Chapter 2

Basic Features of the CHALLENGE and IRIX™ Architectures

IRIX overview<$startrange>The architecture of the CHALLENGE, Onyx, and POWERCHALLENGE computers provides multiple CPUs, a large real memory, a high-speed system bus, and fast I/O channels. (For brevity, the phrase Challenge/Onyx is used to refer to these machines as a single type.)

The IRIX operating system normally manages the hardware resources so as to optimize the throughput of a large number of UNIX* applications, both batch and interactive.

This chapter gives a high-level summary of the standard operational methods of IRIX, and points out how they can sometimes conflict with the needs of a real-time program.

If you already know IRIX and the Challenge/Onyx architecture, you can skip to Chapter 3, “How IRIX™ and REACT/Pro™ Support Real-Time Programs,”, which introduces the features you can use to create fully deterministic system behavior for real-time programs.

Multiprocessor Architecture

Challenge/Onyx architectureFigure 2-1 shows a simplified, high-level view of the Challenge/Onyx architecture.

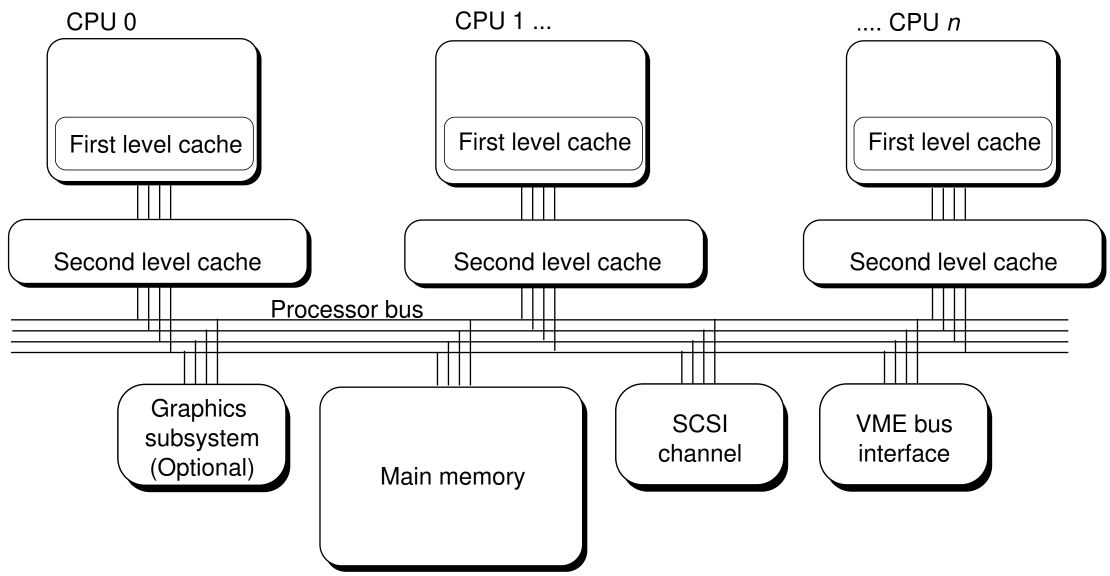

Figure 2-1 Symmetric Multiprocessor Architectureprocessor bus:diagram

CPUs, Memory, and the System Bus

processor bus:capacitybus:processor. <Italics>See<Default Para Font> processor bus<$nopage>CPU:relation to bus and memorymultiprocessor architectureA Challenge/Onyx system contains from 2 to as many as 36 CPUs. All are functionally identical. The CPUs are connected to each other and to a single memory by the processor bus. The processor bus carries 128-bit parallel packets at a data rate of 1.2 Gigabytes/second. An important feature of the bus design is that it is "fair," that is, there is a very low probability of any CPU on it starving for access. This helps to make real-time program timings determinate and repeatable.

IRIX:kernelkernel:multiprocessor usememory:mainThere is a single physical memory (shown as "main memory" in Figure 2-1) that is accessed equally by all CPUs. For example, there is a single image of the UNIX kernel in memory, and any of the CPUs could be executing instructions from it, in any combination, at any time.

Concurrent Execution

concurrent executionThe Challenge/Onyx computers permit true concurrency—two or more CPUs executing the same program at the same instant. However, most ordinary UNIX programs execute in only one CPU at a time.

lock:used by kernelsemaphore:used by kernelkernel:multiprocessor useTwo or more CPUs, executing on behalf of different processes, can enter the IRIX kernel simultaneously. The kernel is written to optimize concurrent use. It uses *semaphores* and *locks* to serialize the use of the data structures that can be used by two or more processes at the same time.

A real-time program may need to use two or more CPUs concurrently in order to finish the work it needs to do in each frame interval. You can structure your real-time program as multiple processes. You can cause these processes to run concurrently on multiple CPUs, and you can use semaphores and locks to protect their common resources. Process creation is discussed later in this chapter, under “Process Management” on page 15.

Memory Hierarchy

cache:architecturememory:hierarchyEach CPU in a Challenge/Onyx system accesses memory through a four-level hierarchy:

- First-level instruction and data caches within the CPU chip provide the fastest access to recently-used data (the cache size depends on the microprocessor model).
- A larger second-level cache on each CPU board stores recently-used instructions and data (this cache size depends on the CPU board model).
- Main memory contains the current state of swapped-in processes.
- Swapped-out virtual pages are kept in the swap partition on disk.

locality of referenceThere is a ratio of roughly 100:1 in access speeds between each level of this hierarchy. There is a large reward of execution speed for a program that maintains *locality of reference*, and so executes mostly out of cache. This is examined in more detail under “Reducing Cache Misses” on page 52. At the other extreme, there is a large penalty of lost time for any program that causes pages to be swapped in and out of memory.

Cache Coherency Updates

cache coherencyEach CPU has two levels of cache that hold copies of memory data. Copies of the same data can exist in multiple caches at the same time. When a CPU writes to its cache memory, it broadcasts the fact on the processor bus. Other CPUs that have cached the same location mark their cached copies as invalid, so that if they need to refer to it again, they will reload the modified data.

This is a greatly oversimplified summary of a complicated protocol that ensures consistent, correct behavior of the multiple CPUs, even when they use the same memory areas. (For details on the subject, refer to one of the MIPS processor books listed in "Other Useful Books" on page xxiii.) Cache coherence is built into the hardware at a low level, and your program does not need to take any special steps to maintain it.

Virtual Memory

address spaceIn general, each UNIX process has its own *address space*. The process sees the address space as a continuous range of memory locations containing the process's code, data, and other resources.

The composition of the address space, and the methods by which a process can share it with other processes, are covered in Chapter 4, "Managing Virtual Memory in a Real-Time Program."

page sizememory:virtualvirtual memorymemory:page sizeThe IRIX kernel manages each process's address space as a set of *pages*. All pages are the same size in one implementation of IRIX. (The page size is 4 KB in 32-bit systems, but larger in 64-bit systems. Programs should always determine the page size dynamically by calling the **getpagesize()** function.)

virtual memory:page faultpage faultSome or all of the pages that represent a process's address space may be stored on disk. When the process attempts to access a page not in memory, it causes a *page fault* interrupt. The kernel suspends the process until it can provide the page contents. If the page has defined contents, the kernel schedules a disk I/O operation to load it. If this is the first use of a stack or heap page, the kernel simply creates a page of zeros. In order to make room for the needed page, the kernel may have to invalidate some other page, and may have to save the contents of the other page to the swap disk.

A page fault causes an unpredictable and possibly lengthy pause in the execution of a process. A real-time program cannot tolerate such delays. However, you can have part or all of your program’s address space locked into memory, so that a page fault cannot occur.

Translation Lookaside Buffer Updates

TLBtranslation lookaside buffer. <Italics>See<Default Para Font> TLB<$nopage>Virtual addresses are mapped to real memory locations using translation tables kept in memory. For speed, each CPU has a cache of recently-used page addresses, called the *translation lookaside buffer (TLB)*.

interrupt:TLBTLB update interruptUnder certain conditions, kernel code executing in one CPU can change the address space mapping in a way that could invalidate TLB entries in other CPUs. In order to synchronize the TLBs, the kernel broadcasts an interrupt to all CPUs. The interrupt service routine in each CPU purges the TLB for that CPU so it will be reloaded with accurate values. Memory accesses immediately after a TLB purge are slow, while the TLB contents are reconstructed. The TLB update interrupt comes at unpredictable times. A real-time program with tight timing constraints cannot tolerate being delayed this way.

However, when you dedicate one or more CPUs to executing your real-time program, you can isolate your dedicated CPUs from TLB interrupts. (For details, see “Isolating a CPU From TLB Interrupts” on page 84).

Device Interrupts

device interruptinterrupt:devicedevice driverWhen a device needs attention, it requests an interrupt. This forces one CPU to trap to an interrupt handler to service the interrupt. The interrupt handler locates a *device driver* that can respond to the interrupt. There are two kinds of device drivers:

- Multiprocessor-aware device drivers can run on any CPU. The interrupt handler enters the code of the device driver immediately, on the CPU that was interrupted.
- Device drivers that are not multiprocessor-aware cannot be executed safely on any CPU. The interrupted CPU in turn interrupts CPU 0, and then returns to the interrupted work. The interrupt handler in CPU 0 calls the old device driver.

Interrupts at the same and lower priority levels are masked off (blocked) in the interrupted CPU while the device driver is running. Other CPUs continue to run, and can even receive interrupts.

The design of multiprocessor-aware device drivers is covered in the *IRIX Device Driver Programmer's Guide* (see page xxiii). Disk and network drivers are always multiprocessor-aware. However, VME device drivers (other than disk drivers) are not required to be multiprocessor-aware.

VME Interrupts

interrupt:VME busVME bus:interrupt levelsbus,VME. <Italics>See<Default Para Font> VME bus<$nopage>Interrupts from the VME bus are grouped into 7 priority levels. Each device on the bus uses a particular level. Higher numbered levels have superior priority (IRQ7 is superior to IRQ1).

interrupt:sprayingBy default, interrupts are "sprayed" (dynamically distributed, in rotation) to all CPUs in order to equalize the load of handling interrupts. You can control this in two ways:

- Designate CPUs that are not to receive sprayed interrupts. You would do this to protect real-time processes in those CPUs from being interrupted by devices not related to real-time work.
- Specify that interrupts of a specified VME interrupt level are to be directed to a specified CPU. You would do this either to group all non-real-time interrupts on a designated CPU, or to direct real-time interrupts to a CPU that is dedicated to handling them.

For details on these actions, see "Minimizing Overhead Work" in Chapter 6.

Interrupt Latency

interrupt:latencyWhen interrupts come from the real-time input and output devices, you are concerned about *interrupt latency*, the amount of time that elapses between the hardware signal and the start of the IRIX kernel's response to it. Interrupt latency has several sources, some of which you can control. (See "Components of Interrupt Response Time" in Chapter 6.)

Interrupt Response Time

device service timedevice driverinterrupt response timeThe time that elapses from the arrival of an interrupt until the system returns to executing user code is *interrupt response time*. It includes interrupt latency, plus the time spent in the device driver (called *device service time*), plus the time IRIX needs to switch program contexts, and other factors. When you take full advantage of the features of IRIX and REACT/Pro and configure the system properly, you can guarantee a maximum 200 microsecond interrupt response time. See “Minimizing Interrupt Response Time” in Chapter 6.

Processor Arrays

The POWERChallenge Array is a collection of two or more POWERChallenge systems, each one of which is a symmetric multiprocessor as described in the preceding topics. Within each “node” of the Array there are multiple CPUs, a system bus, and a single memory. The nodes are connected by a high-speed network, HIPPI or FDDI.

The real-time features discussed in this book apply within one Challenge/Onyx system, whether it stands alone or is a node in an Array system. You can distribute an application across multiple nodes of an Array using the Message-Passing Interface (MPI) standard. However, the MPI standard does not provide for guaranteed message latencies. As a result, you cannot distribute a real-time application across nodes of an Array. You can run multiple, real-time, applications in different nodes of an Array, but you cannot synchronize them at real-time levels of determinacy.

Process Management

process<$startrange>A *process* is one executable instance of a program. The IRIX kernel creates new processes, and by default it attempts to schedule their shared use of the hardware in a fair and effective way. You can alter the default scheduling to favor a real-time program in several different ways.

Process Composition

kernel:process managementprocess:attributesprocess:compositionA process consists of an address space containing the program text and data, and a number of *process attributes* managed by the IRIX kernel. A few examples of process attributes are

- process IDa unique process ID number
- machine register contents, representing the current instruction and stack level as well as working data
- user IDgroup IDUNIX user and group identities
- current directorycurrent working directory for file searches
- signal handler:as process attributesignal-handling status

For a more complete list, refer to the fork(2) reference page and read the list of attributes that a new process does and does not inherit from its parent.

Process Creation

device driverThere are two system calls that create a process. They differ in that one creates a new address space and the other does not.

Normal Process Creation With fork()

address space:duplicated by <Function>fork()<Default Para Font>process:created with <Function>fork()<Default Para Font><Function>fork()<Default Para Font>The conventional method of creating a new process in UNIX is to issue the **fork()** system call. It creates a “child” process, which is a copy of the “parent” process that issued the call. The address space of the child is a duplicate of the parent’s address space, as are most of its attributes, including its machine register contents. Only the return value of **fork()** differs. The use of **fork()** is shown in Example 2-1.

Example 2-1 Schematic of Using fork()<Function>fork()<Default Para Font>:example

```
int childProcId;
switch(childProcId = fork())
{
case 0:
   { /* this is executed by the child process */ }
   break;
case -1:
   { /* parent process, no child process created */ }
   break;
default:
   { /* parent process, child process exists */ }
}
```

copy on write page statueaddress space:copy on writeIRIX does not physically duplicate all the pages of the parent's address space. That would waste a great deal of time. Instead, the page translation table that defines the child's address space initially refers to the physical pages of the parent's address space. However, the table designates these pages as "copy on write."

Whenever the child process writes into a page, it causes a hardware trap. The kernel then makes a duplicate of that one page so that the child has a unique copy into which it can write. Thus only the pages that are written are copied, and then only when the child uses them.

Address Space Replacement With exec()

process:attributes initialized by <Function>exec()address space:replaced by <Function>exec()<Function>exec()<Default Para Font>The **exec()** system call is the means by which UNIX "loads a program." This call replaces the entire address space with a new one based on a program image loaded from an executable file. The **exec()** call also initializes many of the process attributes (refer to the exec(2) reference page for details).

The combination of **fork()** and **exec()** suits the needs of a command shell. The way a UNIX command shell launches a program is to **fork()**, creating a new process. In the new process (case 0 in Example 2-1) it calls **exec()**, replacing the new address space. As a result, in the great majority of **fork()** calls, the child's address space is completely replaced before more than one or two of its pages have been copied.

However, **fork()** is not well-suited to building a program designed as a number of small, cooperating processes—the kind of design that your real-time application needs if it is to exploit multiple CPUs.

Lightweight Process Creation With sproc()

address space:shared by lightweight processesprocess:lightweight. <Italics>See<Default Para Font> lightweight process<$nopage>lightweight process:created with <Function>sproc()<Function>sproc()The **sproc()** system call is unique to IRIX. It creates a new process that shares its parent's address space. The new process has its own machine registers and its own memory region for its stack. Otherwise, both processes execute concurrently using the same program text and data, and sharing many process attributes. A parent process and its children by **sproc()** constitute a *process group*.

lightweight process:preferred for real-time usereal-time program:lightweight processes preferredFor several reasons, you should use **sproc()** if you structure your real-time application as multiple, cooperating processes:

- lightweight process:less work to createThe kernel does less work to create a process with **sproc()**. For example, it does not have to build a page table to describe a new address space.
- The parent process can initialize disk files, device files, global data structures, memory-mapped I/O, and other objects, and all these are automatically available to the child processes.
- The parent and all child processes have write access to global data, and can use high-performance semaphores and locks to regulate access.
- There is only one address space to lock into memory, no matter how many processes use it.

Process Scheduling

scheduling:assumptionskernel:scheduling assumptionsprocess schedulingWhen managing a mix of programs, the IRIX kernel attempts to keep all CPUs busy and all processes advancing, and is generally successful at this. (For details, see "Using Priorities and Scheduling Queues" on page 67.) By default, the IRIX kernel schedules processes to execute under these assumptions:

- There are far more processes (dozens to hundreds) than there are CPUs to execute them.
- The system's resources should be shared among all processes as equitably as possible.
- Most processes spend most of their time waiting for input or output.
- As long as a process makes some progress (is not blocked indefinitely), its exact rate of progress is not crucial ("the system is busy" is always a valid excuse for slow response).

real-time program:and scheduler assumptionsHowever, when a real-time program is running, the assumptions for scheduling must change: there is typically only one real-time program in a system; you are prepared to give it all of the system's resources if necessary; it spends very little time waiting for input. Most important, its precise rate of progress is an integral part of its design, and "the system is busy" is never an excuse.

Your real-time program can give itself a high scheduling priority or, if it cannot tolerate time-sharing at all, it can seize one or more CPUs and dedicate them to its exclusive use. The specific calls are surveyed in Chapter 3, “How IRIX™ and REACT/Pro™ Support Real-Time Programs” and covered in detail in Chapter 6, “Controlling CPU Workload”.

I/O Scheduling

process:blocked by I/OWhen a process initiates I/O, IRIX usually suspends the process until data transfer is complete. By understanding the I/O system, and by using the Asynchronous I/O feature, you can make sure that a real-time process is not blocked in this way.

Disk I/O

When a process requests disk input, it is blocked until the data has been read and copied into the designated buffer. When a process requests disk output, it is blocked until the data has been copied into a kernel buffer or until the disk write is complete, depending on the options used when the file was opened.

VME Bus I/O

VME bus:and process schedulingYour program can perform I/O to the VME bus in three ways: programmed I/O (PIO), direct memory access (DMA) from VME Bus Master devices, and a unique form of DMA from VME Bus Slave devices.

When it uses programmed I/O, your program polls the device registers or memory as if they were variables in memory, and does not block. Your real-time program can do PIO in a time-critical process.

VME-bus I/O using either form of DMA generally does delay the requesting process until the DMA transfer is complete. All of these methods are discussed under “Program Access to the VME Bus” on page 168.

Other I/O

file descriptor:returned by <Function>open() device:opening <Function>open()In general, UNIX allows your process to open any device for I/O with the **open()** call. You specify a pathname designating one of the device special files found in the */dev* directory. The **open()** call returns a file descriptor which you can pass to the **read()** or **write()**

functions. For device files, these functions are routed directly to the device driver for the device. Through this means your program can read or write serial devices, SCSI devices, and (in SGI systems other than Challenge/Onyx), devices on the GIO or EISA bus.

A call to a device driver for input or output normally blocks the calling process until the data has been transferred.

Asynchronous I/O

asynchronous I/OTypically, a real-time process cannot allow itself to be blocked for I/O. *Asynchronous I/O* is a feature of IRIX that gives you the ability to schedule I/O to be done in a separate process. This process—created automatically for you—requests the I/O and waits for it, while your real-time process continues to execute. For details on asynchronous I/O, see Chapter 8, “Optimizing Disk I/O for a Real-Time Program.”process<$endrange>

IRIX overview<$endrange>