **pthread_rwlock_init()** passing a *pthread_rwlockattr_t* object that you have set up with attribute values.

The first two methods initialize the read-write lock to default attributes.

By default, only threads within a process share a read-write lock. Using **pthread_rwlockattr_setpshared()**, you can allow any thread (from any process) with access to the read-write lock memory location to claim the read-write lock. Enable read-write lock sharing by changing the default PTHREAD_PROCESS_PRIVATE attribute to PTHREAD_PROCESS_SHARED.

Using Read-Write Locks

The functions for claiming, releasing, and using read-write locks are summarized in Table 13-12.

Table 13-12   Functions for Using Read-Write Locks

Function	Purpose
pthread_rwlock_wrlock(3P)	Apply a write lock, blocking until it is available.
pthread_rwlock_trywrlock(3P)	Test a write lock and acquire it if it is available, else return an error.
pthread_rwlock_rdlock(3P)	Apply a read lock, blocking until it is available.
pthread_rwlock_tryrdlock(3P)	Test a read lock and acquire it if it is available, else return an error.
pthread_rwlock_unlock(3P)	Release a read or a write lock.

To determine where read-write locks should be used, examine the memory variables and other objects (such as files) that can be accessed from multiple threads. Create a read lock for each set of shared objects that are used together. Ensure that the code acquires the write lock before it modifies the shared objects. You acquire a write lock by calling **pthread_rwlock_wrlock()**, and release it with **pthread_rwlock_unlock()**. A read lock is acquired by calling **pthread_rwlock_rdlock()**, and released with **pthread_rwlock_unlock()**. When a thread must not be blocked, it can use **pthread_rwlock_trywrlock()** or **pthread_rwlock_tryrdlock()** to test the lock and apply it only if it is available.

Message-Passing Parallelism

In a message-passing model, your parallel application consists of multiple, independent processes, each with its own address space. Each process shares data and coordinates with the others by passing messages, using a formal interface. The formal interface makes the program independent of the medium over which the message go. The processes of the program can be in a single computer, with messages moving via memory-to-memory copy; but it is possible to distribute the program in different machines, with messages passing over a network.

The Message-Passing Toolkit package supports three libraries on which you can build a message-passing application. The Cray Shared-Memory (SHMEM) library supports message passing in a single system. Message-Passing Interface (MPI) and Parallel Virtual Machine (PVM) models support distribution. High-level overviews of these are given under "Message-Passing Models" on page 245.

Choosing a Message-Passing Model

There are five considerations in choosing among the three message-passing models: compatibility, portability, scope, latency, and bandwidth.

Compatibility   If you are starting with an existing program that uses one of the three models, or if you want to reuse code from such a program, or if you personally are highly familiar with one of the three, you will likely choose that model in order to minimize development time.

Portability   The SHMEM library is portable among all Silicon Graphics/Cray systems, including both IRIX and UNICOS/MK. However, it is not supported on systems of other types. Both MPI and PVM are industry standard libraries that are widely available in public-domain implementations.

Scope   The SHMEM library can be used only within a single multiprocessor such as Cray T3E or an Origin2000. You can use MPI or PVM to distribute a program across all nodes in an Silicon Graphics Array, or across a heterogeneous network.

Latency   Latency, the start-up delay inherent in sending any one message of any size, is the shortest in SHMEM. MPI within a single system is a close second (both use memory-to-memory copy).

                  MPI latency across an Array using the Silicon Graphics-proprietary HIPPI Bypass is an order of magnitude greater. MPI or PVM latency using ordinary HIPPI or TCP/IP is greater still.

Bandwidth   The rate at which the bits of a message are sent is the highest in SHMEM and MPI within a single system. MPI bandwidth over a HIPPI link is next, followed by PVM.

If you require the highest performance within a *single* Cray or Silicon Graphics system, use SHMEM. For the highest performance in an Array system linked with HIPPI, use MPI. Use PVM only when compatibility or portability is an overriding consideration.

Choosing Between MPI and PVM

When your program must be able to use the resources of multiple systems, you choose between MPI and PVM. In many ways, MPI and PVM are similar:

- Each is designed, specified, and implemented by third parties that have no direct interest in selling hardware.

- Support for each is available over the Internet at low or no cost.

- Each defines portable, high-level functions that are used by a group of processes to make contact and exchange data without having to be aware of the communication medium.

- Each supports C and Fortran 77.

- Each provides for automatic conversion between different representations of the same kind of data so that processes can be distributed over a heterogeneous computer network.

Another difference between MPI and PVM is in the support for the "topology" (the interconnect pattern: grid, torus, or tree) of the communicating processes. In MPI, the group size and topology are fixed when the group is created. This permits low-overhead group operations. The lack of run-time flexibility is not usually a problem because the topology is normally inherent in the algorithmic design. In PVM, group composition is dynamic, which requires the use of a "group server" process and causes more overhead in common group-related operations.

Other differences are found in the design details of the two interfaces. MPI, for example, supports asynchronous and multiple message traffic, so that a process can wait for any of a list of message-receive calls to complete and can initiate concurrent sending and receiving. MPI provides for a "context" qualifier as part of the "envelope" of each message. This permits you to build encapsulated libraries that exchange data independently of the data exchanged by the client modules. MPI also provides several elegant data-exchange functions for use by a program that is emulating an SPMD parallel architecture.

PVM is possibly more suitable for distributing a program across a heterogeneous network that includes both uniprocessors and multiprocessors, and includes computers from multiple vendors. When the application runs in the environment of a Silicon Graphics Array system, MPI is the recommended interface.

Differences Between PVM and MPI

This section discusses the main differences between PVM and MPI from the programmer's perspective, focusing mainly on PVM functions that are not available in MPI.

Although to a large extent the library calls of MPI and PVM provide similar functionality, some PVM calls do not have a counterpart in MPI, and vice versa. Additionally, the semantics of some of the equivalent calls are inherently different for the two libraries (owing, for example, to the concept of dynamic groups in PVM). Hence, the process of converting a PVM program into an MPI program can be straightforward or complicated, depending on the particular PVM calls in the program and how they are used. For many PVM programs, conversion is straightforward.

In addition to a message-passing library, PVM also provides the concept of a *parallel virtual machine session*. A user starts this session before invoking any PVM programs; in other words, PVM provides the means to establish a parallel environment from which a user invokes a parallel program.

Additionally, PVM includes a *console*, which is useful for monitoring and controlling the states of the machines in the *virtual machine* and the state of execution of a PVM job. Most PVM console commands have corresponding library calls.

The MPI standard does not provide mechanisms for specifying the initial allocation of processes to an MPI computation and their binding to physical processors. Mechanisms to do so at load time or run time are left to individual vendor implementations. However, this difference between the two paradigms is not, by itself, significant for most programs, and should not affect the port from PVM to MPI.

The chief differences between the current versions of PVM and MPI libraries are as follows:

- PVM supports dynamic spawning of tasks, whereas MPI does not.

- PVM supports dynamic process groups; that is, groups whose membership can change dynamically at any time during a computation. MPI does not support dynamic process groups.

  MPI does not provide a mechanism to build a group from scratch, but only from other groups that have been defined previously. Closely related to groups in MPI are communicators, which specify the communication context for a communication operation and an ordered process group that shares this communication context. The chief difference between PVM groups and MPI communicators is that any PVM task can join/leave a group independently, whereas in MPI all communicator operations are collective.

- A PVM task can add or delete a host from the virtual machine, thereby dynamically changing the number of machines a program runs on. This is not available in MPI.

- A PVM program (or any of its tasks) can request various kinds of information from the PVM library about the collection of hosts on which it is running, the tasks that make up the program, and a task's parent. The MPI library does not provide such calls.

- Some of the collective communication calls in PVM (for instance, **pvm_reduce()**) are nonblocking. The MPI collective communication routines are not required to return as soon as their participation in the collective communication is complete.

- PVM provides two methods of signaling other PVM tasks: sending a UNIX signal to another task, and notifying a task about an event (from a set of predefined events) by sending it a message with a user-specified tag that the application can check. A PVM call is also provided through which a task can kill another PVM task. These functions are not available in MPI.

- A task can leave/unenroll from a PVM session as many times as it wants, whereas an MPI task must initialize/finalize exactly once.

- A PVM task need not explicitly enroll: the first PVM call enrolls the calling task into a PVM session. An MPI task must call **MPI_Init()** before calling any other MPI routine and it must call this routine only once.

- A PVM task can be registered by another task as responsible for adding new PVM hosts, or as a PVM resource manager, or as responsible for starting new PVM tasks. These features are not available in MPI.

299

- A PVM task can multicast data to a set of tasks. As opposed to a broadcast, this multicast does not require the participating tasks to be members of a group. MPI does not have a routine to do multicasts.

- PVM tasks can be started in debug mode (that is, under the control of a debugger of the user's choice). This capability is not specified in the MPI standard, although it can be provided on top of MPI in some cases.

- In PVM, a user can use the **pvm_catchout()** routine to specify collection of task outputs in various ways. The MPI standard does not specify any means to do this.

- PVM includes a receive routine with a timeout capability, which allows the user to block on a receive for a user-specified amount of time. MPI does not have a corresponding call.

- PVM includes a routine that allows users to define their own receive contexts to be used by subsequent PVM receive routines. Communicators in MPI provide this type of functionality to a limited extent.

On the other hand, MPI provides several features that are not available in PVM, including a variety of communication modes, communicators, derived data types, additional group management facilities, and virtual process topologies, as well as a larger set of collective communication calls.

Working With Fonts

Chapter 15, "Working With Fonts"

Describes the use of fonts and font metric files within the X Window System, and the installation of bit-mapped and Type 1 fonts.

Working With Fonts

This chapter describes how to work with fonts on Silicon Graphics computers. It begins with an introduction to fonts and digital typography. Then it explains which fonts are available and how to install additional fonts. It also covers how to download outline fonts in the Type 1 format to a PostScript printer.

This chapter contains these sections:

- "Font Basics" defines fonts and provides some general background information.

- "Using Fonts With the X Window System" discusses some of the most useful font utilities of the X Window System.

- "Installing and Adding Font and Font Metric Files" explains how to install and add font files and font metric files for system-wide use.

- "Downloading a Type 1 Font to a PostScript Printer" explains how to download a Type 1 font to a PostScript printer.

Font Basics

Fonts are collections of characters. A font contains the information about the shape, size, and position of each character in a character set. That information is needed by programs that process characters, such as editing, word-processing, desktop publishing, multimedia, titling, and prepress application programs. Almost all software components in a computer system use fonts to display messages, prompts, titles, and so forth.

Binary digits are used to represent all types of information stored in a digital computer, including fonts. Digital typography deals with the style, arrangement, and appearance of typeset matter in digital systems. If you want to use font and font metric files to correctly typeset text on a digital computer, you need to know some basics about digital typography. This section contains a brief introduction to fonts and digital typography. You may want to read a book on typography for more in-depth information.

This section covers the following topics:

- "Terminology" introduces a few basic terms.

- "How Resolution Affects Font Size" describes horizontal and vertical resolution, pixels, and bitmap fonts.

- "Font Names" explains the differences between PostScript and X Windows font names.

- "Writing Programs That Need to Use Fonts" covers X programs, Display Postscript (DPS) programs, and IRIS GL and IRIS GL/X programs.

Terminology

Before discussing how to use fonts, consider these terms.

Typography

Typography is the art and technique of working with type. In traditional typography, the term *type* refers to a piece of wood or metal with a raised image of a character or characters on its upper face. Such pieces of wood or metal are assembled into lines and pages, which are printed by a letterpress process. What typographers do with type is called typesetting or composition. Type can also refer to the images produced by using such pieces of wood or metal.

Traditional typesetting is seldom used today. In modern typography, *type* usually refers to the images produced on typesetting or composition systems, which do not use wooden or metal type, such as photo and digital composition systems. The typography on a digital system, such as a digital computer, is called digital typography.

Digital typography is based on a hierarchy of objects called *characters*, *fonts*, and *font families* (or typefaces). Numeric values or measurements related to those objects can be divided into *character metrics*, *font metrics*, and *typeface metrics*. Sometimes all information about a font family, or typeface, is stored in a set of font files, but sometimes metric information for a set of font files is stored in a separate file called the font metric file.

Character

A character is a graphical or mathematical representation of a glyph. Letters, digits, punctuation marks, mathematical symbols, and cursors are examples of glyphs.

Font

A font is a set of characters, that is, a set of representations of characters. In a *bitmap* font, the shape of each character is represented by a rectangular array of bit values, 1 or 0, forming a bitmap of the shape. In an *outline* font, the shape of a character is represented by a mathematical description of its outline.

A distinction exists between a base and composite font. A *base font* is a set of characters of the same size and style. Characters in a base font usually match one another in size, style, weight, and slant because their shape, size, position, and spacing have been carefully designed by a skilled font designer. A *composite font* is composed of base fonts with various attributes, for example roman and italic, or book weight and semibold.

Font Family, or Typeface

A professional font designer usually creates an entire *font family*, or *typeface*, composed of a variety of base fonts with related forms, rather than a single font. A base font family, or typeface, is a set of base fonts with the same style or design. A composite font family, or composite typeface, is composed of base font families. A base font family can consist of bitmap fonts in certain sizes, a scalable font that can be used to produce bitmap fonts in different sizes, or both.

How Resolution Affects Font Size

The images on most output devices, such as laser printers and video monitors, are created by coloring a rectangular array of small dots or pixels (picture elements). The number of dots or pixels that can be drawn per unit of length in a horizontal direction is called the *horizontal resolution*, while the number of pixels that can be drawn per unit of length in a vertical direction is called the *vertical resolution*. The most commonly used unit of measure for resolution is the number of dots per inch (dpi). Resolution is a device-dependent unit of measure.

To display the resolution of your video monitor, enter this command:

```
xdpyinfo | grep resol
```

You should get a response similar to this:

```
resolution: 93x93 dots per inch
```

The first number is the horizontal resolution; the second the vertical resolution.

If you draw a single character at a given resolution, the *bounding box* of the character is the smallest rectangle that enclose that character.

If you display all of the characters in a font in the same place (without advancing), you get a composite image of those characters. If you then draw the smallest rectangle that encloses that composite image, you have the bounding box for the font. The size of a font is usually measured in the vertical direction. That size is usually not smaller than the height of the font bounding box, but it can be greater than that height. It may include additional vertical spacing that is considered part of the font design.

Typographers use small units of measure called *points* to specify font size. A point is approximately equal to 1/72 of an inch. The exact value is 1/72.27 (0.013837) of an inch, or 0.351 mm.

A point is a device-independent unit of measure. Its size does not depend on the resolution of an output device. A 12-point font should have approximately the same size on different output devices, regardless of the resolution of those devices.

If the resolution of an output device is equal to 72 dots per inch (dpi), the size of a dot or pixel is approximately equal to the size of a point. If the resolution of an output device is greater than 72 dpi, the size of a dot or pixel is smaller than the size of a point, and vice versa. You can use the following formula to calculate a pixel size from a point size:

```
pixel-size = point-size x device-resolution / 72.27
```

A bitmap font is usually designed for a particular resolution. Such a font has the point size specified by its designer only when it is used on an output device whose resolution matches the resolution for which that font was designed. This is because a font designer specifies a fixed bitmap for each character. If a pixel is smaller than a point, characters will be smaller than intended, and vice versa.

Font Names

When a font is designed, it is assigned a name such as *Courier Oblique*. This font belongs to a font family called *Courier*, which includes:

- Courier
- Courier Bold
- Courier Bold Oblique
- Courier Oblique

When the PostScript page description software language was developed by Adobe Systems, the spaces embedded in font names were replaced with dashes. PostScript font names look like this:

```
Courier
Courier-Bold
Courier-BoldOblique
HeiseiMin-W3--Adobe-Japan1-2
```

The size of a font is usually not part of the name of a scalable font because it can be scaled to any size. Bitmap fonts are usually designed in specific sizes. They are referred to by names such as 12-point Courier or 10-pixel Courier Bold.

The X Consortium specified 14-part font names for the X Window System. Each name is in effect a complete description of the font.

Figure 15-1 shows an example 14-part name for a bitmap font, with each part labeled. Point sizes in X font names are specified in *decipoints* (tenths of a point).
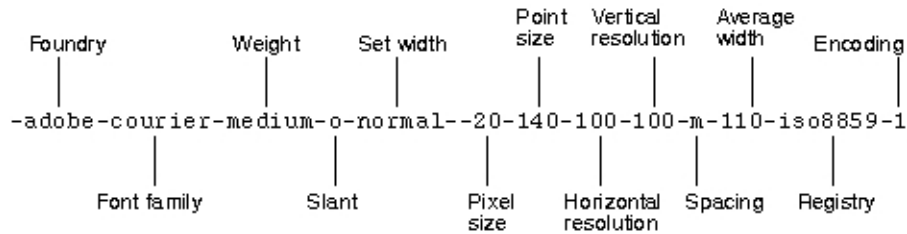
Figure 15-1   X Window System Font Name Example

Writing Programs That Need to Use Fonts

You can write different types of programs for Silicon Graphics computers, for example, X, Display PostScript (DPS), IRIS GL, OpenGL, and mixed-model programs. Some of your programs need fonts.

How a program accesses font files depends on the program type:

- X programs access fonts by calling X font functions, such as **XListFonts()** and **XLoadFont()**.

- DPS programs access fonts by calling X and DPS functions, or by using PostScript.

- IRIS GL and IRIS GL/X mixed-model programs usually access fonts by calling font management (fm) functions from the IRIS GL Font Manager library (**fmenumerate()** and **fmfindfont()**, for example).

Most fonts are installed when you install the X Window System (X11 Execution Environment). Some fonts are installed with other software components, such as DPS and IRIS Showcase. Some bitmap fonts are installed when you install a language module, such as the Japanese Language Module (JLM). Some outline fonts are installed when you install a font module, such as the Japanese Font Module (JFM). However, most fonts are shared by the X Window System, DPS (which is an extension of the X Window System), IRIS GL Font Manager, Impressario, and other software components.

To maintain compatibility and portability, it is best not to access font files directly from an application program because font formats, font names, font contents, and the location of font directories may change. Your program should use the Application Programming Interfaces (APIs) specified for the X Window System, DPS, and IRIS GL Font Manager, or call even higher level functions for the 2D and 3D text available from toolkits such as IRIS Inventor and IRIS Performer.

Using Fonts With the X Window System

This section describes how to use fonts with the X Window System. The X Window System has several font utilities. This section covers a few of the most useful utilities and includes:

- "Listing and Viewing Fonts" explains using the *xlsfonts* command.

- "Viewing Fonts" describes the *xfd* command.

- "Getting the Current X Font Path" covers the *xset* command.

- "Changing the X Font Path" explains the *xset fp* command.

For a complete description of the utilities, refer to your X Window System documentation.

Listing and Viewing Fonts

Getting a List of Font Names and Font Aliases

To find out which font names and font aliases are known to the X Window System, use the command *xlsfonts.*For more information about that command, see the reference page xlsfonts(1). If you enter the command:

```
xlsfonts | more
```

the resulting display contains entries such as:

```
-adobe-courier-bold-o-normal--0-0-0-0-m-0-iso8859-1
-adobe-courier-bold-o-normal--14-100-100-100-m-90-iso8859-1
-sgi-screen-medium-r-normal--14-140-72-72-m-70-iso8859-1
screen14
```

The first entry is an example of a 14-part X name for an outline (scalable) font. Numeric parts of font names are set to zero for outline fonts, because those fonts can be scaled to various sizes. The second and third entries are examples of 14-part X font names for bitmap fonts, while the last entry is an alias for the third entry. An X or DPS program can get a list of available fonts by calling **XListFonts()** or the function **XListFontsWithInfo()**.

Viewing Fonts

To see what a particular font looks like, use the command *xfd*, and specify a font name or font alias known to the X Window System by using the option *-fn*. For example, to display the 14-point Adobe Courier Bold font, enter:

```
xfd -fn -adobe-courier-bold-r-normal--14-140-75-75-m-90-iso8859-1
```

To request a Utopia Regular font scaled to the size of 28 points, enter:

```
xfd -fn -adobe-utopia-medium-r-normal--0-280-0-0-p-0-iso8859-1
```

You can use an asterisk (*) to indicate that any value is acceptable for a part of an X font name. However, asterisks in a command must be protected from the shell with quotes. For example, enter:

```
xfd -fn "-*-itc bookman-demi-i-normal--11-80-100-100-p-63-iso8859-1"
```

to indicate that *xfd* can use an ITC Bookman Demi Italic font from any foundry.

The *xfd* command displays all characters in a specified font, as shown in Figure 15-2.
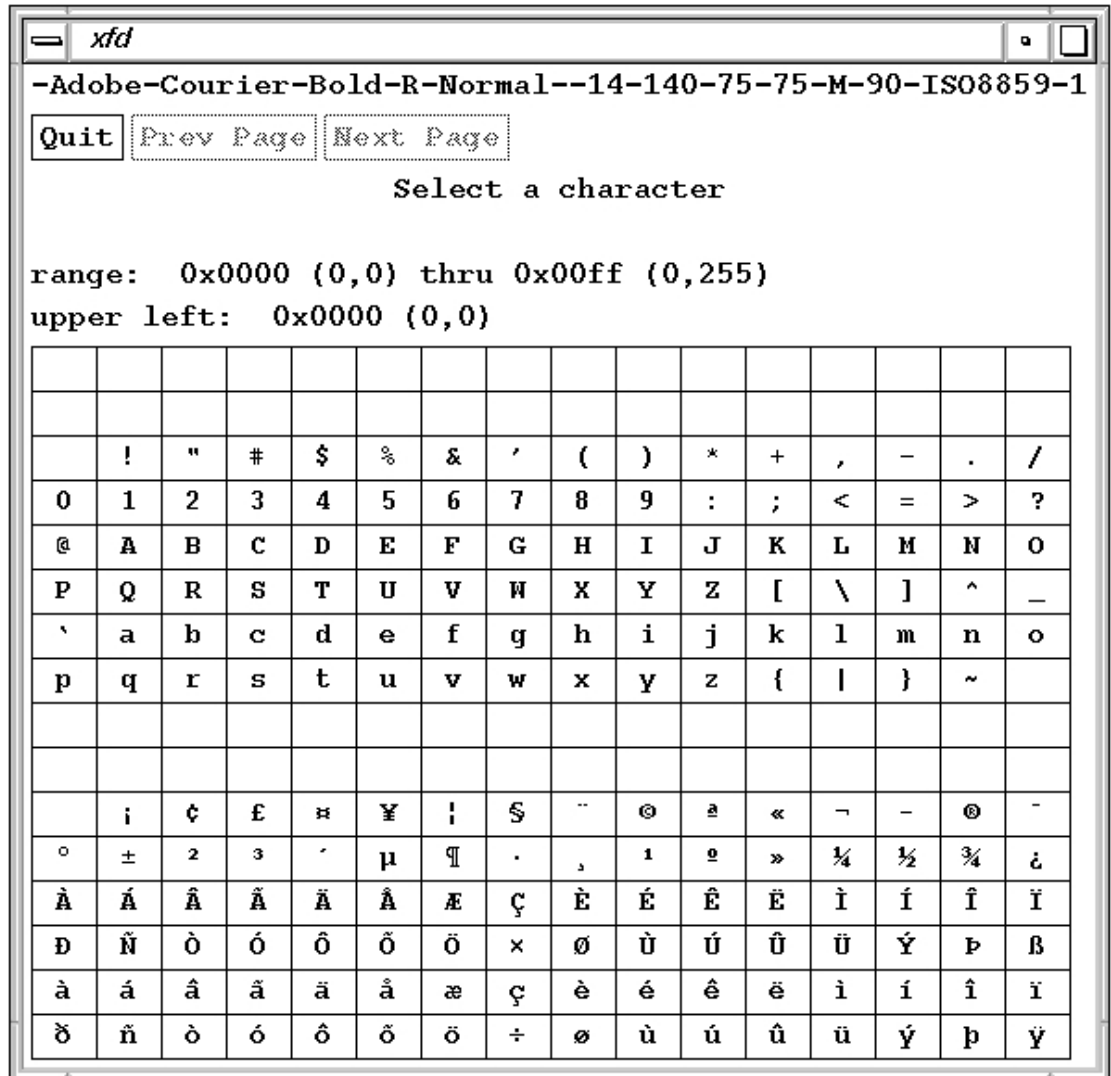
Figure 15-2   Sample Display From xfd

To open a shell window that uses a certain font, enter:

xwsh -fn *font-name*

Getting the Current X Font Path

The X system locates font files along a path, similar to the execution path used to find executable files. To display the current X font path, enter this command:

xset q

In addition to other information, the *xset* utility displays font path information that may look like this:

```
Font Path:
/usr/lib/X11/fonts/100dpi/,/usr/lib/X11/fonts/75dpi/,
/usr/lib/X11/fonts/misc/,/usr/lib/X11/fonts/Type1/,
/usr/lib/X11/fonts/Speedo/,/usr/lib/X11/fonts/CID/
```

The X Window System checks the resolution of your video monitor. If that resolution is closer to 75 dpi than 100 dpi, it puts the directory *75dpi* ahead of the directory *100dpi* in the X font path.

Changing the X Font Path

You can change the default X font path by using the option *fp=* on an *xset* command line. For example, enter:

xset fp=*newpath*

This command changes the X font path to the new font path (*newpath*).

Installing and Adding Font and Font Metric Files

This section explains where the various types of font and font metric files are installed by default, and how you can add one of your font or font metric files to the IRIX operating system.

This section describes the following topics:

- "Locations of Font and Font Metric Files" covers the conventional directories and names for font files.

- "Adding Font and Font Metric Files" details adding a bitmap and outline font, and adding a font metric file.

Locations of Font and Font Metric Files

By default, font and font metric files are installed in the directories listed in Table 15-1.

Table 15-1   Font and Font Metric Directories

Directory Path	Conventional Contents
/usr/lib/DPS/outline/base	Outline font files in the Adobe Type 1 format
/usr/lib/X11/fonts/Type1	Symbolic links to font files in */usr/lib/DPS/outline/base*
/usr/lib/DPS/AFM	Adobe Font Metric (AFM) files
/usr/lib/X11/fonts/100dpi	Bitmap fonts designed for the screen resolution of 100 dpi
/usr/lib/X11/fonts/75dpi	Bitmap fonts designed for the screen resolution of 75 dpi
/usr/lib/X11/fonts/misc	Miscellaneous other bitmap fonts
/usr/lib/X11/fonts/Speedo	Outline font files in the Bitstream Speedo™ format
/usr/lib/X11/fonts/CID	AFM, CCM, CFM, CIDFont and CMap files for large outline fonts in the Adobe CID-keyed format

The X Window System, Display PostScript, IRIS GL Font Manager, Impressario, and other software components use the directories listed in Table 15-1 by default. The locations of font files are made known to the X Window System in two ways:

• Within each directory specified in the X font path, a file named *fonts.dir* contains a directory of filenames with their corresponding 14-part font names. For example, to see the font names available in */usr/lib/X11/fonts/100dpi*, use the command

```
more /usr/lib/X11/fonts/100dpi/fonts.dir
```

This file is created by *mkfontdir* (see the mkfontdir(1) reference page).

• The files */usr/lib/X11/fonts/ps2xlfd_map** are used by the X Window System and the IRIS Font Manager to map PostScript names or short font names to 14-part X font names, and vice versa. The IRIS Font Manager does not use any bitmap fonts that do not have an entry in those files.

In IRIX 6.5, the twelve bitmap and outline fonts appear in the install directories:

• Dutch 801 Roman, Dutch 801 Italic, Dutch 801 Bold, Dutch 801 Bold Italic

• Swiss 721 Roman, Swiss 721 Italic, Swiss 721 Bold, Swiss 721 Bold Italic

• Courier 10-Pitch Roman, Courier 10-Pitch Italic, Courier 10-Pitch Bold, and Courier 10-Pitch Bold Italic

Each font contains 1015 characters. Those characters adhere to the International Organization for Standardization ISO8859-1 through ISO8859-10 and the Minimum European Subset (MES) of ISO10646-1 or Unicode 2.0. For more information about MES, use a web browser to open http://www.indigo.ie/egt/standards/mes.html.

Swiss 721 fonts are installed when you install the subsystem *x_eoe.sw.Xfonts*. The rest of the fonts are in the subsystem *x_eoe.sw.Xunicodefonts*. The subsystem *x_eoe.sw.Xfonts* is installed by default, while the subsystem *x_eoe.sw.Xunicodefonts* is optional. The new fonts are sometimes referred to as Unicode fonts, because they include Unicode character maps, and you can use Unicode codes to access characters in those fonts.

Conventions for Bitmap Font Filenames

The names of bitmap font files are specified according to the following conventions:

- Most filenames begin with three or four letters unique to the font family, such as *cour* for the Courier family, or *8x13* for a utility bitmap family.

- When a family has different style variants such as Roman and Italic, the next character of the filename is an uppercase letter to indicate the style, for example *courO* for Courier Oblique, or *8x13B* for a utility bold font.

- The last two characters of the filename are two digits giving the nominal size of the font in points, as in *courO18*.

- Most bitmap files are of the Portable Compiled Format (PCF) type and have the file suffix *.pcf*, as in *courO18.pcf* or *8x13B.pcf*.

- Files are compressed using the *compress* command (see the compress(1) reference page), and, therefore, have the terminal suffix *.Z* as in *courO18.pcf.Z*.

- Exceptions to these conventions are names such as *Swiss721-Bold--Bitstream-Unicode-0_10.pcf.Z* which are bitmap fonts in the directory */usr/lib/X11/fonts/100dpi*. These fonts use this format:

  CIDFontName--*CMapName_PointSize*`.pcf.Z`

  where *CIDFontName* is the font family name, *CMapName* specifies the character map name, and *PointSize* is the nominal size of the font. These names are reserved for bitmap CID-keyed fonts.

In */usr/lib/DPS/AFM* there is one font metric file per typeface. When you install a font module, such as the Japanese Font Module, metric files for CID-keyed fonts are stored in the directory */usr/lib/X11/fonts/CID/character-collection/AFM*. Font metric files are primarily used by text-processing and desktop-publishing programs to, for example, generate PostScript code for a specified document.

Creating Font Aliases

If you do not want to use long X font names, you can specify shorter aliases for those names. Silicon Graphics uses a file called *fonts.alias* to specify short aliases for fonts. There can be a *fonts.alias* file in an X font directory. For example, see the file *fonts.alias* in the directory */usr/lib/X11/fonts/100dpi*.

A typical font alias looks like this:

```
fixed -misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso8859-1
```

This associates the short alias "fixed" to the longer name that follows it. The alias file can also be used to specify alternate conventions for the component parts of a 14-part font name. For example, the following entry creates an alias that uses "regular" instead of "medium" for the weight component:

```
-adobe-utopia-regular-i-normal--14-100-100-100-p-74-iso8859-1
-adobe-utopia-medium-i-normal--14-100-100-100-p-74-iso8859-1
```

To specify your own font aliases in a font directory, store them in a file called *fonts.alias.local* in that directory. That way your entries do not disappear when you upgrade your system software.

Adding Font and Font Metric Files

When you purchase a font or obtain a font that is in the public domain, you need to add that font to your system and possibly to your printer in order to use it. Adobe Systems donated bitmap, outline, and font metric files for the Utopia font family to the X Consortium. This section shows how the font and font metric files for Utopia Regular were added to the IRIX operating system. Other font and font metric files can be added in a similar way.

You need superuser privilege to make changes to X font directories. Before you make any changes to any IRIX directory, make a copy of its contents so that you can restore that directory if anything goes wrong. For example, your font files may not be in the right format, and they may interfere with the access of Silicon Graphics font files. Keep a log of the changes you make, and mention those changes when you report a problem with font files to Silicon Graphics; otherwise, it may be very difficult or impossible for other people to reproduce any problems that you might report.

Adding a Bitmap Font

The procedure in this section shows how to add Utopia Regular bitmap fonts to IRIX. Other fonts can be added in a similar way.

To add the Utopia bitmap fonts to the X Window System, Display PostScript, and IRIS GL Font Manager, follow these steps:

1. Log in as root.

2. Choose names for the installed bitmap files. Refer to the naming conventions for existing bitmap font files (see "Conventions for Bitmap Font Filenames" on page 315) and use names with a consistent format when you create new font names. For example, Adobe provided Utopia Regular bitmap font files designed for the resolutions of 100 and 75 dpi. The original names of these files were *UTRG_10.bdf* through *UTRG_24.bdf*

    Filenames closer to IRIX conventions are *utopR10* through *utopR24* (followed by the appropriate file suffixes).

3. Convert files in Bitmap Distribution Format (BDF) to Portable Compiled Format (PCF) font files.

    BDF font files are text (ASCII) files. You can think of them as source font files. You can put BDF font files into an X font directory, but normal practice is to use only binary font formats such as the PCF (*.pcf*) or compressed PCF format (*.pcf.Z*) for performance reasons.

    Use the *bdftopcf* command to convert a BDF font file to a PCF font file (see the bdftopcf(1) reference page). For example, Adobe provided two sets of Utopia Regular bitmap font files that were designed for the resolutions of 100 and 75 dpi. These files were in the extended Bitmap BDF 2.1 format. The original names of the bitmap files were *UTRG_10.bdf* through *UTRG_24.bdf*. One of them could be converted with the following command:

    ```
    bdftopcf -o utopR10.pcf UTRG_10.bdf
    ```

    However, you normally want to compress the PCF file as well. You can compress a PCF file by entering a command such as:

    compress utopR10**.pcf**

    But you could combine both steps simply as follows:

    ```
    bdftopcf UTRG_10.bdf | compress -c >utopR10.pcf.Z
    ```

4. Move the bitmap font files to the appropriate directory, */usr/lib/X11/fonts/100dpi* or */usr/lib/X11/fonts/75dpi*. You can of course combine this step with the format conversion step as follows:

```
bdftopcf UTRG_10.bdf | compress -c
          >/usr/lib/X11/fonts/100dpi/utopR10.pcf.Z
```

You can tell the resolution for which a font was designed by the name of the directory in which the font designer stored the font files, or by the information in the header of a bitmap font file. In a BDF 2.1 font file, the horizontal and vertical resolution are specified in the X font name. They are also specified after the point size as the second and third numeric values in a SIZE entry. For example, the entry:

```
SIZE 8 100 100
```

within the file indicates an 8-point font that was designed for the horizontal and vertical resolution of 100 dpi.

5. For Type 1 PostScript font families, there is one entry per font family in the file */usr/lib/X11/fonts/ps2xlfd_map*. For each Japanese font family shipped by Silicon Graphics, there is an entry in the file */usr/lib/X11/fonts/ps2xlfd_map.japanese*.

When adding a new Type 1 font, insert an entry in the appropriate file for each style variation in the font family. It is not necessary to have an entry for each bitmap size. For example, the entries in *ps2xlfd_map* for the Utopia fonts are:

```
Utopia-Bold -adobe-utopia-bold-r-normal--0-0-0-0-p-0-iso8859-1
Utopia-BoldItalic -adobe-utopia-bold-i-normal--0-0-0-0-p-0-iso8859-1
Utopia-Italic -adobe-utopia-medium-i-normal--0-0-0-0-p-0-iso8859-1
Utopia-Regular -adobe-utopia-medium-r-normal--0-0-0-0-p-0-iso8859-1
```

The first field is the PostScript font name, as specified in the outline font file (see "Adding an Outline Font" on page 319). The second field is the X 14-part font name with 0 for all specific dimension values.

When you add your own bitmap or outline fonts, put their entries in a file called */usr/lib/X11/fonts/ps2xlfd_map.local*. That way your entries do not disappear when you upgrade your system software.

Make sure that there is no overlap between your entries and the entries in other *ps2xlfd_map** files.

6. If you want to establish alias names for any of the new fonts, create or edit *fonts.alias* files in the appropriate directories (see "Creating Font Aliases" on page 315).

7. Invoke the *mkfontdir* command to rebuild the *fonts.dir* database in each directory where you added bitmap files. Enter the command:

```
mkfontdir /usr/libX11/fonts/*dpi
```

to create a new *fonts.dir* (fonts directory) file in the *100dpi* and *75dpi* directories.

8. Use the *xset* command to notify the window system to rebuild its list of fonts:

```
xset fp rehash
```

9. To check whether the fonts you added are known to the X Window System, enter:

```
xlsfonts > /tmp/fontlist
```

The names of the fonts you added should appear on the list of font names and aliases produced by *xlsfonts*.

Bitmap fonts should now be added to the X Window System and the IRIS GL Font Manager. Since DPS needs both outline and bitmap fonts for each supported typeface, it first checks which outline fonts are stored in the directory *usr/lib/DPS/outline/base*. Then it looks for the corresponding bitmap fonts in other X font directories. It ignores all other bitmap fonts. Therefore, DPS ignores the bitmap fonts you added until you add the corresponding outline fonts.

Adding an Outline Font

To add the Utopia Regular outline font to the X Window System, Display PostScript, and the IRIS GL Font Manager, follow these steps:

You can install only Adobe text (ASCII) Type 1 font files or compatibles, not binary Type 1 font files and not Type 3 font files. Display PostScript can handle Type 3 font files, but the X Window System and IRIS GL Font Manager cannot.

1. Log in as root.

2. Convert the file to Printer Font ASCII (PFA) format if necessary. Printer Font Binary (PFB) files are not supported. To convert *.pfb* files to *.pfa* files, use the *pfb2pfa* command shipped with IRIX version 5.3 and higher (see the pfb2pfa(1) reference page). For example, to convert the Adobe file UTRG____.pfb, enter

```
pfb2pfa UTRG____.pfb UTRG____.pfa
```

3. Look at the names of existing outline font files in the directory
    /usr/lib/DPS/outline/base. Display PostScript requires that the name of each outline
    font file match the PostScript font name specified in the /FontName entry in the
    header of that outline font file. For example, if you enter:

    grep /FontName Courier-Bold

    in the directory */usr/lib/DPS/outline/base*, you get:

    ```
    /FontName /Courier-Bold def
    ```

    The name revealed is used for the filename of the outline font, the filename of the
    metric file, and in the */usr/lib/X11/fonts/ps2xlfd.map* file.

    For example, Adobe provided the Utopia Regular outline font file *UTRG____.pfa*,
    which is an outline font file in the Type 1 format. To find the PostScript font name
    for this font, enter:

    ```
    grep /FontName UTRG____.pfa
    ```

    You should get the response:

    ```
    /FontName Utopia-Regular def
    ```

    When this font was added to IRIX, the name of the file *UTRG____.pfa* was changed
    to *Utopia-Regular.*

4. Put the file *Utopia-Regular* in the directory */usr/lib/DPS/outline/base*, because that
    outline font is in the Type 1 format. If you have an outline font in the Speedo format,
    put it in the directory:

    ```
    /usr/lib/X11/fonts/Speedo
    ```

5. To add the Utopia Regular font and font metric files to Display PostScript, enter:

    ```
    /usr/bin/X11/makepsres -o /usr/lib/DPS/DPSFonts.upr
    /usr/lib/DPS/outline/base /usr/lib/DPS/AFM
    ```

    You should now be able to access the font file you added via Display PostScript.

6. For most font families shipped by Silicon Graphics, there is one entry per font family in the file:

    `/usr/lib/X11/fonts/ps2xlfd_map`

    as described in "Adding a Bitmap Font." The same entry is used for both bitmap and outline fonts.

    If you add your own (local) bitmap or outline fonts, put an entry for each font family in the file called:

    `/usr/lib/X11/fonts/ps2xlfd_map.local`

    You can use entries in the file *ps2xlfd_map* as templates for entries in the file *ps2xlfd_map.local*.

    If the file *ps2xlfd_map.local* does not exist, log in as root, and create it.

    You can now access the font you added via the IRIS GL Font Manager.

7. Display PostScript is an extension of the X Window System. To add an outline font in the Type 1 format to the rest of the X Window System, in any directory, enter the commands:

    ```
    type1xfonts
    xset fp rehash
    ```

    This re-creates symbolic links in the directory */usr/lib/X11/fonts/Type1* that point to outline font files in the directory */usr/lib/DPS/outline/base*, and instructs the X Window System to check which fonts are available.

8. To check whether the outline fonts you added are known to the X Window System, enter:

    xlsfonts | grep *family-name*

The entries for the outline fonts you added should appear on the list of font names and aliases produced by *xlsfonts*.

Adding a Font Metric File

Adobe Font Metric (AFM) files are primarily used by application programs—for example, to generate PostScript code for a specified document. Follow these steps to add a font metric file for an outline font in the Type 1 format:

1.   Log in as root.

2.   Put Adobe Font Metric files in the directory */usr/lib/DPS/AFM*.

The name of an AFM file must match the PostScript font name as given in the file */usr/lib/X11/fonts/ps2xlfd_map* (see "Locations of Font and Font Metric Files" on page 313).

For example, Adobe provided the Utopia Regular font metric file *UTRG____.AFM*. When this font was added to IRIX, the name was changed to *Utopia-Regular* to correspond to the line

```
Utopia-Regular -adobe-utopia-medium-r-normal--0-0-0-0-p-0-iso8859-1
```

in */usr/lib/X11/fonts/ps2xlfd_map*.

The file was put in the directory */usr/lib/DPS/AFM*.

Font metric files for a large outline font in the CID-keyed format should be put in the directory */usr/lib/X11/fonts/CID/character-collection/AFM*. There is one AFM file for each CIDFont file, and one AFM file for each CID-keyed font.

Downloading a Type 1 Font to a PostScript Printer

Some outline fonts are usually built into a PostScript printer. You can find out which fonts are known to the PostScript interpreter in your printer by sending the following file to that printer:

```
%!
% Produce a list of available fonts
/f 100 string def
/Times-Roman findfont 12 scalefont setfont
/y 700 def
72 y moveto
FontDirectory {
   pop f cvs show 72 /y y 13 sub def y moveto
} forall
showpage
```

Utopia fonts are not usually built into PS printers. If you try to print a document that requires a Utopia font on a PS printer that does not have that font, a warning message about the replacement of a missing font with a Courier font is sent to the file */usr/spool/lp/log* on the system to which that PS printer is attached.

You can download a Type 1 font to a PS printer in either of the following two ways:

- You can insert a Type 1 font file at the beginning of the PostScript file that needs that font. You should have a statement that starts with:

  ```
  %!
  ```

  Put this statement at the beginning of your PS file. If you have two such lines, delete the second one.

  When you download a font this way, the font is available only while your print job is being processed.

- You can make a copy of a Type 1 font file, and then insert the statement:

  ```
  serverdict begin 0 exitserver
  ```

  after the first group of comment statements (lines that start with %) if no password has been specified for your printer; otherwise, replace 0 in the above statement with the password for your printer. Then send the edited file to your printer.

  When you download a font this way, the warning message:

  ```
  %%[ exitserver: permanent state may be changed ]%%
  ```

  is sent to the file */usr/spool/lp/log* on the system to which the printer is attached.

  The permanent state of the printer is not really changed. Downloaded fonts disappear when you reset the printer by switching its power off and on. If there is not enough memory for additional fonts, you receive a message about a Virtual Memory (VM) error, and the font is not downloaded.

  If you again send the program that produces a list of available fonts to your printer, you should see the PostScript names of the fonts you downloaded on that list.

Internationalizing Your Application

Internationalization is the process of generalizing an application so that it can easily be customized—or *localized*—to run in more than one language environment. You can provide internationalized software that will produce output in a user's native language, format data (such as currency values and dates) according to local standards, and tailor software to a specific culture.

This chapter describes how to create such an application. It contains the following major sections:

- "Overview of Internationalization" presents an introduction to internationalization and defines some common terms.

- "Using Locales" explains how to set the current locale and limitations of the locale system.

- "Character Sets, Codesets, and Encodings" describes various ways of encoding characters, the traditional ASCII being just one of these.

- "Cultural Items" discusses the ways in which different cultures affect the way a string can be viewed, for example in outputting or collating.

- "Locale-Specific Behavior" covers native language support (NLS) and the NLS database, regular expressions, and cultural data.

- "Strings and Message Catalogs"describes how to create and use catalogs of messages to send diagnostic information to users in various locales.

- "Internationalization Support in X11R6" describes internationalization support provided by X11, Release 6 (including features from X11R5).

- "Internationalization Support in Motif" points to information describing how to internationalize a Motif application.

- "Translating User Input" discusses the translation of keyboard events into programmatic character strings for a variety of keyboards.

- "GUI Concerns" discusses internationalizing applications that use graphical user interfaces (GUIs)

- "Popular Encodings" presents some common non-ASCII encodings.

For a list of ISO 3166 country names and abbreviations, see Appendix A, "ISO 3166 Country Names and Abbreviations." You can find detailed information about fonts in Chapter 15, "Working With Fonts." Also, you can find additional information about internationalizing an application in the *IRIX Interactive Desktop Integration Guide*.

Overview of Internationalization

Internationalized software can be made to produce output in a user's native language, to format data (such as dates and currency values) according to the user's local customs, and to otherwise make the software easier to use for users from a culture other than that of the original software developer. As computers become more widely used in non-American cultures, it becomes increasingly important that developers stop relying on the conventions of American programming and the English language in their programs. This chapter provides information on how to make your applications more widely accessible.

This section presents the following topics:

- "Some Definitions of Internationalization" covers locales, internationalization, localization, nationalized software, and multilingual software.

- "Areas of Concern in Internationalizing Software" points out a few concerns to watch for when internationalizing your software.

- "Standards" covers standard-compliant features.

- "Internationalizing Your Application: The Basic Steps" lists the procedures to use when internationalizing an icon.

- "Additional Reading on Internationalization" provides references you can consult for additional information about internationalization.

Some Definitions of Internationalization

This section defines some of the terms used in this chapter.

Locale

Locale refers to a set of local customs that determine many aspects of software input and output formatting, including natural language, culture, character sets and encodings, and formatting and sorting rules. The locale of a program is the set of such parameters that are currently selected. For information on the method for selecting locales, see "Additional Reading on Internationalization" below.

Internationalization (i18n)

Internationalization is the process of making a program capable of running in multiple locales without recompiling. To put it another way, an internationalized program is one that can be easily localized without changing the program itself. (See "Localization (l10n)," below, for an explanation of the term "localization.")

Note: The word "internationalization" consists of an *i* followed by 18 letters followed by an *n*. It is thus often abbreviated "i18n" in informal writing. On similar principles, "localization" is often abbreviated "l10n."

A program written for a specific locale may be difficult to run in a different environment. Rewriting such a program to operate in each desired environment would be tedious and costly.

Your goal as a developer should thus be to write *locale-independent* programs, programs that make no assumptions about languages, local customs, or coded character sets. Such internationalized applications can run in a user's native environment following native conventions with native messages, without recompiling or relinking. A single copy of an internationalized program can be used by a world of different users.

Localization (l10n)

Localization is the act of providing an internationalized application with the environment and data it needs to operate in a particular locale. For example, adding German system messages to IRIX is a part of localizing IRIX for the German locale.

329

Nationalized Software

Nationalized programs run in only one language and are governed by one set of customs; in other words, in a nationalized program the locale is built into the application. Even if the application doesn't use ASCII or English, as long as it is a single-language program it is nationalized, not internationalized. Most older UNIX programs can be thought of as being nationalized for the United States.

Consider two applications, *hello* and *bonjour*. The application *hello* always produces the output

```
Hello, world.
```

and *bonjour* always produces

```
Bonjour, tout le monde.
```

Neither *hello* nor *bonjour* are internationalized; they are both nationalized.

There are no special requirements for writing or porting nationalized applications, whether they are text or graphics programs. Terminal-based programs work on suitable terminals, including internationalized terminal emulators. "Suitable" means that the terminal supports any necessary fonts and understands the encoding of the application output. Graphics programs simply do as they have always done. Applications using existing interfaces to operate in non-English or non-ASCII environments should continue to compile and run under an internationalized operating system.

Multilingual Software

A *multilingual* program is one that uses several different locales at the same time. Examples are described in "Multilingual Support" on page 339.

Areas of Concern in Internationalizing Software

Few developers will have to pay attention to more than a few items described in this section. Most will need to catalog their strings. Some will need to use library routines for character sorting or locale-dependent date, time, or number formatting. A few whose applications use the eighth bit of 8-bit characters inappropriately will need to stop doing so. The few applications that do arithmetic to manipulate characters will need to be cleaned up. Some GUI designers will have to spend just a little more time thinking. But for the large majority of developers, there isn't much to do.

The information presented in the following sections addresses internationalization issues pertinent to a developer; some sections, however, may not be relevant to your applications.

Standards

IRIX internationalization includes these standards-compliant features, among others:

- ANSI C and POSIX (ISO 9945-1): Locale

- *X/OPEN Portability Guide, Issue 4* (XPG/4): XPG/4 message catalogs, interpretation of locale strings

- UNIX System V Release 4: Multi-National Language Support (MNLS) message catalogs

- X11R5 and X11R6: Input methods, text rendering, resource files

Internationalizing Your Application: The Basic Steps

To internationalize your icon, follow these steps:

1. Call **setlocale()** as soon as possible to put the process into the desired locale. See "Setting the Current Locale" on page 334 for instructions.

2. Make your application 8-bit clean. (An application is 8-bit clean if it does not use the high bit of any data byte to convey special information.) See "Eight-Bit Cleanliness" on page 341 for instructions.

3. If you're writing a multilingual application, you must do one of two things:

   - fork, and then call **setlocale()** differently in each process

   - call **setlocale()** repeatedly as necessary to change from language to language

   See "Multilingual Support" on page 339 for more information.

4. Use wide character (WC) or multibyte (MB) characters and strings to allow for more than one byte per character (this is needed for Asian languages, which often require two or even four bytes per character). See "Character Representation" on page 342 for more information.

5. Do not rely on ASCII and English sorting rules. Locale-specific collation should be performed with **strcoll()** and **strxfm()**. (These are table-driven functions; the tables are supplied as part of locale support.) See "Collating Strings" on page 348 for more information.

6. Use the **localeconv()** function to find out about general details of numeric formatting. Use **strfmon()** to format currency amounts in particular. See "Specifying Numbers and Money" on page 349 for more information.

7. Use **strftime()** to format dates and times (**strftime()** gives a host of options for displaying locale-specific dates and times.) See "Formatting Dates and Times" on page 351 for more information.

8. Avoid arithmetic on character values. Use the macros in *ctype.h* to get information about a given character. (These macros are table-driven and locale-sensitive.) If you prefer, you can use the functions that correspond to these macros instead. "Character Classification and ctype" on page 351 provides more detailed information on these macros and functions.

9. If you do your own regular expression parsing and matching, use the XPG/4 extensions to traditional regular expression syntax for internationalized software. See "Regular Expressions" on page 353 for more information.

10. Where possible, use the XPG/4, rather than the MNLS interface in order to maximize portability. See "Strings and Message Catalogs" on page 366 for more information.

11. Provide a catalog for your locale. See "SVR4 MNLS Message Catalogs" on page 370 for more information.

12. The File Typing Rule (FTR) strings that are used to customize the IRIX Interactive desktop can be Internationalized. See "Internationalizing File Typing Rule Strings With MNLS" on page 374 for more information.

13. Use message catalogs for **printf()** format strings that take linguistic parameters, and allow localizers to localize the format strings as well as text strings. See "Variably Ordered Referencing of printf() Arguments" on page 375 for more information.

14. If you're using Xlib, initialize Xlib's internationalization state after calling **setlocale()**. See "Initialization for Xlib Programming" on page 379 for more information.

15. Specify a default fontset suitable for the default locale. Make sure that the application accepts localized fontset specifications via resources (or message catalogs) or command-line options. See "Fontsets" on page 380 for more information.

16. Use X11R5 and X11R6 text rendering routines that understand multibyte and wide character strings, not the X11R4 text rendering routines **XDrawText()**, **XDrawString()**, and **XDrawImageString()**. See "Text Rendering Routines" on page 382 for more information.

17. Use X11R5 and X11R6 MB and WC versions of width and extents interrogation routines. See "New Text Extents Functions" on page 382 for more information.

18. If you are writing a toolkit text object, or if you can't use a toolkit to manage event processing for you, then you have to deal with input methods. Follow the instructions in "Translating User Input" on page 385.

19. Use resources to label any object that employs some sort of text label. Your application's app-defaults file should specify every reasonable string resource. See "X Resources for Strings" on page 401 for more information.

20. Use dynamic layout objects that calculate layout depending on the natural (localized) size of the objects involved. Some IRIS IM widgets providing these services are XmForm, XmPanedWindow, and XmRowColumn. See "Dynamic Layout" on page 402 for more information. If you can't use dynamic layout objects, refer to "Layout" on page 402 for instructions.

21. Make sure that all icons and other pictographic representations used by your application are localizable. See "Icons" on page 403 for more information.

Additional Reading on Internationalization

For more information on internationalization, refer to:

- O'Reilly Volume 1, *Xlib Programming Manual*
- *X Window System*, by Robert Scheifler and Jim Gettys
- *X/Open Portability Guide*
- *OSF/Motif Style Guide*

Using Locales

An internationalized system is capable of presenting and receiving data understandably in a number of different formats, cultures, languages, and character sets. An application running in an internationalized system must indicate how it wants the system to behave. IRIX uses the concept of a locale to convey that information.

A process can have only one locale at a time. Most internationalization interfaces rely on the locale of the current process being set properly; the locale governs the behavior of certain library routines.

This section covers the following topics:

- "Setting the Current Locale" explains categories, locales, strings, location of locale-specific data, and locale naming conventions.

- "Limitations of the Locale System" describes multilingual support, misuses of locales, and encoding.

You can find additional information in "Locale-Specific Behavior" on page 353, which describes native language support, regular expressions, and cultural data.

Setting the Current Locale

Applications begin in the *C* locale. (*C* is the name used to indicate the system default locale; it usually corresponds to American English.) Applications should therefore call **setlocale()** as soon as possible to put the process into the desired locale. The syntax for **setlocale()** is:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

The call almost always looks either like this:

```
if (setlocale(LC_ALL, "") == NULL)
    exit_with_error();
```

or like this:

```
if (setlocale(LC_ALL, "") == NULL)
    setlocale(LC_ALL, "C");
```

Details of the two parameters are given in the next two sections.

Using Locale Categories

Applications need not perform every aspect of their work in the same locale. Although this approach is not recommended, an application could (for example) perform most of its activities in the English locale but use French sorting rules. You can use locale categories to do this kind of locale-mixing. (Mixing locale categories is not the same as multilingual support—see "Multilingual Support.")

The *category* argument is a symbolic constant that tells **setlocale()** which items in a locale to change. Table 16-1 lists the available category choices.

Table 16-1   Locale Categories

Category	Affects
LC_ALL	All categories below
LC_COLLATE	Regular expressions, **strcoll()**, and **strxfrm()**
LC_CTYPE	Regular expressions and ctype routines (such as **islower()**)
LC_MESSAGES	**gettxt()**, **pfmt()**, and **nl_langinfo()**
LC_MONETARY	**localeconv()** and **strfmon()**
LC_NUMERIC	Decimal-point character for formatted I/O and nonmonetary formatting information returned by **localeconv()**
LC_TIME	**ascftime()**, **cftime()**, **getdate()**, and **strftime()**

Categories correspond to databases that contain relevant information for each defined locale. The locations of these databases are given in the "Location of Locale-Specific Data" on page 337.

Setting the Locale

The **setlocale()** function attempts to set the locale of the specified category to the specified locale. You should almost always pass the empty string as the *locale* parameter to conform to user preferences.

On success, **setlocale()** returns the new value of the category. If **setlocale()** couldn't set the category to the value requested, it returns NULL and does not change locale.

Empty String

An empty string passed as the *locale* parameter is special. It specifies that the locale should be chosen based on environment variables. This is the way a user specifies a preferred locale, and that preference should almost always be honored. The variables are checked hierarchically, depending on category, as shown in Table 16-2; for instance, if the category is LC_COLLATE, an empty-string locale parameter indicates that the locale should be chosen based on the value of the environment variable LC_COLLATE—or, if that value is undefined, the value of the environment variable LANG, which should contain the name of the locale that the user wishes to work in.

Table 16-2   Category Environment Variables

Category	First Environment Variable	Second Environment Variable
LC_COLLATE	LC_COLLATE	LANG
LC_CTYPE	LC_CTYPE	LANG
LC_MESSAGES	LC_MESSAGES	LANG
LC_MONETARY	LC_MONETARY	LANG
LC_NUMERIC	LC_NUMERIC	LANG
LC_TIME	LC_TIME	LANG

Specifying the category LC_ALL attempts to set each category individually to the value of the appropriate environment variable.

If no non-null environment variable is available, **setlocale()** returns the name of the current locale.

Nonempty Strings in Calls to setlocale()

Here are the possibilities for specifying the *locale* parameter:

NULL   Specifying a null pointer argument—not the same as the empty string—causes **setlocale()** to return the name of the current locale.

"C"   Specifying a locale value of the single-character string "C" requests whatever locale the system uses as a default. (Note that this is a string and not just a character.)

Other strings   Request a particular locale by specifying its name. This overrides any user preferences and should only be done with good reason.

Location of Locale-Specific Data

Except for XPG/4 message catalogs, locale-specific data (that is, the "compiled" files containing the collation information, monetary information, and so on) are located in */usr/lib/locale/*<locale>/<category>, where <locale> and <category> are the names of the locale and category, respectively. For example, the database for the LC_COLLATE category of the French locale *fr* would be in */usr/lib/locale/fr/LC_COLLATE*.

There will probably be multiple locales symbolically linked to each other, usually in cases where a specific locale name points to the more general case. For example, */usr/lib/locale/POSIX* might point to */usr/lib/locale/C*.

Locale Naming Conventions

A locale string is of the form

language[*_territory*[*.encoding*]][*@modifier*] . . .

where

- *language* is the two-letter ISO 639 abbreviation for the language name.

- *territory* is the two-uppercase-letter ISO 3166 abbreviation for the territory name. (For a list of these abbreviations, see the table in Appendix A, "ISO 3166 Country Names and Abbreviations.")

- *encoding* is the name of the character encoding (mapping between numbers and characters). For western languages, this is typically the codeset, such as 8859-1 or ASCII. For Asian languages, where an encoding may encode multiple codesets, the encodings themselves have names, such as UJIS or EUC (these encodings are described later in this section). "Character Sets, Codesets, and Encodings" on page 340 discusses codesets and encodings.

- *modifier*s are not actually part of the locale name definition; they give more specific information about the desired localized behavior of an application. For example, under X11R5 or X11R6, a user can select an input method with modifiers. (To use the *xwnmo* Input Method server provided by Silicon Graphics, for example, add **@im=_XWNMO** to the locale string.) No standards exist for this part of a locale string.

Language data is implementation specific; databases for the language *en* (English) might contain British cultural data in England and American cultural data in the United States. If other than the default settings are required, the territory field may be used. For example, the above cases could be more strictly defined by setting LANG to *en_GB* or *en_US*. Full rigor might lead to *en_GB.ISO 8859-1* for England and *en_US.ISO 8859-1* for the USA.

ANSI C has defined a special locale value of *C*. The *C* locale is guaranteed to work on all compliant systems and provides the user with the system's default locale. This default is typically American English and ASCII, but need not be. POSIX has also defined a special locale value, *POSIX*, which is identical to the *C* locale.

The length of the locale string may not exceed NL_LANGMAX characters (NL_LANGMAX is defined in */usr/include/limits.h*). However, XPG/4 recommends that this string (not counting modifiers) not exceed 14 characters.

Limitations of the Locale System

This section explains multilingual support, misuse of locales, and the absence of filesystem information for encoding types.

Multilingual Support

There can be only one locale at a time associated with any given process in an internationalized system. Therefore, although multilingual applications—which give the appearance of using more than one locale at a time—can be created, internationalization does not provide inherent support for them. Here are two examples of multilingual programs:

- An application creates and maintains windows on four different displays, operated by four different users. The program has a single controlling process, which is associated with only one locale at any given time. However, the application can switch back and forth between locales as it switches between users, so the four users may each use a different locale.

- In a sophisticated editing system with a complex user interface, a user may wish to operate the interface in one language while entering or editing text in another. For instance, a user whose first language is German may wish to compose a Japanese document, using Japanese input and text manipulation, but with the user interface operating in German. (There is no standard interface for such behavior.)

In writing a multilingual application, the first task is identifying the locales for the program to run in and when they apply. (There is no standard method for performing this task.) Once the application has chosen the desired locales, it must do one of the following:

- fork, and then call **setlocale()** differently in each process

- call **setlocale()** repeatedly as necessary to change from language to language

Misuse of Locales

The LANG environment variable and the locale variables provide the freedom to configure a locale, but they do not protect the user from creating a nonsensical combination of settings. For example, you are allowed to set LANG to *fr* (French) and LC_COLLATE to *ja_JP.EUC* (Japanese). In such a case, string routines would assume text encoded in 8859-1—except for the sorting routines, which might assume French text and Japanese sorting rules. This would likely result in arbitrary-seeming behavior.

No Filesystem Information for Encoding Types

The IRIX filesystem does not contain information about what encoding should be associated with any given data. Thus, applications must assume that data presented to an application in some locale is properly encoded for that locale. In other words, a file is interpreted differently depending on locale; there is no way to ask the file what it thinks its encoding is.

For example, you may have created a file while in a Japanese locale using EUC. Later, you might try printing it while in a French locale. The results will likely resemble a random collection of Latin 1 characters.

This problem applies to almost all stored strings. Most strings are uninterpreted sequences of nonzero bytes. This includes, for example, filenames. You can, if you want to, name your files using Chinese characters in a Chinese locale, but the names will look odd to anyone who runs */bin/ls* on the same filesystem using a non-Chinese locale.

Character Sets, Codesets, and Encodings

One major difference between nationalized and internationalized software is the availability in internationalized software of a wide variety of methods for encoding characters. Developers of internationalized software no longer have the convenience of always being able to assume ASCII. Three terms that describe groupings of characters are the following:

character set   An abstract collection of characters.

codeset   A character set with exactly one associated numerical encoding for each character. The English alphabet is a character set; ASCII is a codeset.

encoding   A set of characters and associated numbers; however, this term is more general than "codeset." A single encoding may include multiple codesets; *Extended UNIX Code (EUC)*, for instance, is an encoding that provides for four codesets in one data stream.

This section describes these topics:

- "Eight-Bit Cleanliness" explains how to make 8-bit clean characters.

- "Character Representation" discusses multibyte and wide characters.

- "Multibyte Characters" covers using and handling multibyte characters, conversions to constant-size characters, and the number of bytes in a character and string.

- "Wide Characters" explains *wchar* strings, support routines, and conversion to multibyte characters.

- "Reading Input Data" covers nonuser-originated data.

For information on installing and using fonts with an application, refer to Chapter 15, "Working With Fonts."

Eight-Bit Cleanliness

A program is *8-bit clean* if it does not use the high bit of any data byte to convey special information. ASCII characters are specified by the low seven bits of a byte, so some programs use the high bit of a data byte as a flag; such programs are not 8-bit clean. Internationalized programs must be 8-bit clean, because they cannot expect data to be in the form of ASCII bytes; non-ASCII character sets usually use all eight bits of each byte to specify the character. But a program must go out of its way to manipulate bytes based on the value of the high bit, and since changing data without cause is seldom desirable, most programs are already 8-bit clean.

The old *csh* (before this problem was fixed in the IRIX 5.0 release) was a good example of a program that was not 8-bit clean; it used the high bit in input strings to distinguish aliases from unaliased commands. An effect of this misuse was that *csh* stripped the eighth bit from all characters. For example, the user command

```
echo I know an architect named Mañosa
```

Produced the response

```
I know an architect named Maqosa
```

Another example is the specification of Internet messages, which calls for 7-bit data. Thus, if *sendmail* fails to strip the 8th bit from a character prior to sending it, it violates a protocol; if it does strip the bit, it could garble a non-ASCII message (this protocol problem is being addressed).

One of the simplest things to do to remove the American bias from a program is to replace the ASCII assumption with the assumption that the Latin 1 codeset will be used. This approach is not true internationalization, but it can make the application usable in most of Western Europe. Latin 1 uses only one byte per character, unlike some other codesets, so 8-bit clean ASCII software should work without modification using the Latin 1 codeset.

Ensuring that code is 8-bit clean is the single most important aspect of internationalizing software.

Another caveat about 8-bit characters applies only to a particular set of circumstances: If you are not using a multibyte character type (see the next section), you should not declare characters as type *signed char*. (The default in IRIX C is for *char* to imply *unsigned char*.) If you try to cast a *signed char* to an *int* (as you must do to use the **ctype()** functions) and the character's high bit is set (as it may be in an 8-bit character set), the high bit is interpreted as a sign bit and extends into the full width of the int.

Character Representation

Western languages usually require only one byte for each character. Asian languages, however, often require two or even four bytes per character, and some Asian encodings allow a variable number of bytes per character.

The two kinds of encodings that allow more than one byte per character are

- multibyte (MB) characters are of variable size
- wide characters (WC or wchar characters) are a fixed number of bytes long)

The application developer must decide where to use WC and MB characters and strings:

- Multibyte strings are almost the default: string I/O uses MB, MB code works for ASCII and ISO 8859, and MB characters use less space than do wide characters. However, manipulating individual characters within a multibyte string is difficult.

  Note: Traditional strings are merely a special case of multibyte strings, where every character happens to be one byte long and there is only one codeset. All MB code, including conversion to and from wchars, works for traditional ASCII, or ISO 8859, strings.

- Applications that do heavy string manipulation typically use WC strings for such activity, because manipulating individual WC characters in a string is much simpler than doing the same thing with MB characters. So wide characters are used as necessary to provide programming ease or runtime speed; however, they take up more space than MB characters.

  Note: WC is system dependent—applications should not use it for I/O strings or communication.

Multibyte Characters

A multibyte character is a series of bytes. The character itself contains information on how many bytes long it is. Multibyte characters are referenced as strings (and are therefore of type *char **); before parsing, a string is indistinguishable from a multibyte character. The zero byte is still used as a string (and MB character) terminator.

A string of MB characters can be considered a null-terminated array of bytes, exactly like a traditional string. A multibyte string may contain characters from multiple codesets. Usually, this is done by incorporating special bytes that indicate that the next character (and only the next character) will be in a different codeset. Very little application code should ever need to be aware of that, though; you should use the available library routines to find out information about multibyte strings rather than look at the underlying byte structure, because that structure varies from one encoding to another. For one example of an encoding that allows characters from multiple codesets, see "EUC" on page 406.

Use of Multibyte Strings

Multibyte strings are very easy to pass around. They efficiently use space (both data and disk space), since "extra" bytes are used only for characters that require them. MB strings can be read and written without regard to their contents, as long as the strings remain intact. Displaying MB strings on a terminal is done with the usual routines: **printf()**, **puts()**, and so on. Many programs (such as *cat*) need never concern themselves with the multibyte nature of MB strings, since they operate on bytes rather than on characters; so MB strings are often used for string I/O.

Manipulation of individual characters in an MB string can be difficult, since finding a particular character or position in a string is nontrivial (see "Handling Multibyte Characters," below). Therefore, it is common to convert to WC strings for that kind of work.

Handling Multibyte Characters

Usually, multibyte characters are handled just like *char* strings. Editing such strings, however, requires some care.

You cannot tell how many bytes are in a particular character until you look at the character. You cannot look at the *n*th character in a string without looking at all the previous *n* - 1 characters, because you cannot tell where a character starts without knowing where the previous character ends. Given a byte, you don't know its position within a character. Thus, we say the string has *state* or is *context-sensitive*; that is, the interpretation we assign to any given byte depends on where we are in a character.

This analysis of characters is locale-dependent, and therefore must be done by routines that understand locale.

Conversion to Constant-Size Characters

Multibyte characters and strings are convertible to wchars using **mbtowc()** for individual characters and **mbstowcs()** for strings (see the mbtowc(3) and mbstowcs() reference pages).

Finding the Number of Bytes in a Character

To find out how many bytes make up a given single MB character, use **mblen()**, as shown in Example 16-1 (see also the mblen(3) reference page).

Example 16-1   Find Number of Bytes in an MB Character

```
#include <stdlib.h>
. . .
size_t n;
int len;
char *pStr;
. . .
len = mblen(pStr, n); /* examine no more than n bytes */
```

It is the application's responsibility to ensure that *pStr* points to the beginning of a character, not to the middle of a character.

The maximum number of bytes in a multibyte character is MB_LEN_MAX, which is defined in *limits.h*. The maximum number of bytes in a character under the current locale is given by the macro MB_CUR_MAX, defined in *stdlib.h*.

How Many Bytes in an MB String?

Since **strlen()** simply counts bytes before the first NULL, it tells you how many bytes are in an MB string.

How Many Characters in an MB String?

When **mbstowcs()** converts MB strings to WC strings, it returns the number of characters converted. This is the simplest way to count characters in an MB string.

Note: Many code segments that deal with individual characters within a string are better served by wide character strings. Because counting often involves conversion, such segments are often better served by working with a WC string, then converting back.

Getting the length without performing the conversion is straightforward, but not as simple. **mbtowc()** converts one character and returns the number of bytes used, but returns the same information without conversion if a NULL is passed as the address of the WC destination. Thus

```
len = mblen(pStr, n);
```

is equivalent to

```
len = mbtowc((wchar_t *) NULL, pStr, n);
```

In fact, **mblen()** calls **mbtowc()** to perform its count. Therefore, counting characters in an MB string without converting would look like the code in Example 16-2.

Example 16-2   Counting MB Characters Without Conversion

```
int cLen;
char *tStr = pStr;

numChars = 0;
cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
while (cLen > 0) {
    tStr += cLen;
    numChars++;
    cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
    if (cLen == -1)
        numChars = cLen; /* invalid MB character */
}
```

Wide Characters

A wide character (*WC* or *wchar*) is a data object of type *wchar_t*, which is guaranteed to be able to hold the system's largest numerical code for a character. *wchar_t* is defined in *stdlib.h*. Under IRIX 4.0.x, **sizeof(wchar_t)** was 1. In IRIX 5.1 and above, it is 4. All wchars on a system are the same size, independent of locale, encoding, or any other factors.

Uses for wchar Strings

The single advantage of WC strings is that all characters are the same size. Thus, a string can be treated as an array, and a program can simply index into the array in order to modify its contents. Most applications' *char* manipulation routines work with little modification other than a type change to *wchar_t*, with appropriate attention to byte count and **sizeof()**.

So, when applications have significant string editing to perform, they typically keep the strings in WC format while doing that editing. Those WC strings may or may not be converted to or from MB strings at other points in the application.

Wide characters are often large and are not as space efficient as multibyte strings. Applications that do not need to perform string editing probably shouldn't use *wchar*s. If an application intends to both maintain and edit large numbers of strings, then the developer needs to make size and complexity trade-off decisions.

Support Routines for Wide Characters

Analogs to the routines defined in *string.h* and *stdio.h* are supplied in *libw.a* and defined in *widec.h*. This includes routines such as **getwchar()**, **putwchar()**, **putws()**, **wscpy()**, **wslen()**, and **wsrchr()** (see the wcstring(3) reference page).

Conversion to MB Characters

Wide characters and strings are convertible to MB strings via **wctomb()** and **wcstombs()**, respectively.

Reading Input Data

Input can be divided into two categories: user events and other data. This section deals with nonuser-originated data, which is assumed to come from file descriptors or streams. User events are discussed in "Translating User Input" on page 385.

It is generally fair to assume that unless otherwise specified, data read by an application is encoded suitably for the current locale. Text strings typically are in MB format.

Streams can be read in WC format by using routines defined in *widec.h*.

Cultural Items

This section discusses several aspects of a locale that may differ between locales. It includes these topics:

- "Collating Strings" describes string collation.
- "Specifying Numbers and Money" explains some monetary formats, and the **printf()** and **localeconv()** functions.
- "Formatting Dates and Times" covers using **strftime()** to format of dates and times.
- "Character Classification and ctype" discusses associations between character codes, and using macros and functions from */usr/lib/ctype.h*.
- "Regular Expressions" presents information for developers who do their own regular expression parsing and matching.

Also see "Cultural Data" for additional information.

Collating Strings

Different locales can have different rules governing collation of strings, even within identical encodings.

In English, sorting rules are extremely simple: each character sorts to exactly one unique place. Under ASCII (C locale), the characters are even in numeric order. However, neither of those statements is necessarily true for other languages and other codesets. It should be noted that the sorting in en_US locale is different from sorting in C locale. As a result, en_US is not equal to C locale. Furthermore:

- Sorting order for a language may be completely unrelated to the (numerical) order of the characters in a given encoding.

- Even with a correctly sorted list of the characters in a character set, you may not be able to sort words properly.

- Locales using identically encoded character sets may use very different sorting rules.

Programs using ASCII can do simple arithmetic on characters and directly calculate sorting relationships; such programs frequently rely on truisms such as the fact that

```
'a' < 'b'
```

in ASCII. But internationalized programs cannot rely on ASCII and English sorting rules. Consider some non-English collation rule types:

- *One-to-Two* mappings collate certain characters as if they were two. For example, the German *ß* collates as if it were "ss."

- *Many-to-One* mappings collate a string of characters as if they were one. For example, Spanish sorts "ch" as one character, following "c" and preceding "d." In Spanish, the following list is in correct alphabetical order: *calle*, *creo*, *chocolate*, *decir*.

- *Don't-Care Character* rules collate certain characters as if they were not present. For example, if "-" were a don't-care character, "co-op" and "coop" would sort identically.

- *First-Vowel* rules sort words based first on the first vowel of the word, then by consonants (which may precede or follow the vowel in question).

- *Primary/Secondary* sorts consider some characters as equals until there is a tie. For example, in French, a, á, à, and â all sort to the same primary location. If two strings (such as "tache" and "tâche") collate to the same primary order, then the secondary sort distinguishes them.

- Special case sorts exist for some Asian languages. For example, Japanese *kanji* has no strict sorting rules. *Kanji* strings can be sorted by the strokes that make up the characters, by the *kana* (phonetic) spellings of the characters, or by other agreed-upon rules.

It should be clear that a programmer cannot hope to collate strings by simple arithmetic or by traditional methods.

Locale-specific collation should be performed with **strcoll()** and **strxfrm()**. These are table-driven functions; the tables are supplied as part of locale support. The value of LC_COLLATE determines which ordering table to use. (See the strcoll(3) and strxfrm(3) reference pages.)

strcoll() has the same interface as **strcmp()** and can be directly substituted into code that uses **strcmp()**. However, **strcoll()** can consume more CPU time, so where it is used in a time-critical loop you may have to redesign.

Specifying Numbers and Money

Format of simple numbers differs from locale to locale. Characters used for decimal radix and group separators vary. Grouping rules may also vary. Even though we assume that decimal numbers are universal, there are some eighteen varying aspects of numeric formatting defined by a locale. Many of these are details of monetary formatting.

For example, Germany uses a comma to denote a decimal radix and a period to denote a group separator. English reverses these. India groups digits by two except for the last three digits before the decimal radix. Many locales have particular formats used for money, some of which are shown in Table 16-3.

Table 16-3 Some Monetary Formats

Country	Positive Format	Negative Format
India	Rs1,02,34,567.89	Rs(1,02,34,567.89)
Italy	L.10.234.567	-L.10.234.567
Japan	¥10,234,567	-¥10,234,567
Netherlands	F10.234.567,89	F-10.234.567,89
Norway	Kr10.234.567,89	Kr10.234.567,89-
Switzerland	SFr10,234,567.89	SFr10,234,567.89C

Using printf()

printf() function, detailed in the printf(3S) reference page, examines LC_NUMERIC and chooses the appropriate decimal radix. If none is available, it tries to use ASCII period. No further locale-specific formatting is done directly by **printf()**. However, see "Variably Ordered Referencing of printf() Arguments," for a way to handle locale-specific ordering of syntactic elements in messages.

Using localeconv()

The **localeconv()** function, detailed in the localeconv(3C) reference page, can be called to find out about numeric formatting data, including the decimal radix (inappropriately called *decimal_point*), the grouping separator (inappropriately called *thousands_sep*), the grouping rules, and a great deal of monetary formatting information.

The **localeconv()** function leaves actual use of formatting information other than the decimal radix to the application.

Using strfmon()

The **strfmon()** function, detailed in the strfmon(3S) reference page, is new with IRIX version 6.2. Like **sprintf()**, **strfmon()** takes an output area, a format string that contains conversion specifications, and one or more argument values to be converted. It creates an output string containing fixed data and converted values.

Only two conversion types are supported: *%i* to convert a double value to international currency representation, and *%n* to convert a double value to national currency representation. You can use **strfmon()** to format currency values as strings, and then use **printf()** or other functions to write the formatted strings.

Formatting Dates and Times

All of these dates can mean the same thing to different people:

```
92.1.4
4/1/92
1/4/92
```

All of these can mean the same time to different people:

```
2:30 PM
14:30
14h30
```

Dates and times can be easily formatted by using **strftime()**, which gives a host of options for displaying locale-specific dates and times. The **ascftime()** and **cftime()** functions give further options, but should be avoided because they do not conform to ANSI and XPG/4 specifications. The old **asctime()** and **ctime()** functions are now obsolete; use **strftime()** instead. For more information, see the strftime(3C) reference page.

Character Classification and ctype

The *ctype.h* header file is described in the ctype(3C) reference page and defines macros to determine various kinds of information about a given character: **isalpha()**, **isupper()**, **islower()**, **isdigit()**, **isxdigit()**, **isalnum()**, **isspace()**, **ispunct()**, **isprint()**, **isgraph()**, **iscntrl()**, and **isascii()**.

When programmers knew that a character set was ASCII, some convenient assumptions could be made about characters and letters. It was common for programmers to do arithmetic with the ASCII code values in order to perform some simple operations. For example, raising a character to upper case could be done by subtracting the difference between the code for *a* and the code for *A*. Numeric characters could be identified by inspection: if they fell between *0* and *9*, they were numeric; otherwise, they weren't. You could tell if a character was (for instance) printable, a letter, or a symbol by comparing to known encoding values. Macros for such activity have long been available in *ctype.h*, but lots of programs did character arithmetic anyway. Since character encoding and linguistic semantics are completely independent, such arithmetic in an internationalized program leads to unpleasant results.

Furthermore, characters exist outside of ASCII that break some non-arithmetic assumptions. Consider the German character *ß* which is a lowercase alphabetic character (letter), yet has no uppercase. Consider also French (as written in France), where the uppercase of *é* is *E*, not *É*.

Clearly, the programmer of an internationalized application has no way of directly computing all the character associations that were available in English under ASCII.

Strict avoidance of arithmetic on character values should remove any trouble in this area. The macros in *ctype.h* are table-driven and are therefore locale-sensitive. If you think of characters as abstract characters rather than as the numbers used to represent them, you can avoid pitfalls in this area.

Regular Expressions

XPG/4 specifies some extensions to traditional regular expression syntax for internationalized software. Few application developers do their own regular expression parsing and matching, however, so we do not include full details here. Briefly, the extensions provide the ability to specify matches based on:

- character class (such as *alpha*, *digit*, *punct*, or *space*)

- equivalence class (for instance, *a*, *á*, *à*, *â*, *A*, *Á*, *À*, and *Â* may be equivalent)

- collating symbols (allowing you to match the Spanish *ch* as one element because it is a single collating token)

- generalization of range specifications of the form $[c_1\text{-}c_2]$ to include the above

If you are processing expressions, see the description of internationalized regular expression grammar in "Using Regular Expressions."

Locale-Specific Behavior

You can internationalize an application so it can span a range of language and cultural environments. This section covers some locale-specific topics you should consider when internationalizing an application. Topics include

- "Overview of Locale-Specific Behavior"

- "Native Language Support and the NLS Database"

- "Using Regular Expressions"

- "Cultural Data"

Much of the information in this section is from the *X/Open Portability Guide*. For additional information on locale-specific behavior, refer to the *X/Open Portability Guide*, *Volume 3*, "XSI *Supplementary Definitions*."

Overview of Locale-Specific Behavior

This section covers

- "Local Customs"
- "Regular Expressions"
- "ANSI X3.159-198X Standard for C"

Local Customs

To meet the requirements of local customs, the X/Open Native Language System (NLS) interface provides a set of library functions that allow cultural data appropriate to the user to be determined at run-time.

Regular Expressions

Regular expressions provide pattern-matching facilities for text. A variety of regular expression support libraries are supplied with IRIX. Most of them parse regular expressions in terms of machine collating sequences, the English language, and the ASCII coded character set.

When a program deals with internationalized input text, it is important to extend regular expression facilities to cover internationalized strings and coded character sets. It is difficult to write regular expressions that apply to more than one language, or to languages with accented/multi-character collating elements because of limitations in syntax.

Application programs can use the *wsregexp* function library, documented in the wsregexp(3W) reference page, to support internationalized regular expression behavior.

ANSI X3.159-198X Standard for C

The American National Standards Committee X3J11 standard for the C programming language includes a number of library functions that are defined to operate internationally; that is, they modify their operation in a manner appropriate to the user's native language and cultural environment.

The X/Open definition includes the international functions in Table 16-4 as defined in *Draft ANSI X3.159, Programming Language C*. ANSI functions that are enhanced by the X/Open definition are marked with an asterisk.

Table 16-4   ANSI Compatible Functions

Function	Function (continued)
atof()	scanf() *
fprintf() *	setlocale()
fscanf() *	sprintf() *
isalnum()	sscanf() *
isalpha()	strcoll()
isgraph()	sterror()
islower()	strftime()
isprint()	strtod()
ispunct()	strxfrm()
isspace()	tolower()
isupper()	toupper()
printf() *	

Draft ANSI X3.159, Programming Language C also defines a number of multi-byte functions, and an additional function for manipulating monetary values. At this stage, the X/Open definition is only guaranteed to work correctly for single-byte 8-bit characters, and thus does not include the multi-byte functions.

In addition, X/Open defines internationalized regular expression compile and match functions, native language message-handling functions, and native language versions of the error-handling functions (see Table 16-5).

Table 16-5   X/Open Additional Functions

Function	Function (continued)
catclose()	regexp()
catgets()	vfprintf()
catopen()	vprintf()
nl_langinfo()	vsprintf()
perror()	

Native Language Support and the NLS Database

The X/Open NLS interface defines the functional capabilities of a generic database that holds various language-dependent entities. This section describes those entities:

- "Configuration Data"
- "Collating Sequence Tables"
- "Character Classification Tables"
- "Shift Tables"
- "Language Information"

Configuration Data

Configuration data identify the languages supported on a system in terms of the recognized settings of language, territory, and codeset. Each valid combination of these settings has its own set of collating sequence, character classification and shift tables, language information data, and message catalogs.

Collating Sequence Tables

Collating sequence tables define the collating sequence for each supported language. The binary values of characters in the associated coded character set are used as indices into the table, individual entries of which indicate the relative position of that character in the language collating sequence. The interface definition supports the following capabilities:

- one-to-one character mappings

- one-to-two character mappings, where certain characters require treatment as if they were two characters

- n-to-one character mappings, where certain character sequences require treatment as if they represented a single character in the collating sequence. The maximum value of N is defined separately for each supported language, where N is a number in the range [1,{NL_NMAX}].

- don't care characters, where certain characters are ignored by the collating sequence

These capabilities extend to providing support for the relative ordering of collating elements within an equivalent class (for example, where two characters are first compared for equality ignoring accents, and if equal, are then ordered by accent sequence).

Character Classification Tables

These contain the lookup tables for character classification. Each character code from the defined coded character set is used as an index into the relevant language lookup table. Each entry language lookup table contains a series of flags identifying the truth or falsehood of a particular language assertion, such as

- upper-case alphabetic character

- lower-case alphabetic character

- punctuation character

- control character

- space character

Shift Tables

Shift tables contain the corresponding upper- and lower-case combinations for each character defined in a coded character set. Thus, the upshifted or downshifted value of a character can be determined by accessing the relevant character entry in the shift table.

Language Information

Language information (or *langinfo*) contains message text specific to a particular localization. The library function **nl_langinfo()** provides a procedural interface to this data, allowing applications to discover cultural and language-specific information at run-time. Individual items of *langinfo* data are identified by constants in *Volume 2, XSI System Interfaces and Headers, <langinfo.h>*.

Information specific to a culture or language includes the following:

- Date and time formats

- Days of the week and months of the year

- Abbreviated names of days and months

- Radix character

- Separator for thousands

- Affirmative and negative responses to yes/no questions

- Currency symbol and its position within a currency value

Using Regular Expressions

Regular expression are used widely throughout the services and are powerful mechanisms for locating and manipulating patterns in text. In order to be compatible with a variety of historic UNIX systems, the IRIX Developer's Option includes the unique regular expression library sets listed in Table 16-6. Note that only the last, wsregexp, supports internationalization.

Table 16-6   Regular Expression Libraries in IRIX

Library Documentation	Type of Support Provided
regcmp(3G)	Function **regcmp()** compiles a pattern string; **regex()** applies the pattern to a target string. Syntax is said to be that of *ed* but "syntax and semantics have been changed slightly" in unspecified ways.
regcmp(1)	Command applies **regcmp()** against a file of pattern strings, generating C code for literal strings that can be included in a source program to preclude having to compile patterns at run-time.
REGEX(3)	Function **re_comp()** compiles a pattern string; **re_exec()** applies the last-compiled pattern against a target string. No means of storing compiled patterns. No documentation of supported syntax, but cross-references ed(1), with which it may or may not be compatible.
regexp(5)	Function **compile()** compiles a pattern string; **step()** or **advance()** applies a stored pattern against a target string. Unusual interface compiles these functions directly into your source module, using macro functions you must define. Pattern syntax clearly documented.
wsregexp(3W)	Function **wsrecompile()** compiles a pattern string; **wsrestep()** or **wsrematch()** applies a pattern against a target. Both pattern and target strings are wide characters. Expression syntax is that of regexp augmented with internationalization expressions.

Internationalized Regular Expressions

A few utilities distributed with IRIX, in particular *grep* (see the grep(1) reference page) support internationalized regular expressions, which provide additional syntax for matching character classes, sequences, or ranges. The internationalized regular expressions supported by the **wsregexp** library are as shown in Table 16-7.

Table 16-7 Character Expressions in Internationalized Regular Expressions

Expression	Description
c	The single character *c* where *c* is not a special character.
[[:*class*:]]	A character class expression. Any character of type **class**, as defined by category **LC_CTYPE** in the program's locale (for example, see **isalpha()**). For *class*, substitute one of the following: *alpha*, a letter *upper*, an upper-case letter *lower*, a lower-case letter *digit*, a decimal digit *xdigit*, a hexadecimal digit *alnum*, an alphanumeric (letter or digit) *space*, a character that produces white space in displayed text *punct*, a punctuation character *print*, a printing character *graph*, a character with a visible representation *cntrl*, a control character
[[=*c*=]]	An equivalence class. Any collation element defined as having the same relative order in the current collation sequence as *c*. As an example, if *A* and *a* belong to the same equivalence class, then both *[[=A=]b]* and *[[=a=]b]* are equivalent to *[Aab]*.

Table 16-7 (continued)   Character Expressions in Internationalized Regular Expressions

Expression	Description
[[.*cc*.]]	A collating symbol. Multi-character collating elements must be represented as collating symbols to distinguish them from single-character collating elements. As an example, if the string *ch* is a valid collating element, then *[[.ch.]]* is treated as an element matching the same string of characters, while *ch* is treated as a simple list of *c* and *h*. If the string is not a valid collating element in the current collating sequence definition, the symbol is treated as an invalid expression.
[*c-c*]	Any collation element in the character expression range *c-c*, where *c* can identify a collating symbol or an equivalence class. If the hyphen character, -, appears immediately after an opening square bracket, or immediately prior to a closing square bracket, it has no special meaning.

Within square brackets, a period (.) that is not part of a *[[.c.]]* sequence, a colon (:) that is not part of a *[[:class:]]* sequence, and an equals sign (=) that is not part of a *[[=c=]]* sequence matches itself.

Table 16-8 shows examples of simple regular expressions.

Table 16-8   Examples of Internationalized Regular Expressions

Pattern	Definition
[[=a=]]bcd	any form of *a* followed by *bcd*
[[.ch.]-e]	any element that collates between *ch* and *e*
[[:lower:]]	any lower case letter

Cultural Data

The items of cultural data listed in Table 16-9 are defined in the C locale.

Table 16-9   Cultural Data Names, Categories, and Settings

Item	Category	Setting for the C Locale
D_T_FMT	LC_TIME	"%a %b %c %H:%M:%S %Y"
D_FMT	LC_TIME	"%m/%d/%y"
T_FMT	LC_TIME	"%H:%M:%S"
AM_STR	LC_TIME	"AM"
PM_STR	LC_TIME	"PM"
DAY_1	LC_TIME	"Sunday"
DAY_2	LC_TIME	"Monday"
DAY_3	LC_TIME	"Tuesday"
DAY_4	LC_TIME	"Wednesday"
DAY_5	LC_TIME	"Thursday"
DAY_6	LC_TIME	"Friday"
DAY_7	LC_TIME	"Saturday"
ABDAY_1	LC_TIME	"Sun"
ABDAY_2	LC_TIME	"Mon"
ABDAY_3	LC_TIME	"Tue"
ABDAY_4	LC_TIME	"Wed"
ABDAY_5	LC_TIME	"Thu"
ABDAY_6	LC_TIME	"Fri"
ABDAY_7	LC_TIME	"Sat"
MON_1	LC_TIME	"January"
MON_2	LC_TIME	"February"

Table 16-9 (continued)   Cultural Data Names, Categories, and Settings

Item	Category	Setting for the C Locale
MON_3	LC_TIME	"March"
MON_4	LC_TIME	"April"
MON_5	LC_TIME	"May"
MON_6	LC_TIME	"June"
MON_7	LC_TIME	"July"
MON_8	LC_TIME	"August"
MON_9	LC_TIME	"September"
MON_10	LC_TIME	"October"
MON_11	LC_TIME	"November"
MON_12	LC_TIME	"December"
ABMON_1	LC_TIME	"Jan"
ABMON_2	LC_TIME	"Feb"
ABMON_3	LC_TIME	"Mar"
ABMON_4	LC_TIME	"Apr"
ABMON_5	LC_TIME	"May"
ABMON_6	LC_TIME	"Jun"
ABMON_7	LC_TIME	"Jul"
ABMON_8	LC_TIME	"Aug"
ABMON_9	LC_TIME	"Sep"
ABMON_10	LC_TIME	"Oct"
ABMON_11	LC_TIME	"Nov"
ABMON_12	LC_TIME	"Dec"
RADIXCHAR	LC_NUMERIC	"."
THOUSEP	LC_NUMERIC	" "

Table 16-9 (continued)   Cultural Data Names, Categories, and Settings

Item	Category	Setting for the C Locale
YESSTR	LC_ALL	"yes"
NOSTR	LC_ALL	"no"
CRNCYSTR	LC_MONENTARY	" "

NLS Interfaces

The NLS interfaces listed here are utilities and library functions.

NLS Utilities

The list below identifies the minimum set of utilities that provide 8-bit transparency on all X/Open compliant systems. The definitions of these commands, in terms of their syntax and parameters, are not changed by the operation of NLS.

ar	*date*	*kill*	*pg*	*tail*	*uulog*
awk	*diff*	*lex*	*pr*	*tar*	*uuname*
cancel	*echo*	*ln*	*ps*	*tee*	*uupick*
cat	*ed*	*lp*	*pwd*	*test*	*uustat*
cc	*egrep*	*lpstat*	*red*	*tr*	*uuto*
cd	*expr*	*ls*	*rm*	*true*	*uux*
chgrp	*false*	*mail*	*rmdir*	*tty*	*wait*
chmod	*fgrep*	*mailx*	*sed*	*umask*	*wc*
chown	*find*	*mkdir*	*sh*	*uname*	*who*
cmp	*gencat*	*mv*	*sleep*	*uniq*	
cp	*grep*	*pack*	*sort*	*unpack*	
cpio	*iconv*	*pcat*	*stty*	*uucp*	

The *cc*, *yacc*, and *lex* commands provide 8-bit transparency for characters contained in character strings, character constants, and comment strings. An 8-bit character string enables a programmer to define default messages in languages other than English. The support of 8-bit characters in identifier names is implementation defined.

The 8-bit operation of commands that communicate with other systems cannot be guaranteed in all circumstances. For example, intersystem mail may be restricted to 7-bit data by the underlying network, 8-bit data and filenames may not be portable to noninternationalized systems, and so forth. Under these circumstances, it is recommended that you use only characters defined in the ASCII 7-bit range of characters for data transfer between machines, and you use only characters defined in the Portable Filename Character Set for naming remote files.

NLS Library Functions

The list below shows library functions usable by internationalized application programs

atof()	**isgraph()**	**scanf()**	**toupper()**
catclose()	**islower()**	**setlocale()**	**vfprintf()**
catgets()	**isprint()**	**sprintf()**	**vprintf()**
catopen()	**ispunct()**	**sscanf()**	**vsprintf()**
fprint()	**isspace()**	**strcoll()**	
fscanf()	**isupper()**	**strerror()**	
gcvt()	**nl_langinfo()**	**strftime()**	
isalnum()	**perror()**	**strtod()**	
isalpha()	**printf()**	**strxfrm()**	
iscntrl()	**regexp()**	**tolower()**	

Also, all functions defined in the *X/Open Portability Guide, Volume 2, XSI System Interfaces and Headers*, and *X/Open Portability Guide, Volume 3, XSI Curses Interface*, provide 8-bit transparency on X/Open compliant systems.

XSI Curses Interface

The XSI curses interface is internationalized. For more information, see the *X/Open Portability Guide, Volume 3, XSI Curses Interface*.

Strings and Message Catalogs

Message catalogs are compiled databases of strings. While a major role of message catalogs is to provide communications text in locale-specific natural language, the strings can be used for any purpose. The idea is that an application uses only strings from a catalog, thus allowing localizers to supply catalogs suitable for a given locale.

Two different and incompatible interfaces to message catalogs exist in IRIX: *MNLS* and *XPG/4*. Developers working on SVR4 or other AT&T code, or related base-system utilities, probably use MNLS. Developers working on independent projects probably use XPG/4. Neither is a solid standard, but XPG/4 is closer to being a standard than MNLS. Thus applications developers who have to choose between the two interfaces are encouraged to use XPG/4 to maximize their portability. XPG/4 seems to be popular in Europe.

This section covers the following topics:

- "XPG/4 Message Catalogs" on page 366
- "SVR4 MNLS Message Catalogs" on page 370
- "Variably Ordered Referencing of printf() Arguments" on page 375

XPG/4 Message Catalogs

The XPG/4 message catalog interface requires that a catalog be opened before it is read, and requires that catalog references specify a catalog descriptor.

Since catalog references include a default to be used in case of failure, applications will work normally without a catalog when in the default locale. This means catalog generation is exclusively the task of localizers. But in order to inform the localizer as to what strings to translate and how they should comprise a catalog, the application developer should provide a catalog for the developer's locale.

Opening and Closing XPG/4 Catalogs

catopen() locates and opens a message catalog file:

```
#include <nl_types.h>
nl_catd catopen(char *name, int unused);
```

The argument *name* is used to locate the catalog. Usually, this is a simple, relative pathname that is combined with environment variables to indicate the path to the catalog (see "XPG/4 Catalog Location" for details). However, the catalog assumes names that begin with "/" are absolute pathnames. Use of a hard-coded pathname like this is strongly discouraged; it doesn't allow the user to specify the catalog's locale through environment variables.

When an application is finished using a message catalog, it should close the catalog and free the descriptor using **catclose()**:

```
int catclose(nl_catd);
```

Using an XPG/4 Catalog

Catalogs contain sets of numbered messages. The application developer must know the contents of the catalog in order to specify the set and number of a message to be obtained.

catgets() is used to retrieve strings from a message catalog (see the catopen(3) and catgets(3) reference pages). Example 16-3 shows a program that reads the first message from the first message set in the appropriate catalog, and displays the result.

Example 16-3   Reading an XPG/4 Catalog

```
#include <stdio.h>
#include <locale.h>
#include <nl_types.h>

#define SET1      1
#define WRLD_MSG  1

int main(){
    nl_catd msgd;
    char *message;
    setlocale(LC_ALL, "");

msgd = catopen("hw",0);
    message = catgets(msgd, SET1, WRLD_MSG,"Hello, world\n");
    printf(message);
    catclose(msgd);
}
```

The previous example uses **printf()** instead of **puts()** in order to make a point: the format string of **printf()** came from a catalog. Note the crucial difference between these two statements:

```
printf(catgets(msgd, set, num, defaultStr));
printf("%s", catgets(msgd, set, num, defaultStr));
```

In the first statement, the catalog provides the **printf()** formatting string, possibly containing conversion specifications and escape sequences. In the second statement, the string from the catalog is treated as data and not interpreted for conversion specifications. For further discussion of issues relating to this important distinction, see "Variably Ordered Referencing of printf() Arguments."

XPG/4 Catalog Location

XPG/4 message catalogs are located using the environment variable NLSPATH. The default NLSPATH is */usr/lib/locale/%L/LC_MESSAGES/%N*, where *%L* is filled in by the LANG environment variable and *%N* is filled in by the *name* argument to **catopen()**. NLSPATH can specify multiple pathnames in ordered precedence, much like the PATH variable. The following is a sample NLSPATH assignment:

```
NLSPATH=/usr/lib/locale/%L/LC_MESSAGES/%N:/usr/local/lib/locale/%L/LC_MESSAGES/
%N:/usr/defaults/%N
```

Creating XPG/4 Message Catalogs

Message catalogs are of this general form (these forms are detailed in the gencat(1) reference page):

```
$set n comment
a message-a\n
b message-b\n
c message-c\n
$quote "
d " message-d "
$this is a comment
```

Each message is identified by a *message number* and a *set*. Sets are often used to separate messages into more easily usable groups, such as error messages, help messages, directives, and so on. Alternatively, you could use a different set for each source file, containing all of that source file's messages.

$set *n* specifies the beginning of set *n*, where *n* is a set identifier in the range from 1 to NL_SETMAX. All messages following the **$set** statement belong to set *n* until either a **$delset** or another **$set** is reached. You can skip set numbers (for example, you can have a set 3 without having a set 2), but the set numbers that you use must be listed in ascending numerical order (and every set must have a number). Any string following the set identifier on the same line is considered a comment.

$delset *n* deletes the set *n* from a message catalog.

$quote *c* specifies a quote character, *c*, which can be used to surround message text so that trailing spaces or null (empty) messages are visible in a message source line. By default, there is no quote character and messages are separated by newlines. To continue a message onto a second line, add a backslash to the end of the first line:

```
$set 1
1 Hello, world.
2 here is a long \
string.\n
3 Hello again.
n message-text-n
```

Message #2 in set #1 is "here is a long string.\n".

Compiling XPG/4 Message Catalogs

After creating the message catalog sources, you need to compile them into binary form using *gencat*, which has the following syntax:

```
gencat catfile msgfile [msgfile ...]
```

where *catfile* is the target message catalog and *msgfile* is the message source file (see the gencat(1) reference page). If an old *catfile* exists, *gencat* attempts to merge new entries with the old. *gencat* "resolves" set and message number conflicts with new information replacing the old.

The *catfile* then needs to be placed in a location where **catopen()** can find it; see the "XPG/4 Catalog Location" on page 368.

SVR4 MNLS Message Catalogs

There are many ways to use strings from MNLS message catalogs. You can get strings directly and then use them, or you can use output routines that search catalogs.

Putting MNLS Strings Into a Catalog

An MNLS catalog source file contains a list of strings separated by new lines. For an empty string, an empty line is used. Strings are referenced by line number in the original source file.

Applications access the catalog by line number, so it's very important not to change the line numbers of existing catalog entries. This means that, when you want to add a new string to an existing catalog source, you should always append it to the end of the file—if you put it in the middle of the file, then you change the line number for subsequent strings.

The following tools can help you compile MNLS message catalogs:

exstr(1)	Searches a C source file for literal strings and lists them, or replaces them with MNLS function calls.
mkmsgs(1)	Creates a message catalog for a particular locale, converting source text lines to the form used by exstr.
srchtxt(1)	Displays selected strings from a message catalog.

When a file of strings is ready to be compiled, simply run *mkmsgs* and put the results in the directory */usr/lib/locale/localename/LC_MESSAGES*.

Using MNLS in Shell Scripts

One difference between MNLS and XPG/4 catalog functions is that the MNLS catalog can be used from commands, and hence it can be used to internationalize a shell script. The following table summarizes MNLS functions that have both a command line and a function library version:

gettxt(1)	Retrieve a string from the catalog.
lfmt(1)	Retrieve a format string, insert arguments, display to *stderr* and to system log or textport.
pfmt(1)	Retrieve a format string, insert arguments, display to *stderr*.

Specifying MNLS Catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified every time a string is needed.

To specify the default message catalog to be used by subsequent calls to MNLS functions that reference catalogs, use **setcat()**:

```
#include <pfmt.h>
char *setcat(const char *catalog);
```

catalog is limited to 14 characters, and may contain no character equal to zero or to the ASCII codes for slash (/) or colon (:). (See the setcat(3) reference page.)

setcat() doesn't check to see if the catalog name is valid; it just stores the string for future reference. For an example of use, see the following topic. The catalog indicated by the string must be found in the directory */usr/lib/locale/localename/LC_MESSAGES*.

Getting Strings From MNLS Message Catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified in each reference call. Strings are read from a catalog via **gettxt()** (see the gettxt(3) reference page):

```
#include <unistd.h>
char *gettxt(const char *msgid, const char *defaultStr);
```

msgid is a string containing two fields separated by a colon:

```
msgfilename:msgnumber
```

The *msgfilename* is a catalog name as described previously in the "Specifying MNLS Catalogs" on page 371. For example, to get message 10 from the *MQ* catalog, you could use either:

```
char *str = gettxt("MQ:10", "Hello, world.\n");
```

or

```
setcat("MQ");
str = gettxt(":10", "Hello, world.\n");
```

Using pfmt()

pfmt() is one of the most important routines dealing with MNLS catalogs, because it is used to produce most system diagnostic messages. **pfmt()** formats like **printf()** and produces standard error message formats (see the pfmt(3) reference page for the function, or pfmt(1) for shell use). It can usually be used in place of **perror()**. For example,

```
pfmt(stderr, MM_ERROR, "MQ:64:Permission denied");
```

would produce, by default (such as when the Mozambique locale is unavailable),

```
ERROR: Permission denied.
```

The syntax of **pfmt()** is

```
#include <pfmt.h>
int pfmt(FILE *stream, long flags, char *format, ... );
```

The *flags* are used to indicate severity, type, or control details to **pfmt()**. The format string includes information specifying which message from which catalog to look for. Flag details are discussed in the following section. The format is discussed in the "Format Strings for pfmt()" on page 373.

Labels, Severity, and Flags

pfmt() flags are composed of several groups; specify no more than one from each group. Specify multiple flags by using OR. The groups are as follows:

output format control	MM_NOSTD, MM_STD
catalog access control	MM_NOGET, MM_GET
severity	MM_HALT, MM_ERROR, MM_WARNING, MM_INFO
action message specification	MM_ACTION

pfmt() prints messages in the form *label:severity:text*. *Severity* is specified in the *flags*. The *text* comes from a message catalog (or a default) as specified in the *format*, and the *label* is specified earlier by the application.

In the example above, if no label has been set, we get only the output:

```
ERROR: Permission denied.
```

Typically, an application sets the label once early in its life; subsequent error messages have the label prepended. For example

```
setlabel("UX:myprog");
...
pfmt(stderr, MM_ERROR, "MQ:64:Permission denied");
```

would produce (by default)

```
UX:myprog: ERROR: Permission denied.
```

For details, consult the pfmt(3) and setlabel(3) reference pages.

Format Strings for pfmt()

pfmt() format strings are of this form:

[[*catalog*:]*messagenum*:]*defaultstring*

The *catalog* field is in the format described in "Specifying MNLS Catalogs" on page 371. *messagenum* is the message number in the catalog to use as the format. *defaultstring* specifies the string to use if the catalog lookup fails for any reason.

An important feature of **pfmt()** is its ability to refer to format arguments in format-specified order just as **printf()** does. See "Variably Ordered Referencing of printf() Arguments" for details.

Using fmtmsg()

fmtmsg() is a comprehensive formatter using the MNLS catalogs and "standard" formats. You probably won't need to use it; most applications should get by with **pfmt()**, **gettxt()**, and **printf()**. Consult the fmtmsg(3) reference page for details.

Internationalizing File Typing Rule Strings With MNLS

You can internationalize the strings defined in the LEGEND and MENUCMD rules in the File Typing Rule (FTR) file. To internationalize these rules, precede the string with the following:

`:[`*catalogname*`:]`*msgnumber*`:`

catalogname is optional and should be a valid MNLS catalog; *msgnumber* is the line number in *catalogname*. If you omit *catalogname*, the *uxsgidesktop* catalog is used by default.

You can use these rules to create your own FTR catalog. For example, an entry looks like this:

```
LEGEND :mycatalog:7:Archive 8mm Tape Drive
```

This entry uses line 7 from the catalog, *mycatalog*, as the LEGEND for this FTR. If *mycatalog* is not available, or line 7 is not accessible from *mycatalog*, "Archive 8mm Tape Drive" is used as the LEGEND.

```
LEGEND :7:Archive 8mm Tape Drive
```

This entry uses line 7 from the *uxsgidesktop* catalog, if available. Otherwise, "Archive 8mm Tape Drive" is used.

The next example,

```
MENUCMD \'mycatalog:9:Eject Tape\' /usr/sbin/eject /dev/tape
```

displays line 9 from *mycatalog*, if available. Otherwise "Eject Tape" is displayed on the menu that pops up when you click an icon that uses this FTR.

You can internationalize strings in the command part of MENUCMD and CMD rules by using *gettxt* or any other convenient policy detailed in this section. For example

```
CMD OPEN xconfirm -t "Tape tool not available"
```

can be internationalized to

```
CMD OPEN xconfirm -t "`gettxt mycatalog:376 'Tape tool not available'`"
```

In this example, *gettxt* is invoked to access line 376 from the catalog, *mycatalog*, and the string returned by *gettxt* is passed to *xconfirm* for display. If line 376 from *mycatalog* is not accessible, then *gettxt* returns the string "Tape tool not available."

For more information about FTRs, see the *IRIX Interactive Desktop Integration Guide*.

Variably Ordered Referencing of printf() Arguments

printf() and its variants can now refer to arguments in any specified order. Consider the following scenario: an application has chosen "house" from a list of objects and "white" from a list of colors. The application wishes to display this choice. The code might look like this:

```
char *obj, *color;
... /* make choices */ ...
printf("%s %s\n", color, obj);
```

The **printf()** call produces this:

```
white house
```

Even once we make sure that *obj* and *color* are localized strings, we are not quite finished. If our locale is Spanish, the **printf()** yields:

```
blanca casa
```

That is incorrect grammar; in Spanish, it should be:

```
casa blanca
```

The solution to this problem is *variably ordered referencing* of **printf()** arguments. The syntax of **printf()** format strings has been expanded to deal with this.

The original definition of **printf()** is that each conversion specification %*T* (where *T* represents any of the **printf()** conversion characters) is implicitly matched to an argument value by position. In order to deal with variably ordered strings, **printf()** allows an argument position index *D* to appear in the conversion specification following the %, so that where a format string contains %*T*, it can now contain %*D*$*T*. The value *D*, set off by a currency symbol ($), selects the argument from the argument list to be used. This means you can write

```
printf("2nd parameter is %2$s; the 1st is %1$s", p1, p2)
```

The *second* parameter is printed *first*, with the first parameter printed second. For example:

```
char *store = "Macy's";
char *obj = "a cup";

printf("At %1$s, I bought %2$s.\n", store, obj);
printf("I bought %2$s at %1$s.\n", store, obj);
```

This code displays

```
At Macy's, I bought a cup.
I bought a cup at Macy's.
```

In English, we are able to come up with strings suitable for either word order; in some other language, we might not be so lucky. Nor can we predict which order such languages might prefer. So the developer has no way of knowing how to create traditional **printf()** format strings suitable for all languages.

Developers should therefore use message catalogs for their **printf()** format strings that take linguistic parameters, and allow localizers to localize the format strings as well as text strings. This means that the localizer has much greater ability to create intelligible text. An internationalized version of the above code appears in Example 16-4.

Example 16-4   Internationalized Code

```
/* internationalized (XPG/4) version */
char *form = catgets(msgd, set, formNum,
                     "At %1$s, I bought %2$s.\n");
char *store = catgets(msgd, set, storeNum, "Macy's");
char *obj = catgets(msgd, set, objNum, "a cup");

printf(form, store, obj);
```

The unlocalized (default) version would produce

```
At Macy's, I bought a cup.
```

A localized version might produce

```
Compré una tasa en Macy's.
```

In practice, variably ordered format strings are found only in message catalogs and not in default strings. The default string usually simply uses the parameters in the order they're given, without the new variable-order format strings.

Internationalization Support in X11R6

X11R6 internationalization support is provided on the X client side; that is, the application must take care of such support instead of relying on the X server. No server changes are necessary, and the protocol is unchanged. Full backward compatibility is preserved, so a new internationalized application can run on an old server.

Note: X11R6 internationalization refers to features in X11R5 and X11R6.

X uses existing internationalization standards to do its internationalization support; there are no X-specific interfaces to set and change locale. Internationalized X applications receive no help from X when attempting multilingual support. No locales or special process states are peculiar to X.

This section covers the following topics:

- "Limitations of X11R6 in Supporting Internationalization" discusses vertical text, character sets, and Xlib interface changes.

- "Resource Names" covers encoding of resource names.

- "Getting X Internationalization Started" describes initialization of Xlib and toolkit programming.

- "Fontsets" explains specifying, creating, and using fontsets.

- "Text Rendering Routines" discusses the *XmbDrawText()*, *XmbDrawString()*, and *XmbDrawImageString()* functions.

- "New Text Extents Functions" describes a few new extents-related functions, including *XFontSetExtents*.

Limitations of X11R6 in Supporting Internationalization

Since X is locale-independent, there are some limitations on its ability to support internationalization. The X protocol and Xlib specification, together with ANSI C and POSIX restrictions, have led to certain choices being made in X11R6. These are described in the following paragraphs.

377

Vertical Text

There is no built-in support for vertical text. Applications may draw strings vertically only by laying out the text manually.

Character Sets

In previous releases of X, there was no general support for character sets other than Latin 1. X11R6, however, does allow other character sets.

X11R6 includes the definition of the *X Portable Character Set*, which is required to exist in all locales supported by Xlib. There is no encoding defined for this set; it is only a character set. The set—which is similar to printable ASCII plus the newline and tab—consists of these characters:

```
abcdefghijklmnoqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
<space> <tab> <newline>
```

The *Host Portable Character Encoding* is the encoding of the X Portable Character Set on the Xlib host. This encoding is part of X, and is thus independent of locale—the coding remains the same for all locales supported by the host.

Strings used or returned by Xlib routines are either in the Host Portable Character Encoding or a locale-specific encoding. The Xlib reference pages specify which encodings are used where. Some string constructs (such as *TextProperty*) contain information regarding their own encoding.

Xlib Interface Change

Full use of X11R6's internationalization features means calling some new routines supplied in the X11R6 Xlib. While all old Xlib applications work with the new Xlib, developers should change their code in places. These are described below.

Resource Names

Resource names are compiled into programs. Because of that, their encoding must be known independent of locale. Trying to add a level of indirection here results in a problem: you're always left with something compiled that can't be localized. Resource names therefore use the X Portable Character Set. The names may be anything; at least they'll mean something to the application author. (If the names were numbers, for example, they would be meaningless to everybody.)

Getting X Internationalization Started

Xlib's internationalization state, like that of *libc*, needs to be initialized.

Initialization for Toolkit Programming

If you're using Xt (with a widget set such as IRIS IM, Motif, or XaW), then don't use **setlocale()**. Instead, use

```
XtSetLanguageProc(NULL, NULL, NULL)
```

If you're using a toolkit other than Xt, call **setlocale()** as early as possible after execution begins.

Initialization for Xlib Programming

Initialize Xlib's internationalization state after calling **setlocale()**. Xlib is being initialized, not a server or server-specific object, so a server connection is not necessary.

Example 16-5   Initializing Xlib for a Locale

```
if ( setlocale(LC_ALL, "") == NULL )
    exit_with_error();
if ( ! XSupportsLocale() )
    exit_with_other_error();
if ( XSetLocaleModifiers("") == NULL)
    give_warning();
```

XSetLocaleModifiers() is required only for input. Just as passing an empty string to **setlocale()** honors the user's environment, so does passing an empty string to **XSetLocaleModifiers()**.

Fontsets

In X11R5 and X11R6, unlike previous releases of X, a string may contain characters from more than one codeset. There are several methods for determining which codeset a given character is in; which method is appropriate depends on the locale and the encoding used.

For information on installing and using fontsets with an application, refer to Chapter 15, "Working With Fonts."

Such multiple-codeset strings usually cannot be rendered using a single font. A *fontset* is a collection of fonts suitable for rendering all codesets represented in a locale's encoding. A fontset includes information to indicate which locale it was created in. Applications create fontsets for their own use; when a program creates a fontset, it is told which of the requested fonts are unavailable.

Example: EUC in Japanese

To render strings encoded in EUC in Japanese, an application would need fonts encoded in 8859-1, JIS X 208, and JIS X 201. The application doesn't need to know which characters in a string go with which font, since it doesn't deal with locale specifics. So it creates a fontset that is made from a list of user-specified fonts (under the assumption that the localizer has provided an appropriate list). Rendering is then done using that fontset. The locale-aware rendering system chooses the appropriate fonts for each character being rendered, from the supplied list. You can find additional information about EUC in "Asian Languages."

Specifying a Fontset

A fontset specification is just a string, enumerating XLFD names of fonts. (See *X Logical Font Description Conventions*, an MIT X Consortium standard, as well as "Font Names" on page 307.) This string can include wild card characters. For example, a specification of 16-point "fixed" fonts might be as follows:

```
char *fontSetSpecString = "*fixed-medium-r-normal*150*";
```

Based on the fonts available, a particular server might expand this to a string such as:

```
-jis-fixed-medium-r-normal--16-150-75-75-c-160-jisx0208.1983-0
-sony-fixed-medium-r-normal--16-150-75-75-c-80-iso8859-1
-sony-fixed-medium-r-normal--16-150-75-75-c-80-jisx0201.1976-0
```

Specifying the fontset by simply enumerating the fonts is perfectly acceptable:

```
char *fontSetSpecString =
"-jis-fixed-medium-r-normal*150-75-75*jisx0208.1983-0,\
-sony-fixed-medium-r-normal*150-75-75*iso8859-1,\
-sony-fixed-medium-r-normal*150-75-75*jisx0201.1976-0";
```

A German locale would work with only the ISO font; a Japanese locale might use all three; a Chinese locale would have trouble with this fontset.

The developer should specify a default fontset suitable for the default locale. Furthermore, developers should ensure that the application accepts localized fontset specifications via resources (or message catalogs) or command line options. Localizers are responsible for providing default fontset specifications suitable for their locales.

Creating a Fontset

Creating fontsets in X is simply a matter of providing a string that names the fonts, as described above.

Example 16-6   Creating a Fontset

```
XFontSet fontset;
char *base_name;  /* should get from resource */
char **missingCharsetList;
int missingCharsetCount;
char *defaultStringForMissingCharsets;

base_name = "*fixed-medium-r*150*"; /* use resources! */

fontset = XCreateFontSet(display, base_name,
                         &missingCharsetList,
                         &missingCharsetCount,
                         &defaultStringForMissingCharsets);
```

The locale in effect at create time is bound to the fontset. Fontsets are freed with **XFreeFontSet()**.

Using a Fontset

Fontsets are used when rendering text with X11R6 **Xmb** or **Xwc** text rendering routines. These routines are described in "Text Rendering Routines."

Text Rendering Routines

X11R6 includes text rendering routines that understand multibyte and wide-character strings. These routines are analogous to the X11R4 text rendering routines **XDrawText()**, **XDrawString()**, and **XDrawImageString()**. The old routines continue to operate, but do not take fontsets, and don't know how to handle characters longer than one byte.

- **XmbDrawText()** and **XwcDrawText()** take lists of *TextItems*, each of which contains (among other things) a string. The strings are rendered using fontsets. These routines allow complex spacing and fontset shifts between strings.

- **XmbDrawString()** and **XwcDrawString()** render a string using a fontset. These routines render in foreground only and use the raster operation from the current graphics context.

- **XmbDrawImageString()** and **XwcDrawImageString** also render a string using a fontset. These routines fill the background rectangle of the entire string with the background, then render the string in the foreground color, ignoring the currently active raster operation.

Consult the appropriate reference pages for more details on these routines.

New Text Extents Functions

X11R6 provides MB and WC versions of **width** and **extents** interrogation routines, supplying the maximum amount of space required to draw any character in a given fontset. These routines depend on fontsets to interpret strings and use locale-specific data.

The *XFontSetExtents* structure contains the two kinds of extents a string can have:

```
typedef struct {
    XRectangle max_ink_extent;
    XRectangle max_logical_extent;
} XFontSetExtents;
```

max_ink_extent gives the maximum boundaries needed to render the drawable characters of a fontset. It considers only the parts of glyphs that would be drawn, and gives distances relative to a constant origin. *max_logical_extent* gives the maximum extent of the *occupied space* of drawable characters of a fontset. The occupied space of a character is a rectangle specifying the minimum distance from other graphical features; other graphics generated by a client should not intersect this rectangle. *max_logical_extent* is used to compute interline spacing and the minimum amount of space needed for a given number of characters.

Here are descriptions of a few of the new extents-related functions (consult the appropriate reference pages for details):

• **XExtentsOfFontSet()** returns an *XFontSetExtents* structure for a fontset.

• **XmbTextEscapement()** and **XwcTextEscapement()** take a string and return the distance in pixels (in the current drawing direction) to the origin of the next character after the string, if the string were drawn. Escapement is always positive, regardless of direction.

• **XmbTextExtents()** and **XwcTextExtents()** take a string and return information detailing the overall rectangle bounding the string's image and the space the string occupies (for spacing purposes).

• **XmbTextPerCharExtents()** and **XwcTextPerCharExtents()** take a string and return ink and logical extents for each character in the string. Use this for redrawing portions of strings or for word justification. If the fontset might include context-dependent drawing, the client cannot assume that it can redraw individual characters and get the same rendering.

• **XContextDependentDrawing()** returns a Boolean telling whether a fontset might include context-dependent drawing.

Internationalization Support in Motif

Your applications can use Motif's internationalization capabilities. Refer to the chapter titled "Internationalization" in the *OSF/Motif Programmer's Guide* for information about the following topics:

- issues in internationalized applications

- compound strings, fonts, and text display

- localizing applications

- advanced topics in internationalization

There are some important points to remember when you internationalize and localize your application:

- At the top of your **main** program, issue the call

  ```
  XtSetLanguageProc(NULL, NULL, NULL);
  ```

- Translate your app-defaults and install it in */usr/lib/X11/$LANG/app-defaults*.

- Motif uses font sets and font lists to display text. Specify a font list in your application defaults file using the following format:

  ```
  *fontList: font-list-string:
  ```

Be sure to separate elements in the *font-list-string* as follows:

- Separate single fonts with a comma (,).

- Separate elements within a font set with a semicolon (;).

- End the string with a colon (:).

An example of specifying a Japanese *fontList* is as follows:

```
*fontList: 7x14;--mincho-*--14-*;--14-*:
```

Translating User Input

This section explains the translation of physical user events into programmatic character strings or special keyboard data (such as "backspace"). This kind of work should be done by toolkits. If you can use a toolkit to manage event processing for you, do so, and blissfully ignore this section. If you are writing a toolkit text object, or are writing a truly extraordinary application, then this section is for you.

This section on translating user input covers these topics:

- "About User Input and Input Methods" on page 385 presents an overview of user input and input methods.

- "About X Keyboard Support" on page 386 covers X keyboard support, including keys, keycodes, keysyms, and composed characters.

- "Input Methods (IMs)" on page 389 describes how input methods are opened and closed.

- "IM Styles" on page 391 discusses the use and naming of IM styles.

- "Input Contexts (ICs)" on page 394 explains an IM styles, IC values, pre-edit and status attributes, and creating and using ICs.

- "Events Under IM Control" on page 398 describes differences in processing events under IM control including *XFilterEvent()* and *LookupString* routines.

About User Input and Input Methods

Just as internationalized programs cannot assume that data is in ASCII, they cannot assume that user input will use any specific keyboard. Keyboards change from country to country and language to language; internationalized software should never assume that a certain position on the keyboard is bound to a certain character, or that a given character will be available as a single keystroke on all keyboards.

No useful physical keyboard—not even one specifically designed for multilingual work—could possibly contain a key for every character we would ever wish to type. Certainly there are characters commonly used in other areas of the world that are not present on most USA keyboards. So methods have been invented that provide for input of almost any known character on even the most naïve keyboards. These schemes are referred to as *input methods* (IMs).

Input methods vary significantly in design, use, and behavior, but there is a single API that developers use to access them. The object is for the application simply to ask for an IM and let the system check the locale and choose the appropriate IM.

Some IMs are complex; others are very simple. The API is designed to be a low-level interface, like Xlib. Usually, only toolkit text object authors must deal with the IM interfaces. However, some applications developers are unable to use toolkit objects, so the concepts are described here.

Reuse Sample Code

A sample program demonstrating some of the concepts in this section is given in Chapter 11 of the *Xlib Programming Manual, Volume One*. Looking carefully at that code may be easier than starting from scratch.

GL Input

The old GL function **qdevice()** has a hard-coded view of a keyboard (see */usr/include/gl/device.h* for details). Some flexibility, particularly for Europe, is available if you queue KEYBD instead of individual keys, but the GL has no general solution to non-ASCII input. There is no supported way to input Chinese (for instance) to the old GL.

OpenGL does not contain input code but leaves that to the operating environment, which in IRIX means X.

In short, support for internationalized input means a departure from **qread()**. Under IRIX, that means using mixed-model input, all the more reason to use a toolkit.

About X Keyboard Support

This section provides some background that may help make the following sections easier to understand.

Keys, Keycodes, and Keysyms

When a client connects to the X server, the server announces its range of *keycodes* and exports a table of *keysyms*. Each key event the client receives has a single byte *keycode*, which directly represents a physical key, and a single byte *state*, which represents currently engaged modifier keys, such as Shift or Alt.

Note: The mapping of state bits to modifiers is done by another table acquired from the server.

Keysyms are well defined, and there has been an attempt to have a keysym for every engraving one might possibly find on any keyboard, anywhere. (An *engraving* is the image imprinted on a physical key.) These are contained in */usr/include/X11/keysymdef.h*. Keysyms represent the engravings on the actual keys, but not their meanings. The server's idea of the keysym table can be changed by clients, and clients may receive *KeyMap* events when this remapping happens, but such events don't happen often.

When a client receives a Key event, it asks Xlib to use the keycode to index into its keysym table to find a list of keysyms. (This list is usually very short. Most keys have only one or two engravings on them.) Using the state byte, Xlib chooses a keysym from the list to find out what was engraved on the key the user pressed.

At this point, the client can choose to act on the keysym itself (if, for instance, it was a backspace) or it can ask for a character string represented by the keysym (or both). Generating such a string is tricky; it is discussed in "Input Methods (IMs)," below.

Details on X keyboard support can be found in *X Window System, Third Edition*, from Digital Press. Details on input methods are also available in that book, as well as in the *Xlib Programming Manual, Volume One*.

Composed Characters

There are two ways to compose characters that do not exist on a keyboard: explicit and implicit. It is common for an application to be modal and switch between the two. For example, Japanese input of kana is often done via implicit composition.

Users switch between a mode where input is interpreted as romaji (Latin characters) and a mode where input is translated to kana.

Furthermore, both styles may operate simultaneously. While an application is supporting implicit composition of certain characters, other characters may be composable via explicit composition.

Not every keystroke produces a character, even if the associated keysym normally implies character text. The event-to-string translation routines figure out what result a given set of keystrokes should produce (see "Using XLookupString(), XwcLookupString(), and XmbLookupString()" in this section).

Character composition from the user's aspect is discussed in the compose(5) and composetable(5) reference pages.

Explicit Composition

> Explicit composition is requested when the user presses the Compose key and then types a key sequence that corresponds to the desired character. For example, to compose the character ñ under some keymaps, you might press the Compose key and then type ~n.
>
> **Note:** The xmodmap(1) reference page tells how to map the XK_Multi_key keysym onto whatever key you want to use as Compose.

Implicit Composition

> Implicit composition mimics many existing European typewriters that have "dead" keys: keys that type a character but do not advance the carriage. When a special "dead" key is struck, the system attempts to compose a character using the next character struck. For example, on a keyboard that had a diaeresis (¨) and an O, but no Ö, you would strike ¨ and then o to compose Ö.
>
> Implicit composition support usually comes with some specified way to leave characters uncomposed.

Supported Keyboards

IRIX currently supports 16 keyboard layouts: American, Belgian, Czech, Danish, English, French, German, Italian, Norwegian, Polish, Portuguese, Russian, Spanish, Swedish, Swiss and Turkish. The American keyboard needs only ASCII.

Input Methods (IMs)

Input methods (IMs) are ways to translate keyboard-input events into text strings. You would use a different input method, for instance, to type on a USA keyboard in Chinese than to type on the same keyboard in English. Nobody would build a keyboard suitable for direct input of the tens of thousands of distinct Chinese characters.

IMs come in two flavors, *front-end* and *back-end*. Both types can use identical application programming interfaces, so you lose no generality by using back-end methods for our examples here.

To use an IM, follow these steps:

1. Open the IM.

2. Find out what the IM can do.

3. Agree upon capabilities to use.

4. Create input contexts with preferences and window(s) specified (see "Input Contexts (ICs)" on page 394).

5. Set the input context focus.

6. Process events.

Although all applications go through the same setup when establishing input methods, the results can vary widely. In a Japanese locale, you might end up with networked communications with an input method server and a *kanji* translation server, with circuitous paths for Key events. But in a Swiss locale for example, it is likely that nothing would occur besides a flag or two being set in Xlib. Since operating in non-Asian locales ends up bypassing almost all of the things that might make input methods expensive, Western users are not noticeably penalized for using Asia-ready applications.

Opening an Input Method

XOpenIM() opens an input method appropriate for the locale and modifiers in effect when it is called (see the XOpenIM(3X11) reference page). The locale is bound to that IM and cannot be changed. (But you could open another IM if you wanted to switch later.) Strings returned by **XmbLookupString()** and **XwcLookupString()** are encoded in the locale that was current when the IM was opened, regardless of current input context.

The syntax is

```
XIM XOpenIM(Display *dpy, XrmDataBase db, char *res_name,
            char *res_class);
```

The *res_name* is the resource name of the application, *res_class* is the resource class, and *db* is the resource database that the input method should use for looking up resources private to itself. Any of these can be NULL. The fragment in Example 16-7 shows how easy it is to open an input method.

Example 16-7   Opening an IM

```
XIM im;
im = XOpenIM(dpy, NULL, NULL, NULL);
if (im == NULL)
    exit_with_error();
```

XOpenIM() finds the IM appropriate for the current locale. If **XSupportsLocale()** has returned good status (see "Initialization for Xlib Programming") and **XOpenIM()** fails, something is amiss with the administration of the system.

XSetLocaleModifiers() determines configure locale modifiers. The local host X locale modifiers announcer (the XMODIFIERS environment variable) is appended to the modifier list to provide default values on the locale host. The modifier list argument is a null-terminated string containing zero or more concatenated expressions of this form:

@category=value

For example, if you want to connect Input Method Server *xwnmo*, set modifiers *_XWNMO* as follows:

```
XSetLocaleModifiers("@im=_XWNMO");
```

Or, set environment variable XMODIFIERS to the string `@im=_XWNMO` and execute

```
XSetLocaleModifiers("");
```

Note: The library routines are not prepared for the possibility of **XSupportsLocale()** succeeding and **XOpenIM()** failing, so it's up to application developers to deal with such an eventuality. (This circumstance could occur, for example, if the IM died after **XSupportsLocale()** was called.) This topic is under some debate in the MIT X consortium. If **XSetLocaleModifiers()** is wrong, **XOpenIM()** will fail.

Most of the complexity associated with IM use comes from configuring an input context to work with the IM. Input contexts are discussed in "Input Contexts (ICs)" on page 394.

To close an input method, call **XCloseIM()**.

IM Styles

If the application requests it, an input method can often supply status information about itself. For example, a Japanese IM may be able to indicate whether it is in Japanese input mode or romaji input mode. An input method can also supply pre-edit information, partial feedback about characters in the process of being composed. The way an IM deals with status and pre-edit information is referred to as an IM style. This section describes styles and their naming.

Root Window

The *Root Window* style has a pre-edit area and a status area in a window owned by the IM as a descendant of the root. The application does not manage the pre-edit data, the pre-edit area, the status data, or the status area. Everything is left to the input method to do in its own window, as illustrated in Figure 16-1.

Main body of window; text input occurs here.

Application window

root window

| Status | Pre-edit information |
| | IM window |
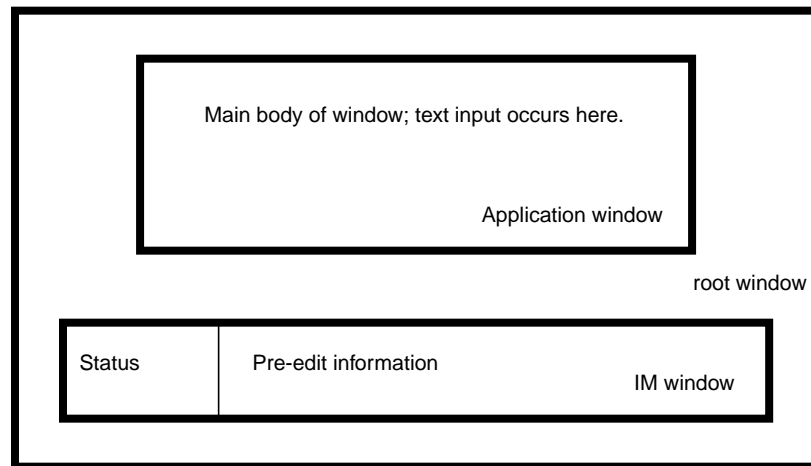
Figure 16-1   Root Window Input

Off-the-Spot

The *Off-the-Spot* style places a pre-edit area and a status area in the window being used, usually in reserved space away from the place where input appears. The application manages the pre-edit area and status area, but allows the IM to update the data there. (The application provides information regarding foreground and background colors, fonts, and so on.) A window using Off-the-Spot input style might look like that shown in Figure 16-2.
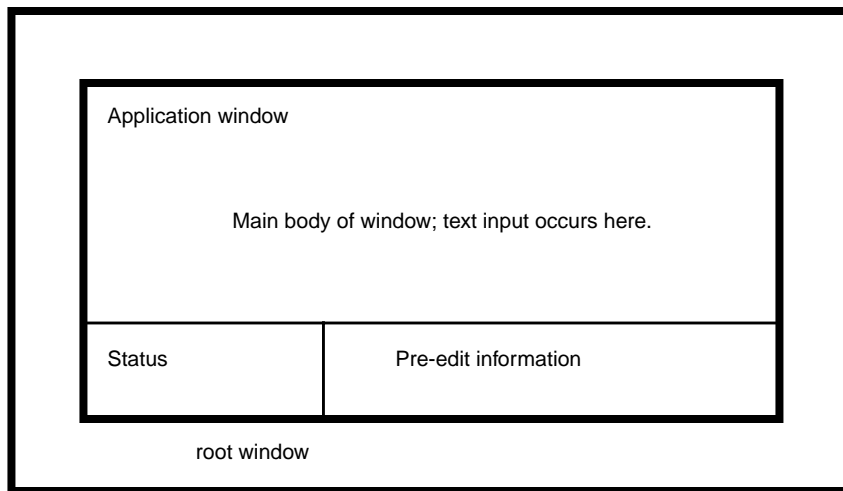
Figure 16-2   Off-the-Spot Input

Over-the-Spot

The *Over-the-Spot* style involves the IM creating a small, pre-edit window over the point of insertion. The window is owned and managed by the IM as a descendant of the root, but it gives the user the impression that input is being entered in the right place; in fact, the pre-edit window often has no borders and is invisible to the user, giving the appearance of On-the-Spot input. The application manages the status area as in Off-the-Spot, but specifies the location of the editing so that the IM can place pre-edit data over that spot.

On-the-Spot

On-the-Spot input is by far the most complex for the application developer. The IM delivers all pre-edit data via callbacks to the application, which must perform in-place editing—complete with insertion and deletion and so on. This approach usually involves a great deal of string and text rendering support at the input generation level, above and beyond the effort required for completed input. Since this may mean a lot of updating of surrounding data or other display management, everything is left to the application. There is little chance an IM could ever know enough about the application to be able to help it provide user feedback. The IM therefore provides status and edit information via callbacks.

Done well, this style can be the most intuitive one for a user.

Setting IM Styles

A style describes how an IM presents its pre-edit and status information to the user. An IM supplies information detailing its presentation capabilities. The information comes in the form of flags combined with OR. The flags to use with each style are as follows:

Root Window   XIMPreeditNothing | XIMStatusNothing

Off-the-Spot   XIMPreeditArea | XIMStatusArea

Over-the-Spot   XIMPreeditPosition | XIMStatusArea

On-the-Spot   XIMPreeditCallbacks | XIMStatusCallbacks

For example, if you wanted a style variable to match an Over-the-Spot IM style, you could write:

```
XIMStyle over = XIMPreeditPosition | XIMStatusArea;
```

If an IM returns *XIMStatusNone* (not to be confused with *XIMStatusNothing*), it means the IM will not supply status information.

Using Styles

An input method supports one or more styles. It's up to the application to find a style that is supported by both the IM and the application. If several exist, the application must choose. If none exist, the application is in trouble.

Input Contexts (ICs)

An input method may be serving multiple clients, or one client with multiple windows, or one client with multiple input styles on one window. The specification of style and client/IM communication is done via *input contexts.* An input context is simply a collection of parameters that together describe how to go about receiving and examining input under a given set of circumstances.

To set up and use an input context:

1. Decide what styles your application can support.

2. Query the IM to find out what styles it supports.

3. Find a match.

4. Determine information that the IC needs in order to work with your application.

5. Create the IC.

6. Employ the IC.

Find an IM Style

The IM may be able to support multiple styles—for example, both Off-the-Spot and Root Window. The application may be able to do, in order of preference, Over-the-Spot, Off-the-Spot, and Root Window. The application should determine that the best match in this case is Off-the-Spot.

First, discover what the IM can do, then set up a variable describing what the application can do, as shown in Example 16-8.

Example 16-8   Finding What a Client Can Do

```
XIMStyles *IMcando;
XIMStyle  clientCanDo; /* note type difference */
XIMStyle  styleWeWillUse = NULL;

XGetImValues(im, XNQueryInputStyle, &IMcando, NULL);

clientCanDo =
/*none*/ XIMPreeditNone | XIMStatusNone |
/*over*/ XIMPreeditPosition | XIMStatusArea |
/*off*/  XIMPreeditArea | XIMStatusArea |
/*root*/ XIMPreeditNothing | XIMStatusNothing;
```

A client should always be able to handle the case of
XIMPreeditNone | XIMStatusNone, which is likely in a Western locale. To the
application, this is not very different from a *RootWindow* style, but it comes with less
overhead.

Once you know what the application can handle, look through the IM styles for a match,
as shown in Example 16-9.

Example 16-9   Setting the Desired IM Style

```
for(i=0; i < IMcando->count_styles; i++) {
    XIMStyle tmpStyle;
    tmpStyle = IMcando->support_styles[i];

    if ( ((tmpStyle & clientCanDo) == tmpStyle) )
            styleWeWillUse = tmpStyle;
}
if (styleWeWillUse = NULL)
    exit_with_error();
XFree(IMcando);

/* styleWeWillUse is set, which is what we were after */
```

IC Values

There are several pieces of information an input method may require, depending on the
input context and style chosen by the application. The input method can acquire any
such information it needs from the input context, ignoring any information that does not
affect the style or IM.

A full description of every item of information available to the IM is supplied in *X
Window System, Third Edition*. The following is a brief list:

XNClientWindow	Specifies to the IM which client window it can display data in or create child windows in. Set once and cannot be changed.
XNFilterEvents	An additional event mask for event selection on the client window.
XNFocusWindow	The window to receive processed (composed) Key events.
XNGeometryCallback	A geometry handler that is called if the client allows an IM to change the geometry of the window.
XNInputStyle	Specifies the style for this IC.

XNResourceClass, *XNResourceName*	The resource class and name to use when the IM looks up resources that vary by IC.
XNStatusAttributes, *XNPreeditAttributes*	The attributes to be used for any status and pre-edit areas (nested, variable-length lists).

Pre-Edit and Status Attributes

When an IM is going to provide state, it needs some simple X information with which to do its work. For example, if an IM is going to draw status information in a client window in an Off-the-Spot style, it needs to know where the area is, what color and font to render text in, and so on. The application gives this data to the IC for use by the IM.

As with the "IC Values" section, full details are available in *X Window System, Third Edition*.

XNArea	A rectangle to be used as a status or pre-edit area.
XNAreaNeeded	The rectangle desired by the attribute writer. Either the application or the IM may provide this information, depending on circumstances.
XNBackgroundPixmap	A pixmap to be used for the background of windows the IM creates.
XNColormap	The colormap to use.
XNCursor	The cursor to use.
XNFontSet	The fontset to use for rendering text.
XNForeground, *XNBackground*	The colors to use for rendering.
XNLineSpacing	The line spacing to be used in the pre-edit window if more than one line is used.
XNSpotLocation	Specifies where the next insertion point is, for use by *XIMPreeditPosition* styles.
XNStdColormap	Specifies that the IM should use **XGetRGBColormaps()** with the supplied property (passed as an Atom) in order to find out which colormap to use.

Creating an Input Context

Creating an input context is a simple matter of calling **XCreateIC()** with a variable-length list of parameters specifying IC values. Example 16-10 shows a simple example that works for the root window.

Example 16-10   Creating an Input Context With XCreateIC()

```
XVaNestedList arglist;
XIC ic;

arglist = XVaCreateNestedList(0, XNFontSet, fontset,
                              XNForeground,
                              WhitePixel(dpy, screen),
                              XNBackground,
                              BlackPixel(dpy, screen),
                              NULL);

ic = XCreateIC(im, XNInputStyle, styleWeWillUse,
               XNClientWindow, window, XNFocusWindow, window,
               XNStatusAttributes, arglist,
               XNPreeditAttributes, arglist, NULL);
XFree(arglist);

if (ic == NULL)
    exit_with_error();
```

Using the IC

A multi-window application may choose to use several input contexts. But for simplicity, assume that the application just wants to get to the internationalized input using one method in one window.

Using the IC is a matter of making sure you check events the IC wants, and of setting IC focus. If you are setting up a window for the first time, you know the event mask you want, and you can use it directly. If you are attaching an IC to a previously configured window, you should query the window and add in the new event mask.

Example 16-11   Using the IC

```
unsigned long imEventMask;

XGetWindowAttributes(dpy, win, &winAtts);
XGetICValues(ic, XNFilterEvents, &imEventMask, NULL);

imEventMask |= winAtts.your_event_mask;
XSelectInput(dpy, window, imEventMask);
XSetICFocus(ic);
```

At this point, the window is ready to be used.

Events Under IM Control

Processing events under input method control is almost the same in X11R6 as it was under R4 and before. There are two essential differences: the **XFilterEvent()** and **X*LookupString()** routines.

Using XFilterEvent()

Every event received by your application should be fed to the IM via **XFilterEvent()**, which returns a value telling you whether or not to disregard the event. IMs asks you to disregard the event if they have extracted the data and plan on giving it to you later, possibly in some other form. All events (not just *KeyPress* and *KeyRelease* events) go to **XFilterEvent()**.

If you compacted the event processing into a single routine, a typical event loop would look something like the code in Example 16-12.

Example 16-12   Event Loop

```
Xevent event;

while (TRUE) {
    XNextEvent(dpy, &event);

    if (XFilterEvent(&event, None))
        continue;

    DealWithEvent(&event);
}
```

Using XLookupString(), XwcLookupString(), and XmbLookupString()

When using an input method, you should replace calls to **XLookupString()** with calls to **XwcLookupString()** or **XmbLookupString()**. The **MB** and **WC** versions have very similar interfaces. The examples below arbitrarily use **XmbLookupString()**, but apply to both versions.

There are two new situations to deal with:

1. The string returned may be long.

2. There may be an interesting keysym returned, an interesting set of characters returned, both, or neither.

Dealing with the former is a matter of maintaining an arena, as in Example 16-13.

To tell the application what to pay attention to for a given event, **XmbLookupString()** returns a status value in a passed parameter, equal to one of the following:

XLookupKeysym   Indicates that the keysym should be checked.

XLookupChars   Indicates that a string has been typed or composed.

XLookupBoth   Means both of the above.

XLookupNone   Means neither is ready for processing.

XBufferOverflow   Means the supplied buffer is too small—call **XmbLookupString()** again with a bigger buffer

XmbLookupString() also returns the length of the string in question. Note that **XmbLookupString()** returns the length of the string in bytes, while **XwcLookupString()** returns the length of the string in characters.

The example below should help show how these functions work. Most event processors perform a switch on the event type; assume you have done that and have received a *KeyPress* event.

Example 16-13   KeyPress Event

```
case KeyPress:
{
    Keysym keysym;
    Status status;
    int buflength;
    static int bufsize = 16;
    static char *buf = NULL;

    if (buf == NULL) {
        buf = malloc(bufsize);
        if (buf < 0) StopSequence();
    }

    buflength = XmbLookupString(ic, &event, buf, bufsize,
                                &keysym, &status);

    /* first, check to see if that worked */
    if (status == XBufferOverflow) {
        buf = realloc(buf, (bufsize = buflength));
        buflength = XmbLookupString(ic, &event, buf, bufsize,
                                    &keysym, &status);
    }

    /* We have a valid status. Check that */
    switch(status) {
    case XLookupKeysym:
        DealWithKeysym(keysym);
        break;
    case XLookupBoth:
        DealWithKeysym(keysym);
        /* **FALL INTO** charcter case */
    case XLookupChars:
        DealWithString(buf, buflength);
    case XLookupNone:
        break;
    } /* end switch(status) */

} /* end case KeyPress segment */
break; /* we are in a switch(event.type) statement */
```

GUI Concerns

It shouldn't be significantly more difficult to internationalize an application with a graphical user interface than an application without such an interface, but there are a few further issues that must be addressed:

- "X Resources for Strings" on page 401 covers labeling objects using X resources.

- "Layout" on page 402 describes creating layouts that are usable after localization.

- "Icons" on page 403 explains some concerns for localizing icons.

X Resources for Strings

Resource lookup mechanisms in Xlib as well as in toolkits monitor locale environment variables when locating resource files. For string constants that are used within toolkit objects, resources provide a simpler solution than do message catalogs.

These are some common objects that should definitely get their text from resources:

- Labels

- Buttons

- Menu items

- Dialog notices and questions

Any object that employs some sort of text label should be labeled via resources. Since the localizer wants to provide strings for the local version of the application, the *app-defaults* file for the application should specify every reasonable string resource. Reference pages should identify all localizable string resources.

Localizers of an application provide a separate resource file for each locale that the application runs in.

Layout

Layout management is of special interest when you cannot predict how large a button or other label might be. The nature of the problem of layout composition and management does not change, but one must construct the layout management without full knowledge of the final appearance.

It's worth noting that localization efforts can be assumed to be "reasonable" in some sense. For example, X resources have always allowed a user to specify an extremely large font for buttons, but applications correctly choose to let such users live with the results. But it's not always that clear what is reasonable and what isn't; you don't always know what will be difficult to translate succinctly in some locale. So while you need not provide for all combinations of resource specifications, you must make the application localizable.

Three main approaches to the layout problem are described below: dynamic layout, constant layout, and localized layout

Dynamic Layout

Most toolkits provide *form*, *pane*, *rowcolumn*, or other layout objects that calculate layout depending on the "natural" (localized) size of the objects involved. Most use some hints provided by the developer that can regulate this layout. For example, some IRIS IM widgets providing these services are *XmForm*, *XmPanedWindow*, and *XmRowColumn*.

Dynamic layout is probably the simplest way to prevent localization difficulties.

Note: The IRIS IM product is the Silicon Graphics port of the OSF/Motif product, and should not be confused with IM, the abbreviation for Input Methods.

Constant Layout

Under certain circumstances, an application may insist on having a predefined layout. When this is so, the application must provide objects that are constructed to allow localization. A "Quit" button that just barely allows room for the Latin 1 string "Quit" is not likely to suffice when localizers attempt to fit their translations into that small space.

In order to enforce constant layout, the developer incurs the heavy responsibility of making sure the objects are localizable. This means a lot of investigation; the "there, that ought to be enough" approach is chancy at best.

Localized Layout

Some toolkits provide for layout control by run-time reading of strings or other data files. Applications that use such toolkits can easily finesse the layout issue by providing the capability for localization of the layout, as well as localization of the contents of the layout. This provides each localizer maximum freedom in presenting the application to the local users. The application developer is responsible for providing localizers with instructions and the mechanisms necessary to produce layout data.

IRIS IM Localization With editres

IRIX provides an interactive method of laying out widgets for IRIS IM and Xaw (the Athena Widget Set): a utility called *editres*. With *editres*, you can construct and edit resources and see how your widgets will look on the screen; the program even generates a usable app-defaults file for you. But note that if you hard-code any resources into your IRIS IM code, you won't be able to edit them using this method.

Icons

Icons attempt to be fairly generic representations of their antecedents. Unfortunately, it is very difficult for a designer to know what is generic or recognizable in other cultures. Therefore, it is important that any pictographic representations used by an application be localizable.

Graphic representations can be stored as strings representing X bitmaps, as names of data files containing pictographs, or in whatever manner the developer thinks best, so long as the developer provides a way for the localizer to produce and deliver localized pictographs.

Popular Encodings

This section discusses three encodings that are commonly used:

- "The ISO 8859 Family" explains the ISO 8859 family of encodings.
- "Asian Languages" describes Asian language encodings.
- "Unicode" covers the ISO 10646 and Unicode.

The ISO 8859 Family

American English is easily representable in 7-bit ASCII. Most other languages are not. For example, the character é is not in ASCII.

Most Western European languages are representable in 8-bit ISO 8859-1, which is commonly known as Latin 1. Latin 1 is a superset of ASCII that includes characters used by several Western European languages (such as ö, £, ñ, ç, ¿).

ISO 8859 comes in nine parts, many of which overlap; all are supersets of ASCII.

The ISO 8859 Character Sets are shown in Table 16-10.

Table 16-10   ISO 8859 Character Sets

Character Set	Common Name	Languages Supported
8859-1	Latin 1	Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish
8859-2	Latin 2	Albanian, Czech, English, German, Hungarian, Polish, Rumanian, Serbo-Croatian, Slovak, Slovene
8859-3	Latin 3	Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish
8859-4	Latin 4	Danish, English, Estonian, Finnish, German, Greenlandic, Lapp, Latvian, Lithuanian, Norwegian, Swedish
8859-5	Latin/Cyrillic	Bulgarian, Byelorussian, English, Macedonian, Russian, Serbo-Croatian, Ukrainian
8859-6	Latin/Arabic	Arabic, English (see ISO 8859-6 specification)
8859-7	Latin/Greek	English, Greek (see ISO 8859-7 specification)
8859-8	Latin/Hebrew	English, Hebrew (see ISO 8859-8 specification)
8859-9	Latin 5	Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, Turkish

IRIX contains over 500 Latin 1 fonts, as well as a few fonts for each of the other 8859-encoded character sets, except 8859-6 and 8859-8. Currently, IRIX contains no fonts for use with the 8859-6 or 8859-8 character sets.

To get the list of ISO-8859 fonts, enter the following:

```
xlsfonts
```

Or you can restrict the amount of output, for example, by typing

```
xlsfonts '*8859-2'
```

To see the encoding, use the *xfd* command. For example:

```
xfd -fn -sgi-screen-medium-r-normal--9-90-72-72-m-60-iso8859-1
```

For more information on *xlsfonts* and *xfd*, and installing and using fonts, refer to Chapter 15, "Working With Fonts."

Asian Languages

Asian languages are commonly ideographic and employ large numbers of characters for their representation. For example, Japanese and Korean can be practically encoded in 16 bits. Daily-use Chinese can be, also, but archives and scholars frequently need more, so Chinese is often encoded with up to four bytes per character.

Some Standards

Various Asian character sets have been developed, some of which are considered standard. Encodings for these sets are less standardized. Asian character sets usually require larger-than-byte character types like those described in "Multibyte Characters." Table 16-11 lists some of these standard character sets. Note that some of these character sets have multiple associated codesets, usually designated by appending the year the codeset was adopted to the character set name. (For example, JIS X 208-1983 is different from JIS X 208-1990.)

Table 16-11 Character Sets for Asian Languages

Language	Character Set Standards	Support
Japanese	JIS X 0201.1976-0	*Katakana*
	JIS X 0208.1983-0	*Kanji*, *kana*, Latin, Greek, Cyrillic, symbols, others
	JIS X 0212.1990-0	Supplemental *kanji*, others
Chinese	GB 2312.1980-0	
Korean	KSC 5601.1987-0	Hangul
Taiwan	CNS 11643	

EUC

EUC is *Extended UNIX Code*, an encoding methodology that supports concurrent use of four codesets in one encoding. It employs two special "shift state" bytes:

```
ss1 = 0x8e
ss2 = 0x8f
```

These are used to identify codesets within a string. The EUC encoding scheme uses the following patterns to indicate which codeset is in use at any given time:

```
Codeset #0: 0xxxxxxx
Codeset #1: 1xxxxxxx [ 1xxxxxxx ...]
Codeset #2: ss1 1xxxxxxx [ 1xxxxxxx ...]
Codeset #3: ss2 1xxxxxxx [ 1xxxxxxx ...]
```

So if *ss1* appears in a string, it means that the next character—however many bytes long it is—should be interpreted as a character from codeset #2. If there are multiple characters in a row from codeset #2, each one is preceded by *ss1*. Similarly, *ss2* indicates that the following character belongs to codeset #3. If any other byte whose high bit is 1 appears in the string (without being preceded by *ss1* or *ss2*), it is interpreted as all or part of a character from codeset #1.

In EUC, codeset #1 is always ASCII. The other codesets are implementation- or user-defined. This is why EUC cannot support Latin 1 in Asian locales.

EUC implementations exist (but are not standardized) for all ideographic Asian languages.

Unicode

The Unicode Consortium has developed a character code system called Unicode. Unicode 2.0 covers most of the modern languages, scripts, CJK (Chinese-Japanese-Korean) scripts, and scientific and mathematical symbols. Each character is represented by a fixed width of 16 bits. Unicode 2.0 implements the characters that are coded in the Basic Multilingual Plane of ISO-10646. For more detailed information, see Unicode Standard, Version 2.0, published by Addison-Wesley; ISBN 0-201-48345-9.

ISO 3166 Country Names and Abbreviations

Table A-1 lists the ISO 3166 country codes, alphabetized by country name (the table reads from left to right, and top to bottom).

Table A-1   ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name	Code
Afghanistan	AF	Albania	AL	Algeria	DZ
American Samoa	AS	Andorra	AD	Angola	AO
Anguilla	AI	Antarctica	AQ	Antigua and Barbuda	AG
Argentina	AR	Aruba	AW	Australia	AU
Austria	AT	Bahamas	BS	Bahrain	BH
Bangladesh	BD	Barbados	BB	Belgium	BE
Belize	BZ	Benin	BJ	Bermuda	BM
Bhutan	BT	Bolivia	BO	Botswana	BW
Bouvet Island	BV	Brazil	BR	British Indian Ocean Territory	IO
Brunei Darussalam	BN	Bulgaria	BG	Burkina Faso	BF
Burma	BU	Burundi	BI	Byelorussia	BY
Cameroon	CM	Canada	CA	Cape Verde	CV
Cayman Islands	KY	Central African Republic	CF	Chad	TD
Chile	CL	China	CN	Christmas Island	CX
Cocos Islands	CC	Colombia	CO	Comoros	KM

409

Table A-1 (continued)   ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name	Code
Congo	CG	Cook Islands	CK	Costa Rica	CR
Cote D'Ivoire	CI	Cuba	CU	Cyprus	CY
Czech Republic	CS	Denmark	DK	Djibouti	DJ
Dominica	DM	Dominican Republic	DO	East Timor	TP
Ecuador	EC	Egypt	EG	El Salvador	SV
Equatorial Guinea	GQ	Ethiopia	ET	Falkland Islands	FK
Faroe Islands	FO	Fiji	FJ	Finland	FI
France	FR	French Guiana	GF	French Polynesia	PF
French Southern Territories	TF	Gabon	GA	Gambia	GM
Germany	DE	Ghana	GH	Gibraltar	GI
Greece	GR	Greenland	GL	Grenada	GD
Guadalupe	GP	Guam	GU	Guatemala	GT
Guinea	GN	Guinea-Bissau	GW	Guyana	GY
Haiti	HT	Heard and McDonald Islands	HM	Honduras	HN
Hong Kong	HK	Hungary	HU	Iceland	IS
India	IN	Indonesia	ID	Iran	IR
Iraq	IQ	Ireland	IE	Israel	IL
Italy	IT	Jamaica	JM	Japan	JP
Jordan	JO	Kampuchea	KH	Kenya	KE
Kiribati	KI	Korea	KP or KR	Kuwait	KW

Table A-1 (continued)   ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name	Code
Laos	LA	Lebanon	LB	Lesotho	LS
Liberia	LR	Libya	LY	Liechtenstein	LI
Luxembourg	LU	Macau	MO	Madagascar	MG
Malawi	MW	Malaysia	MY	Maldives	MV
Mali	ML	Malta	MT	Marshall Islands	MH
Martinique	MQ	Mauritania	MR	Mauritius	MU
Mexico	MX	Micronesia	FM	Monaco	MC
Mongolia	MN	Montserrat	MS	Morocco	MA
Mozambique	MZ	Namibia	NA	Nauru	NR
Nepal	NP	Netherlands	NL	Netherlands Antilles	AN
Neutral Zone	NT	New Caledonia	NC	New Zealand	NZ
Nicaragua	NI	Niger	NE	Nigeria	NG
Niue	NU	Norfolk Island	NF	Northern Mariana Islands	MP
Norway	NO	Oman	OM	Pakistan	PK
Palau	PW	Panama	PA	Pangaea	GE
Papua New Guinea	PG	Paraguay	PY	Peru	PE
Philippines	PH	Pitcairn	PN	Poland	PL
Portugal	PT	Puerto Rico	PR	Qatar	QA
Quebec	QC	Reunion	RE	Romania	RO
Rwanda	RW	Saint Kitts and Nevis	KN	Saint Lucia	LC
Saint Vincent and the Grenadines	VC	Samoa	WS	San Marino	SM

Table A-1 (continued)   ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name	Code
Sao Tome and Principe	ST	Saudi Arabia	SA	Senegal	SN
Seychelles	SC	Sierra Leone	SL	Singapore	SG
Solomon Islands	SB	Somalia	SO	South Africa	ZA
Spain	ES	Sri Lanka	LK	St. Helena	SH
St. Pierre and Miquelon	PM	Sudan	SD	Suriname	SR
Svalbard and Jan Mayen Islands	SJ	Swaziland	SZ	Sweden	SE
Switzerland	CH	Syrian Arab Republic	SY	Taiwan	TW
Tanzania	TZ	Thailand	TH	Togo	TG
Tokelau	TK	Tonga	TO	Trinidad and Tobago	TT
Tunisia	TN	Turkey	TR	Turks and Caicos Islands	TC
Tuvalu	TV	Uganda	UG	Ukraine	UA
United Arab Emirates	AE	United Kingdom	GB	United States Minor Outlying Islands	UM
Uruguay	UY	Vanuatu	VU	Vatican City State	VA
Venezuela	VE	Viet Nam	VN	Virgin Islands (British)	VG
Virgin Islands (USA)	VI	Wallis and Futuna Islands	WF	Western Sahara	EH
Yemen	YE or YD	Yugoslavia (Former)	YU	Zaire	ZR
Zambia	ZM	Zimbabwe	ZW		

Index

F

fcntl()
  example code, 233
file, mapping into memory, 15, 26
file access permissions and **mmap()**, 17
file and record locking, 171-190
  across systems, 188
  deadlocks, 186
  efficiency, comparative, 188
  F_GETLK, 183
  F_SETLK, 181
  F_TEST, 185
  F_ULOCK, 183
  F_UNLCK, 183
  failure, 180
  file permissions and, 174
  forking, 186
  lock information, 183
  locking a file, 176
  mandatore, 186
  multiple read locks, 183
  NFS with, 188
  opening files, 175
  order of lock removal, 183
  overview, 172-174
  removing locks, 179
  setting locks, 179
file descriptor
  with asynchronous I/O, 194
  with **mmap()**, 13
file typing rules, 374
  LEGEND, 374
  MENUCMD, 374
Finland country code, 410
fonts, 303-324
  accessing, 308
  adding, 313-322
    bitmap font, 316-319
    font files, 316

  font metric file, 322
  outline font, 319-322
  Utopia Regular font files, 316
  aliases, 309
  character, defined, 305
  display characters, 310
  downloading, 323
  images, 306
  installing, 313-322
  missing fonts, 323
  names, 307, 309
  opening a shell window, 312
  path, 312
  pixels, 306
  point size, 306
  PostScript printers, 323
  programming access, 308
  resolution and size, 306
  scaling, 310
  Speedo format, 313
  Type 1 font, 313, 323-324
  typeface, defined, 305
  using APIs, 308
  Utopia fonts, 323
  viewing, 310
  virtual memory, 324
  xfd command, 310
  X Window System, 307, 309-322
fontsets, 380-381
  creating, 381
  specifying, 380
  using, 381
fork()
  defines address space, 5
  new address space copy-on-write, 10
forking, 186
France country code, 410
fsync(), 192
ftruncate() on memory-mapped file, 18

virtual swap, 7-9
  SIGKILL from, 8
  See also address space
VME PIO, 20
VPN. *See* virtual page number

W

wait(), 259
wait, timed, 127
warning messages
  lp log file, 324
wide characters. *See* internationalization, wide
    characters
write(), 192
  direct, 225
  synchronous, 223
  with guaranteed-rate I/O, 232, 234

X

xfd command, 310
XFilterEvent(), 398
XFontSetExtents, 382
XLFD font names. *See* internationalization, X
    Window System, fontsets
Xlib changes, 378
XLookupString()<Default Para Fon>, 399
XmbLookupString(), 399
XSetLocaleModifiers(), 390
XwcLookupString(), 399
X Window System
  fonts. *See* fonts
  installing fonts. *See* fonts, installing
  internationalization changes, 377
  limitations, 377

Y

yielding, 127
You, 8

Z

Zambia country code, 412

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document

- Omission of material that you expected to find

- Technical errors

- Relevance of the material to the job you had to do

- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2478-007.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:

   - On the Internet: techpubs@sgi.com

   - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs

- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801

- To send your comments by **traditional mail**, use this address:

Technical Publications
Silicon Graphics, Inc.
1600 Amphitheatre Parkway, M/S 535
Mountain View, California 94043-1351