Chapter 1

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:

- "Defining the Address Space" on page 3 tells what the address space is and how it is created.
- "Interrogating the Memory System" on page 10 summarizes the ways your program can get information about the address space.
- "Mapping Segments of Memory" on page 11 documents the different ways that you can create new memory segments with predefined contents.
- "Locking Pages in Memory" on page 24 discusses when and how to lock pages of virtual memory to avoid page faults.
- "Additional Memory Features" on page 29 summarizes functions for address space management.

Defining the Address Space

Each user-level process has a *virtual address space*. This term means nothing more than 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 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 xxii.

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 CPU hardware.

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. The 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 left undefined so that an attempt to reference them (for example, through an uninitialized pointer variable) causes a hardware exception.

Typically, the 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.

Page Numbers and Offsets

IRIX manages memory in units of a page. The size of a page can differ from one system to another. The size when 32-bit addressing is used is 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)

The page size when 64-bit addressing is used is greater than 4,096 bytes, and can differ between versions of IRIX, but 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 10.

Page tables, built by IRIX during a **fork()** or **exec()** call, define the address space 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 SIGBUS error.

Addresses are *defined*, that is, made available for potential use, in one of four 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 in the address space of the new process.

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 11.

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 entry in the page tables. A defined address is always related to a *backing store*, a source from which its contents can be retrieved. A page in the 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, 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:

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*. The limits can be queried with **getrlimit()** and changed 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 find the limits of swap space by making repeated calls to **malloc()**.
- A large memory allocation by one program can fill swap, 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()**, which never returns NULL until the program exceeds its resource limit.

  Instead, when finally a program 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 (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. 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()**, the backing store file is specified at creation time (see "Mapping Segments of Memory" on page 11).

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 valid 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.

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	uiPageSize = getpagesize(); ulPageSize = 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);

Table 1-1 (continued) Memory System Calls

Memory Information	System Call Invocation
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 {
   long freemem; /* pages of free memory */
   long availsmem; /* total real+swap memory space */
   long availrmem; /* available real memory space */
   long bufmem; /* not useful */
   long physmem; /* total real memory space */
};
```

A sample program that applies **swapctl()** and **sysmp()** to display these numbers is shipped in the 4DGifts example directory. See *~4Dgifts/examples/unix/irix/freevmen.c*

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 portion of VME A24 or A32 bus address space (when a VME bus exists on the system)
- a segment initialized to binary zero
- a view of the kernel’s private address space or of physical memory

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.

The 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()** function (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 special device 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 10).

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 22.

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 29).

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.
MAP_AUTOGROW	Extend the object when the process stores beyond its end (not POSIX)
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 22.

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.

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 25.

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 20), **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 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.

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 Pages in Memory" on page 24.

Mapped File Sizes

Since 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. To map an entire file:

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 ObjectServer 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 map the new portion of the file as 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.

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 30).

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 4, “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 to represent the segment.

  Typically the file is located in */var/tmp*, but it can be anywhere. The file permissions determine the access permitted to other processes.
- In another process, get a file descriptor using **open()** or the POSIX function **shm_open()**, specifying the file path.
- In that other process, use **mmap()** specifying the file descriptor of the file.

Tip: The **usinit()** function is a more convenient and general way of creating a shared arena. It is based on **mmap()** but adds other services—see “Allocating in a Shared Arena” on page 39. You can also create a shared segment of memory using SVR4-compatible shared memory functions; see “System V Shared Memory” on page 101.

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 uservme(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.

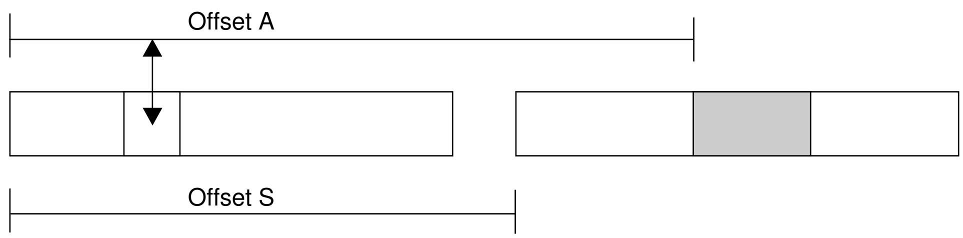

Figure 1-1 Segments With a Fixed Offset Relationship

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 since 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 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 with a frame rate above a few Hertz cannot tolerate such interruptions. The solution is to lock some or all of the pages of your address space into memory. A page fault cannot occur on a locked page.

Locking Functions

You use one of two functions to lock segments into physical memory:

mpin() Locks a specified range of pages into memory

plock() Locks all program text, or all data, or the entire address space

The two functions have the same effect. They differ only in how you specify the pages to be locked. (Refer to the mpin(2) and plock(2) reference pages.)

Using **mpin()** 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. The best use of **mpin()** is to lock a mapped memory segment, since you know its starting address and length immediately after creating it.

Both **plock()** and **mpin()** define all pages of the specified segments before locking them. 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.

You use either **munpin()** or **punlock()** to unlock pages, allowing the kernel to reclaim them when necessary. Locked pages of an address space are unlocked when the last process using the address space terminates.

Locking Program Text and Data

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 **mpin()** to lock executable, mapped segments.)

When you specify the data option, the function 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 Files Into Memory

If you map a file before you lock the data segment into memory (see “Mapping a File for I/O” on page 16), the mapped file is read into the locked pages. If you map a file after locking the data segment, 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 that:

- 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 the 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.

You can use **mpin()** to lock any portion of a mapped segment, and **munpin()** to unlock portions that are not needed. A call to **mpin()** implies a wait while the contents of that portion of the file are read, so this call should be made in an asynchronous process.

Reducing Cache Misses

When performance requirements are high, you become concerned, not with the loss of milliseconds to a page fault, but with the loss of microseconds to a cache miss. When your program accesses instructions or data that are not in cache memory, the CPU requests a load of a “cache line” of 128 bytes from main memory. Possibly hundreds of CPU clock cycles pass while the cache is being loaded. Due to the pipeline architecture of the CPU, it can often continue to work during this delay. However, multiple successive cache misses can bring effective work to a halt for tens of microseconds.

Locality of Reference

The key to good cache performance is to maintain strong locality of reference. This can be restated as a rule of thumb: “Keep things that are used together, close together.” Or, “Extract the greatest possible use from any 128-byte cache line before touching another.” You must decide how to apply these principles in the context of your program design. Some possible techniques:

- When designing a large data structure, group small fields together at one end of the structure. Do not mix small and large fields.
- Consolidate frequently-tested switches, flags, and pointers into a single record so they tend to stay in cache.
- Avoid searching linked lists of structures. Each time a process visits a link merely to find the address of the next link, it is likely to incur a cache miss. Worse, a search over a long list fills the cache with unneeded links, driving out useful data.
- Avoid striding through a large array of structures (such as an array of graphics library objects) visiting only one or two fields in each structure. Whenever possible, arrange the data so that any sequential scan visits and uses every byte before moving on.
- Use inline function definitions for functions that are called within innermost loops. Do not use inline definitions indiscriminately, however, because they increase the total size of the binary, potentially causing more cache misses in non-looping code.
- Use **memalign()** to allocate important structures on 128-byte boundaries, so as to ensure the structures fit in the smallest number of cache lines (see the memalign(3) reference page).

Cache Mapping in Challenge and Onyx Systems

The cache design in the Challenge/Onyx line depends on the CPU model in use. The basic Challenge uses the IP19 board with R4*x*00 processors. This CPU board uses a simple algorithm to assign a memory location to a cache line: the address of a byte of data is taken modulo the cache size to generate the cache address. This means that two words that are separated in main memory by an exact multiple of the cache size are always loaded to the same cache location.

Only one of the words can occupy the cache at a time, so if your program alternates between words, it will have a cache miss on each reference. It is surprisingly easy to create this situation. The following code fragment causes bad performance in an R4K Challenge system with a 1 MB cache.

```
float part1[262144]; /* 1 MB */
float part2[262144]; /* adjacent 1 MB */
for (j=0;j<262144;++j) part1[j] = part2[j];
```

In that code fragment, the words of each array hash to the identical cache lines, so each assignment in the loop incurs two cache misses. (Some systems have caches of different sizes, but the same principle applies.)

Note: The cache in the POWER Challenge, which uses the R8000 or R10000 CPU, *does not* use simple modulus mapping; it is an associative memory that is much more resistant to cache conflicts.

Multiprocessor Cache Conflicts

In a multiprocessor system such as the Challenge, when one CPU modifies cached data, it broadcasts the fact on the bus. Any other CPU holding that same cache line marks it invalid. If another CPU then needs to refer to the so-called "dirty" cache line, it has to fetch the modified version from the first CPU. This takes even longer than reloading the cache line from main memory.

These conflicts can cause cache delays when the processes in two or more CPUs are working on the same data concurrently. There is no conflict so long as all CPUs are *reading* the data. Each works from its own cache copy in that case. But whenever one CPU modifies the data, all other CPUs suffer a cache miss on the next access to the same data.

In general the only way to avoid such conflicts is to separate the readers and writers in time. Arrange the program so that data is updated occasionally in a burst, then used for a longer period.

Detecting Cache Problems

There are relatively few tools for detecting or fixing cache problems in code. You can combine the two IRIX profiling tools, *pixie* and *prof* (see the pixie(1) and prof(1) reference pages), to arrive at a tentative diagnosis.

The *pixie* tool modifies the executable of a program so that every basic block is counted during execution. Its output ranks functions by the absolute count of instructions they executed.

The *prof* tool samples the instruction counter of the program while the program is executing. Its output ranks functions by the amount of time that the CPU spent in their code.

Normally the output of these tools should agree on the location of the hot spots in a program. However, if *prof* shows that a function is taking more time than is justified by its *pixie* execution count, that function may be running slowly due to cache-miss problems.

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:

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 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.