Integer Arithmetic, Logical, and Shift Instructions

These instructions perform basic operations on integer or general-purpose registers. In some of the older CISC[1] ISAs, an arithmetic, logical, or shift instruction may use a memory location as one (or more) of its operands. In this case the processor needs to perform an implicit memory load before performing the specified integer operation and/or an implicit memory store afterward. Most modern ISAs, especially RISC[2] ISAs, separate the process of loading an operand from memory from the process of performing the actual arithmetic, logical, or shift operation.

Floating-Point Instructions

These instructions perform floating-point operations. The operands of the instructions are generally, but not necessarily, held in typed registers called floating-point registers. In the Intel IA-32 instruction set, the floating-point registers are arranged as a stack, and most floating-point instructions refer to the registers only implicitly via the floating-point stack pointer. The principal difference from integer instructions is that the values in the registers are interpreted to be in a specified floating-point format. The standard format is the IEEE floating-point format, which specifies not only the encoding of floating-point data but also the acceptable results of operations performed on floating-point data.

Branch and Jump Instructions

While the previous categories of instructions change the contents of the data-holding resources within the processor, branch and jump instructions simply change the flow of control. In stored-program machines, this change is accomplished by changing the PC, thereby changing the memory location from where the next instruction is to be fetched. Conditional branches test values in registers and branch

based on the outcome. The register to be checked can be a general-purpose register or a special condition code register set as a side effect of executing other instructions. ISAs also include jump instructions, which are similar to branches, except they transfer control unconditionally[3] and are often used for transferring control to a logically different piece of code from the current one (e.g., to a subroutine or procedure). The targets of jump instructions are sometimes not known at compile time; hence jumps can take place indirectly through the contents of a register. Some jumps have the side effect of saving the location (PC value) of the instruction immediately following the jump in a *link* register. If a procedure is invoked using one of these jump and link instructions, a procedure return can be implemented as an indirect jump to the PC value saved in the link register.

A.3 The System ISA: Resource Management

While the user portion of the ISA is primarily for getting computational work done, the system part of the ISA is for management of system resources. The operating system receives requests for accessing or altering resources and services them according to its own well-defined resource-management polices. The system ISA contains the actual mechanisms by which the OS communicates with the underlying hardware in order to carry out its requested services and resource-management decisions.

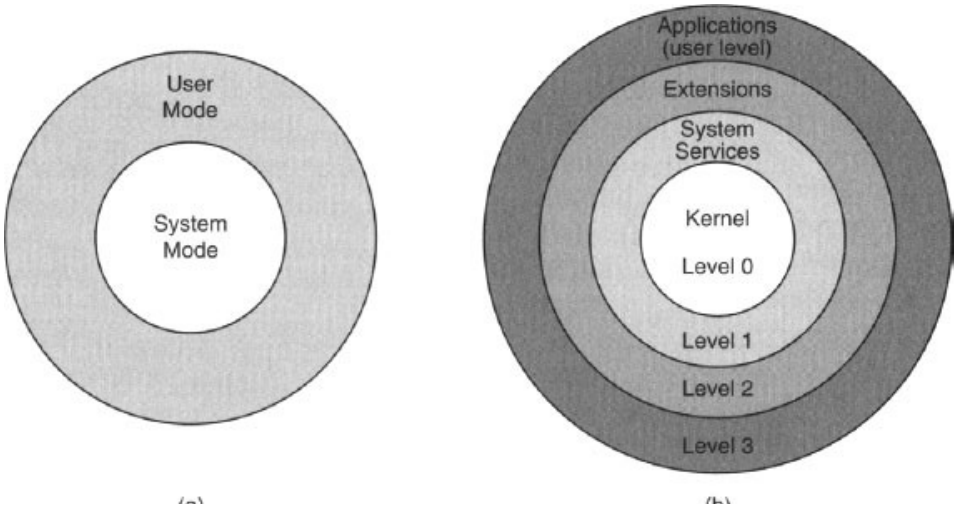
A.3.1 Privilege Levels

A modern processor can simultaneously support multiple applications or processes, each of which has access to system resources, including real main memory, secondary memory, and other parts of the I/O system. Even though many user programs may be operating

on the machine at the same time, the use of resources by any user program should not be seen or affected by the use of the same or other resources by any other program, unless it explicitly grants this permission to the other program. This protection is made possible by the operating system, which also ensures that resources are allocated fairly among all user programs.

To do this, the operating system must enjoy certain privileges not normally extended to a user program. These privileges typically involve the allocation, accessing, and altering of physical resources within the system. Special privileges to system resources are permitted by defining modes of operation in the ISA. Certain resources can be made accessible in one mode and not in the others. Usually an ISA specifies at least two modes of operation, a *system mode,* in which all resources are accessible to software, and a *user mode,* in which only certain restricted resources are accessible. System mode is sometimes referred to as *supervisor mode, kernel mode,* or *privileged mode.*

Many operating systems (including UNIX and its derivatives) rely on only two privilege levels. On the other hand, the Intel IA-32 ISA supports up to four levels, shown in Figure A.8. Both Windows and Linux, when implemented on an IA-32, use only the innermost level, level 0, for the OS and level 3 for user applications.



(a) (b)

Figure A.8 Privilege Rings in a System. *(a) Simple systems have two levels of privilege. (b) Intel's IA-32 allows four rings, with the operating system kernel executing in the innermost level, level 0, and user applications executing in level 3.*

At any point in time the mode of operation is part of the system state, stored in a small special register or as part of a larger special register. When there are only two privilege levels in a system, the encoding of the mode requires just one bit. Modification of the mode bit(*s*) must be tightly controlled so that a user process cannot arbitrarily change the mode and thereby extend its privileges at will. A controlled change of the privilege level, for example, to perform I/O, is typically effected either explicitly via a system call instruction or implicitly through a trap. A system call instruction transfers control (changes the PC) to a special location specified by the ISA, and the OS is designed so that this location is part of the OS's code region. In addition to the control transfer, the privilege is automatically changed to system mode.

Some ISAs include instructions that behave differently depending on the current privilege level. For example, an instruction that modifies a privileged resource could be designed to perform no operation (act as a *no-op* instruction) if the processor is in the user mode. Thus a prohibited operation cannot be performed by the user. As described in **Chapter 8**, however, such instructions are problematic from the point of view of implementation of virtual machines; it is preferable for the hardware to transfer control of such illegal instructions to the operating system via a trap so that the OS can then take appropriate action.

In the following sections we briefly describe those parts of the architected state visible only to the more highly privileged operating system and then provide more detailed discussion of the way instructions in the system ISA are used for managing privileged hardware resources.

A.2.2 System Register Architecture

A.3.2 System Register Architecture

Most ISAs include special registers to assist with the task of managing hardware resources. These registers are sometimes given only secondary importance because they are not exposed to the application and because the compiler usually does not use them. However, they are very important in system-level virtual machine implementations, because they often are the source of some of the thorniest problems that arise in the implementation of such VMs. Some of the more important registers are shown in Table A.2.

System Clock Register

This register records the number of clock ticks that have elapsed from the last time the register was reset to zero. A clock tick may be measured either in time (milliseconds) or in processor clock cycles.

TABLE A.2 Typical System Registers

System Registers	Examples in PowerPC	Examples in IA-32
System clock register	Time base register	Model specific; time-stamp counter in newer models; clock in I/O system in older models
Trap and interrupt register	Data storage interrupt register Interrupt save/restore register	None, indirect through interrupt table
Trap and interrupt mask register	Machine state register	Interrupt enable flag in EFLAGS register
Translation table pointers	Storage description register Address space register	Page directory base address (control register 3)

Trap and Interrupt Registers

These registers record information about the occurrence of traps and interrupts so that trap or interrupt-handling code can take the appropriate action. Typically, for each trap or interrupt condition, there is a corresponding bit in the register that is set when the trap or interrupt occurs.

Trap and Interrupt Mask Registers

There are situations when the processor should not be interrupted, for example, when another interrupt is already being handled. Interrupts are typically categorized into ordered classes, an interrupt belonging to a higher class having priority over an interrupt belonging to a lower class. At any given point in an execution the mask register specifies those classes of interrupts that will be ignored. Similarly, software may wish to ignore certain trap conditions, and the trap mask register indicates which should be masked off.

Translation Table Pointers

The mappings of logical pages and memory segments to real memory are usually kept in memory resident tables, and the locations of these tables are pointed to by page and/or segment table pointer registers. Because the page/segment table pointer registers are used for managing hardware resources, access to them must be controlled. While in all cases the writing of these registers must be limited to the OS, in many cases user-privilege processes must be prevented from reading these registers as well. For example, reading a page table pointer register may reveal properties of other processes, and this can potentially contribute to security leaks. Also, as discussed in **Chapter 8**, reading resource-related registers in user mode is problematic when constructing system virtual machines.

A.3.3 ISA Support for Managing the Processor Resource

The processor is perhaps the most important system resource, and, like the other hardware resources, its use is managed by the operating system. The ISA support required for doing this, however, is minimal. First, the operating system must be able to give control to a user process. This is done via a system-return instruction that causes the transfer of control (a jump) to the desired point in the

user program and a privilege-mode change to user mode. Second, to make sure that it will eventually get back control from the processor, the operating system can set an interval timer so that when the time interval elapses, there will be an interrupt that returns control to the OS (interrupts are described in more detail in Section A.3.6). The interval timer may be an architected counter, i.e., one of the system registers that can be read or written in system mode. Or it may be built into the I/O system and can be read or written in a manner similar to the way an I/O device would be accessed (Section A.3.5). Control is also given back to the operating system when a user process executes a system-call instruction or when there is a trap or interrupt.

Hence, the only ISA features needed for managing the processor resource are system-call and return instructions and an interrupting interval timer, along with a mechanism for setting it. Other traps and interrupts are not necessary for managing the processor resource; rather, they are mechanisms by which the operating system can be put into a privileged state where it can deal with other resource-management conditions.

A.3.4 ISA Support for Managing the Memory Resource

By far, the most extensive part of the system ISA is memory related. The memory architecture includes relatively elaborate data structures (page and/or segment tables) for keeping track of the usage of memory, along with mechanisms for manipulating the data structures. The key issue is that there is a limited amount of real memory that must be allocated to a number of programs, each of which has an (often larger) architected amount of logical memory. In general, this means that the real memory must be shared, and most of the time a given process cannot be given as much real memory as the size of its logical address space would allow.

The system ISA also provides a way of protecting the memory in use by one program from the others. That is, the ISA must be able to limit the access of a program to only those parts of memory to which it has been granted permission by the OS, whether to read it, write it, or execute instructions held in it. These mechanisms can also be used to protect a program from itself, in a sense. For example, parts of memory that hold instructions can be protected from being accidentally overwritten by a store instruction due to a program bug.

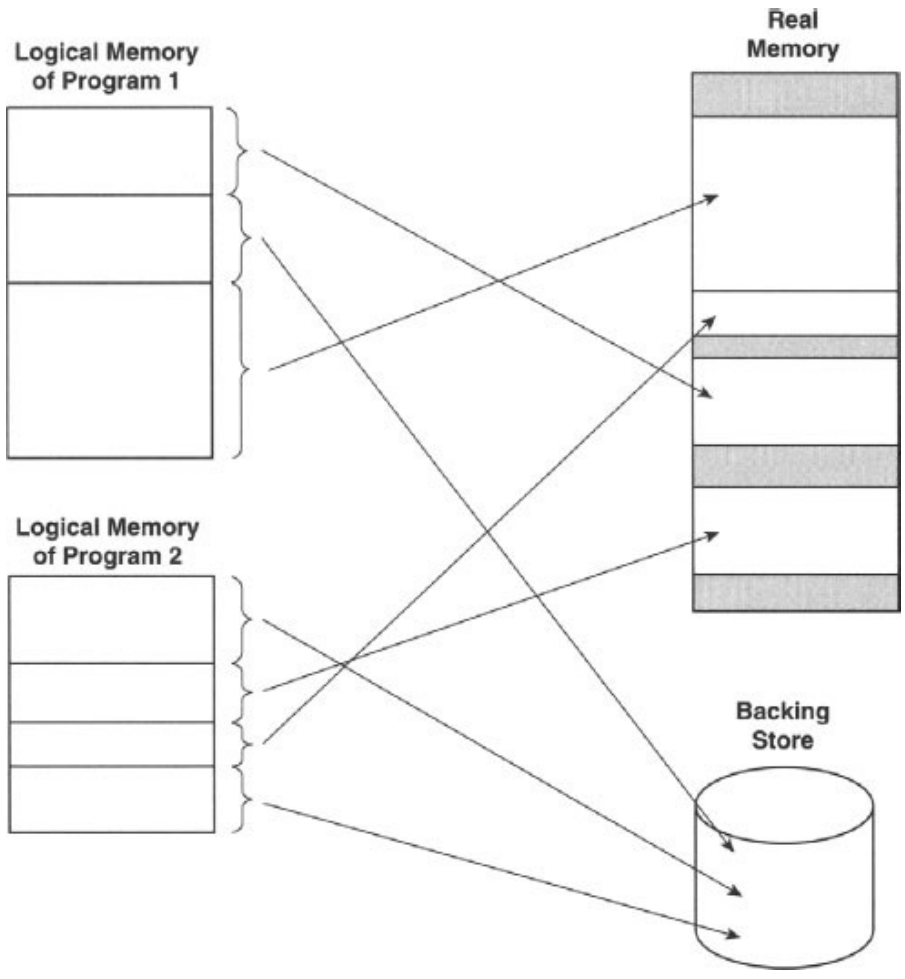


Figure A.9 Logical-to-Real Memory Mapping. *Some regions of memory are mapped to real memory; others reside on a backing store, usually a disk.*

Figure A.9 illustrates the way real memory resources can be allocated to two programs, each with its own logical memory space. Some portions of each of the programs have real memory allocated

Some portions of each of the programs have real memory allocated to them, and other portions do not. Some of the portions of logical memory that are not assigned real memory have their contents held in a *backing store,* usually a disk. Other portions of the logical memory may be unused. The operating system keeps track of the portions of the logical memory space in main memory and in a backing store. The OS moves memory contents back and forth between real memory and the backing store based on its own internal memory-management policies. Thus, a particular real memory location could represent different logical memory locations at different points in time.

With this approach, program memory has become a *virtualized* resource. The user process can directly address memory resources that are defined by its own logical address space. However, this logical address space represents only a virtual view of memory. The actual real memory resources are assigned in a very different manner. It is for this reason that such a system is called a *virtual memory* system and that logical addresses are often referred to as *virtual addresses.*

For assigning logical memory to real memory, the respective memory address spaces are normally divided into blocks. There are two general approaches for doing this. In one, the blocks are of arbitrary size and are called *segments.* In the other, the blocks are all of fixed size and are called *pages.* Managing the real memory is simplified if the basic elements for management have the same fixed size, so just about all ISAs today use paging. However, as we saw in Figure A.7b, some ISAs add segments on top of paging. When this is done, each segment is composed of some number of pages.

When pages are used, a logical address can be interpreted as having two parts, a page number and an offset within the page. For example, if 32-bit addresses and 4KB pages are used, then the low-order 12 bits identify an offset within a page, and the upper 20 bits form

the page number.

Page Tables

To support logical-to-real memory mapping, a data structure known as the *page table* is used. An example page table that maps logical pages to real pages is shown in Figure A.10, with the page number of the virtual address used as an index into the table. Page table structures can get more complex, but the one shown is adequate for understanding the basic operations. Each entry in the table contains information about the real memory location of an accessed virtual page. When the virtual memory space is larger than the real memory space, as it often is, some of the virtual pages may not be mapped to real memory. Hence there is also a valid bit associated with each page entry to indicate whether or not the page is mapped. Note that the page table can also be used for data that is shared among multiple programs if the same real page is mapped into the virtual address spaces of the sharing programs.

As suggested earlier, in Section A.2.2, it is usually possible to restrict the type of access to a location in memory given to an application program. The granularity at which such access protection is specified is generally the page, and hence each entry in the page table also stores access-protection information. These are the “prot” fields in the page table of Figure A.10. The types of operations that can be performed on a page may be determined by a program’s privilege level, e.g., whether it is in supervisor or user mode. The three types of accesses that are commonly controlled are *read, write,* and *execute.* Read and write accesses are checked during execution of load and store instructions, while execute accesses are checked when instructions are fetched for execution. One way of specifying the access protection of a page is to provide three bits, each corresponding to the read, write, execute (R, W, E) privileges. Note that the access protection given to a page is not static. The same real page could have one set of protections when the machine is operating in the

user mode and a different set when operating in the system mode.

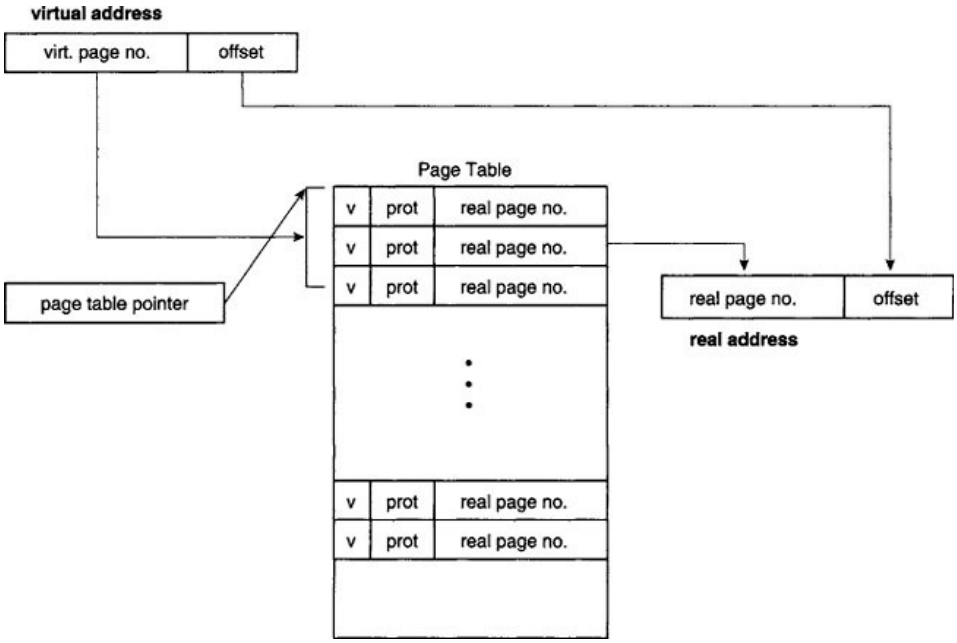


Figure A.10 Page Table for Mapping a Linear Address Space.

Translation Lookaside Buffers

The page table is a relatively large structure and is generally held in main memory. In theory, the page table is consulted for every load, store, or instruction fetch. Actually doing so, however, would be very slow. To make memory accesses much faster, a small associative memory structure called a *translation lookaside buffer* (TLB) is used for caching recent address translations, as depicted in Figure A.11. The TLB is very much like the cache memory described in Section A.1.2 and similarly relies on the principle of locality. When an address needs to be translated, the TLB is accessed by associatively indexing it using the virtual page number of the logical address. On a match, the table returns the real page number for the referenced logical page. As the principle of locality suggests, the number of distinct pages referenced is typically small and changes slowly during the execution of the program. Thus the size of the TLB need not be large, and access to the TLB can be relatively fast. Each entry in the TLB also contains the page-protection bits copied from

the page table.

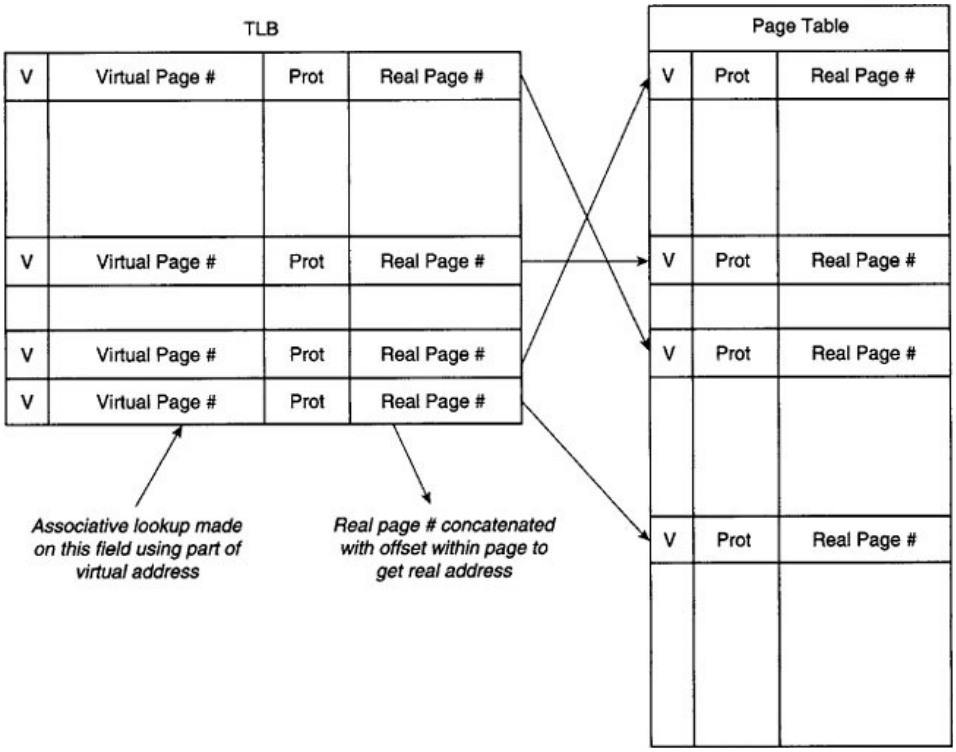


Figure A.11 Structure of a Translation Lookaside Buffer (TLB).

There are three possibilities when a page address is presented to the TLB.

- It matches a TLB entry, and the protection associated with the entry permits the type of access being requested; i.e., there is a *TLB hit.* This is the common case, and the address is correctly translated to a real address. In many microarchitectures, a cache memory access proceeds in parallel with the TLB access, so there is little, if any, performance penalty due to the address translation process.
- It could match an entry in the TLB, but the protection bits do not allow the requested access type. In this case, an access exception is generated, and there is a trap to the operating system.
- There may be no entry with a matching virtual page number. This is referred to as a *TLB miss.* There are two reasons a reference can miss in the TLB. The first is that a mapping exists for the address in the page table but not in the TLB. The second reason is that the referenced page is not currently mapped to any page in real memory. In the first case, either the hardware or the operating system copies the mapping from the page table into the TLB, making room, if necessary, by

deleting some older TLB entry. In the second case, an addressing exception is generated, and there is a trap to the operating system. The operating system uses a page-replacement algorithm to replace one of the existing real pages in memory with the requested page from the backing store. In doing so, it may have to rewrite the contents of the replaced page to the backing store if the page was modified while it was in real memory.

Page Table and TLB Interaction

Both the page table and the TLB are important parts of the address-translation process, and their use involves the interaction between hardware and software. The actual architected interface between hardware and software may occur at either the page table or the TLB, depending on the ISA. If the TLB is defined in the ISA, then the actual implementation of the page table is done in software as part of the operating system. If the page table is defined as part of the ISA, then the TLB is part of the hardware implementation and is largely transparent to the software. In the latter case, occasionally the ISA may provide a "purge TLB" instruction that makes the TLB less than completely transparent. The two alternatives are shown in Table A.3.

With an architected page table, the ISA defines a specific method for mapping virtual addresses to real addresses as well as a specific format for each page table entry. The page table itself usually resides in memory, and the page table pointer register indicates the base (first entry) of the page table. If an accessed virtual page is not present in the page table, i.e., if the valid bit is false, a page fault is generated and control is transferred to a special location defined in the ISA that contains the routine to handle the fault (part of the OS). The ISA also specifies where the system should save information regarding the accessed page. For example, the address that caused the fault may be placed in a special ISA-defined control register.

TABLE A.3 Architected TLB Versus Architected Page Table

	Architected TLB	Architected Page Table

TLB entry format	Defined in ISA	Left to hardware implementation
TLB configuration	Defined in ISA	Left to hardware implementation
Page table entry format	Left to OS implementation	Defined in ISA
Page table configuration	Left to OS implementation	Defined in ISA
Miss in TLB	Causes TLB fault to OS	Hardware accesses page table
Miss in page table	Detected by TLB fault-handling software	Causes page fault
New entry in TLB	Made by OS	Made by hardware
New entry in page table	Made by OS	Made by OS

With an architected TLB, the ISA defines the specific method format for TLB entries as well as the size and access method. The page table is part of the OS implementation. The hardware implementation is unaware of the page table's presence. There are special instructions in the ISA that allow the operating system to read and write TLB entries. If an address is not in the TLB, then the hardware undertakes an action similar to a page fault; i.e., it saves the faulting address in a special control register and traps to an ISA-defined memory location. The operating system checks its page table, accesses the page from secondary storage if necessary, and, when the page is present in real memory, updates the TLB with relevant information.

For a more detailed description of address translation issues, the interested reader is referred to an article by **Jacob and Mudge (1998)**.

A.3.5 Managing Input/Output Resources

Typically, the part of an ISA dealing with I/O is relatively small. Because of the wide variety of I/O devices and relatively infrequent I/O management operations, it is more appropriate to deal with the specifics of I/O in software, particularly in the OS software. The ISA needs to provide only a mechanism for addressing I/O devices and to transfer information to and from the devices.

Some ISAs have explicit *I/O instructions*. These instructions usually have a form similar to load and store instructions, but the addresses are completely separate from main memory addresses. The address

indicates a specific device (or a register that is part of the device controller). The execution of an I/O instruction causes signals leaving the processor to be activated in a pattern that corresponds to the address and/or data values of the I/O instruction. The system designer is responsible for using these signals to connect I/O devices (e.g., through a bus) and to inform an OS programmer of the I/O signaling conventions. The IBM System/360 and its successors, as well as the Intel IA-32, are examples of computers that provide such I/O instructions in their ISAs.

Many recent processors incorporate another form of I/O, called *memory-mapped I/O,* sometimes in addition to I/O instructions. With memory-mapped I/O, a specific region of the real memory address space is reserved for accessing I/O devices (see Figure A.3). These addresses do not correspond to real memory locations; rather, loads and stores directed to these addresses are interpreted by the memory controller as commands sent to an I/O device. Different memory-mapped locations are used for sending commands to different devices and for different types of requests to the same device.

In order to protect I/O resources in general-purpose computer systems, I/O instructions are usually privileged and can only be invoked by the operating system. Similarly, with memory-mapped I/O, the real memory addresses used for I/O are never mapped to user-accessible pages, so only the OS has access to them. Because I/O operations are only available to the OS, user programs invoke OS calls that first check to make sure the user program should be able to make the requested access and then use I/O instructions to complete the request.

Finally, interrupts are a part of most I/O architectures. They are used by the I/O system to get the attention of the OS by forcing a transfer of control into OS interrupt handler code. The I/O system may interrupt the OS when it is finished with some requested I/O operation or when there is some condition, such as a device error

operation or when there is some condition, such as a device error, that requires attention. Interrupts are discussed in more detail in the next subsection.

A.3.6 Traps and Interrupts

Traps and interrupts are an important mechanism for transferring control to the operating system when events needing special attention occur. Because they often involve changing the privilege mode and crossing protection boundaries, they are an important consideration in virtual machine implementations.

A *trap* is a transfer of control that occurs as a side effect of an instruction's execution. Traps are generally triggered by an *exception* condition, which is typically an unusual condition detected during the execution of an instruction. Examples of exception conditions include arithmetic overflows, page faults, violations of memory-access privileges, and illegal or unimplemented opcodes. The ISA usually specifies the exception conditions associated with each instruction.

Interrupts occur due to conditions unrelated to the execution of a specific instruction. Interrupts are caused by events occurring external to the process currently executing. Examples of such external events include I/O interrupts, which occur due to events outside the processor, and timer interrupts, which signal the end of a time interval specified by the operating system.

As described in Section A.3.2, the ISA generally provides a mechanism for disabling certain traps or classes of traps. The usual way to accomplish this is through the use of bits held in a mask register. Not all traps can be disabled, however. For example, it makes little logical sense to disable a page fault, and hence there is usually no mask bit associated with a page fault. Also, some traps may be disabled in user mode. For example, an application may not want any

special action to be taken if some arithmetic operation causes an overflow. Other traps, for example, a trap to indicate a memory-access violation, may be masked only in system mode. Often two different mask registers are used, one of which can be written only in supervisor mode and the other which can be written in any mode.

If an exception condition occurs, the corresponding trap bit held in the trap control register is set. If the mask bit corresponding to this trap is set, the trap is disabled and hence no special action is taken. If the mask bit is not set, a sequence of events follows.

1. Instruction execution temporarily ceases and the processor places itself in a "precise" state with respect to the trapping instruction:
   - All instructions prior to the trapping instruction are completed and make all their specified register and memory modifications.
   - Depending on the specification in the ISA, the instruction causing the exception either completes, as in the case of overflow exceptions, or does not cause any change of state, as in the case of page faults.
   - None of the instructions following the trapping instructions modify the process state (registers or memory) in any way.
2. After a precise state is achieved, the program counter of the executing program is saved in an ISA-specified location, either in a control register or in a special memory location. Some or all of the registers (both general-purpose and control registers) may also be saved by the hardware implementation. Modern RISC processors leave the job of saving registers to the trap- or interrupt-handling software.
3. The processor is placed in a privileged mode and control is transferred to a memory location that is specified as part of the ISA. Ordinarily, this location is in the operating system, so at this point the OS gains control of the processor.
4. The operating system saves any remaining critical parts of the state of the trapping process that have not already been saved by hardware, e.g., the registers.
5. The OS code at this point may directly handle the trap, or it may analyze the situation further in order to determine the address of the code that specifically handles the trapping condition. This trap-handling code may be OS code, or in some cases, as in arithmetic overflows, it may be user code. In those cases where the user process has defined a trap handler, the OS passes control back to the user trap handler.
6. After trap handling has been completed, the OS (or user trap handler) restores the

process's precise state, which had been saved earlier, and jumps back to the location in the user program that caused the exception.

Constructing the precise state of the system at the time of the exception is a way of ensuring deterministic behavior of the program. This requirement, however, introduces significant complexity in most implementations of an ISA, including VM implementations. Some ISAs relax this requirement for certain types of exceptions. For example, the PowerPC does not require precise state to be produced in the case of floating-point exceptions under normal operation. For debugging purposes, however, it does provide a mode in which these exceptions are precisely reported.

ISAs also provide instructions whose principal function is to give control to the operating system via a traplike mechanism. Like branches, such traps may sometimes be conditionally invoked. One important type of explicit trap is invoked by the *system-call* instruction. A system-call instruction is executed when a program needs some service from the operating system, e.g., an I/O operation. The actions taken during a system-call-invoked trap are similar to those described earlier; the main additional feature is that the user may specify arguments and parameters in registers or in a block of memory so that the operating system knows exactly what service is requested. This specification is not part of the ISA but part of the application binary interface (ABI) of the system.

Interrupts are treated in a manner similar to traps. The precise state of the processor must be produced in the case of an interrupt. However, because the interrupt is externally caused, there is some flexibility in an implementation regarding the exact point at which the currently running process is stopped and control returned to the operating system.

Interrupts, like traps, can also be disabled through mask bits held in a special control register. Masking interrupts is useful because there are times when the system software is unable to cleanly handle the

interrupt condition. Such a situation arises, for example, when another interrupt has just occurred and software is still in the process of handling it. Masks can also be used to help establish priorities of interrupt conditions; lower-priority interrupts can then be masked off by high-priority interrupts. As in the case of traps, certain interrupts related to the real environment of the processor, such as a "power fail" interrupt or a "high-temperature" interrupt, are not maskable, for they need immediate attention. Most other interrupts related to I/O are maskable.

A.4 Operating System Organization

Operating systems are complex pieces of software that are responsible for managing system resources and for handling requests from programs for services utilizing the resources. A typical OS is divided into major components that handle processor scheduling, memory management, and I/O. Figure A.12 shows the major blocks of the Linux operating system with its various interfaces, including the system-call interface and the device-driver interface. The following subsections describe the ways an OS manages the major system resources.

A.4.1 Processor Management

The operating system time-multiplexes a computer system's processor(*s*) among several processes that may be active in the system. It does this using a scheduling algorithm, often based on priorities, that determines which processes should run and for how long. The scheduler maintains a ready queue that holds information regarding processes that are ready to run. When a processor becomes available, the scheduler selects a process from the ready queue, sets the interval timer according to the amount of time the scheduler decides it should have, and jumps to the process.

If the application process generates an enabled trap condition or performs a system-call instruction, control is given back to the OS. Otherwise, the interval timer will expire, and there will be an interrupt to the OS. In the event of a trap, the OS will invoke the appropriate trap handler (or may terminate the process). If there is a system call or some other condition that requires service from the OS, e.g., an I/O request or a page fault, the OS will set up the needed I/O operation and remove the process from the ready queue. When the service completes, e.g., when an I/O device completes its operation, an interrupt is generated, and the OS puts the process back on the ready queue.

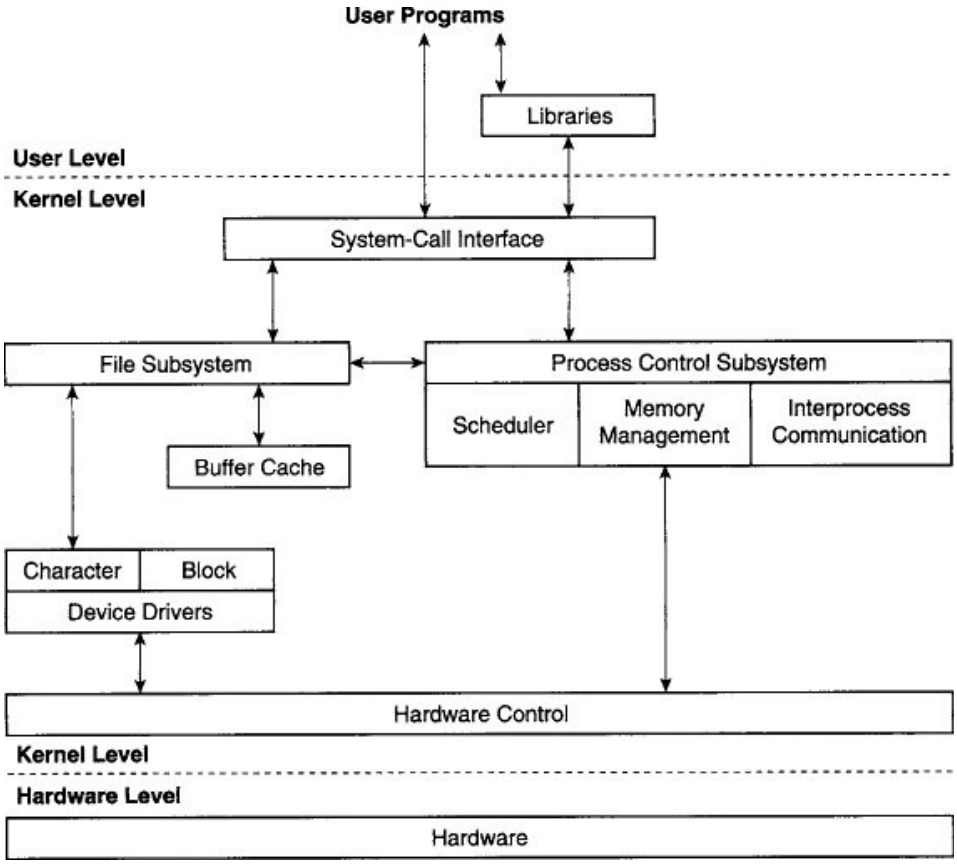


Figure A.12 Linux Architecture.

A.4.2 Memory Management

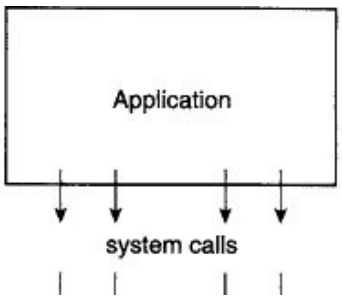
Memory management involves the sharing of real memory among

the processes running on the system. The memory manager interacts between the hardware and the user process via the page table and/or TLB, as described earlier, in Section A.3.4. When a process generates a page or TLB fault, the OS takes over and either updates the TLB from the page table if the TLB is architected or handles the fault by scheduling a disk I/O operation to the backing store in order to retrieve the page. In many cases, the OS also prefetches a number of adjacent pages and keeps them in a buffer cache in anticipation of their immediate need.

The OS relies on the locality principle and attempts to give each running process those pages that are actively being used, also referred to as its *working set.* Good page-replacement algorithms, for example variations of the least-recently-used (LRU) algorithm, help to control the number of page faults incurred by a process.

A.4.3 Input/Output Management

As noted earlier, an operating system abstracts most of the details of hardware devices and makes these I/O devices accessible through well-defined interfaces. As shown in Figure A.13, there are really two major interfaces that come into play when an application invokes an I/O service. The service is invoked using a system call, which transfers control to the operating system. The operating system itself uses an interface to a set of software routines that convert generic hardware requests into specific commands to hardware devices. This layer is referred to as the *device driver layer,* and the interface to this layer is through device driver calls.



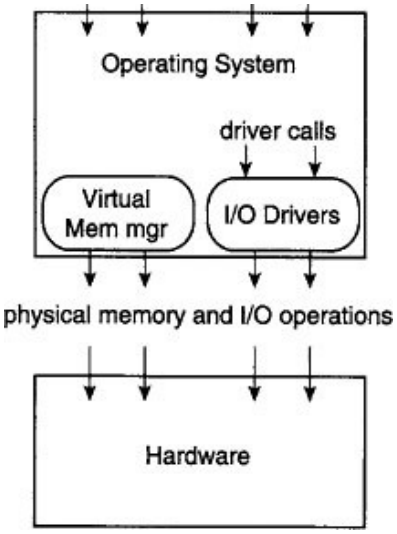


Figure A.13 Interfaces Are Related to I/O Operation.

The device driver approach is the common technique for conducting interactions between the operating system and a hardware device. The device driver takes care of device-specific aspects of performing an I/O transaction. For example, when a file system uses a block-device interface to write to a disk, the device driver converts a device-independent request to appropriate requests specific to the type of disk controller chip that is physically attached to the system. These specific requests are then conveyed to the I/O device via I/O instructions or memory-mapped loads and stores.

There are two types of device drivers in Linux, character device drivers and block device drivers. Character device drivers communicate directly with the user program, with no buffering in between. A terminal is an example of a character device driver; communication between the device driver and the hardware is performed a character at a time. In contrast, the communication between the driver and the hardware is at a much higher granularity in the case of block device drivers, like those for disks. In this case, user programs access information through an area reserved for the transfer of a block of data.

A.5 The Operating System Interface

We now consider ways in which running processes request service or management functions from the operating system. To do this, we focus our attention on interfaces 2 and 3 depicted in **Figure 1.4**. Combined with the user-mode portion of the ISA, these two interfaces form the ABI and the API. The ABI is of most interest because it is the direct interface into the OS. However, the API is also of interest because most user programs access the ABI only through calls to the libraries that make up an API.

Many of the functions specified in an ABI involve the management of hardware resources by the operating system. As discussed earlier, entry into the operating system can occur either through traps and interrupts or through a system-call instruction. The abstraction of traps and interrupts in the ABI is commonly referred to as a *signal*, to be described in Section A.5.4. An example of a system call in the Linux operating system as it is invoked through an API is shown here:

```
# include <syscall.h>
extern int syscall(int, ...);
int file_close(int filedescriptor)
{
    return syscall(SYS_close, filedescriptor);
}
```

The first argument in a system call is a unique identifying number. The Linux kernel uses this number as an index into a table of system-call entry points. Each entry in the table points to the location where the specified system call resides in memory, along with the number of arguments that should be passed to it. The number of arguments supported by a system call in the ABI often depends on the ISA. On Intel's IA-32 architecture, for example, the number of hardware registers available limits the number of arguments to five, in addition to the first argument, which is the system-call num-

ber. However, a register can point to data structures in main memory that hold additional argument-related information. Different system calls can be categorized by the nature of the hardware resource they manage. We next describe these categories along with some examples from the Linux operating system.

A.5.1 Process-Management System Calls

This class of system calls either helps create new processes, terminate processes, or manage various aspects of processes running on the system. A Linux **fork()** system call, for example, creates a new process, called a *child process,* that is an exact copy of the parent process except for its identifying process ID number. An **exec()** system call (or one of its several variations) is used to load and execute a program in the current virtual address space. So starting a new process generally involves a **fork** immediately followed by an **exec.** A process continues execution until it terminates either voluntarily, through an **exit()** system call, or involuntarily, by receiving a signal (to be discussed later). The parent process can determine whether a child process has terminated using the **wait()** system call. Other process-related system calls include **sleep(),** which can be used by a process to voluntarily relinquish use of the processor for a specified time, or **wakeup(),** which supplies a signal to wake up a sleeping process. There are other system calls, such as **setpriority()** and **getrusage(),** that either set parameters of a process or provide information about the resources being used by the process.

A.5.2 Memory-Management System Calls

Each process is provided with its own virtual address space, which the OS manages. Even though the virtual address space belongs to the user process, the management of the space is largely controlled through the operating system. The user may request a block of memory, for example, by using the **malloc()** API routine, which employs sophisticated algorithms to make optimum utilization of the

ploys sophisticated algorithms to make optimum utilization of the address space. The **malloc()** call actually invokes a system call, **sbrk(),** to ensure that the size of the data region, also referred to as the *heap,* is sufficiently large to perform the allocation successfully. The user invokes another library routine, **free(),** which frees the region pointed to by an argument in the call. The user can also change the protections of a set of pages in its space using the **mprotect()** system call. For example, an application that wishes to protect itself from writing over a region of data could invoke **mprotect()** on the page containing the data and designate that page RO (read-only).

Multiple processes that need to communicate with each other often do so by sharing sections of memory. System calls such as **shmget()** allow a memory segment in the virtual address space of one process to be mapped in the virtual address space of another process. That is, both processes map the same memory regions in their page tables.

A.5.3 Input/Output System Calls

A program running under the control of an OS does not invoke the routines in the device driver directly; rather, it makes a device-independent request such as **open()** or **read()** through a system-call interface. The separation of the responsibilities of the operating system from the device driver, as shown in Figure A.13, enables the addition of new devices to a computer system without changing the operating system.

User programs in Linux communicate with the kernel using generic file-system commands such as **open(), read(), write(),** and **close().** The **open()** command associates a file descriptor with a device that has been installed in the system with a file-system name like **/dev/abc.** A program called the Virtual Filesystem Switch registers the device and enables the mapping of the **read()** request, for example, with the appropriate routine in the device driver associated with /

dev/abc, as shown in Figure A.14.

A.5.4 Signals: Abstracting Exception Conditions

An ISA supports traps and interrupts, as described earlier. In Linux and other UNIX systems, these are made available to a process via a set of *signals* that are delivered asynchronously to an application through *signal handlers.* Default signal handlers defined by the system take some default action on receipt of a signal. The default action may stop the process, may terminate the process, perhaps with the generation of a core (memory) dump, or may ignore the signal. However, the application program can override the default action through the use of the **sigvec()** system call. If the application program decides to take specific action, it invokes a *signal handler,* which specifies the action. Almost any signal can be ignored by the process. The only exceptions are the SIGSTOP and SIGKILL signals—these signals provide a way for the user or the system to stop or terminate a runaway process.

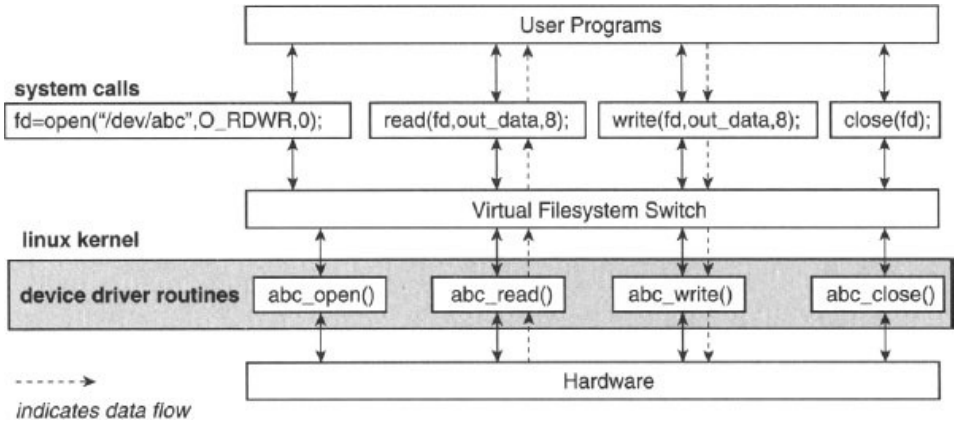


Figure A.14 Linux I/O Programming Interface.

There are two ways in which a signal gets *posted* to a process. It may be due to a hardware trap or interrupt, or it may be generated in software by another process, for example, through the **kill()** system call. Like hardware interrupts, signals may be masked by a process.

If a signal arrives and it is in the masked set, it gets recorded in a list of pending signals, but no action is taken until the signal is unmasked. A signal can be added to the masked list using the **sigblock()** system call, while the entire set of masks can be set together using the **sigsetmask()** system call. Normal execution is resumed after a signal has been handled by using the **sigreturn()** system call.

A.6 System Initialization

An often-overlooked aspect of an ISA is system initialization, beginning with power-up or system reset to the point at which an application program is ready to run. However, this is an important aspect to be considered in the design of system virtual machines, because these exact operations have to be reproduced whenever a new guest system VM is brought to life by a host.

The term used to describe the process of initializing the system is *bootstrapping* (the system lifts itself up by its bootstraps). Bootstrapping involves a series of actions, each action bringing to life a capability that allows the next action to be performed, until eventually the entire system is brought to life. The process of bootstrapping is a highly privileged operation because it involves the initialization of many important resources in the system. Here is a brief chronology of the actions that take place during a system initialization.

1. A processor "reset" is activated through a pin or a signal that enters the processor.
2. Some of the architected resources are loaded with ISA-defined initial values. For example,

- The program counter is set to an initial value, e.g., all zeros.
- Control registers are set to initial values so that initial program execution can proceed; for example, all traps and interrupts will be masked off.
- Page translation is turned off. (It will be turned on as part of the software initialization process.)
- Some, possibly all, of the general-purpose registers are set to initial values. Since

- Some, possibly all, of the general-purpose registers are set to initial values. Since addresses to memory locations are computed using values in registers, registers have to be loaded in order to gain addressability to memory.

3. Instruction fetch begins at the initial location stored in the program counter. This location usually contains a hardwired set of instructions. Typically these instructions are in a block of read-only memory referred to as the *boot ROM.* The code contained in the boot ROM is small and is just sufficient to initialize a boot device from which more code can be brought in. This device may be a tape reader or a floppy disk or the first track of a hard drive.
4. Initializing code is brought in from the boot device and executed. The execution of this code allows other critical aspects of the system to be initialized. For example, the keyboard and monitor are enabled. This code is also able to access larger storage areas, most importantly, a hard drive (or other device) that contains the operating system.
5. The operating system is loaded, and control is transferred to it.
6. The operating system initializes various internal tables, turns on page mapping, locates other I/O devices by activating the I/O buses, performs a cursory check of the file system, and prompts the user to log in.
7. The system is now ready for running an application program.

Some interesting aspects of the boot process follow.

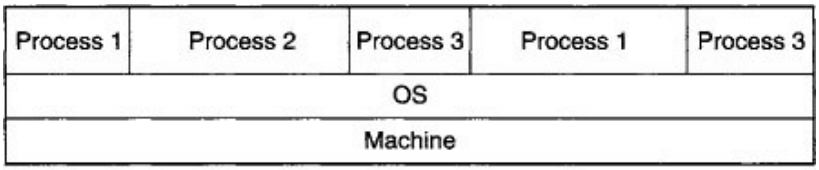
- It is becoming common to initialize network connectivity early in the process of bootstrapping so that the rest of the initialization proceeds using data and code from a remote system.
- With increases in ROM density, many of the functions that were loaded from floppy disks or CD-ROMs can be accommodated in the boot ROM.
- Many systems also perform a self-test early in the bootstrapping process to ensure that critical hardware elements are functioning properly.

A.7 Multiprocessor Architecture

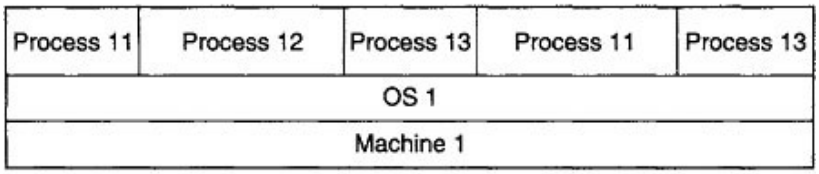
Computers are generally capable of handling multiple processes simultaneously. Uniprocessors achieve this through a technique called *multiprogramming.* In multiprogramming, only one process runs on the processor at a specific point in time. The appearance of parallel execution of processes is achieved by running each process for a maximum period of time, typically around 10 msec. At the end of this “time slice,” the processor state is saved in memory, and the context of another process is brought into the processor’s registers.

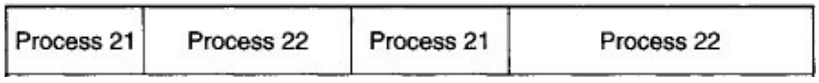
Thus there is a performance degradation of a process due to this sharing of resources with other processes, though this degradation can be reduced by scheduling the context switch at points in the computation where the process would have to wait for some external event such as I/O. Figure A.15a shows the multiplexing of processes in a conventional uniprocessor multiprogrammed system with a single hardware processor and operating system. The multiple user processes run on the same OS, which time-multiplexes the processor and other hardware resources among these processes.

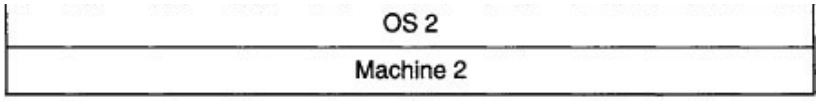
Multiprocessors, on the other hand, employ multiple processors to achieve parallel execution of processes. The multiplicity of program counters, hardware registers, execution units, caches, and sometimes even memory are used to handle multiple processes. If each process is designed to run on a single processor, there is no degradation of performance of the process as a result of executing other processes on the other processors. However, there are programs that are designed in such a way that the multiple processes running on the different processors need to communicate with each other, for example, to share data values or to signal events. The performance of the program is often determined by the efficiency of implementation of this communication.



(a)







(b)

Figure A.15 Comparison of Activity over Time in a Multiprogrammed System and a Clustered Multiprocessing System. *(a) In multiprogramming, several tasks or processes running under a single operating system are multiplexed in time. (b) In multiprocessing, the processes are distributed among multiple processors; however, the number of operating systems cannot exceed the number of processors.*

Further parallelism in execution can be obtained by implementing multiprogramming on each of the processors of a multiprocessor system, as shown in Figure A.15b. The figure illustrates the case where an operating system and its application processes are dedicated to a specific processor in the multiprocessing system. It is also possible to design the multiprocessor with just one operating system that controls the allocation of processes to processors. In this case, a process may run on one processor at one time and on another at another time.

Historically, many systems have been designed around a single processor, and such uniprocessor systems remain an important system class. Today, however, multiprocessors are in widespread use as servers and are well on their way to becoming commonplace for high-end desktop computing. For server implementations, multiprocessor systems with large main memories, disk systems, and high-bandwidth network facilities provide important economies of scale as well as improved opportunities for balancing resources among a large number of simultaneously active application programs. System software employs systemwide data structures for managing processor, memory, and communication resources. When an application thread is initiated, system software assigns it resources from the pool of available processors, and, as it runs, system software provides it with memory, disk, and communication resources.

A.7.1 Types of multiprocessing

There are two main paradigms for supporting multiprocessing: *clus-*

tered computing and *shared-memory multiprocessing.* In many server environments, e.g., Web or database servers, the individual application processes are often independent of each other. This type of workload works well on clustered systems, where the hardware associated with each processor, including the memory and sometimes the disks, is separated from the hardware of other processors on the system. System software in this case typically consists of different copies of an operating system running on each processor, as shown in Figure A.15b.

There are other multiprocessor applications where a single program actually consists of a number of closely cooperating parallel processes. Many scientific and engineering applications are multithreaded in this way, as are compute-intensive commercial applications such as data mining. Many other commercial database applications also are structured with multiple processes sharing data, and they are best implemented on a multiprocessor system with a single operating system. The operating system manages the memory in the system as one large structure shared symmetrically among all the processors and hence the name *shared-memory processing,* or SMP. The shared memory acts like a bulletin board and may be used either to pass data values from one process in the system to another or to signal the occurrence of an event to other processors that may be waiting for that event to occur. Reliable management of this communication of data and events needs special synchronization primitives to be supported by the ISA. The two types of multiprocessor systems are illustrated in Figure A.16.

There also are systems that are hybrids of the clustered and shared-memory forms of multiprocessing. For example, *distributed shared memory* (DSM) systems are implemented like clustered systems but are capable of supporting a single operating system image across the multiple processors. It is hard to achieve as low a latency of communication on such a system as on a shared-memory system. A lot of research has been conducted in this area, and several differ-

lot of research has been conducted in this area, and several different hardware and software techniques have been proposed to improve or hide the communication latency. But this form of computing did not gain popularity, possibly because of the high sensitivity of the performance of applications to the way the algorithms are written and data structures are laid out in memory.

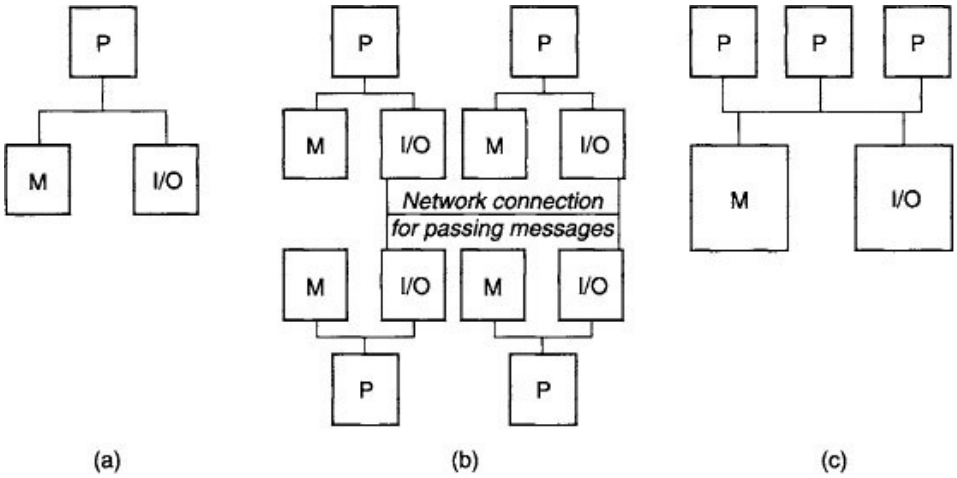


Figure A.16 Comparison of Uniprocessor and Two Common Multiprocessor Systems. *(a) Uniprocessor; (b) message-passing multiprocessor system; (c) shared-memory multiprocessor system (P = processor, M = memory).*

On the other hand, a different type of hybrid appears to be emerging as an important player—*SMP clusters*. Many server systems today are clusters of nodes, where each node itself is a small shared-memory multiprocessor. Communication between processors in a node is through shared memory, while communication between nodes is through messages. Unlike the distributed shared-memory paradigm, there is no effort made in hardware to improve the communication latency between nodes; consequently, operating systems are not expected to span multiple nodes in an SMP cluster.

A.7.2 Clustered Systems

The most common form of cluster is simply a network of computers. Users access the cluster through *terminals*. Terminals may be simply devices that allow the user to type in requests and receive information to be displayed on a screen. More commonly, terminals are

complete workstations or PCs comprising processors, memory, disks, and peripherals. When connected to a network, the terminal can access data located at other nodes in the network. The network itself may be spread over a large area or may be quite compact, with high-speed interconnections. A degree of isolation is achieved on a cluster by the fact that applications generally run on the processor at a local node and cannot easily be affected by applications running on other nodes in the network.

Besides the terminal nodes just described, the network in a cluster typically also has various types of servers attached to it. We have already seen the need for file servers that are repositories of data for users of a cluster. There can also be Web servers, which are nodes that serve as access points to the World Wide Web, mail servers, which are nodes that provide mail services, acting as a repository for new and old mail, and print servers, which service printing jobs from other nodes.

Clustering is also common in very large server systems. Here each node in a cluster may be a large shared-memory multiprocessing system, for example, a 32-way (processor) or 64-way system. Multiple nodes of this type are interconnected by a fast, high-bandwidth, optical network, as shown in Figure A.17. In the largest of such systems, the high-bandwidth interconnection may be a switch rather than a bus. Examples of such clusters are the IBM Parallel Sysplex system for the mainframes (**Nick et al. 1997**) or the HP Superdome Hyperplex system (**Charlu 1999**). Originally these systems provided a solution to expand beyond the capabilities of a single box. Their attractive features include a suite of software system management tools that allow workload balancing and graceful degradation from failures. Additionally, communication routines are provided to enable an application to use resources associated with multiple nodes in a cluster.

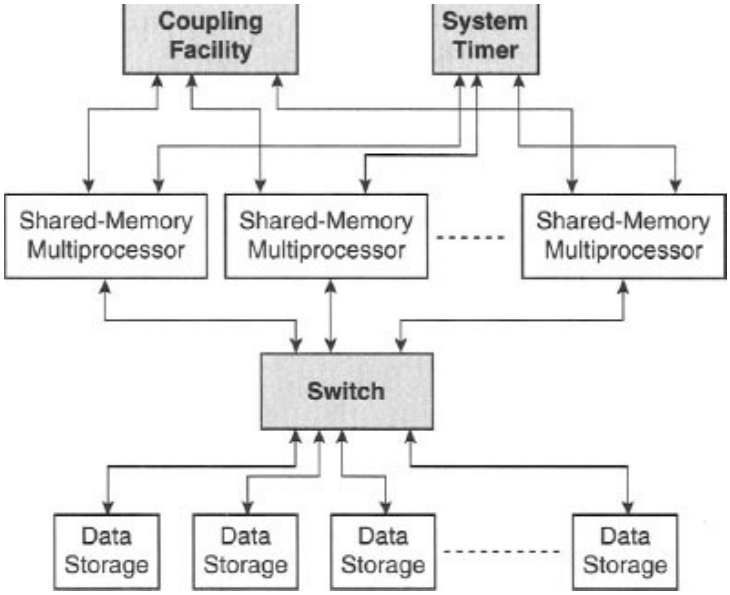


Figure A.17 Typical Large Database Cluster. *The coupling facility provides hardware support for synchronization and coherence between nodes of the cluster. The system timer ensures that the clocks on all nodes are synchronized. The data storage, typically on large disks, is shared between nodes and accessed through a high-bandwidth fiber-optic switch.*

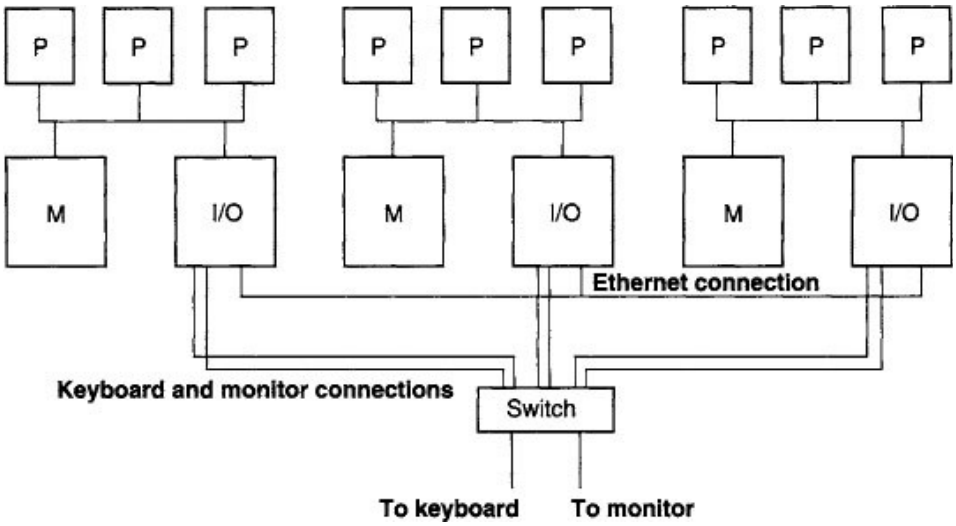


Figure A.18 A Cluster of SMP Nodes. *Beowulf clusters are typically built using inexpensive off-the-shelf tower-type desktop computers connected together through Ethernet and sharing a single monitor and keyboard for the use of the system administrator.*

The availability of inexpensive commodity uniprocessor systems has led to various architectures that combine commodity components to build inexpensive multiprocessor systems. The nodes in some cluster systems, such as the Beowulf cluster systems (**Ridge et al. 1997**), are off-the-shelf uniprocessor or small-scale SMP system boxes with off-the-shelf disks, connected together through commod-

ity networks such as 100Mb/sec Ethernet and running an inexpensive (or free) operating system such as Linux (see Figure A.18). A denser form of the Beowulf cluster, called the *blade server*, is rapidly gaining popularity. Blade servers essentially sacrifice I/O expandability, eliminate the monitor, keyboard, case, and power supply of desktop boxes, and put the uniprocessor or SMP subsystem in a thin package called a *blade*. These blades are then slid into standard racks, where a backplane provides the Ethernet connection and other wiring as well as power.

From a programming point of view, the distinction between shared-memory and clustered types of multiprocessor systems arises from the way processes communicate with each other. In shared-memory systems, two processes communicate by writing to or reading from memory locations accessible to both. In clustered systems, processes communicate by sending messages, and hence they are also referred to as *message-passing* multiprocessor systems. Unlike a shared-memory system, a clustered system presents more of a challenge to an application programmer. In both cases, the programmer is required to organize and partition a program into threads of computation, each to be executed on a different processor. In the cluster case, the programmer is also responsible for the task of synchronizing the computation using messages passed between the threads. Clustered systems would not have gained as much popularity as they have were it not for the software tools and libraries that have been introduced to help develop applications for such systems. Message passing has been recognized as a technique applicable to both closely coupled clusters and more loosely coupled distributed multiprocessor systems. This has led to the development of standard interfaces that allow a programmer to abstract the communication between processors in both types of environments. One of the more widely used interfaces of this type is the message-passing interface (MPI) (**Pacheco 1996**), which specifies a set of library routines that enable processes to communicate with each other and to exchange data. Portability of applications between systems that implement

MPI is one of the biggest strengths of the interface.

A.7.3 Evolution of Shared-Memory Systems

Many commercial applications benefit from having a shared-memory paradigm rather than a clustered design. The shared-memory paradigm is more convenient to reason with and to write programs for, but the hardware needed to support these systems is more complex and specialized as compared to those for message-passing systems. The complexity in hardware arises from the need to maintain coherence in memory locations, especially in modern memory hierarchies, where memory elements can reside at various levels of cache. Message-passing systems avoid this complexity by shifting this responsibility to the application programmer or to library routines.

The ease of programming shared-memory systems, along with innovative hardware techniques, has led to the development of larger and larger SMP systems over the last decade. Today, we find large shared-memory multiprocessor systems having 128 or more processors.

The ISA of most processors today includes special conventions for the access of shared-memory locations. As with all aspects of an ISA, these conventions represent a contract between the programmer and an implementation of the ISA. A programmer writes a program assuming that accesses to variables shared among processors obey certain rules, while the hardware designer (the processor designer or the system designer) ensures that the rules used by the programmer are indeed guaranteed by the implementation.

Memory Coherence Models

Memory coherence refers to the visibility of a write to a given memory location by all other processors in the system. Ideally, one would

like the value written to a memory location to be instantaneously visible to all other processors, meaning that a subsequent read to that location by any other processor should see the new value. This "instantaneous" solution is impractical, however, because of the several layers in the memory hierarchy of most modern systems. So a more practical definition of coherence is adopted—*memory coherence* is said to be implemented on a multiprocessor system if the order of writes to a given location by one processor is maintained when observed by any other processor in the system. Notice that this definition gives flexibility to a hardware designer to delay the observation of changes made by one processor on the other processors. Over the years, there have been several implementation protocols developed to facilitate coherence. These protocols are largely implementation alternatives and are typically not part of an ISA specification.

Consider the example of Figure A.19. Assume that processors 1 and 2 both have write-through caches. When each processor writes to location 50, it does so to a cached version of location 50. As a result, a read following a write will reflect the value in the cached version of location 50. Because the caches are write-through, the value gets written out to memory, though it takes longer for processor 2's write request to reach the memory. In the meantime, the cached value in processor 2 is evicted. When the value of location 50 is requested the next time, both processors get value 1. After a sufficiently long time, when processor 2's write reaches memory and neither processor has a cached copy of location 50, both processors read value 2 from memory. Thus, assuming a starting value of 0, processor 1 sees a transition from 0 to 1 to 2 in location 50, while processor 2 sees the transition from 0 to 2 to 1 to 2 for the same location. This is a violation of the memory coherence rule because it would never be possible for processor 1 to see the 0–2–1–2 order of values observed for processor 2. We note here that such a situation would not have occurred either on a multiprogrammed system or on a multiprocessor

system having instantaneous access to memory—models that most programmers are comfortable working with

There are several ways to avoid such memory coherence violations. One way is to ensure that when a value is written to the cache of a processor, copies of that location cached in all other processors are invalidated and, additionally, to ensure that any read request by a processor is satisfied from a cache that has the most recent valid copy, if the memory itself has not yet received the valid copy. The reader is referred to **Culler and Singh (1999)** for a study of various cache coherence protocols that have been designed to ensure memory coherence.

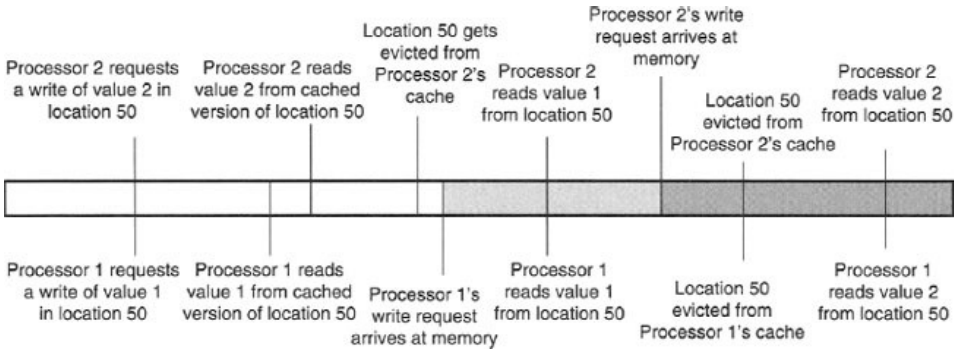


Figure A.19 Violation of Memory Coherence. *The bottom part of the figure shows a write followed by three reads of location 50 by processor 1, while the top part shows the same sequence for processor 2. The shaded region indicates the time when the memory contains value 1, while the hatched region indicates the time when it contains value 2. Note that the values read from the caches are different from the values in memory. The sequence of values returned for processor 1 due to reads is 0–1–1–2, while that returned for processor 2 is 0–2–1–2.*

Memory Consistency Models

Memory consistency characterizes the order in which accesses by one processor to different locations in memory are observed by another processor. This is in contrast to memory coherence, which deals with the order of writes to a single location by a processor. Memory consistency concerns itself not only with accesses to different locations but with both types of accesses, reads, and writes. In general, the more constraining the consistency rules are, the easier it is for a programmer to develop and debug a program but the

harder it is for a hardware designer to produce a high-performance implementation of the processor. An implementation that forces all processors to wait until a written value is visible to all processors in the system provides a maximum level of consistency but is uninteresting because of its high overhead. The frequent, and long, waits would negate a lot of the parallelism benefits expected from executing the application on such a multiprocessor system.

The *sequential consistency* model, described first by Leslie **Lamport (1979)**, is an elegant and natural model. A multiprocessor system maintains sequential consistency if the set of observable memory access orderings made by a multithreaded program is a subset of the observable orderings when the program is run on a multiprogrammed uniprocessor system. It is a natural definition, and it is restrictive but still flexible enough to allow the exploitation of most of the benefits from multiprocessing. Even in a multiprogrammed execution of a multithreaded program, the set of observable access orderings is large, with the variation in ordering arising from the variation of points in time that context switches occur and the variation in allocation of waiting threads to the processor. This is illustrated in Figure A.20.

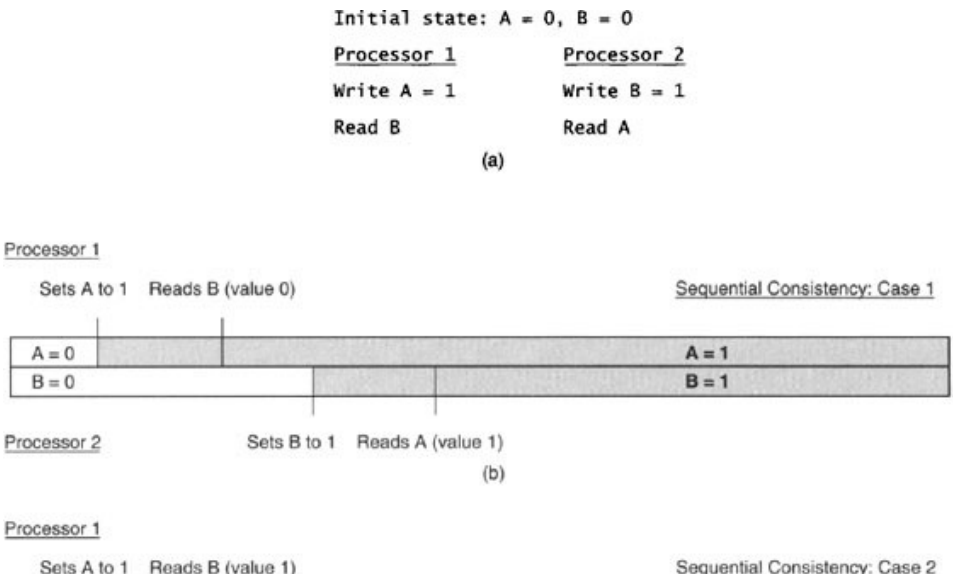
We will simply state here, without proof, that a sufficient condition for sequential consistency in a multiprocessor system is the following: For every pair of accesses to memory by any one processor, the first access in program order is observed by all other processors before the second access is observed. Any implementation that satisfies this condition guarantees sequential consistency in the system.

Processor designers are always looking for ways to improve the single-thread performance in their implementations. Modern superscalar implementations attempt to increase throughput by performing independent operations in the instruction stream in an order that may be different from the original program order. Sequential consistency can get in the way of designing fast uniprocessor imple-

mentations. The general philosophy in these implementations is that programmers generally know the exact points in their programs where they would like ordering preserved, and hence it is not necessary to force an implementation to maintain sequential consistency always. Let us now examine some possible relaxations.

When a pair of memory accesses is made by a processor, various conditions in the hardware, specifically the latency of completing these accesses, may cause the second operation to appear to occur before the first. This potential violation of pair order is referred to as a *hazard.* There are four kinds of possible hazards.

- **A read—read (RR) hazard:** This means that if there are two reads that occur in program order, the second read is seen to occur after the first read. One would assume that when there are no intervening writes, such a reordering of reads should not pose any problem. This is indeed the case with single-threaded programs. However, in multithreaded programs on a multiprocessor system, there is the possibility that a processor accessing one of these locations for writing causes the values read to be different from the values read in an alternative implementation that forces a read-to-read ordering.
- **A write—read (WR) hazard:** This refers to a reordering of memory accesses such that a read occurs before a write, even though program order requires the read to be after the write. A processor is always designed to avoid such a hazard when the locations addressed by the write and read operations are identical.



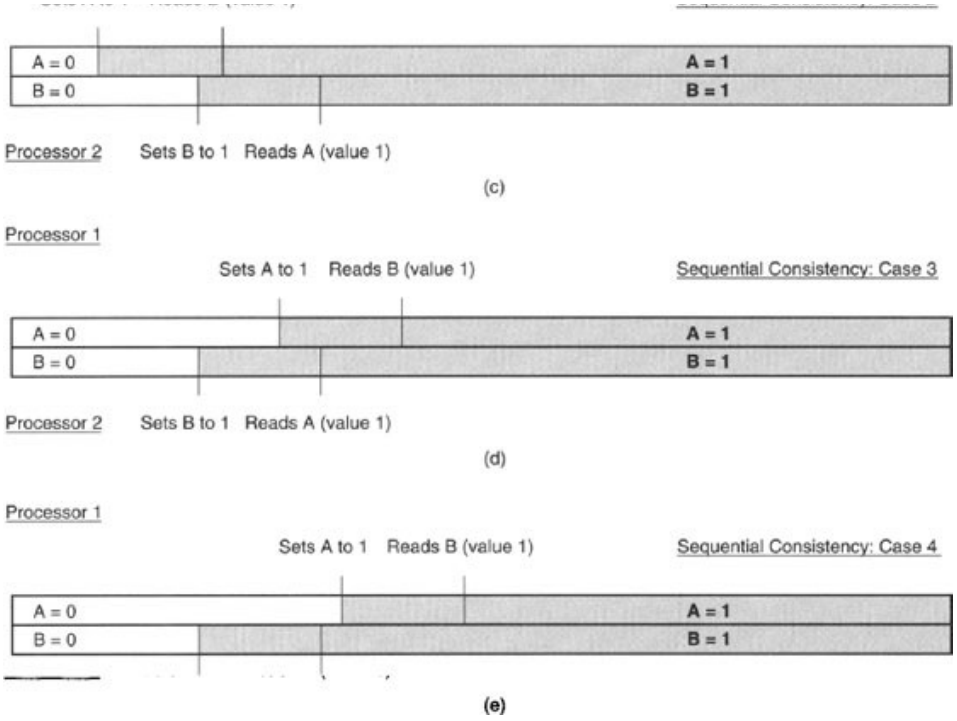


Figure A.20 Sequential Consistency. *If the program shown in (a) were executed in a multiprogrammed environment, the only possible results are those shown in (b) through (e). Hence any sequentially consistent system must produce one of these results. In particular, it is impossible for the values read to both be 0 at the end of these sequences.*

- **A read—write (RW) hazard:** This occurs when a write occurs before a read, even though program order requires the write to be after the read.
- **A write—write (WW) hazard:** Such a hazard occurs when two writes are performed in an order opposite to what was required in the program. Once again, correct execution on a single thread ensures that such a hazard cannot take place when the two referenced locations are identical.

An ISA that disallows all four types of hazards in any of its implementations is said to support a strong consistency model. Most processor ISAs specify consistency rules that are weaker than such a strong model. Both the IBM System/390 and the IA-32 processors originally specified one such relaxed ordering rule, referred to today as *processor consistency*. This rule allows the relaxation of the WR ordering; i.e., it permits operands of an instruction to be fetched even before the result of a previous instruction is written all the way to memory. In the absence of such a relaxation, pipelining of operations in a processor would be seriously hampered because instructions would not be allowed to start access of operands even when these operands are independent of the values written by im-

mediately preceding instructions.

Figure A.21 illustrates how a multiprocessor system implementing a processor consistency model can produce results that are not possible if the system followed the sequential consistency model. The program example is taken from Figure A.20. In the example, if both processors perform the reads before the writes, then it is possible for both processors to see old values simultaneously at the end of the program segment. On the other hand, as Figures A.20b through e illustrate, at least one of the processors must see a new value if the system implemented sequential consistency.

With the advent of superscalar processors that decouple memory access from instruction execution, memory models began to specify even weaker consistency rules. For example, *release consistency* (**Gharachorloo et al. 1990**), which is now supported by many modern architectures, relaxes all memory-ordering requirements except for explicit synchronization operations. The responsibility for correctness in the execution of a multithreaded program is thus shifted completely to the programmer. Special instructions, called *memory barrier instructions,* are provided to allow the programmer or the compiler to impose ordering relationships wherever desired. Examples of memory barrier instructions include the MEMBAR instruction in Sun's SPARC architecture and SYNC in the PowerPC architecture. Both of these ensure that the results of operations after the memory barrier are not seen before the results of operations that occur earlier than the memory barrier have been recorded. Thus, if a strict ordering is desired in the example in Figure A.21a, a memory barrier placed between the writes and the reads would prevent the load instructions from being completed before the results of the store instructions have been recorded.

```
Initial state: A = 0, B = 0
Processor 1          Processor 2
Write A = 1          Write B = 1
```

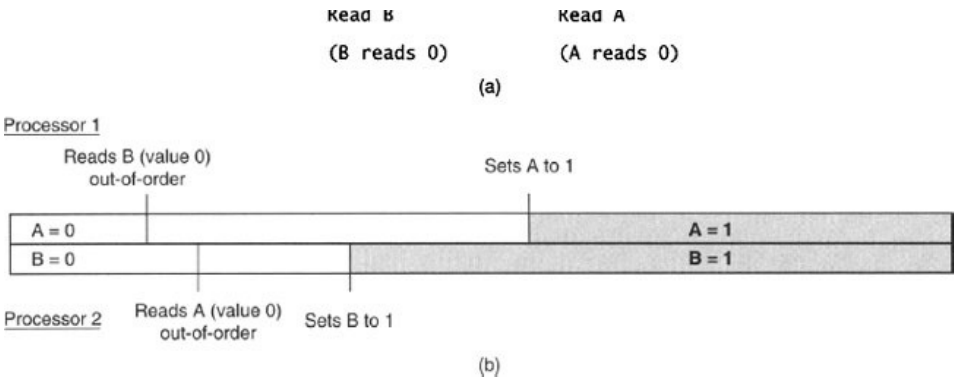


Figure A.21 Sample Program to Illustrate Processor Consistency. *In processor consistency, the reads are allowed to go ahead of preceding writes as long as they are not to the same location. If this occurs, the reads result in a value of 0 for both A and B. This combination is not one of the possibilities shown in Figures A.20b through e.*

A.8 Example Instruction Set Architectures

Most of the examples in this book involve the use of either the PowerPC ISA or the IA-32 ISA. Hence we will describe some of the salient characteristics of these two architectures in the rest of this appendix.

A.8.1 PowerPC ISA

The PowerPC architecture (**IBM 1994**) was defined as a joint effort by IBM, Apple, and Motorola in 1993. However, it was very closely derived from the earlier IBM POWER ISA. Currently PowerPC is used by IBM in its pSeries, eSeries, and iSeries lines of computers and by Apple in its Macintosh line. Versions of the PowerPC architecture are also used in embedded systems developed by IBM and Motorola, in the game machines sold by Nintendo, in the Sony-Toshiba-IBM (STI) Cell processor, and in the IBM BlueGene supercomputer.

The PowerPC ISA is an example of a RISC ISA. The original 32-bit ISA was extended to a 64-bit ISA because of the need to address larger memories and because of the performance advantages of operating

memories and because of the performance advantages of operating on larger units of data. We begin with a description of the state (registers and memory architecture) and then summarize the actual instruction set.

Registers

The PowerPC register set is illustrated in Figure A.22. There are 32 general-purpose registers and 32 floating-point registers. The floating-point registers are 64 bits wide, while the general-purpose registers are 32 bits or 64 bits, depending on the version of the architecture implemented. Both versions can be simultaneously supported in an implementation, in which case the physical registers are 64 bits wide, while only the lower 32 bits will be active when operating in the 32-bit mode.

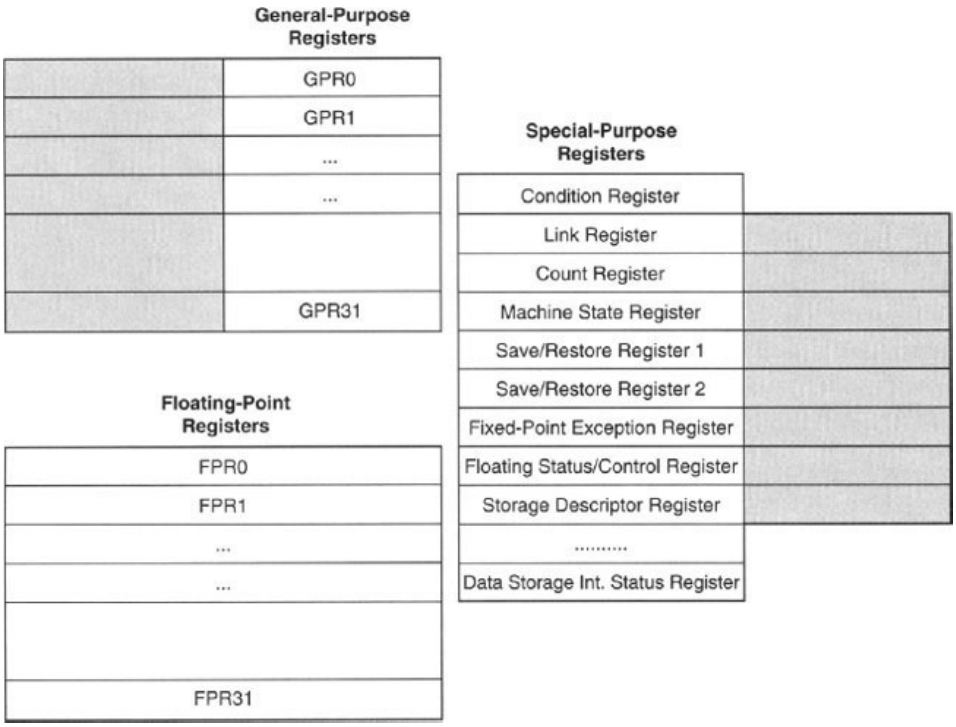


Figure A.22 PowerPC Register Set. *Shaded areas indicate registers that change size from 32 bits to 64 bits when going to 64-bit mode.*

The main special-purpose registers used by an application are the link register and the count register, each 32 or 64 bits wide, depending on the mode, and the 32-bit condition register, divided into eight

ing on the mode, and the 32-bit condition register, divided into eight independently addressable fields, one of which is set as a side effect of executing certain instructions. The fixed-point exception register contains fields set or used by fixed-point instructions, while the bits in floating-point status and control register allow the monitoring of certain conditions resulting from the execution of floating-point instructions. Both these registers contain "summary" bits that can be set implicitly by the execution of an instruction but must be reset explicitly. These bits provide summary information about whether a certain event occurred since they were last reset.

The remaining registers are generally not used during the compilation of a user program. These include a machine state register, MSR, which provides information about the current state of the machine, for example, whether it is in 32-bit mode or 64-bit mode, whether it is in privileged state, and whether address translation is on. There are two registers, SRR0 and SRR1, the machine state status and restore registers, respectively, that are used to save and restore information on interrupts. Other special-purpose registers include the data storage interrupt status register, DSISR, which provides information about the cause of storage and alignment interrupts, and a set of software-use registers SPGR0—3, specially designated for use by the operating system.

Memory Architecture

The address used by a PowerPC application is generally called the *effective address*. The term *virtual address* refers to a larger flat address space that includes the effective address spaces of all processes active on the system. Finally, the term *real address* is used in the conventional way to mean locations in real memory.

The PowerPC memory architecture implements segmentation, with segments that contain up to 2^{28} bytes (256MB). Pages are 2^{12} bytes (4KB) in size. The effective address space is 2^{32} bytes in 32-bit mode and 2^{64} bytes in 64-bit mode. The real address space is also limited

and 2^{64} bytes in 64-bit mode. The real address space is also limited to either 2^{32} bytes in 32-bit mode or 2^{64} bytes in 64-bit mode. The virtual address space is much larger, up to 2^{52} bytes in 32-bit mode or 2^{80} bytes in 64-bit mode.

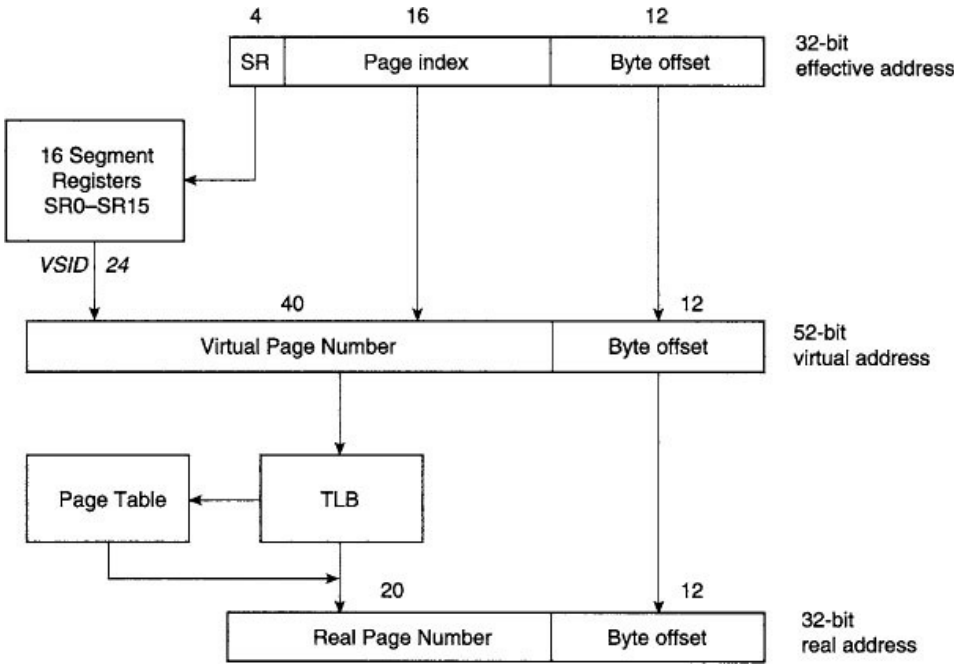


Figure A.23 Address Translation in the 32-bit PowerPC ISA.

In the 32-bit mode, there are 16 segment registers, SR0—15, that hold the locations in virtual memory of the 16 segments available in the effective address space. The top four bits of a 32-bit effective address point to one of the segment registers, which contains the top 24 bits of the segment address (refer to Figure A.23). This 24-bit virtual segment ID (VSID) along with the 16-bit page index from the effective address forms a 40-bit virtual page number (VPN). The VPN is then converted to a real page number (RPN) by referring to the page table whose location in real memory is determined using the storage descriptor register, SDR1.

The PowerPC has an architected page table; that is, the format of entries in the page table is defined in the ISA. In order to make references to the page table more efficient, the table is accessed associatively via hashing. The VPN is hashed in two different ways to arrive

at two pointers, each of which points to an entry group (PTEG) consisting of eight page table entries. A page fault occurs only if the required page cannot be found in either page table group. The probability of a page fault depends on various factors, including the hashing function used, the number of PTEGs implemented, and reference characteristics of the application.

The PowerPC architecture does not mandate the use of a translation lookaside buffer. However, it does provide instructions to maintain a TLB if one is implemented (as it always will be, practically speaking). For example, the **tlbie** instruction can be used to invalidate an entry in the page table when an entry is replaced in the hashed page table. All entries in the TLB can be invalidated (flushed) simultaneously using a **tlbia** instruction.

In 64-bit mode, the effective number of segments supported is 2^{36} instead of 16. The mapping of an effective segment ID to a virtual segment ID is done by means of a segment table pointed to by a special register called the *address space register* (ASR). Once again, in order to make accesses to this table more efficient, two hashed pointers are used to access two sets of segment table entry groups (STEGs), each set having eight segment table entries. A miss here causes a segment fault. A segment lookaside buffer (SLB) is used for avoiding a translation on each reference. Instructions (**slbie** and **slbia**) are provided to invalidate a single SLB entry or the entire SLB.

Instruction Set Summary

PowerPC uses a RISC instruction set that has fixed 32-bit-width instructions. All storage operations are conducted through load and store instructions. Thus there are no instructions that perform arithmetic or logical operations on the contents of memory. Any such operation requires that the operand be brought in from memory into one of the registers before being manipulated.

The PowerPC divides the user instruction set into three groups: the branch processor instructions, the fixed-point processor instructions, and the floating-point processor instructions.

The branch processor handles instructions that set or use the condition register, CR, or that change the flow of control modifying the program counter to deviate from the sequential execution of instructions. These latter instructions, the branch instructions, have one of the two following formats.

b	target	Branch to target address obtained by adding the address of the current instruction with the 24-bit offset specified in the instruction.
bc	cond, target	Branch conditionally to target obtained by adding the address of the current instruction with the 14-bit offset specified in the instruction. The condition for branching is determined by condition register bit BI and field BO in the instruction.

A useful variant of the branch instruction is the branch-and-link instruction. When the LK bit in the branch instruction is set, it indicates that a side effect of executing either the b or the bc instruction is to place the address of the immediately following instruction into LR, the link register. This is a useful way of saving the return address while branching to subroutines.

bl	target	Branch to target and save return address in link register.
bcl	target	Conditionally branch and link to target.

The branch processor includes instructions that allow various logical and transfer operations on individual bits or fields of the CR instruction. Another important instruction in the branch processor is the system-call instruction.

sc	Branch to a special location determined by the contents of the MSR, and save state information in SRR0 and SRR1. The processor enters system mode after executing this instruction.

The fixed-point processor executes instructions that involve use of the general-purpose registers, GPR0—31. Load and store instructions use the general-purpose registers to specify the address of the memory location involved and hence also involve the fixed-point processor. Here are examples of load and store instructions.

ld	rt, d(ra)	Load register rt with the contents of memory addressed by (ra) + d. The d field is treated as a signed quantity.
ldx	rt, ra, rb	Load register rt with the contents of memory addressed by (ra) + (rb).
lwz	rt, d(ra)	Load lower 32 bits of register rt from contents of memory addressed by (ra) + d; zero out the upper 32 bits of register rt.
lwzx	rt, ra, rb	Load lower 32 bits of register rt from contents of memory addressed by (ra) + (rb); zero out the upper 32 bits of register rt.
stw	rt, d(ra)	Store the 32 low-order bits of register rt in the memory location addressed by (ra) + d. The d field is treated as a signed quantity.
stwx	rt, ra, rb	Store the 32 low-order bits of register rt in the memory location addressed by (ra) + (rb).

The remaining fixed-point instructions are largely similar to those found in most RISC ISAs. The arithmetic and logical instructions operate on operands, both of which may be in general-purpose registers, or one of which may be a constant defined in the instruction. Instructions are nondestructive—the result register may be different from both operand registers. Examples follow.

add	rt, ra, rb	Add the contents of ra and rb and put the result in rt.
subf.	rt, ra, rb	Subtract the contents of ra from rb and put the result in rt. The dot in the mnemonic indicates that as a side effect of the instruction, bits in field 0 (first four bits) of the condition register must be set according to whether the result is negative, zero, or overflowed. This is called a *record-form instruction*, because it sets the record bit in the instruction.

Because branches are taken conditionally on the contents of the condition register, the compiler must either use a record form instruction or set any of the fields in the CR explicitly using a compare instruction. Here's an example.

cmpi	3, ra, rb	Compare the contents of registers ra and rb, treating them as unsigned integers, and set field 3 (bits 12–15) of the condition register according to whether ra is less than rb, rb is less than ra, or both are identical.

The PowerPC also has a rich set of rotate and shift instructions. These instructions optionally perform a mask operation during the shift process, which allows a wide variety of common operations in compilers and interpreters to be performed in a single instruction. An example is the following instruction.

rlwimi	ra, rs, sh, mb, me	Rotate the contents of rs left by the amount specified in sh, and merge the masked part into register ra. The part of ra that is changed is determined by a mask of 1's extending from the mb position to the me position.
rlwinm	ra, rs, sh, mb, me	Rotate the contents of rs left by the amount specified in sh, and use mask to extract a part of the rotated value to be transferred to ra.

A.8.2 The Intel IA-32 ISA

The IA-32 ISA (**Intel 1999**), informally known as the x86, descended from the Intel 8086 through the 286, 386, 486, and Pentium series. The 8086 started as a 16-bit microcontroller chip, primarily for use in embedded systems. An 8086 derivative, the 8088, was selected by IBM for the first PCs in 1981, and soon the x86 ISA evolved to a 32-bit general-purpose architecture. Recently, it has been further extended to 64 bits. Currently the IA-32 is probably the most widely used general-purpose architecture, with applications ranging from laptops to desktops to servers.

The IA-32 ISA is an example of a CISC ISA. The instruction set is more "cluttered" than "complex," there having been several instructions sets that are fundamentally more complex. The clutter comes from its evolutionary history and extensions, coupled with the need to maintain backward compatibility at each step. In this book we feature the 32-bit IA-32 architecture; the earlier 16-bit version of the ISA and the newer 64-bit version are not used in this book. Just as we did with the PowerPC description, we will begin with a descrip

we did with the PowerPC description, we will begin with a description of the state (registers and memory architecture) and then summarize the actual instruction set.

Registers

The IA-32 register set is illustrated in Figure A.24. There are eight general-purpose registers, six segment registers, and a floating-point stack with eight elements. The general-purpose registers are 32 bits wide, the segment registers are 16 bits wide, and the elements of the floating-point stack are 80 bits wide. For executing 16-bit instructions, the low-order 16 bits of the general-purpose registers are used. Some of the general-purpose registers have the special uses for certain instructions, some of which are listed below.

eax—Accumulator for operands and results

ebx—Pointer to data held in the DS segment

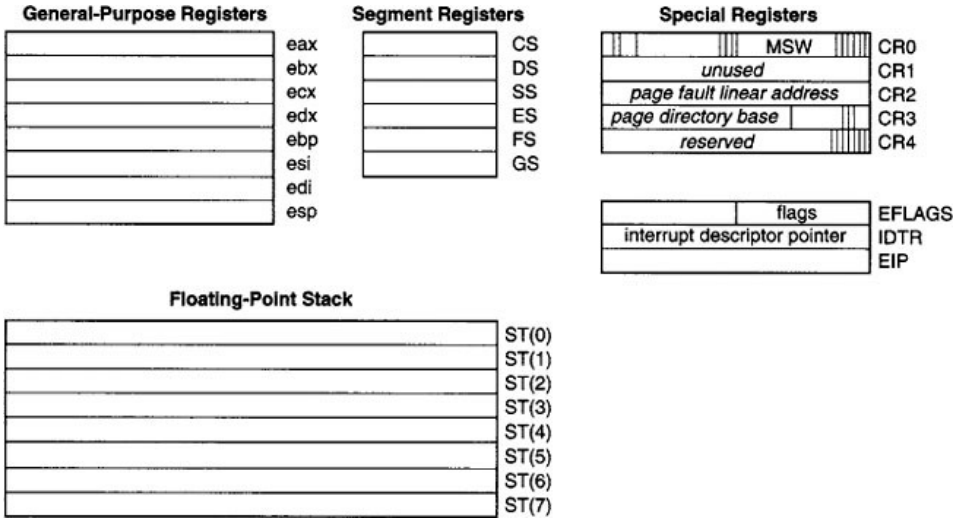


Figure A.24 IA-32 Register Set. *Registers shown are for 32-bit protected-mode operation.*

ecx—Counter for string operations and loops

edx—I/O pointer

esi—Pointer to data held in the DS segment; source pointer for string