Chapter 4

Managing Virtual Memory in a Real-Time Program

When planning a real-time 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 are:

- "Defining the Address Space" on page 43 tells what the address space is and how it is created.
- "Interrogating the Memory System" on page 48 summarizes the ways your program can get information about the address space.
- "Locking Pages in Memory" on page 49 discusses when and how to lock pages of virtual memory to avoid page faults.

This chapter is a condensed version of a longer discussion of virtual memory management which can be found in the book *Topics In IRIX Programming*. In particular, *Topics In IRIX Programming* has detailed information on memory mapping.

The the structure of physical and virtual address spaces is discussed in the *IRIX Device Driver Programmer's Guide* and the MIPS architecture documents listed on page xxiii.

Defining the Address Space

Each process has a *virtual address space*; in other words, a set of memory addresses that the process can use. When 32-bit addressing is in use, the addresses can range from 0 to 0x7fffffff; that is, 2^{31} numbers, for a total theoretical size of 2 gigabytes. (Numbers greater than 2^{31} are reserved for kernel addresses.) 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.)

Address Space Boundaries

A process has at least 3 segments of usable addresses:

- A text segment contains the executable image of the program. The text segment is always read-only.
- A data segment contains the "heap" of allocated data space.
- A stack segment contains the function-call stack.

Another text segment is created for each dynamic shared object (DSO) with which a process is linked. A process can create additional data segments in various ways described later in the chapter.

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.

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 principle is the same. The less-significant bits of an address specify an offset within a page, while the more-significant bits specify the VPN.

The actual size of a page in the present system can be learned with **getpagesize()** as noted under "Interrogating the Memory System" on page 48.

Address Definition

Most of the possible addresses in an address space are undefined, that is, not entered 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()**, any addresses that were defined in the parent's address space are defined in the address space of the new process (see "Normal Process Creation With fork()" on page 16).
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	Your program 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 in the book *Topics in IRIX Programming*.
Allocation	The **brk()** function extends the segment devoted to data (the *heap*) to a specific virtual address. The **malloc()** function allocates memory for use, calling **brk()** as required. (See the brk(2) and malloc(3) reference pages). The more commonly used library version of **malloc()** calls the underlying **malloc()** (see the malloc(3x) reference page).

When an address is defined, it is entered in the page tables and 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 the 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).

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 4-1 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()** (see "Lightweight Process Creation With sproc()" on page 17) and does not supply a stack area, 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.

Page Validation

Although an address is defined, the corresponding page is not necessarily loaded in physical memory. The sum of the 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. Invalid pages can be any of the following:

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

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

Page validation takes from 10 to 50 milliseconds, a delay that a real-time program normally cannot tolerate.

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 (see "Signals" on page 36) 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.

Interrogating the Memory System

You can get information about the state of the memory system with the system calls shown in Table 4-1.

Table 4-1 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);
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+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*

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. If your real-time frame rate exceeds a few Hertz, your program 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

There are two functions that you use 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 mapped memory segments, since you know their starting addresses and lengths immediately after creating them.

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.

Locking pages in memory of course reduces the memory that is available for all other programs in the system. Locking a large program will increase 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. 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 one in Example 4-2

Example 4-2 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 only need to do this from 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, 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.

In a real-time program you cannot tolerate a delay to read a file page. However, a very large file can easily exceed the capacity of physical memory.

One alternative for a large file is to not map it, but to use conventional read and write access to it. However, this alternative forfeits the convenience of referring to the file as if it were an array in memory.

Another 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 vehicle simulator could lock the parts of a scenery file that the 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 the frame rate is 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 (see Figure 2-1 on "Multiprocessor Architecture" on page 10 and "Memory Hierarchy" on page 11), 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.

In a normal program, delays due to cache misses are not noticeable because the overall average speed of the program is satisfactory. However, for a real-time program with a frame rate above 50 Hz, a cache miss can cause the unpredictable loss of a useful fraction of one frame interval.

Note: In addition to the following guidelines, the IRIX kernel assists you in maintaining good cache use with special scheduling rules. See "Understanding Affinity Scheduling" on page 72.

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 will 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 will visit and use 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/Onyx

The cache design in the Challenge/Onyx line depends on the CPU model in use. The basic Challenge/Onyx uses the IP19 with an R4000 processor. 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 a Challenge/Onyx 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 Challenge/Onyx systems have caches of different sizes, but the same principle applies.)

Note: The cache in the R8000-based POWER Challenge *does not* use simple modulus mapping; it is an associative memory that is much more resistant to cache conflicts.

Multiprocessor Cache Conflicts

As described under "Memory Hierarchy" on page 11, 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 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. When using the REACT/Pro Frame Scheduler, plan the schedule so the process that updates the data runs in a different minor frame from processes that read the data.

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.