Computer architecture

Lecture 1: Computer architecture

Computer architecture

There are two parts to computer architecture:

1. Instruction set architecture(ISA).
2. Hardware system architecture (HSA).

ISA includes the specification that determine how machine language programmers will interact with the computer.
A computer is generally viewed in terms of its ISA, which determines the computational characteristics of the computer.

HAS deals with the computer's major hardware subsystems, including its central processing unit(CPU), I/O system and storage system.

Classification of computer architectures:

1. **Von Neumann architecture**

also known as the **von Neumann model** is a computer architecture based on a 1945 description by the mathematician and physicist John von Neumann and others.

Von Neumann machine meet the following criteria:

1. It has three basic hardware subsystems
    - CPU
    - The main memory system
    - I/O system

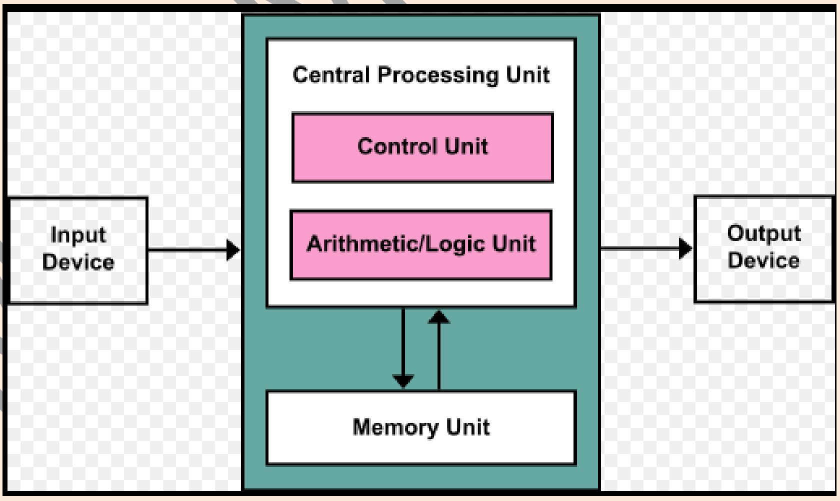
2. It is a stored program computer.

3. It carries out instruction sequentially

    The CPU executes one program at a time.

The word has evolved to mean any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the von Neumann often limits the performance of the system.[3]

The design of a von Neumann architecture machine is simpler than a Harvard architecture machine—which is also a stored-program system but has one dedicated set of address and data buses for reading and writing to memory, and another set of address and data buses to fetch instructions.



2. **Non-Von Newman Machine:**

The classification includes the following categories:
1.Single Instruction stream, Single data stream (SISD)
2. Single Instruction stream, Multiple data stream (SIMD)
3. Multiple Instruction stream, Single data stream (MISD)
4. Multiple Instruction stream, Multiple data stream (MIMD)

MEMORY LOCATIONS AND OPERATIONS

The memory can be modeled as an array of millions of adjacent cells, each capable of storing a binary digit (bit), having value of 1 or 0. These cells are organized in the form of groups of fixed number of cells or bits. 8 bits is called a byte. size of a word ranges from 16 to 64 bits. It is, however, customary to express the size of the memory in terms of bytes. For example, the size of a typical memory of a personal computer is 256 Mbytes. In order to be able to move a word in and out of the memory, a distinct address has to be assigned to each word. This address will be used to determine the location in the memory in which a given word is to be stored. This is called a memory write operation. Similarly, the address will be used to determine the memory location from which a word is to be retrieved from the memory. This is called a memory read operation.

INSTRUCTION SET ARCHITECTURE AND DESIGN

Three basic steps are needed in order for the CPU to perform a write operation into a specified memory location:

1. The word to be stored into the memory location is first loaded by the CPU into a specified register, called the memory data register (MDR).
2. The address of the location into which the word is to be stored is loaded by the CPU into a specified register, called the memory address register (MAR).
3. A signal, called write, is issued by the CPU indicating that the word stored in the MDR is to be stored in the memory location whose address in loaded in the MAR.

Figure 2.2 illustrates the operation of writing the word given by 7E (in hex) into the memory location whose address is 2005. Part a of the figure shows the status of the registers and memory locations involved in the write operation before the execution of the operation. Part b of the figure shows the status after the execution of the operation.
It is worth mentioning that the MDR and the MAR are registers used exclusively by the CPU and are not accessible to the programmer.
Similar to the write operation, three basic steps are needed in order to perform a memory read operation:

1. The address of the location from which the word is to be read is loaded into the MAR.
2. A signal, called read, is issued by the CPU indicating that the word whose address is in the MAR is to be read into the MDR.
3. After some time, corresponding to the memory delay in reading the specified word, the required word will be loaded by the memory into the MDR ready for use by the CPU.

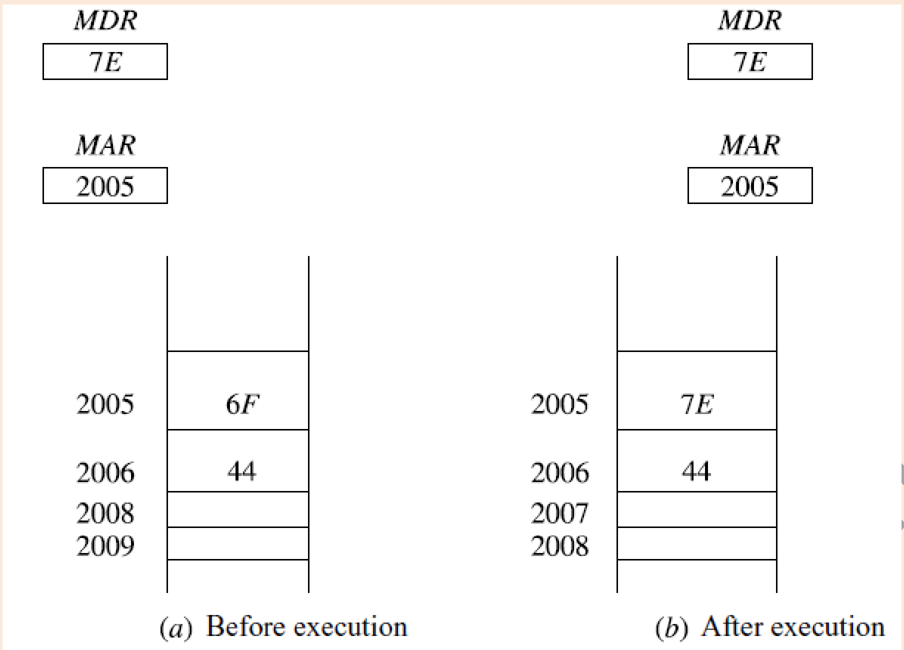


Figure 2.2 Illustration of the memory write operation

Figure 2.3 illustrates the operation of reading the word stored in the memory location whose address is 2010. Part a of the figure shows the status of the registers and memory locations involved in the read operation before the execution of the operation. Part b of the figure shows the status after the read operation.

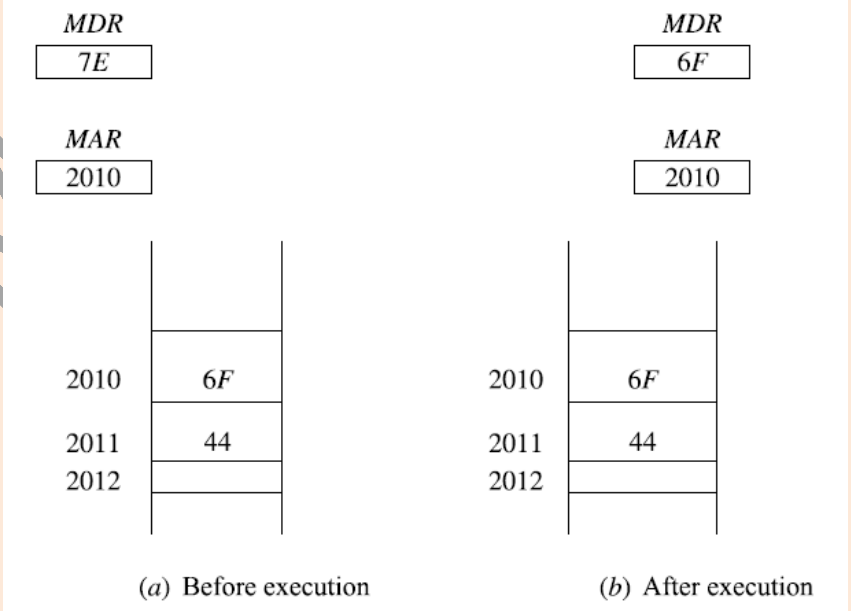


Figure 2.3 Illustration of the memory read operation

ADDRESSING MODES

Information involved in any operation performed by the CPU needs to be addressed. In computer terminology, such information is called the operand. Therefore, any instruction issued by the processor must carry at least two types of information.

These are the operation to be performed, encoded in what is called the op-code field, and the address information of the operand on which the operation is to be performed, encoded in what is called the address field.

Instructions can be classified based on the number of operands as: three-address, two-address, one-and-half-address, one-address, and zero-address. We explain these classes together with simple examples in the following paragraphs.

It should be noted that in presenting these examples, we would use the convention operation, source, destination to express any instruction. In that convention, operation represents the operation to be performed, for example, add, subtract, write, or read.

The source field represents the source operand(s). The source operand can be a constant, a value stored in a register, or a value stored in the memory. The destination field represents the place where the result of the operation is to be stored, for example, a register or a memory location.

for example a three-address instruction the instruction

    ADD A,B,C

The instruction adds the contents of memory location A to the contents of memory location B and stores the result in memory location C.

A two-address instruction example:

    ADD A,B

In this case, the contents of memory location A are added to the contents of memory location B and the result put in contents of memory location B.

A one-address instruction takes the form

    ADD B

the instruction adds the content of the accumulator reg. to the content of memory location B and stores the result back into the accumulator .

Between the two- and the one-address instruction, there can be a one-and-half address instruction. Consider, for example, the instruction

    ADD B,R1

In this case, the instruction adds the contents of register R1 to the contents of memory location B and stores the result in register R1.

It is interesting to indicate that there exist zero-address instructions. These are the instructions that use stack operation.

These are the push and the pop operations. Figure 2.4 illustrates these two operations. As can be seen, a specific register, called the stack pointer (SP), is used to indicate the stack location that can be addressed.

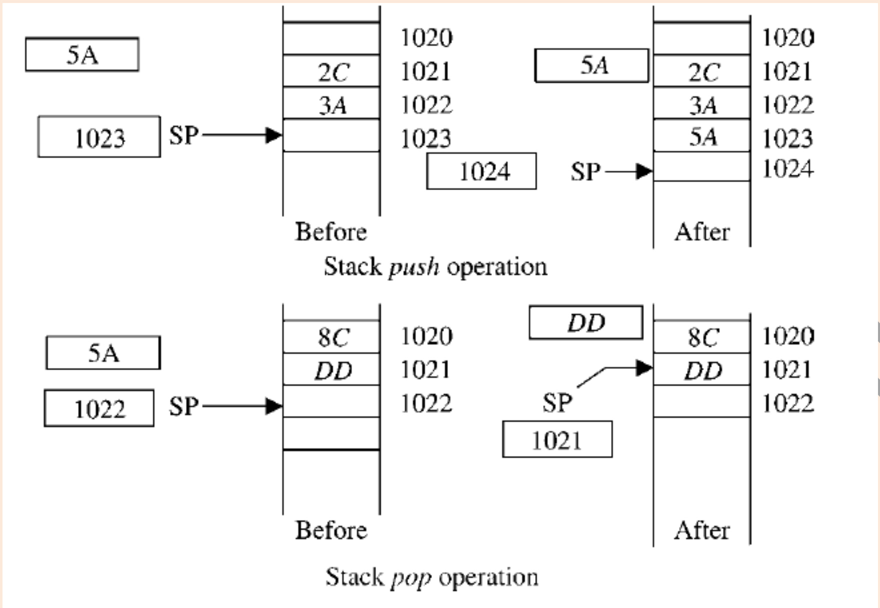


Figure 2.4 The stack push and pop operations

Processing Unit Design

we focus our attention on the main component of any computer system, the central processing unit (CPU). The primary function of the CPU is to execute a set of instructions stored in the computer's memory. A simple CPU consists of a set of registers, an arithmetic logic unit (ALU), and a control unit (CU). In what follows, the reader will be introduced to the organization and main operations of the CPU.

1. CPU BASICS

A typical CPU has three major components: (1) register set, (2) arithmetic logic unit (ALU), and (3) control unit (CU). The register set differs from one computer architecture to another. It is usually a combination of general-purpose and special purpose registers. General-purpose registers are used for any purpose, hence the name general purpose. Special-purpose registers have specific functions within the CPU. For example, the program counter (PC) is a special-purpose register that is used to hold the address of the instruction to be executed next. Another example of special-purpose registers is the instruction register (IR), which is used to hold the instruction that is currently executed. The ALU provides the circuitry needed to perform the arithmetic, logical and shift operations demanded of the instruction set. we have covered a number of arithmetic operations and circuits used to support computation in an ALU. The control unit is the entity responsible for fetching the instruction to be executed from the main memory and decoding and then executing it. Figure 5.1 shows the main components of the CPU and its interactions with the memory system and the input/ output devices.

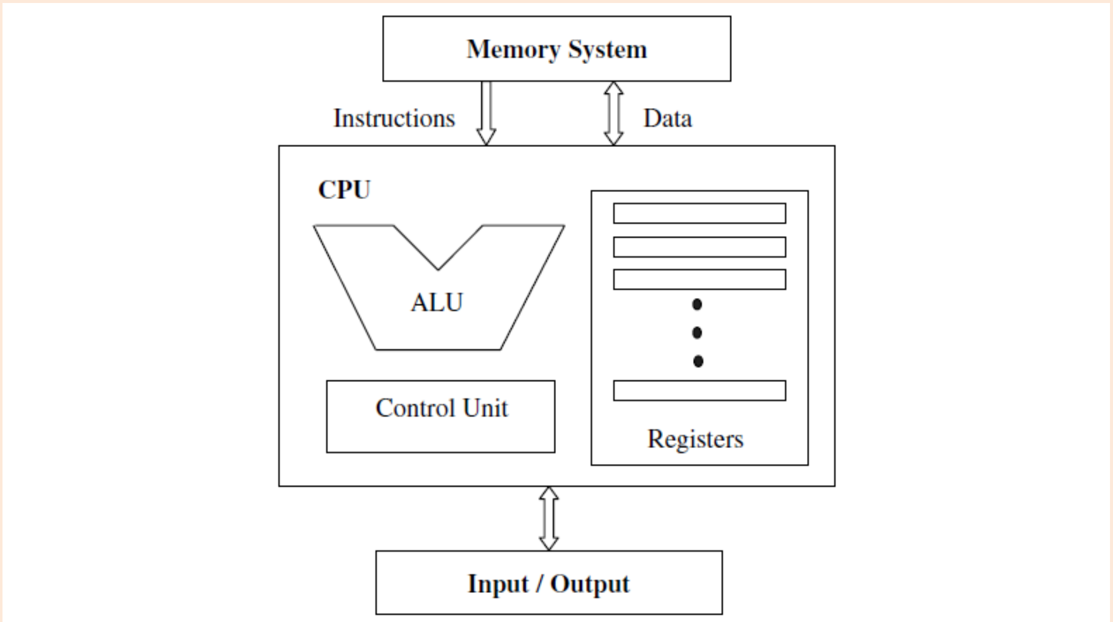


Figure 5.1 Central processing unit main components and interactions with the memory and I/O

The CPU fetches instructions from memory, reads and writes data from and to memory, and transfers data from and to input/output devices. A typical and simple execution cycle can be summarized as follows:

1. The next instruction to be executed, whose address is obtained from the PC, is fetched from the memory and stored in the IR.
2. The instruction is decoded.
3. Operands are fetched from the memory and stored in CPU registers, if needed.
4. The instruction is executed.
5. Results are transferred from CPU registers to the memory, if needed.

The execution cycle is repeated as long as there are more instructions to execute. A check for pending interrupts is usually included in the cycle. Examples of interrupts include I/O device request, arithmetic overflow, or a page fault.
When an interrupt request is encountered, a transfer to an interrupt handling routine takes place. Interrupt handling routines are programs

that are invoked to collect the state of the currently executing program, correct the cause of the interrupt, and restore the state of the program.
The actions of the CPU during an execution cycle are defined by micro-orders issued by the control unit. These micro-orders are individual control signals sent over dedicated control lines. For example, let us assume that we want to execute an instruction that moves the contents of register X to register Y. Let us also assume that both registers are connected to the data bus, D. The control unit will issue a control signal to tell register X to place its contents on the data bus D. After some delay, another control signal will be sent to tell register Y to read from data bus D. The activation of the control signals is determined using either hardwired control or microprogramming.

2. REGISTER SET

Registers are essentially extremely fast memory locations within the CPU that are used to create and store the results of CPU operations and other calculations. Different computers have different register sets. They differ in the number of registers, register types, and the length of each register. They also differ in the usage of each register. General-purpose registers can be used for multiple purposes and assigned to a variety of functions by the programmer. Special-purpose registers are restricted to only specific functions. In some cases, some registers are used only to hold data and cannot be used in the calculations of operand addresses. The length of a data register must be long enough to hold values of most data types. Some machines allow two contiguous registers to hold double-length values. Address registers may be dedicated to a particular addressing mode or may be used as address general purpose. Address registers must be long enough to hold the largest address. The number of registers in a particular architecture affects the instruction set design.
A very small number of registers may result in an increase in memory references. Another type of registers is used to hold processor status bits, or flags. These bits are set by the CPU as the result of the execution of an operation. The status bits can be tested at a later time as part of another operation.

Memory Access Registers

Two registers are essential in memory write and read operations: the memory data register (MDR) and memory address register (MAR). The MDR and MAR are used exclusively by the CPU and are not directly accessible to programmers.
In order to perform a write operation into a specified memory location, the MDR and MAR are used as follows:

1. The word to be stored into the memory location is first loaded by the CPU into MDR.
2. The address of the location into which the word is to be stored is loaded by the CPU into a MAR.
3. A write signal is issued by the CPU.

Similarly, to perform a memory read operation, the MDR and MAR are used as follows:
1. The address of the location from which the word is to be read is loaded into the MAR.
2. A read signal is issued by the CPU.
3. The required word will be loaded by the memory into the MDR ready for use by the CPU.

Lecture 2: DIRECT MEMORY ACCESS (DMA)

The main idea of direct memory access (DMA) is to enable peripheral devices to cut out the “middle man” role of the CPU in data transfer. It allows peripheral devices to transfer data directly from and to memory without the intervention of the CPU. Having peripheral devices access memory directly would allow the CPU to do other work, which would lead to improved performance, especially in the cases of large transfers.
The DMA controller is a piece of hardware that controls one or more peripheral devices. It allows devices to transfer data to or from the system’s memory without the help of the processor. In a typical DMA transfer, some event notifies the DMA controller that data needs to be transferred to or from memory. Both the DMA and CPU use memory bus and only one or the other can use the memory at the same time. The DMA controller then sends a request to the CPU asking its permission to use the bus. The CPU returns an acknowledgment to the DMA controller granting it bus access. The DMA can now take control of the bus to independently conduct memory transfer. When the transfer is complete the DMA relinquishes its control of the bus to the CPU. Processors that support DMA provide one or more input signals that the bus requester can assert to gain control of the bus and one or more output signals that the CPU asserts to indicate it has relinquished the bus. Figure 8.10
shows how the DMA controller shares the CPU’s memory bus.

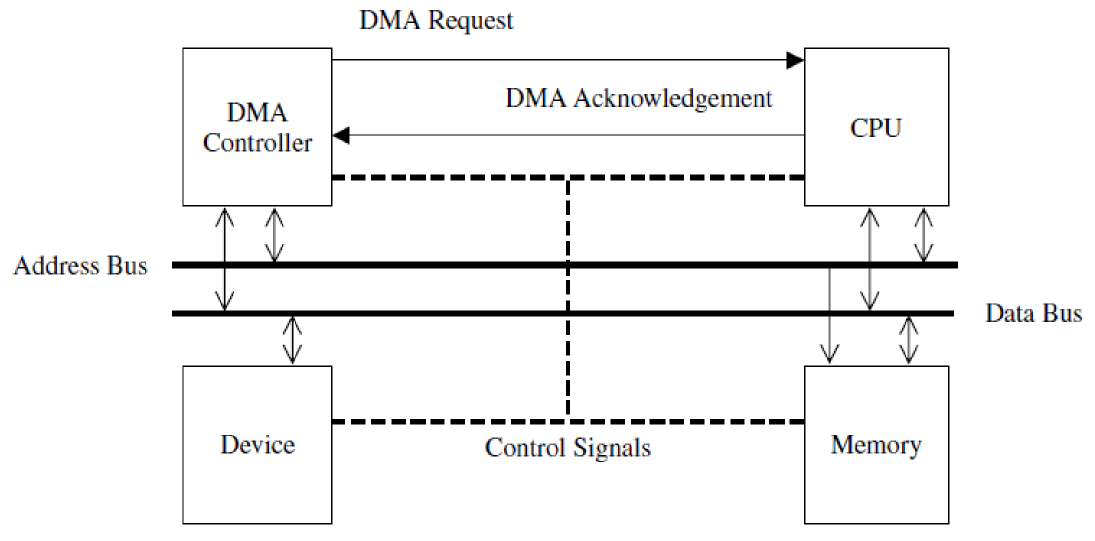


Figure 8.10 DMA controller shares the CPU’s memory bus

Direct memory access controllers require initialization by the CPU. Typical setup parameters include the address of the source area, the address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete. A DMA controller has an address register, a word count register, and a control register. The address register contains an address that specifies the memory location of the data to be transferred. It is typically possible to have the DMA controller automatically increment the address register after each word transfer, so that the next transfer will be from the next memory location. The word count register holds the number of words to be transferred.

The word count is decremented by one after each word transfer. The control register specifies the transfer mode.

Direct memory access data transfer can be performed in burst mode or single cycle mode. In burst mode, the DMA controller keeps control of the bus until all the data has been transferred to (from) memory from (to) the peripheral device.

This mode of transfer is needed for fast devices where data transfer cannot be stopped until the entire transfer is done. In single-cycle mode (cycle stealing), the DMA controller relinquishes the bus after each transfer of one data word. This minimizes the amount of time that the DMA controller keeps the CPU from controlling the bus, but it requires that the bus request/acknowledge sequence be performed for every single transfer. This overhead can result in a degradation of the performance.

The single-cycle mode is preferred if the system cannot tolerate more than a few cycles of added interrupt latency or if the peripheral devices can buffer very large amounts of data, causing the DMA controller to tie up the bus for an excessive amount of time.

The following steps summarize the DMA operations:

1. DMA controller initiates data transfer.
2. Data is moved (increasing the address in memory, and reducing the count of words to be moved).
3. When word count reaches zero, the DMA informs the CPU of the termination by means of an interrupt.
4. The CPU regains access to the memory bus.

A DMA controller may have multiple channels. Each channel has associated with it an address register and a count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. While the transfer is taking place, the CPU is free to do other things. When the transfer is complete, the CPU is interrupted.

Direct memory access channels cannot be shared between device drivers. A device driver must be able to determine which DMA channel to use. Some devices have a fixed DMA channel, while others are more flexible, where the device driver can simply pick a free DMA channel to use.

Linux tracks the usage of the DMA channels using a vector of dma_chan data structures (one per DMA channel). The dma_chan data structure contains just two fields, a pointer to a string describing the owner of the DMA channel and a flag indicating if the DMA channel is allocated or not.

Memory System Architecture:

The memory of computer system can be divided into three main groups:

1. Internal processer memory
   This comprises a small set of high-speed registers used as a working memory for temporary storage of instructions and data.
2. Main memory
   This is a relatively large fast memory used for program and data storage during execution.
3. Secondary memory
   This is generally much larger in capacity but also much slower than main memory. It is use for storing system program and large data files. This type of memory has the following groups:
   - Magnetic tape
   - Floppy disk
   - Hard disk
   - CD-ROM
   - Flash

Read Only Memory (ROM)

ROM type:

1. Programmable ROM(PROM)
2. Erasable PROM(EPROM)
3. Electrically EPROM(EEPROM)

Random Access Memory (RAM)

A memory unit, in which any particular word can be accessed independently, is known as a *random-access memory* (RAM). In a random-access memory the time required for accessing a word is the same for all words. CPU can read and write. RAM is a volatile which main that it loses their information content wherever the power is turned off, thus it used as temporary storage.

Semiconductor RAM

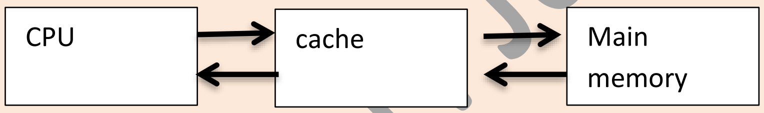
There are two types of memories: ***static*** **(SRAM) and** ***dynamic*** **(DRAM)**.

SRAM
Static memories hold the stored data for long time as the power is on, or until new data are stored in them. This memory used mostly in CPU register and other high speed device, although some computers use them for caches and main memory. It is currently the faster and most expensive of the semiconductor memory.

DRAM
There devices are made with cells that store data as charge of capacitors together with a single transistor, unfortunately the capacitor slowly lose their charges due to leakage so periodic charge, refreshing is necessary to maintain data storage.
The implementation of the refreshing circuit may appear as a disadvantage for DRAM design; but, in general, a cell of a static memory requires more transistors than a cell of a dynamic memory. So the refreshing circuit is considered an acceptable and unavoidable cost of DRAM.

Lecture 3:Cache Memory

One problem designers face in constructing a processor is the bottleneck associated with memory speeds. Because fetches from main memory require considerably more time when compared to the overall speeds in the processor, designers spend a lot of time and effort making memory speeds as fast as possible.
One way to make the memory appear faster is to reduce the number of times main memory has to be accessed. If a small amount of fast memory is installed and at any point in time part of a program is loaded in this fast memory, then, due to the property of locality of reference, the number of references to the main memory will be reduced. Such a fast memory unit, used temporarily to store a portion of the data and instructions (from the main memory) for immediate use, is known as *cache memory*.
Cache memory is a small fast memory placed between a processor and main memory.



Because cache memory is expensive, a computer system can have only a limited amount of it installed. Therefore, in a computer system there is a relatively large and slower main memory coupled together with a smaller, faster cache memory. The cache contains copies of some blocks of the main memory. Therefore, when the CPU requests a word (if the word is in the fast cache), there will be no need to go to the larger, slower main memory.
The performance of a system can be greatly improved if the cache is placed on the same chip as the processor. In this case, the outputs of the cache can be connected to the ALU and registers through short wires, significantly reducing access time.
Cache operation. When the CPU generates an address for memory reference, the generated address is first sent to the cache. Based on the contents of the cache, a *hit* or a *miss* occurs. A hit occurs when the requested word is already present in the cache. In contrast, a miss happens when the requested word is not in the cache.
Two types of operations can be requested by the CPU: a read request and a write request. When the CPU generates a read request for a word in memory, the generated request is first sent to the cache to check if the word currently resides in the cache. If the word is not found in the cache

(i.e., a read miss), the requested word is supplied by the main memory. If the cache is full, a predetermined replacement policy is used to swap out a word from the cache in order to accommodate the new word. If the requested word is found in the cache (i.e., a read hit), the word is supplied by the cache. Thus no fetch from main memory is required. This speeds up the system considerably.
When the CPU generates a write request for a word in memory, the generated request is first sent to the cache to check if the word currently resides in the cache. If the word is not found in the cache (i.e., a write miss), a copy of the word is brought from the memory into the cache. Next, a write operation is performed. Also, a write operation is performed when the word is found in the cache (i.e., a write hit).

To perform a write operation, there are two main approaches that the hardware may employ: *write through*, and *write back*. In the write-through method, the word is modified in both the cache and the main memory. The advantage of the write-through method is that the main memory always has consistent data with the cache.
However, it has the disadvantage of slowing down the CPU because all write operations require subsequent accesses to the main memory, which are time consuming.
In the write-back method, every word in the cache has a bit associated with it, called a *dirty bit* (also called an *inconsistent bit*), which tells if it has been changed while in the cache. In this case, the word in the cache may be modified during the write operation, and the dirty bit is set. All changes to a word are performed in the cache. When it is time for a word to be swapped out of the cache, it checks to see if the word's dirty bit is set: if it is, it is written back to the main memory in its updated form.
The advantage of the write-back method is that as long as a word stays in the cache it may be modified several times and, for the CPU, it does not matter if the word in the main memory has not been updated.
The disadvantage of the write-back method is that, although only one extra bit has to be associated with each word, it makes the design of the system slightly more complex.
Basic cache organization. The basic motivation behind using cache memories in computer systems is their speed. Most of the time, the presence of the cache is not apparent to the user. Since it is desirable that very little time be wasted when searching for words in a cache, usually the cache is managed through hardware-based algorithms. The translation of the memory address, specified by the CPU, into the possible location of the corresponding word in the cache is referred to as a *mapping* process. Based on the mapping process used, cache organization can be classified into three types:

1. Associative-mapping cache
2. Direct-mapping cache
3. Set-associative mapping cache

The following sections explain these cache organizations. To illustrate these three different cache Organizations, the memory organization shown in Figure 2.39c is used. In this figure, the CPU communicates with the cache as well as the main memory. The main memory stores 64K words (16-bit address) of 16 bits each. The cache is capable of storing 256 of these words at any given time. Also, in the following discussion it is assumed that the CPU generates a read request and not a write request. (The write request would be handled in a similar way.)

Associative Mapping. In an associative-mapping cache (also referred to as fully associative cache), both the address and the contents are stored as one word in the cache. As a result, a memory word is allowed to be stored at any location in the cache, making it the most flexible cache organization. Figure 2.47 shows the organization of an associative-mapping cache for a system with 16-bit addressing and 16-bit data. Note that the words are stored at arbitrary locations regardless of their absolute addresses in the main memory.
The organization of an associative-mapping cache can be viewed as a combination of an associative memory and a RAM, as shown in Figure 2.47 (all numbers are in hexadecimal). Since each associative memory cell is many times more expensive than a RAM cell, only the addresses of the words are stored in the associative part, while the data can be stored in the RAM part of the cache because only the address is used for associative search. This will not increase the access time of the cache significantly, but will result in a significant drop in cost. In this organization, when the CPU generates an address for memory reference, it is passed into the argument register and is compared, in parallel, with the address fields of all words currently stored in the cache for a matching address. Once the location has been determined, the corresponding data can be accessed from the RAM.

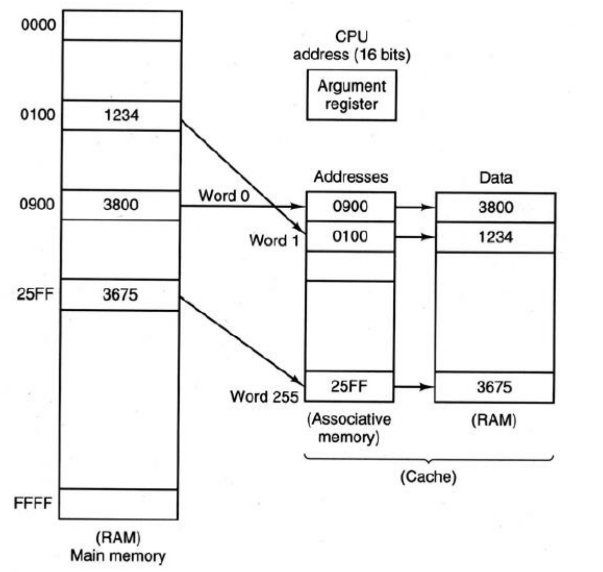


Figure 2.47 Associative-mapping cache (all numbers are in hexadecimal).

The major disadvantage of this method is its need for a large associative memory, which is very expensive and increases the access time of the cache.

Lecture 4: *Direct Mapping*.

In a direct-mapping cache, the requested memory address is divided into two parts, an *index* field, which refers to the lower part of the address, and a *tag* field, which refers to the upper part. The index is used as an address to a location in the cache where the data are located. At this index, a tag and a data value are stored in the cache. If the tag of the requested memory address matches the tag of cache, the data value is sent to the CPU. Otherwise, the main memory is accessed, and the corresponding data value is fetched and sent to the CPU. The data value, along with the tag part of its address, also replaces any word currently occupying the corresponding index location in the cache.

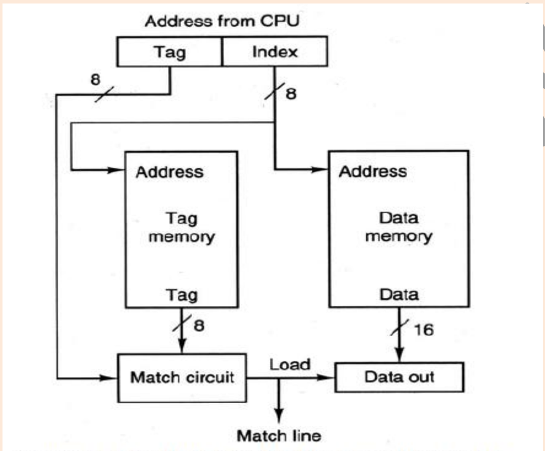


Figure 2.48 Architecture of a direct-mapping cache.

Figure 2.48 represents an architecture for the direct-mapping cache. The design consists of three main components: data memory, tag memory, and match circuit. The data memory holds the cached data. The tag memory holds the tag associated with each cached datum and has an entry for each word of the data memory. The match circuit sets the match line to 1, indicating that the referenced word is in the cache.

An example illustrating the direct-mapping cache operation is shown in Figure 2.49 (all numbers are in hexadecimal). In this example, the memory address consists of 16 bits and the cache has 256 words. The eight least significant bits of the address constitute the index field, and the remaining eight bits constitute the tag field. The 8 index bits determine the address of a word in the tag and data memories. Each word in the tag memory has 8 bits, and each word in the data memory has 16 bits. Initially, the content of address 0900 is stored in the cache. Now, if the CPU wants to read the contents of address 0100, the index (00) matches, but the tag (01) is now different. So the content of main memory is accessed, and the data word 1234 is transferred to the CPU. The tag memory and the data memory words at index address 00 are then replaced with 01 and 1234, respectively.

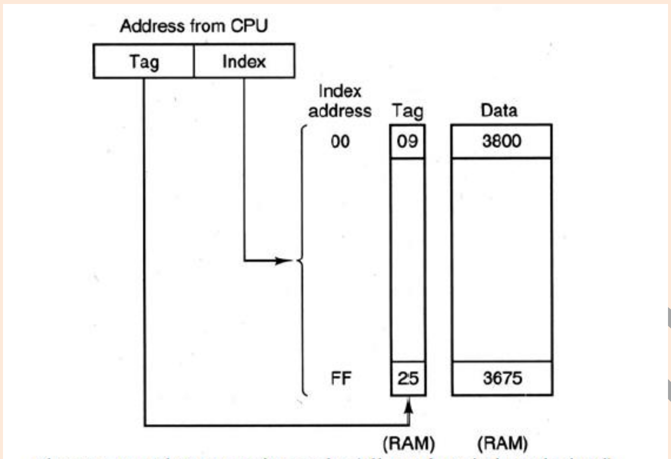


Figure 2.49 Direct-mapping cache (all numbers in hexadecimal).

The advantage of direct mapping over associative mapping is that it requires less overhead in terms of the number of bits per word in the cache. The major disadvantage is that the performance can drop considerably if two or more words having the same index but different tags are accessed frequently. For example, memory addresses 0100 and 0200 both have to be put in the cache at position 00, so a great deal of time is spent swapping them back and forth. This slows the system down, thus defeating the purpose of the cache in the first place. However, considering the property of locality of reference, the probability of having two words with the same index is low. Such words are located 2*k* bits apart in the main memory, where *k* denotes the number of bits in the index field. In our example, such a situation will only occur if the CPU requests reference to words that are 256 (28) words apart. To further reduce the effects of such situations often an expanded version of the direct-mapping cache, called a *set-associative cache*, is used.
The following section describes the basic structure of a set-associative mapped cache.

Set-Associative Mapping. The set-associative mapping cache organization (also referred to as set associative cache) is an extension of the direct-mapping cache. It solves the problem of direct mapping by providing storage for more than one data value with the same index. For example, a set-associative cache with *m* memory blocks, called *m*-way set associative, can store *m* data values having the same index, along with

their tags. Figure 2.50 represents an architecture for a set-associative mapping cache with *m* memory blocks. Each memory block has the same structure as a direct-mapping cache. To determine that a referenced word is in the cache, its tag is compared with the tag of cached data in all memory blocks in parallel. A match in any of the memory blocks will enable (set to 1) the signal match line to indicate that the data are in the cache. If a match occurs, the corresponding data value is passed on to the CPU.
Otherwise, the data value is brought in from the main memory and sent to the CPU. The data value, along with its tag, is then stored in one of the memory blocks.

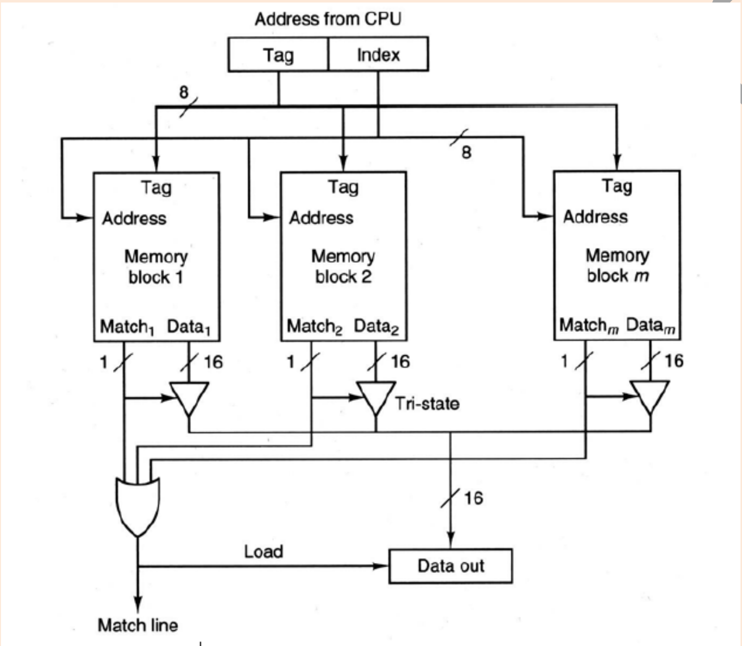


Figure 2.50 Architecture of an *m*-way set-associative mapping cache.

An example illustrating the set-associative mapping cache operation is shown in Figure 2.51 (all numbers are in hexadecimal). This figure represents a two-way set-associative mapping cache. The content of address 0900 is stored in the cache under index 00 and tag 09. If the CPU wants to access address 0100, the index (00) matches, but the tag is now different. Therefore, the content of main memory is accessed, and the data value 1234 is transferred to the CPU. This data with its tag (01) is stored in the second memory block of the cache. When there is no space for a particular index in the cache, one of the two data values stored under that index will be replaced according to some predetermined replacement policy (discussed next).

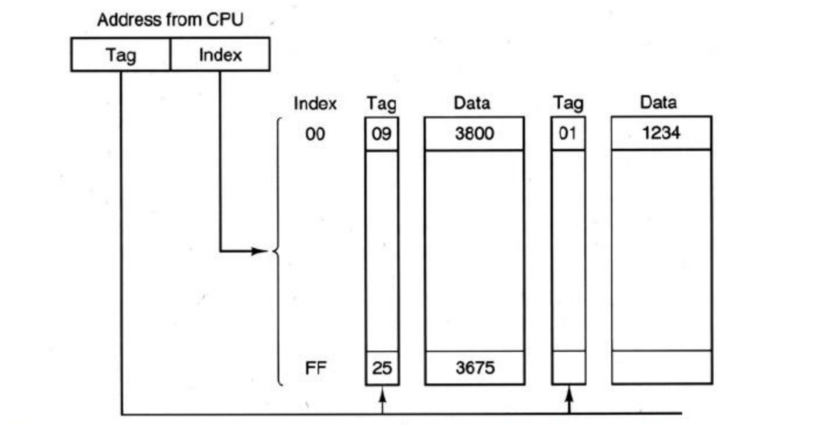


Figure 2.51 Two-way set-associative mapping cache (all numbers are in hexadecimal).

Lecture 5: Replacement strategies.

Sooner or later, cache will become full. When this occurs, a mechanism should be there to replace a word with the newly accessed data from memory. In general, there are three main strategies for determining which word should be swapped out from the cache; they are called *random, least frequently used*, and *least recently used* replacement policies.

Random Replacement. This method picks a word at random and replaces that word with the newly accessed data. This method is easy to implement in hardware, and it is faster than most other algorithms.
The disadvantage is that the words most likely to be used again have as much of a chance of being swapped out as a word that is likely not to be used again. This disadvantage diminishes as the cache size increases.

Least Frequently Used. This method replaces the data that are used the least. It assumes that data that are not referenced frequently are not needed as much. For each word, a counter is kept for the total number of times the word has been used since it was brought into the cache. The word with the lowest count is the word to be swapped out. The advantage of this method is that a frequently used word is more likely to remain in cache than a word that has not been used often. One disadvantage is that words that have recently been brought into the cache have a low count total, despite the fact that they are likely to be used again. Another disadvantage is that this method is more difficult to implement in terms of hardware and is thus more expensive.

Least Recently Used. This method has the best performance per cost compared with the other techniques and is often implemented in real-world systems. The idea behind this replacement method is that a word that has not been used for a long period of time has a lesser chance of being needed in the near future according to the property of temporal locality. Thus, this method retains words in the cache that are more likely to be used again. To do this, a mechanism is used to keep track of which words have been accessed most recently. The word that will be swapped out is the word that has not been used for the longest period of time. One way to implement such a mechanism is to assign a counter to each word in the cache. Each time the cache is accessed, each word's counter is incremented, and the word's counter that was accessed is reset to zero. In this manner, the word with the highest count is the one that was least recently used.

Virtual Memory

Another technique used to improve system performance is called *virtual memory*. As the name implies, virtual memory is the illusion of a much larger main memory size (logical view) than what actually exists (physical view). Prior to the advent of virtual memory, if a program's address space exceeded the actual available memory, the programmer was responsible for breaking up the program into smaller pieces called *overlays*. Each overlay then could fit in main memory. The basic process was to store all these overlays in secondary memory, such as on a disk, and to load individual overlays into main memory as they were needed.
This process required knowledge of where the overlays were to be stored on disk, knowledge of input/output operations involved with accessing the overlays, and keeping track of the entire overlay process. This was a very complex and tedious process that made the complexity of programming a computer even more difficult.
The concept of virtual memory was created to relieve the programmer of this burden and to let the computer manage this process. Virtual memory allows the user to write programs that grow beyond the bounds of physical memory and still execute properly. It also allows for multiprogramming, by which main memory is shared among many users on a dynamic basis. With multiprogramming, portions of several programs are placed in the main memory at the same time, and the processor switches its time back and forth among these programs. The processor executes one program for a brief period of time (called a *quantum* or *time-slice*) and then switches to another program; this process continues until each program is completed. When virtual memory is used, the addresses used by the programmer are seen by the system as *virtual addresses*, which are so called because they are mapped onto the addresses of physical memory and therefore do not access the same physical memory address from one execution of an instruction to the next. Virtual addresses, also called *logical addresses*, are generated by the processor during the compile time and are translated into physical addresses at run time. The two main methods for achieving a virtual memory environment are *paging* and *segmentation*. Each is explained next.

Paging. Paging is the technique of breaking a program (referred to in the following as process) into smaller blocks of identical size and storing these blocks in secondary storage in the form of *pages*. By taking advantage of the locality of reference, these pages can then be loaded into main memory, a few at a time, into blocks of the same size called *frames* and executed just as if the entire process were in memory.

For this method to work properly, each process must maintain a *page table* in main memory. Figure 2.53 shows how a paging scheme works. The base register, which each process has, points to the beginning of the process's page table. Page tables have an entry for each page that the process contains. These entries usually contain a *load* field of one bit, an *address* field, and an *access* field. The load field specifies whether the page has been brought into main memory. The address field specifies the frame number of the frame into which the page is loaded. The address of the page within main memory is evaluated by multiplying the frame number and the frame size. (Since frame size is usually a power of 2, shifting is often used for multiplying frame number by frame size.) If a page has not been loaded, the address of the page within secondary memory is held in this field. The access field specifies the type of operation that can be performed on a block. It determines whether a block is read only, read/write, or executable.

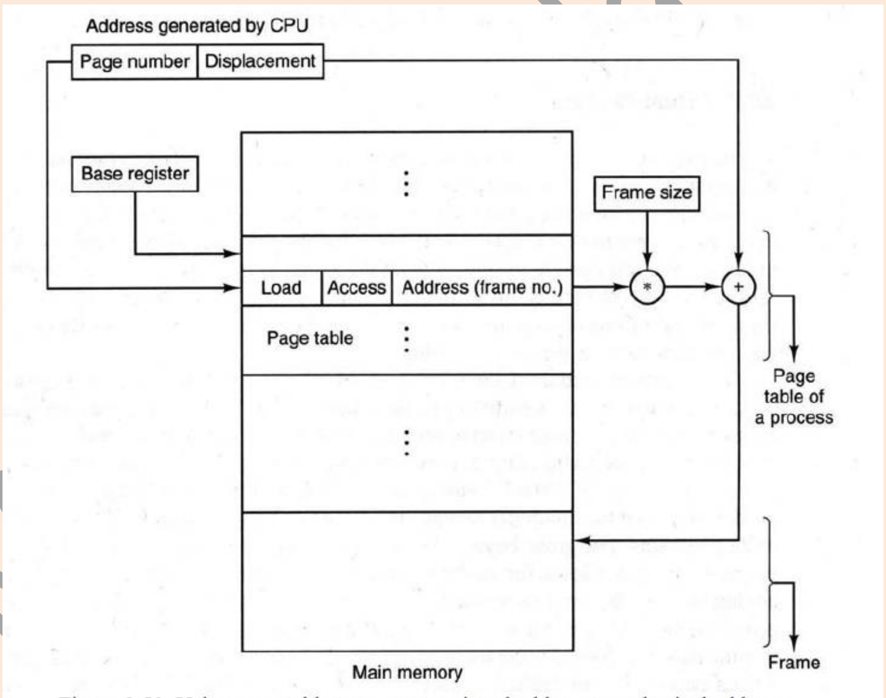
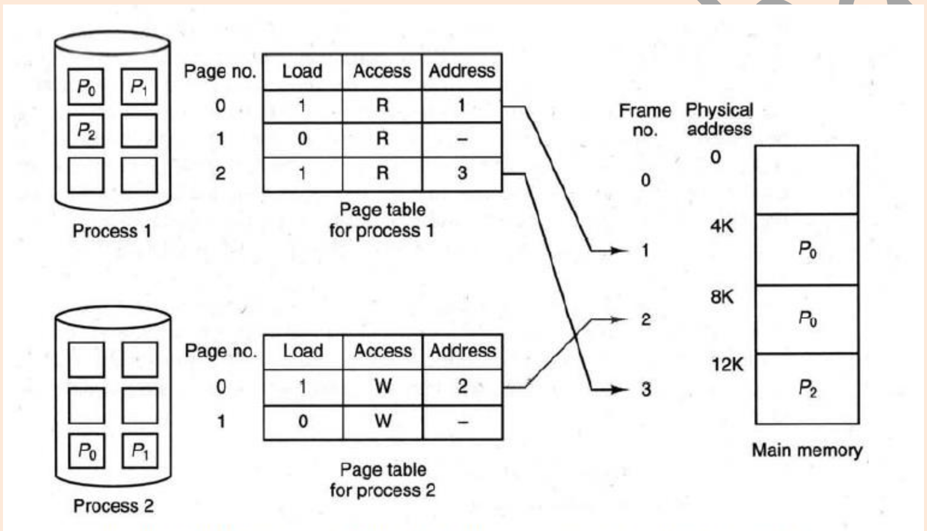


Figure 2.53 Using page table to convert a virtual address to a physical address.

When an access to a variable or an instruction that is not currently loaded into the memory is encountered, a *page fault* occurs, and the page that contains the necessary variable or instruction is brought into the memory. The page is stored in a free frame, if one exists. If a free frame does not exist, one of the process's own frames must be given up, and the new page will be stored in its place. Which frame is given up and whether the

old page is written back to secondary storage depend on which of several page replacement algorithms (discussed later in this section) is used.
As an example, Figure 2.54 shows the contents of page tables for two processes, process 1 and process 2. Process 1 consists of three pages, *P*0, *P*1, and *P*2, whereas process 2 has only two pages, *P*0 and *P*1. Assume that all the pages of process 1 have read access only and the pages of process 2 have read/write access; this is denoted by *R* and *W* in each page table. Because each frame has 4096 (4K) bytes, the physical address of the beginning of each frame is computed by the product of the frame number and 4096. Therefore, given that *P*0 and *P*2 of process 1 are loaded into frames 1 and 3, their beginning address in main memory will be 4K = (1 * 4096) and 12K = (3 * 4096), respectively.



Page no.	Load	Access	Address
0	1	R	1
1	0	R	–
2	1	R	3

Page no.	Load	Access	Address
0	1	W	2
1	0	W	–

Figure 2.54 Page tables for two different processes, Process 1 and Process 2.

The process of converting a virtual address to a physical address can be sped up by using a high-speed lookup table called a *translation lookaside buffer* (TLB). The page number of the virtual address is fed to the TLB where it is translated to a frame number of the physical address. The TLB can be implemented as an associative memory that has an entry for each of the most recently or likely referenced pages. Each entry contains a page number and other relevant information similar to the page table entry, essentially the frame number and access type. For a given page number, an entry of the TLB that matches (a hit) this page number is used to provide the corresponding frame number. If a match cannot be found in the TLB (a miss), the page table of the corresponding process will be located and used to produce the frame number.
The efficiency of the virtual memory system depends on minimizing the number of page faults. Because the access time of secondary memory is much higher than the access time of main memory, an excessive number

of page faults can slow the system dramatically. When a page fault occurs, a page in the main memory must be located and identified as one not needed at the present time so that it can be written back to the secondary memory. Then the requested page can be loaded into this newly freed frame of main memory.

Obviously, paging increases the processing time of a process substantially, because two disk accesses would be required along with the execution of a replacement algorithm. There is an alternative, however, which at times can reduce the number of the disk accesses to just one. This reduction is achieved by adding to the hardware an extra bit to each frame, called a *dirty* bit (also called an *inconsistent* bit). If some modification has taken place to a particular frame, the corresponding dirty bit is set to 1. If the dirty bit for frame *f* is 1, for example, and in order to create an available frame, *f* has been chosen as the frame to swap out, then two disk accesses would be required. If the dirty bit is 0 (meaning that there were no modifications on *f* since it was last loaded), there would be no need to write *f* back to disk. Because the original state of *f* is still on disk (remember that the frames in main memory contain copies of the pages in secondary memory) and no modifications have been made to *f* while in main memory, the page frame containing *f* can simply be overwritten by the newly requested page.

Most replacement algorithms consider the principle of locality when selecting a frame to replace. The principle of locality states that over a given amount of time the addresses generated will fall within a small portion of the virtual address space and that these generated addresses will change slowly with time. Two possible replacement algorithms are

1. First in, first out (FIFO)
2. Least recently used (LRU)

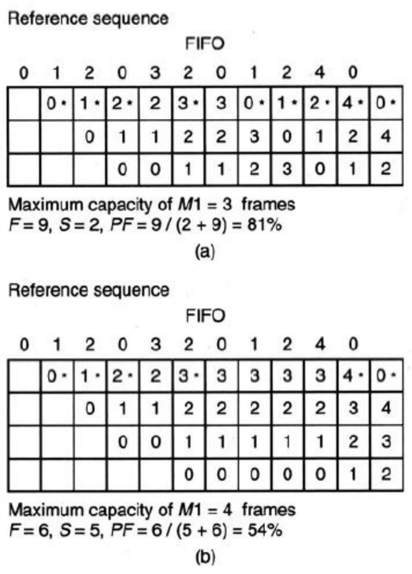
Before the discussion of replacement algorithms, you should note that the efficiency of the algorithm is based on the page size (Z) and the number of pages (N) the main memory ($M1$) can contain. If Z = 100 bytes and N = 3, then $M1 = N * Z = 3 * 100 = 300$ bytes.

Another concern with replacement algorithms is the page fault frequency (PF). The PF is determined by the number of page faults (F) that occurs in an entire execution of a process divided by F plus the number of no-fault references (S): $PF = F / (S + F)$. The PF should be as low a percentage as possible in order to minimize disk accesses. The PF is affected by page size and the number of page frames.

First In, First Out. First in, first out (FIFO) is one of the simplest algorithms to employ. As the name implies, the first page loaded will be the first page to be removed from main memory. Figure 2.55a demonstrates how FIFO works as well as how the page fault frequency (*PF*) is determined by using a table.

The number of rows in the table represents the number of available frames, and the columns represent each reference to a page. These references come from a given reference sequence 0, 1, 2, 0, 3, 2, 0, 1, 2, 4, 0, where the first reference is to page 0, the second is to page 1, the third is to page 2, and so on. When a page fault occurs, the corresponding page number is put in the top row, representing its precedence, and marked with an asterisk. The previous pages in the table are moved down. Once the page frames are filled and a page fault occurs, the page in the bottom page frame is removed or swapped out. (Keep in mind that this table is used only to visualize a page's current precedence. The movement of these pages does not imply that they are actually being shifted around in *M*1. A counter can be associated with each page to determine the oldest page.)

Once the last reference in the reference sequence is loaded, the *PF* can be calculated. The number of asterisks appearing in the top row equals page faults (*F*) and the items in the top row that do not contain an asterisk equals success (*S*). Considering Figure 2.55, when *M*1 contains three frames, *PF* is 81% for the above reference sequence. When *M*1 contains four frames, *PF* reduces to 54%. That is, *PF* is improved by increasing the number of page frames to 4.


Reference sequence

FIFO

0	1	2	0	3	2	0	1	2	4	0	
	0•	1•	2•	2	3•	3	0•	1•	2•	4•	0•
		0	1	1	2	2	3	0	1	2	4
			0	0	1	1	2	3	0	1	2

Maximum capacity of *M*1 = 3 frames
$F = 9$, $S = 2$, $PF = 9 / (2 + 9) = 81\%$

(a)

Reference sequence

FIFO

0	1	2	0	3	2	0	1	2	4	0	
	0•	1•	2•	2	3•	3	3	3	3	4•	0•
		0	1	1	2	2	2	2	2	3	4
			0	0	1	1	1	1	1	2	3
					0	0	0	0	0	1	2

Maximum capacity of *M*1 = 4 frames
$F = 6$, $S = 5$, $PF = 6 / (5 + 6) = 54\%$

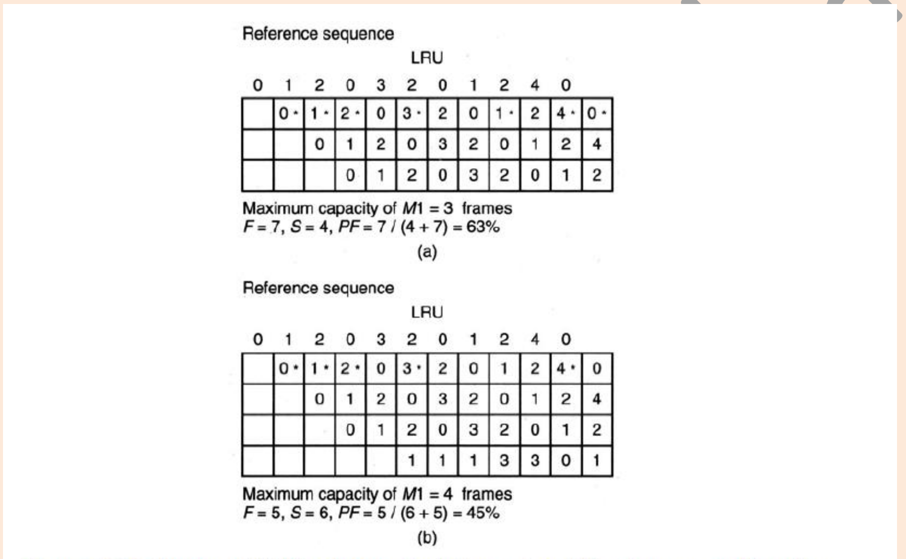
(b)

Figure 2.55 Performance of the FIFO replacement technique on two different memory configurations.

The disadvantage of FIFO is that it may significantly increase the time it takes for a process to execute because it does not take into consideration the principle of locality and consequently may replace heavily used frames as well as rarely used frames with equal probability. For example, if an early frame contains a global variable that is in constant use, this frame will be one of the first to be replaced. During the next access to the variable, another page fault will occur, and the frame will have to be reloaded, replacing yet another page.

Lecture 6: *Least Recently Used*.

The least recently used (LRU) method will replace the frame that has not been used for the longest time. In this method, when a page is referenced that is already in *M*1, it is placed in the top row and the other pages are shifted down. In other words, the most used pages are kept at the top. See Figure 2.56a. An improvement of *PF* is made using LRU by adding a page frame to *M*1. See Figure 2.56b.
In general, LRU is more efficient than FIFO, but it requires more hardware (usually a counter or a stack) to keep track of the least and most recently used pages.



Reference sequence

LRU

0	1	2	0	3	2	0	1	2	4	0	
	0 *	1 *	2 *	0	3 *	2	0	1 *	2	4 *	0 *
		0	1	2	0	3	2	0	1	2	4
			0	1	2	0	3	2	0	1	2

Maximum capacity of $M1 = 3$ frames
$F = 7$, $S = 4$, $PF = 7 / (4 + 7) = 63\%$

(a)

Reference sequence

LRU

0	1	2	0	3	2	0	1	2	4	0	
	0 *	1 *	2 *	0	3 *	2	0	1	2	4 *	0
		0	1	2	0	3	2	0	1	2	4
			0	1	2	0	3	2	0	1	2
					1	1	1	3	3	0	1

Maximum capacity of $M1 = 4$ frames
$F = 5$, $S = 6$, $PF = 5 / (6 + 5) = 45\%$

(b)

Figure 2.56 Performance of LRU replacement technique on two different memory configurations.

Segmentation. Another method of swapping between secondary and main memory is called *segmentation*.
In segmentation, a program is broken into variable-length sections known as *segments*. For example, a segment can be a data set or a function within the program. Each process keeps a segment table within main memory that contains basically the same information as the page table. However, unlike pages, segments have variable lengths, and they can start anywhere in the memory; therefore, removing one segment from main memory may not provide enough space for another segment.
There are several strategies for placing a given segment into the main memory. Among the most well known strategies are *first fit*, *best fit*, and *worst fit*. Each of these strategies maintains a list that represents the size and position of the free storage blocks in the main memory. This list is used for finding a suitable block size for the given segment. The following is an explanation.

First Fit. This strategy puts the given segment into the first suitable free storage. It searches through the free storage list until it finds a block of free storage that is large enough for the segment; then it allocates a block of memory for the segment.
The main advantage of this strategy is that it encourages free storage areas to become available at high memory addresses by assigning segments to the low-memory addresses whenever possible. However, this strategy will produce free areas that may be too small to hold a segment. This phenomenon is known as *fragmentation*. When fragmentation occurs, eventually some sort of compaction algorithm will have to be run to collect all the small free areas into one large one. This causes some overhead, which degrades the performance.

Best Fit. This strategy allocates the smallest available free storage block that is large enough to hold the segment. It searches through the free storage list until it finds the smallest block of storage that is large enough for the segment. To prevent searching the entire list, the free storage list is usually sorted according to the increasing block size. Unfortunately, like first fit, this strategy also causes fragmentation. In fact, it may create many small blocks that are almost useless.

Worst Fit. This strategy allocates the largest available free storage block for the segment. It searches the free storage list for the largest block. The list is usually sorted according to the decreasing block size.
Again, the worst fit, like the other two strategies, causes fragmentation. However, in contrast to first fit and best fit, worst fit reduces the number of small blocks by always allocating the largest block for the segment.

Lecture 7: Performance Measures

Performance Measures

- To assess the performance of a computer, several important issues need to be considered.
- The number of performance measures that are used to assess computers is a controversial topic.
- For example, a user of a computer measures its performance based on the time taken to execute a given job (program).
- A laboratory engineer measures the performance of his system by the total amount of work done in a given time.
- While the user considers the program execution time a measure for performance, the laboratory engineer considers the throughput a more important measure for performance.
- A metric for assessing the performance of a computer helps comparing alternative designs.

Performance Factors

- Performance analysis should help answering questions such as how fast can a program be executed using a given computer.
- In order to answer such a question, we need to determine the time taken by a computer to execute a given job.
- We define the **clock cycle** time as the time between two consecutive rising (trailing) edges of a periodic clock signal.

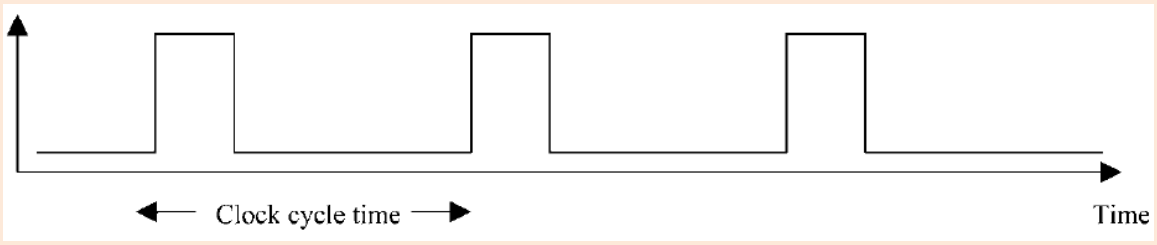


Figure 1. Clock signal

- Clock cycles allow counting unit computations, because the storage of computation results is synchronized with rising (trailing) clock edges.

Clock Cycles: is the time required to execute a job by a computer.

Cycle Count (CC): the number of CPU clock cycles for executing a job.

Cycle Time (CT): the time required to complete on cycle.

The time taken by the CPU to execute a job can be expressed as:

CPU time = CC * CT

Example:

What is the required time for a CPU to execute an instruction if the number of CPU clock cycle (CC) is 5 and the time for each cycle (CT) is 2ms?

Solution:

CPU time= 5 * 2 = 10ms

<u>Cycles per Instruction (CPI):</u> is the average number of clock cycles per instruction for a program.

- It may be easier to count the number of instructions executed in a given program as compared to counting the number of CPU clock cycles needed for executing that program.
- Therefore, the average number of clock cycles per instruction (CPI) has been used as an alternate performance measure.
- The following equation shows how to compute the CPI.

CPU time = Instruction count * CPI * Clock cycle time

$$CPU\ time = \frac{Instruction\ count \times CPI}{Clock\ rate}$$

Example:

Consider that we have two computers. The first one includes a 500 MHz Pentium III processor which takes 2 ms to run a program with 200,000 instructions. While the second on includes a 300 MHz Ultra Sparc processor which takes 1.8 ms to run the same program with 230,000 instructions. Find the CPI for processor. Which one has the better performance?

Solution:

CPI = (CPU time * clock rate) / Instruction count

$CPI_{Pentium} = 2*10^{-3} * 500*10^{6} / 2*10^{5} = 5.00$

$CPI_{SPARC} = 1.8*10^{-3} * 300*10^{6} / 2.3*10^{5} = 2.35$

Sparc computer has the better performance since it has the less CPI.

The different between HDD and SSD

	HDD	SSD
Power	More power consumption	Less power consumption
Speed	Slow	Fast in open files, games, programs
Capacity	Bigger capacity	Smaller capacity
Weight	Heavier	Lighter
Noise	Noisy	No noise
Price	Cheaper	Expensive
Age limit	Less	More

Lecture 8: I/O system

The control of devices connected to the computer is a major concern of operating-system designers. Because I/O devices vary so widely in their function and speed (consider a mouse, a hard disk, and a tape robot), varied methods are needed to control them. These methods form the I/O subsystem of the kernel, which separates the rest of the kernel from the complexities of managing I/O devices.
I/O-device technology exhibits two conflicting trends. On the one hand, we see increasing standardization of software and hardware interfaces. This trend helps us to incorporate improved device generations into existing computers and operating systems. On the other hand, we see an increasingly broad variety of I/O devices. Some new devices are so unlike previous devices that it is a challenge to incorporate them into our computers and operating systems. This challenge is met by a combination of hardware and software techniques. The basic I/O hardware elements, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and oddities of different devices, the kernel of an operating system is structured to use device-driver modules. The **device drivers** present a uniform device access interface to the I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

I/O Hardware

Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks, tapes), transmission devices (network connections, Bluetooth), and human-interface devices (screen, keyboard, mouse, audio in and out). Other devices are more specialized, such as those involved in the steering of a jet. In these aircraft, a human gives input to the flight computer via a joystick and foot pedals, and the computer sends output commands that cause motors to move rudders and flaps and fuels to the engines. Despite the incredible variety of I/O devices, though, we need only a few concepts to understand how the devices are attached and how the software can control the hardware.
A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point, or **port**—for example, a serial port. If devices share a common set of wires, the connection is called a bus. A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B*, and device *B* has a cable that plugs into device *C*, and device *C*

plugs into a port on the computer, this arrangement is called a **daisy chain**. A daisy chain usually operates as a bus.

Buses are used widely in computer architecture and vary in their signaling methods, speed, throughput, and connection methods. A typical PC bus structure appears in Figure 13.1. In the figure, a **PCI bus** (the common PC system bus) connects the processor–memory subsystem to fast devices, and an **expansion bus** connects relatively slow devices, such as the keyboard and serial and USB ports. In the upper-right portion of the figure, four disks are connected together on a **Small Computer System Interface (SCSI)** bus plugged into a SCSI controller. Other common buses used to interconnect main parts of a computer include **PCI Express (PCIe)**, with throughput of up to 16 GB per second, and **Hyper Transport**, with throughput of up to 25 GB per second.

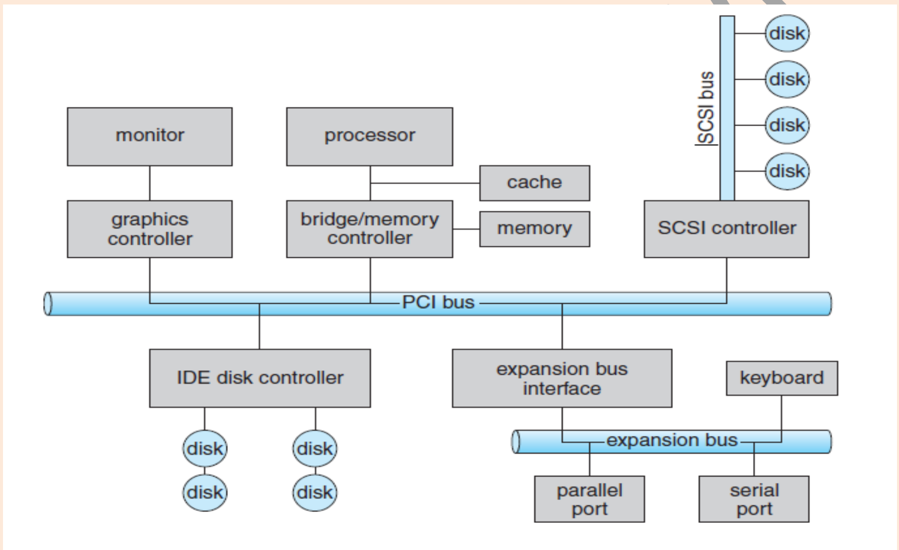


Figure 13.1 A typical PC bus structure.

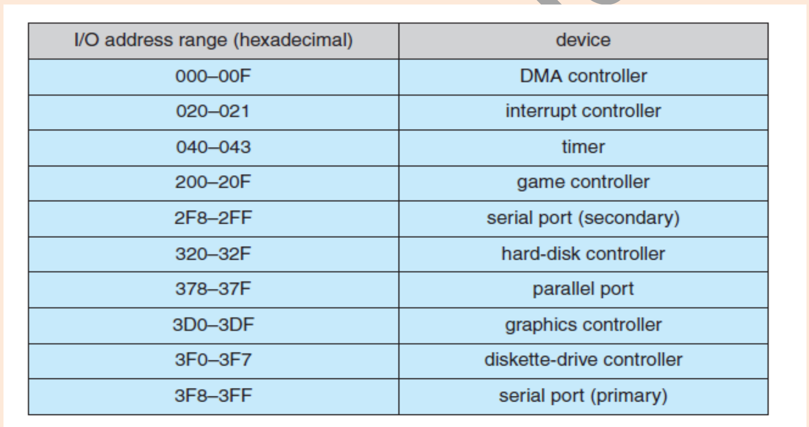
A **controller** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip (or portion of a chip) in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a host adapter) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers. If you look at a disk drive, you will see a circuit board attached to one side. This board is the disk controller. It implements the disk side of the protocol for some kind of connection—SCSI or Serial Advanced Technology Attachment

(SATA), for instance. It has microcode and a processor to do many tasks, such as bad-sector mapping, prefetching, buffering, and caching.

How can the processor give commands and data to a controller to accomplish an I/O transfer? The short answer is that the controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. One way in which this communication can occur is through the use of special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, the device controller can support memory-mapped I/O. In this case, the device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers at their mapped locations in physical memory.

Some systems use both techniques. For instance, PCs use I/O instructions to control some devices and memory-mapped I/O to control others.

Figure 13.2 shows the usual I/O port addresses for PCs.



I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure 13.2 Device I/O port locations on PCs (partial).

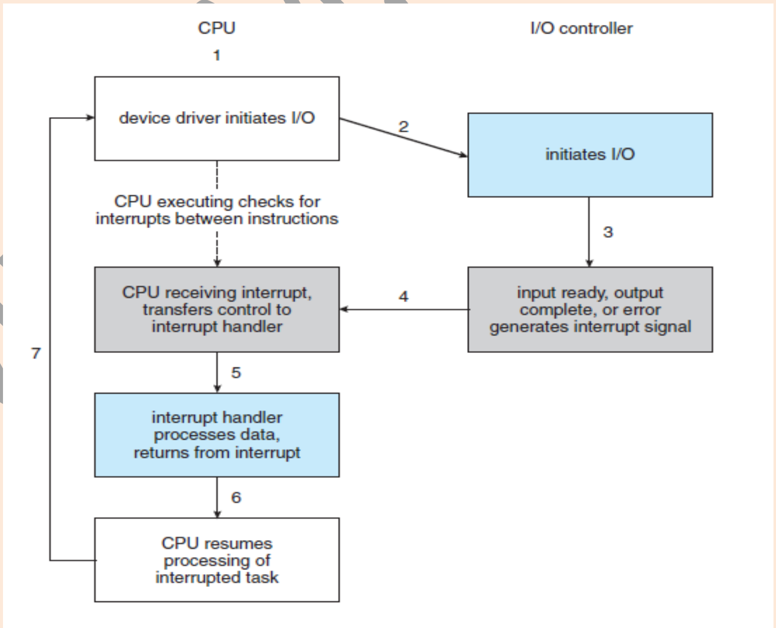
The graphics controller has I/O ports for basic control operations, but the controller has a large memory mapped region to hold screen contents. The process sends output to the screen by writing data into the memory-mapped region. The controller generates the screen image based on the contents of this memory. This technique is simple to use. Moreover, writing millions of bytes to the graphics memory is faster than issuing millions of I/O instructions. But the ease of writing to a memory-mapped I/O controller is offset by a disadvantage. Because a common type of software fault is a write through an incorrect pointer to an unintended region of memory, a memory-mapped device register is vulnerable to accidental modification. Of course, protected memory helps to reduce this risk. An I/O port typically consists of four registers, called the status, control, data-in, and data-out registers.

• The data-in register is read by the host to get input.
• The data-out register is written by the host to send output.
• The status register contains bits that can be read by the host.
These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
• The control register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another bit enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port.
The data registers are typically 1 to 4 bytes in size. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

Polling
The complete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple. We explain handshaking with an example. Assume that 2 bits are used to coordinate the producer–consumer relationship between the controller and the host. The controller indicates its state through the busy bit in the status register. (Recall that to set a bit means to write a 1 into the bit and to clear a bit means to write a 0 into it.) The controller sets the busy bit when it is busy working and clears the busy bit when it is ready to accept the next command. The host signals its wishes via the command-ready bit in the command register. The host sets the command-ready bit when a command is available for the controller to execute.
For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the busy bit.
5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device.
6. The controller clears the command-ready bit, clears the error bit in the

status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.
This loop is repeated for each byte.
In step 1, the host is busy-waiting or polling: it is in a loop, reading the status register over and over until the busy bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task. How, then, does the host know when the controller has become idle? For some devices, the host must service the device quickly, or data will be lost. For instance, when data are streaming in on a serial port or from a keyboard, the small buffer on the controller will overflow and data will be lost if the host waits too long before returning to read the bytes.
In many computer architectures, three CPU-instruction cycles are sufficient to poll a device: read a device register, logical--and to extract a status bit, and branch if not zero. Clearly, the basic polling operation is efficient. But polling becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service, while other useful CPU processing remains undone. In such instances, it may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to require the CPU to poll repeatedly for an I/O completion. The hardware mechanism that enables a device to notify the CPU is called an interrupt


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task
Figure 13.3 Interrupt-driven I/O cycle.

Interrupts

The basic interrupt mechanism works as follows. The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the interrupt-handler routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches it to the interrupt handler, and the handler clears the interrupt by servicing the device. Figure 13.3 summarizes the interrupt-driven I/O cycle. We stress interrupt management in this chapter because even single-user modern systems manage hundreds of interrupts per second and servers hundreds of thousands per second.
The basic interrupt mechanism just described enables the CPU to respond to an asynchronous event, as when a device controller becomes ready for service. In a modern operating system, however, we need more sophisticated interrupt-handling features.

1. We need the ability to defer interrupt handling during critical processing.
2. We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.
3. We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

In modern computer hardware, these three features are provided by the CPU and by the interrupt-controller hardware.
Most CPUs have two interrupt request lines. One is the nonmaskable interrupt, which is reserved for events such as unrecoverable memory errors.
The second interrupt line is maskable: it can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted.
The maskable interrupt is used by device controllers to request service.
The interrupt mechanism accepts an address—a number that selects a specific interrupt-handling routine from a small set. In most architectures, this address is an offset in a table called the interrupt vector. This vector contains the memory addresses of specialized interrupt handlers. The

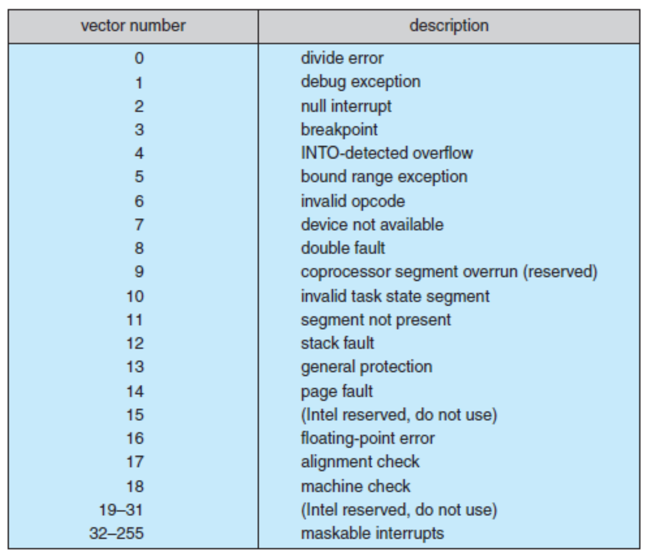
purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service. In practice, however, computers have more devices (and, hence, interrupt handlers) than they have address elements in the interrupt vector.

A common way to solve this problem is to use interrupt chaining, in which each element in the interrupt vector points to the head of a list of interrupt handlers. When an interrupt is raised, the handlers on the corresponding list are called one by one, until one is found that can service the request. This structure is a compromise between the overhead of a huge interrupt table and the inefficiency of dispatching to a single interrupt handler.

Figure 13.4 illustrates the design of the interrupt vector for the Intel Pentium processor. The events from 0 to 31, which are nonmaskable, are used to signal various error conditions. The events from 32 to 255, which are maskable, are used for purposes such as device-generated interrupts.

The interrupt mechanism also implements a system of interrupt priority levels. These levels enable the CPU to defer the handling of low-priority interrupts without masking all interrupts and makes it possible for a high priority interrupt to preempt the execution of a low-priority interrupt.

A modern operating system interacts with the interrupt mechanism in several ways. At boot time, the operating system probes the hardware buses to determine what devices are present and installs the corresponding interrupt handlers into the interrupt vector. During I/O, the various device controllers raise interrupts when they are ready for service. These interrupts signify that output has completed, or that input data are available, or that a failure has been detected. The interrupt mechanism is also used to handle a wide variety of exceptions, such as dividing by 0, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode. The events that trigger interrupts have a common property: they are occurrences that induce the operating system to execute an urgent, self-contained routine.



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 13.4 Intel Pentium processor event-vector table.

An operating system has other good uses for an efficient hardware and software mechanism that saves a small amount of processor state and then calls a privileged routine in the kernel. For example, many operating systems use the interrupt mechanism for virtual memory paging. A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel. This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.
Another example is found in the implementation of system calls. Usually, a program uses library calls to issue system calls. The library routines check the arguments given by the application, build a data structure to convey the arguments to the kernel, and then execute a special instruction called a software interrupt, or trap. This instruction has an operand that identifies the desired kernel service. When a process executes the trap instruction, the interrupt hardware saves the state of the user code, switches to kernel mode, and dispatches to the kernel routine that implements the requested service. The trap is given a relatively low interrupt priority compared with those assigned to device interrupts—executing a system call on behalf of an application is less urgent than servicing a device controller before its FIFO queue overflows and loses data.
Interrupts can also be used to manage the flow of control within the kernel. For example, consider one example of the processing required to complete a disk read. One step is to copy data from kernel space to the user buffer.

This copying is time consuming but not urgent—it should not block other high-priority interrupt handling. Another step is to start the next pending I/O for that disk drive. This step has higher priority. If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes. Consequently, a pair of interrupt handlers implements the kernel code that completes a disk read. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low-priority interrupt to complete the work. Later, when the CPU is not occupied with high priority work, the low-priority interrupt will be dispatched. The corresponding handler completes the user-level I/O by copying data from kernel buffers to the application space and then calling the scheduler to place the application on the ready queue.

A threaded kernel architecture is well suited to implement multiple interrupt priorities and to enforce the precedence of interrupt handling over background processing in kernel and application routines. We illustrate this point with the Solaris kernel. In Solaris, interrupt handlers are executed as kernel threads. A range of high priorities is reserved for these threads.

These priorities give interrupt handlers precedence over application code and kernel housekeeping and implement the priority relationships among interrupt handlers. The priorities cause the Solaris thread scheduler to preempt low priority interrupt handlers in favor of higher-priority ones, and the threaded implementation enables multiprocessor hardware to run several interrupt handlers concurrently.

In summary, interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel. To enable the most urgent work to be done first, modern computers use a system of interrupt priorities. Device controllers, hardware faults, and system calls all raise interrupts to trigger kernel routines. Because interrupts are used so heavily for time-sensitive processing, efficient interrupt handling is required for good system performance.

Lecture 10: Direct Memory Access

For a device that does large transfers, such as a disk drive, it seems wasteful to use an expensive general-purpose processor to watch status bits and to feed data into a controller register one byte at a time—a process termed programmed I/O (PIO). Many computers avoid burdening the main CPU with PIO by offloading some of this work to a special-purpose processor called a direct-memory-access (DMA) controller. To initiate a DMA transfer, the host writes a DMA command block into memory. This block contains a pointer to the source of a transfer, a pointer to the destination of the transfer, and a count of the number of bytes to be transferred. The CPU writes the address of this command block to the DMA controller, then goes on with other work. The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU. A simple DMA controller is a standard component in all modern computers, from smartphones to mainframes.

Handshaking between the DMA controller and the device controller is performed via a pair of wires called **DMA-request** and **DMA-acknowledge**. The device controller places a signal on the DMA-request wire when a word of data is available for transfer. This signal causes the DMA controller to seize the memory bus, place the desired address on the memory-address wires, and place a signal on the DMA-acknowledge wire. When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory and removes the DMA-request signal.

When the entire transfer is finished, the DMA controller interrupts the CPU. This process is depicted in Figure 13.5. When the DMA controller seizes the memory bus, the CPU is momentarily prevented from accessing main memory, although it can still access data items in its primary and secondary caches. Although this **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance. Some computer architectures use physical memory addresses for DMA, but others perform **direct virtual memory access (DVMA)**, using virtual addresses that undergo translation to physical addresses. DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly. This discipline protects data from access-control violations and also protects the system from erroneous use of device controllers that could cause a system crash. Instead, the operating system exports functions that a sufficiently privileged process can use to access low-level operations on the underlying hardware. On kernels without memory protection, processes can access device controllers directly. This direct access can be used to achieve high performance, since it can avoid kernel communication, context switches, and layers of kernel software. Unfortunately, it interferes with system security and stability. The trend in general-purpose operating systems is to protect memory and devices so that the system can try to guard against erroneous or malicious applications.

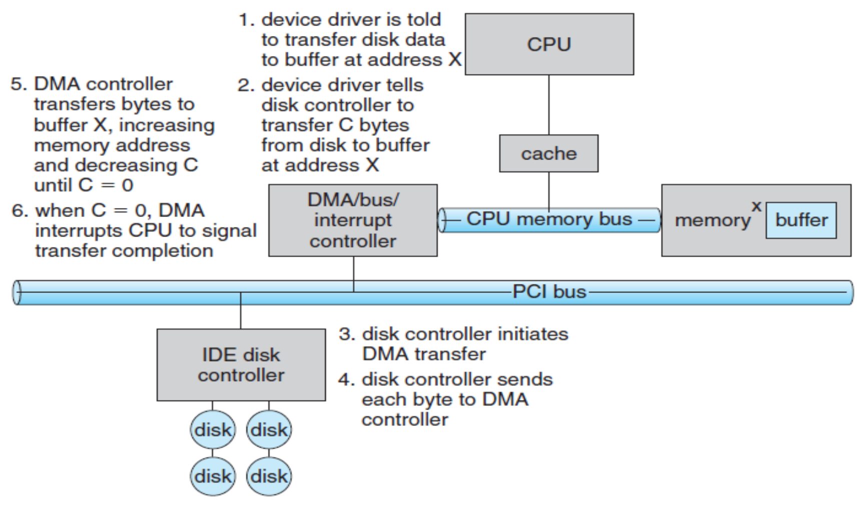

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
DMA/bus/interrupt controller
CPU memory bus
memory
X
buffer
PCI bus
IDE disk controller
disk
disk
disk
disk

Figure 13.5 Steps in a DMA transfer.

I/O Hardware Summary

Although the hardware aspects of I/O are complex when considered at the level of detail of electronics-hardware design, the concepts that we have just described are sufficient to enable us to understand many I/O features of operating systems. Let's review the main concepts:

- A bus
- A controller
- An I/O port and its registers
- The handshaking relationship between the host and a device controller
- The execution of this handshaking in a polling loop or via interrupts
- The offloading of this work to a DMA controller for large transfers

We gave a basic example of the handshaking that takes place between a device controller and the host earlier in this section. In reality, the wide variety of available devices poses a problem for operating-system implementers. Each kind of device has its own set of capabilities, control-bit definitions, and protocols for interacting with the host—and they are all different. How can the operating system be designed so that we can attach new devices to the computer without rewriting the operating system? And when the devices vary so widely, how can the operating system give a convenient, uniform I/O interface to applications? We address those questions next.

Application I/O Interface

In this section, we discuss structuring techniques and interfaces for the operating system that enable I/O devices to be treated in a standard, uniform way. We explain, for instance, how an application can open a file on a disk without knowing what kind of disk it is and how new disks and other devices can be added to a computer without disruption of the operating system.

Like other complex software-engineering problems, the approach here involves abstraction, encapsulation, and software layering. Specifically, we can abstract away the detailed differences in I/O devices by identifying a few general kinds. Each general kind is accessed through a standardized set of functions—an **interface**. The differences are encapsulated in kernel modules called device drivers that internally are custom-tailored to specific devices but that export one of the standard interfaces. Figure 13.6 illustrates how the I/O-related portions of the kernel are structured in software layers.

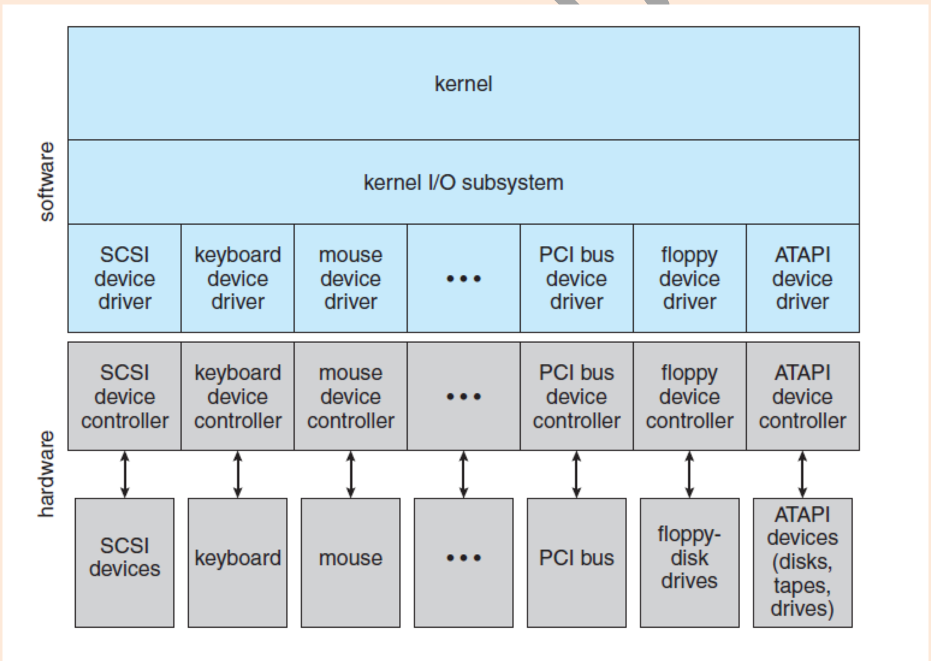
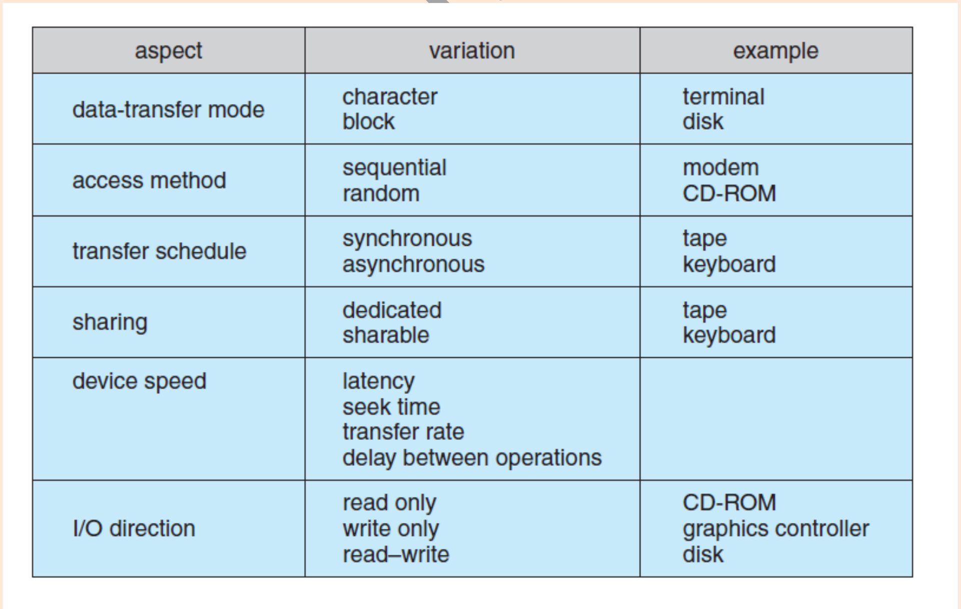


Figure 13.6 A kernel I/O structure.

The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls encapsulate the behavior of devices in a few generic classes that hide hardware differences from applications. Making the I/O subsystem independent of the hardware simplifies the job of the operating-system developer. It also benefits the hardware manufacturers. They either design new devices to be compatible with an existing host controller interface (such as SATA), or they write device drivers to interface the new hardware to popular operating systems. Thus, we can attach new peripherals to a computer without waiting for the operating-system vendor to develop support code.

Unfortunately for device-hardware manufacturers, each type of operating system has its own standards for the device-driver interface. A given device may ship with multiple device drivers—for instance, drivers for Windows, Linux, AIX, and Mac OS X. Devices vary on many dimensions, as illustrated in Figure 13.7.

- **Character-stream or block**. A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.
- **Sequential or random access**. A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.
- **Synchronous or asynchronous**. A synchronous device performs data transfers with predictable response times, in coordination with other aspects of the system. An asynchronous device exhibits irregular or unpredictable response times not coordinated with other computer events.
- **Sharable or dedicated**. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.



aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Figure 13.7 Characteristics of I/O devices.

- **Speed of operation**. Device speeds range from a few bytes per second to a few gigabytes per second.

• **Read–write, read only, or write only**. Some devices perform both input and output, but others support only one data transfer direction.

For the purpose of application access, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types. The resulting styles of device access have been found to be useful and broadly applicable. Although the exact system calls may differ across operating systems, the device categories are fairly standard. The major access conventions include block I/O, character-stream I/O, memory-mapped file access, and network sockets. Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer.

Some operating systems provide a set of system calls for graphical display, video, and audio devices.

Most operating systems also have an **escape** (or **back door**) that transparently passes arbitrary commands from an application to a device driver. In UNIX, this system call is ioctl() (for "I/O control"). The ioctl() system call enables an application to access any functionality that can be implemented by any device driver, without the need to invent a new system call. The ioctl() system call has three arguments. The first is a file descriptor that connects the application to the driver by referring to a hardware device managed by that driver. The second is an integer that selects one of the commands implemented in the driver. The third is a pointer to an arbitrary data structure in memory that enables the application and driver to communicate any necessary control information or data.

Block and Character Devices

The **block-device interface** captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device is expected to understand commands such as read() and write(). If it is a random-access device, it is also expected to have a seek() command to specify which block to transfer next. Applications normally access such a device through a file-system interface. We can see that read(), write(), and seek() capture the essential behaviors of block-storage devices, so that applications are insulated from the low-level differences among those devices.

The operating system itself, as well as special applications such as data base management systems, may prefer to access a block device as a simple linear array of blocks. This mode of access is sometimes called **raw I/O**. If the application performs its own buffering, then using a file system would cause extra, unneeded buffering. Likewise, if an application provides its own locking of file blocks or regions, then any operating-system locking services would be redundant at the least and contradictory at the worst. To avoid these conflicts, raw-device access passes control of the device directly to the application, letting the operating system step out of the way. Unfortunately, no operating-system services are then performed on this device. A compromise that is becoming common is for the operating system to allow a mode of operation on a file that disables buffering and locking. In the UNIX world, this is called **direct I/O**.

Memory-mapped file access can be layered on top of block-device drivers.

Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory. The system call that maps a file into memory returns the virtual memory address that contains a copy of the file. The actual data transfers are performed only when needed to satisfy access to the memory image. Because the transfers are handled by the same mechanism as that used for demand-paged virtual memory access, memory-mapped I/O is efficient. Memory mapping is also convenient for programmers—access to a memory-mapped file is as simple as reading from and writing to memory. Operating systems that offer virtual memory commonly use the mapping interface for kernel services. For instance, to execute a program, the operating system maps the executable into memory and then transfers control to the entry address of the executable. The mapping interface is also commonly used for kernel access to swap space on disk.

A keyboard is an example of a device that is accessed through a **character stream interface**. The basic system calls in this interface enable an application to get() or put() one character. On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services (for example, when a user types a backspace, the preceding character is removed from the input stream). This style of access is convenient for input devices such as keyboards, mice, and modems that produce data for input "spontaneously" —that is, at times that cannot necessarily be predicted by the application. This access style is also good for output devices such as printers and audio boards, which naturally fit the concept of a linear stream of bytes.

Lecture 11: Network Devices

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O,most operating systems provide a network I/O interface that is different fromthe read()–write()–seek() interface used for disks. One interface available in many operating systems, including UNIX and Windows, is the network **socket** interface.
Think of a wall socket for electricity: any electrical appliance can be plugged in. By analogy, the system calls in the socket interface enable an application to create a socket, to connect a local socket to a remote address (which plugs this application into a socket created by another application), to listen for any remote application to plug into the local socket, and to send and receive packets over the connection. To support the implementation of servers, the socket interface also provides a function called select() that manages a set of sockets. A call to select() returns information about which sockets have a packet waiting to be received and which sockets have room to accept a packet to be sent. The use of select() eliminates the polling and busy waiting that would otherwise be necessary for network I/O. These functions encapsulate the essential behaviors of networks, greatly facilitating the creation of distributed applications that can use any underlying network hardware and protocol stack.
Many other approaches to interprocess communication and network communication have been implemented. For instance, Windows provides one interface to the network interface card and a second interface to the network protocols.

Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:
• Give the current time.
• Give the elapsed time.
• Set a timer to trigger operation *X* at time *T*.
These functions are used heavily by the operating system, as well as by time sensitive applications. Unfortunately, the system calls that implement these functions are not standardized across operating systems.
The hardware to measure elapsed time and to trigger operations is called a **programmable interval timer**. It can be set to wait a certain amount of time and then generate an interrupt, and it can be set to do this once or to repeat the process to generate periodic interrupts. The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice.

The disk I/O subsystem uses it to invoke the periodic flushing of dirty cache buffers to disk, and the network subsystem uses it to cancel operations that are proceeding too slowly because of network congestion or failures. The operating system may also provide an interface for user processes to use timers. The operating system can support more timer requests than the number of timer hardware channels by simulating virtual clocks. To do so, the kernel (or the timer device driver) maintains a list of interrupts wanted by its own routines and by user requests, sorted in earliest-time-first order. It sets the timer for the earliest time. When the timer interrupts, the kernel signals the requester and reloads the timer with the next earliest time.
On many computers, the interrupt rate generated by the hardware clock is between 18 and 60 ticks per second. This resolution is coarse, since a modern computer can execute hundreds of millions of instructions per second. The precision of triggers is limited by the coarse resolution of the timer, together with the overhead of maintaining virtual clocks. Furthermore, if the timer ticks are used to maintain the system time-of-day clock, the system clock can drift. In most computers, the hardware clock is constructed from a high frequency counter. In some computers, the value of this counter can be read from a device register, in which case the counter can be considered a high resolution clock. Although this clock does not generate interrupts, it offers accurate measurements of time intervals.

Nonblocking and Asynchronous I/O

Another aspect of the system-call interface relates to the choice between blocking I/O and nonblocking I/O. When an application issues a **blocking** system call, the execution of the application is suspended. The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution. When it resumes execution, it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Nevertheless, most operating systems use blocking system calls for the application interface, because blocking application code is easier to understand than nonblocking application code.
Some user-level processes need **nonblocking** I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen. Another example is a video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.
One way an application writer can overlap execution with I/O is to write

a multithreaded application. Some threads can perform blocking system calls, while others continue executing. Some operating systems provide nonblocking I/O system calls. A nonblocking call does not halt the execution of the application for an extended time. Instead, it returns quickly, with a return value that indicates how many bytes were transferred.
An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete. The application continues to execute its code. The completion of the I/O at some future time is communicated to the application, either through the setting of some variable in the address space of the application or through the triggering of a signal or software interrupt or a call-back routine that is executed outside the linear control flow of the application. The difference between nonblocking and asynchronous system calls is that a nonblocking read() returns immediately with whatever data are available—the full number of bytes requested, fewer, or none at all. An asynchronous read() call requests a transfer that will be performed in its entirety but will complete at some future time. These two I/O methods are shown in Figure 13.8.

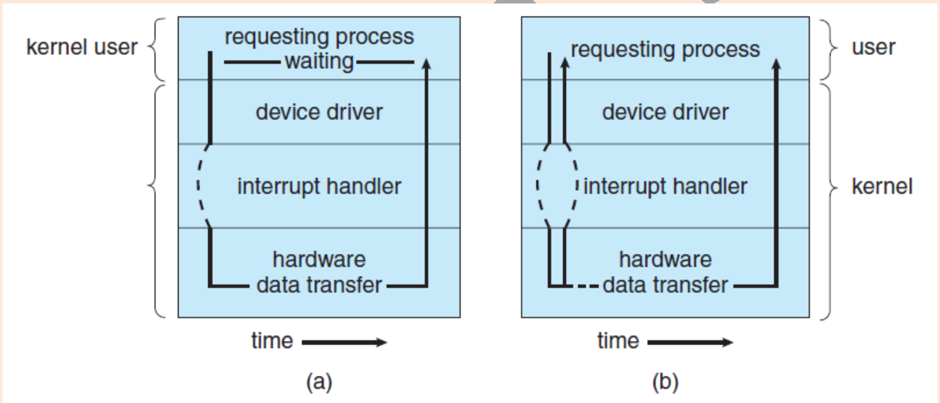


Figure 13.8 Two I/O methods: (a) synchronous and (b) asynchronous.

Asynchronous activities occur throughout modern operating systems. Frequently, they are not exposed to users or applications but rather are contained within the operating-system operation. Disk and network I/O are useful examples. By default, when an application issues a network send request or a disk write request, the operating system notes the request, buffers the I/O, and returns to the application. When possible, to optimize overall system performance, the operating system completes the request. If a system failure occurs in the interim, the application will lose any "in-flight" requests. Therefore, operating systems usually put a limit on how long they will buffer a request. Some versions of UNIX flush their disk buffers every 30 seconds, for example, or each request is flushed within 30 seconds of its occurrence. Data consistency within applications is maintained by the kernel, which reads data from its buffers before issuing I/O requests to devices, assuring that data not yet written are nevertheless returned to a requesting reader. Note that multiple threads performing I/O to the same file might not receive consistent data, depending on how the kernel implements its I/O. In this situation, the threads may need to use locking protocols. Some I/O requests need to be performed immediately, so I/O system calls usually have a way to indicate that a given request, or I/O to a specific device, should be performed synchronously.

A good example of nonblocking behavior is the select() system call for network sockets. This system call takes an argument that specifies a maximum waiting time. By setting it to 0, an application can poll for network activity without blocking. But using select() introduces extra overhead, because the select() call only checks whether I/O is possible. For a data transfer, select() must be followed by some kind of read() or write() command. A variation on this approach, found in Mach, is a blocking multiple-read call. It specifies desired reads for several devices in one system call and returns as soon as any one of them completes.

Vectored I/O

Some operating systems provide another major variation of I/O via their applications interfaces. **vectored I/O** allows one system call to perform multiple I/O operations involving multiple locations. For example, the UNIX read v that vector or writes from that vector to a destination. The same transfer could be caused by several individual invocations of system calls, but this **scatter– gather** method is useful for a variety of reasons. Multiple separate buffers can have their contents transferred via one system call, avoiding context-switching and system-call overhead. Without vectored I/O, the data might first need to be transferred to a larger buffer in the right order and then transmitted, which is inefficient. In addition, some versions of scatter–gather provide atomicity, assuring that all the I/O is done without interruption (and avoiding corruption of data if other threads are also performing I/Oinvolving those buffers). When possible, programmers make use of scatter–gather I/O features to increase throughput and decrease system overhead.

Kernel I/O Subsystem

Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—are provided by the kernel's I/O subsystem and build on the hardware and device driver infrastructure. The I/O subsystem is also responsible for protecting itself from errant processes and malicious users.

I/O Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. Here is a simple example to illustrate. Suppose that a disk arm is near the beginning of a disk and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in the order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling.

Operating-system developers implement scheduling by maintaining await queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests. For instance, requests from the virtual memory subsystem may take priority over application requests. Several scheduling algorithms for disk I/O are detailed in Section 10.4. When a kernel supports asynchronous I/O, it must be able to keep track of many I/O requests at the same time. For this purpose, the operating system might attach the wait queue to a **device-status table**. The kernel manages this table, which contains an entry for each I/O device, as shown in Figure 13.9.

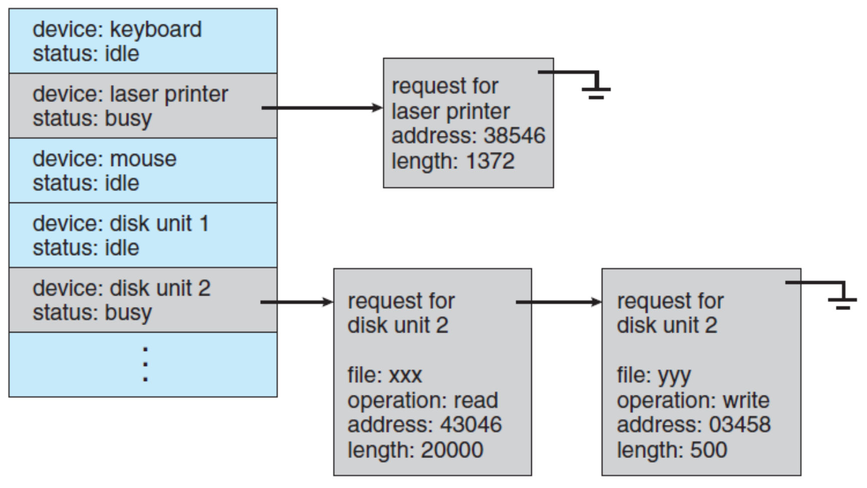


Figure 13.9 Device-status table.

Each table entry indicates the device's type, address, and state (not functioning, idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device.

Scheduling I/O operations is one way in which the I/O subsystem improves the efficiency of the computer. Another way is by using storage space in main memory or on disk via buffering, caching, and spooling.

Buffering

A **buffer**, of course, is a memory area that stores data being transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation. Since the disk write is not instantaneous and the modem still needs a place to store additional incoming data, two buffers are used. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This **double buffering** decouples the producer of data from the consumer, thus relaxing timing requirements between them. The need for this decoupling is illustrated in Figure 13.10, which lists the enormous differences in device speeds for typical computer hardware.

A second use of buffering is to provide adaptations for devices that have different data-transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and reassembly of messages.

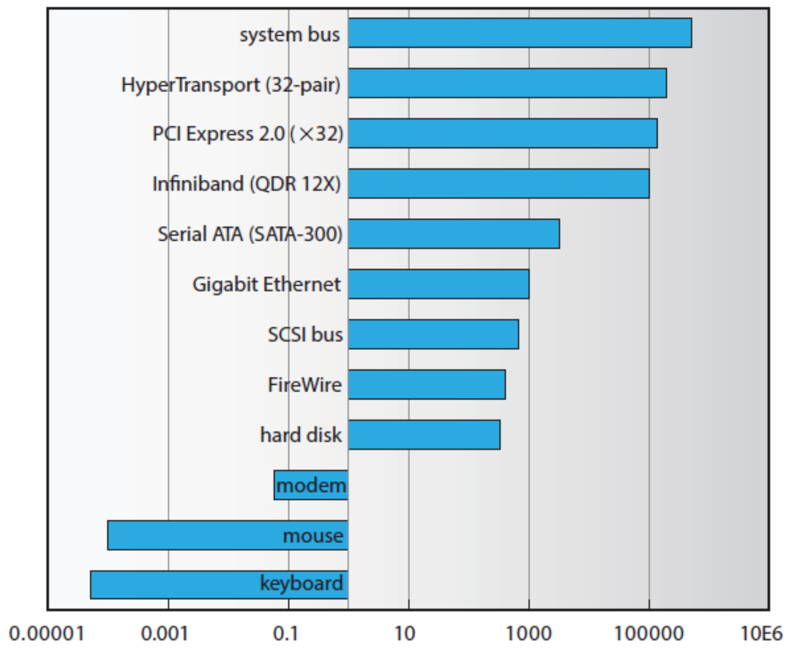


Figure 13.10 Sun Enterprise 6000 device-transfer rates (logarithmic).

At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.
A third use of buffering is to support copy semantics for application I/O. An example will clarify the meaning of “copy semantics.” Suppose that an application has a buffer of data that it wishes to write to disk. It calls the write() system call, providing a pointer to the buffer and an integer specifying the number of bytes to write. After the system call returns, what happens if the application changes the contents of the buffer? With **copy semantics**, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application’s buffer. A simple way in which the operating system can guarantee copy semantics is for the write() system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect. Copying of data between kernel buffers and application data space is common in operating systems, despite the overhead that this operation introduces, because of the clean semantics. The same effect can

be obtained more efficiently by clever use of virtual memory mapping and copy-on-write page protection.

Lecture 12: Caching

A **cache** is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU’s secondary and primary caches. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, holds a copy on faster storage of an item that resides elsewhere.

Caching and buffering are distinct functions, but sometimes a region of memory can be used for both purposes. For instance, to preserve copy semantics and to enable efficient scheduling of disk I/O, the operating system uses buffers in main memory to hold disk data. These buffers are also used as a cache, to improve the I/O efficiency for files that are shared by applications or that are being written and reread rapidly. When the kernel receives a file I/O request, the kernel first accesses the buffer cache to see whether that region of the file is already available in main memory. If it is, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules.

Spooling and Device Reservation

A **spool** is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application’s output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer.

The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In others, it is handled by an in-kernel thread. In either case, the operating system provides a control interface that enables users and system administrators to display the queue, remove unwanted jobs before those jobs print, suspend printing while the printer is serviced, and so on. Some devices, such as tape drives and printers, cannot usefully multiplex the I/O requests of multiple concurrent applications. Spooling is one way operating systems can coordinate concurrent output. Another way to deal with concurrent device access is to provide explicit facilities for coordination. Some operating systems (including VMS) provide support for exclusive device access by enabling a process to allocate an idle device and to deallocate that device when it is no longer needed.

Other operating systems enforce a limit of one open file handle to such a device. Many operating systems provide functions that enable processes to coordinate exclusive access among themselves. For instance, Windows provides system calls to wait until a device object becomes available. It also has a parameter to the OpenFile() system call that declares the types of access to be permitted to other concurrent threads. On these systems, it is up to the applications to avoid deadlock.

Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical malfunction. Devices and I/O transfers can fail in many ways, either for transient reasons, as when a network becomes overloaded, or for "permanent" reasons, as when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures. For instance, a disk read() failure results in a read() retry, and a network send() error results in a resend(), if the protocol so specifies.

Unfortunately, if an important component experiences a permanent failure, the operating system is unlikely to recover.

As a general rule, an I/O system call will return one bit of information about the status of the call, signifying either success or failure. In the UNIX operating system, an additional integer variable named errno is used to return an error code—one of about a hundred values—indicating the general nature of the failure (for example, argument out of range, bad pointer, or file not open). By contrast, some hardware can provide highly detailed error information, although many current operating systems are not designed to convey this information to the application. For instance, a failure of a SCSI device is reported by the SCSI protocol in three levels of detail: a **sense key** that identifies the general nature of the failure, such as a hardware error or an illegal request; an **additional sense code** that states the category of failure, such as a bad command parameter or a self-test failure; and an **additional sense-code qualifier** that gives even more detail, such as which command parameter was in error or which hardware subsystem failed its self-test. Further, many SCSI devices maintain internal pages of error-log information that can be requested by the host—but seldom are.

I/O Protection

Errors are closely related to the issue of protection. A user process may accidentally or purposely attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions. We can use various

mechanisms to ensure that such disruptions cannot take place in the system.
To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf (Figure 13.11). The operating system, executing in monitor mode, checks that the request is valid and, if it is, does the I/O requested. The operating system then returns to the user.
In addition, any memory-mapped and I/O port memory locations must be protected from user access by the memory-protection system. Note that a kernel cannot simply deny all user access. Most graphics games and video editing and playback software need direct access to memory-mapped graphics controller memory to speed the performance of the graphics, for example. The kernel might in this case provide a locking mechanism to allow a section of graphics memory (representing a window on screen) to be allocated to one process at a time.

Kernel Data Structures

The kernel needs to keep state information about the use of I/O components. It does so through a variety of in-kernel data structures, such as the open-file

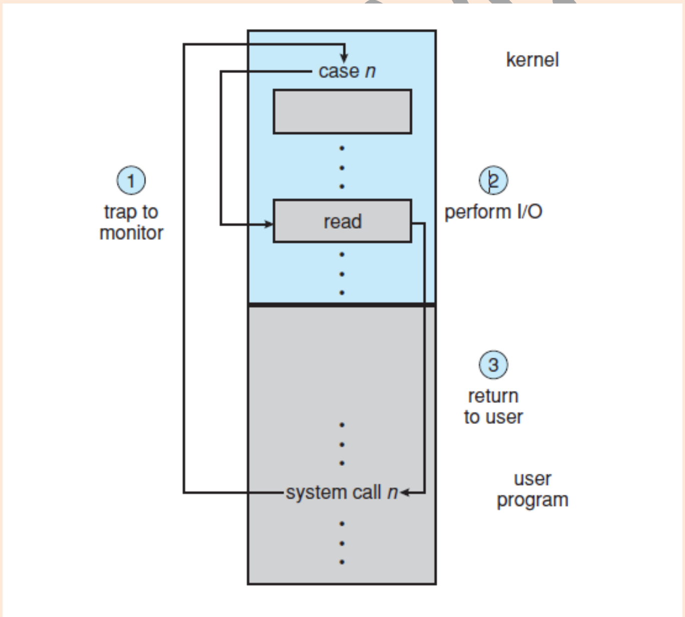


Figure 13.11 Use of a system call to perform I/O.

table structure from Section 12.1. The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities.

UNIX provides file-system access to a variety of entities, such as user files, raw devices, and the address spaces of processes. Although each of these entities supports a read() operation, the semantics differ. For instance, to read a user file, the kernel needs to probe the buffer cache before deciding whether to perform a disk I/O. To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size and is aligned on a sector boundary. To read a process image, it is merely necessary to copy data from memory. UNIX encapsulates these differences within a uniform structure by using an object-oriented technique. The open-file record, shown in Figure 13.12, contains a dispatch table that holds pointers to the appropriate routines, depending on the type of file.

Some operating systems use object-oriented methods even more extensively. For instance, Windows uses a message-passing implementation for I/O.

An I/O request is converted into a message that is sent through the kernel to the I/O manager and then to the device driver, each of which may change the message contents. For output, the message contains the data to be written. For input, the message contains a buffer to receive the data. The message-passing approach can add overhead, by comparison with procedural techniques that use shared data structures, but it simplifies the structure and design of the I/O system and adds flexibility.

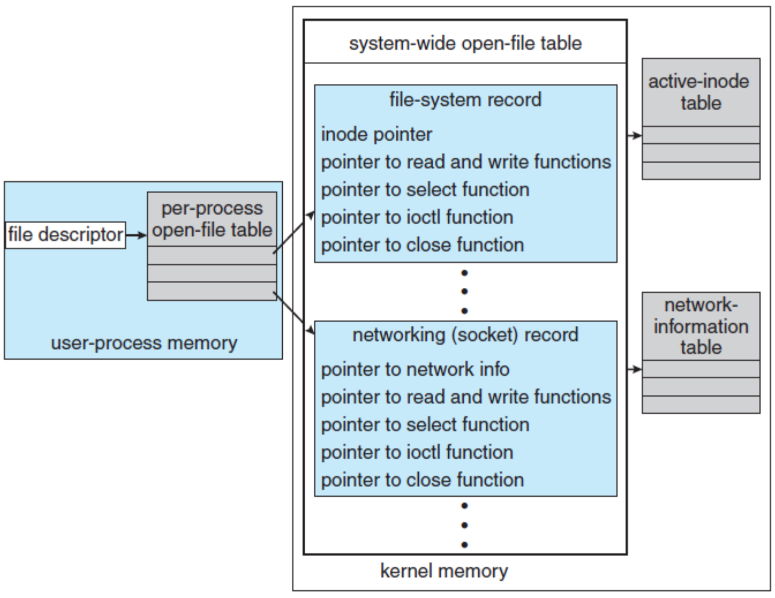

system-wide open-file table
file-system record
inode pointer
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
active-inode table
file descriptor
per-process open-file table
user-process memory
networking (socket) record
pointer to network info
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
network-information table
kernel memory

Figure 13.12 UNIX I/O kernel structure.

Kernel I/O Subsystem Summary

In summary, the I/O subsystem coordinates an extensive collection of services that are available to applications and to other parts of the kernel. The I/O subsystem supervises these procedures:

- Management of the name space for files and devices
- Access control to files and devices
- Operation control (for example, a modem cannot seek())
- File-system space allocation
- Device allocation
- Buffering, caching, and spooling
- I/O scheduling
- Device-status monitoring, error handling, and failure recovery
- Device-driver configuration and initialization

The upper levels of the I/O subsystem access devices via the uniform interface provided by the device drivers.

Lecture 13: Transforming I/O Requests to Hardware Operations:

Earlier, we described the handshaking between a device driver and a device controller, but we did not explain how the operating system connects an application request to a set of network wires or to a specific disk sector.

Consider, for example, reading a file from disk. The application refers to the data by a file name. Within a disk, the file system maps from the file name through the file-system directories to obtain the space allocation of the file. For instance, in MS-DOS, the name maps to a number that indicates an entry in the file-access table, and that table entry tells which disk blocks are allocated to the file. In UNIX, the name maps to an inode number, and the corresponding inode contains the space-allocation information. But how is the connection made from the file name to the disk controller (the hardware port address or the memory-mapped controller registers)?

One method is that used by MS-DOS, a relatively simple operating system.

The first part of an MS-DOS file name, preceding the colon, is a string that identifies a specific hardware device. For example, C: is the first part of every file name on the primary hard disk. The fact that C: represents the primary hard disk is built into the operating system; C: is mapped to a specific port address through a device table. Because of the colon separator, the device name space is separate from the file-system name space. This separation makes it easy for the operating system to associate extra functionality with each device. For instance, it is easy to invoke spooling on any files written to the printer.

If, instead, the device name space is incorporated in the regular file-system name space, as it is in UNIX, the normal file-system name services are provided automatically. If the file system provides ownership and access control to all file names, then devices have owners and access control. Since files are stored on devices, such an interface provides access to the I/O system at two levels. Names can be used to access the devices themselves or to access the files stored on the devices.

UNIX represents device names in the regular file-system name space. Unlike an MS-DOS file name, which has a colon separator, a UNIX path name has no clear separation of the device portion. In fact, no part of the path name is the name of a device. UNIX has a **mount table** that associates prefixes of path names with specific device names. To resolve a path name, UNIX looks up the name in the mount table to find the longest matching prefix; the corresponding entry in the mount table gives the device name. This device name also has the form of a name in the file-system name space. When UNIX looks up this name in the file-system directory structures, it finds not an inode number but a <major,

minor> device number. The major device number identifies a device driver that should be called to handle I/O to this device. The minor device number is passed to the device driver to index into a device table. The corresponding device-table entry gives the port address or the memory-mapped address of the device controller.
Modern operating systems gain significant flexibility from the multiple stages of lookup tables in the path between a request and a physical device controller. The mechanisms that pass requests between applications and drivers are general. Thus, we can introduce new devices and drivers into a computer without recompiling the kernel. In fact, some operating systems have the ability to load device drivers on demand. At

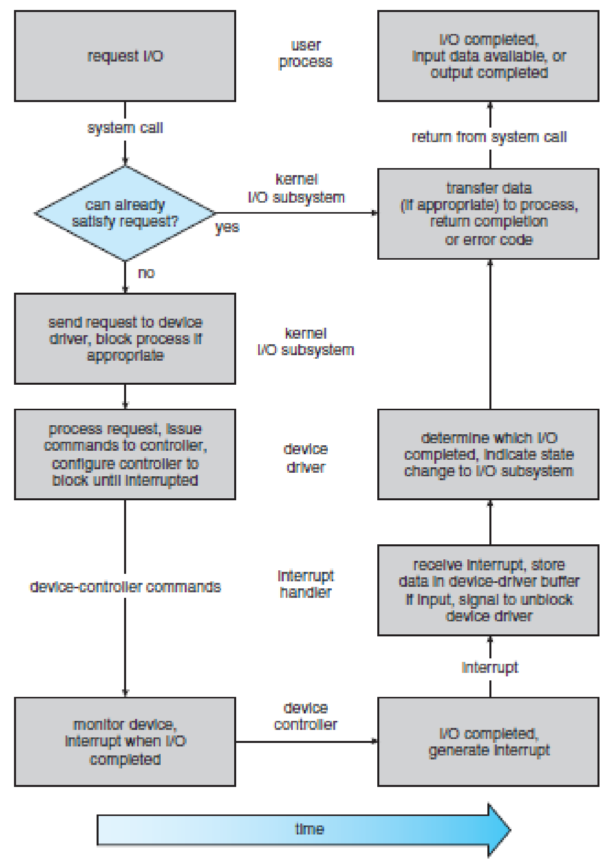

request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
kernel I/O subsystem
can already satisfy request?
yes
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

Figure 13.13 The life cycle of an I/O request.

boot time, the system first probes the hardware buses to determine what devices are present. It then loads in the necessary drivers, either immediately or when first required by an I/O request.
We next describe the typical life cycle of a blocking read request, as depicted in Figure 13.13. The figure suggests that an I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.

1. A process issues a blocking read() system call to a file descriptor of a file that has been opened previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.
3. Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.
4. The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.
5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request has completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.
9. The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

STREAMS

UNIX System V has an interesting mechanism, called **STREAMS**, that enables an application to assemble pipelines of driver code dynamically. A stream is a full-duplex connection between a device driver and a user-level process. It consists of a **stream head** that interfaces with the user process, a **driver end** that controls the device, and zero or more **stream modules** between the stream head and the driver end. Each of these components contains a pair of queues —a read queue and a write queue.

Message passing is used to transfer data between queues. The STREAMS structure is shown in Figure 13.14.
Modules provide the functionality of STREAMS processing; they are ***pushed*** onto a stream by use of the ioctl() system call. For example, a process can open a serial-port device via a stream and can push on a module to handle input editing. Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support **flow control**. Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them. A queue that supports flow control buffers messages and does not accept messages without sufficient buffer space. This process involves exchanges of control messages between queues in adjacent modules.

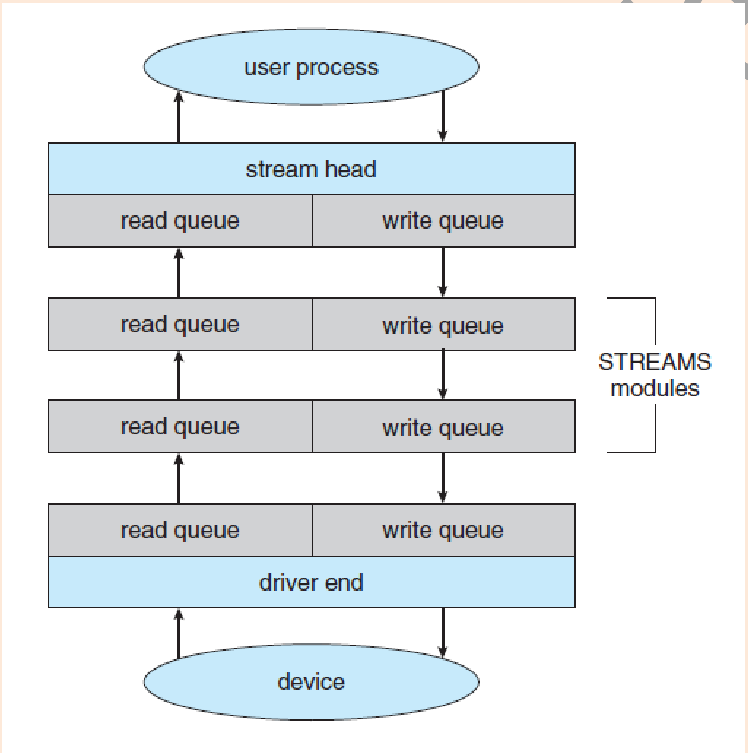


Figure 13.14 The STREAMS structure.

A user process writes data to a device using either the write() or putmsg() system call. The write() system call writes raw data to the stream, whereas putmsg() allows the user process to specify a message. Regardless of the system call used by the user process, the stream head copies the data into a message and delivers it to the queue for the next module in line. This copying of messages continues until the message is copied to the driver end and hence the device. Similarly, the user process reads data from the stream head using either the read() or getmsg() system call. If read() is used, the stream head gets a message from its adjacent queue and returns ordinary data (an unstructured byte stream) to the process. If getmsg() is used, a message is returned to the process.

STREAMS I/O is asynchronous (or nonblocking) except when the user process communicates with the stream head. When writing to the stream, the user process will block, assuming the next queue uses flow control, until there is room to copy the message. Likewise, the user process will block when reading from the stream until data are available.

As mentioned, the driver end—like the stream head and modules—has a read and write queue. However, the driver end must respond to interrupts, such as one triggered when a frame is ready to be read from a network. Unlike the stream head, which may block if it is unable to copy a message to the next queue in line, the driver end must handle all incoming data. Drivers must support flow control as well. However, if a device's buffer is full, the card whose input buffer is full. The network card must simply drop further messages until there is enough buffer space to store incoming messages.

The benefit of using STREAMS is that it provides a framework for a modular and incremental approach to writing device drivers and network protocols. Modules may be used by different streams and hence by different devices. For example, a networking module may be used by both an Ethernet network card and a 802.11 wireless network card. Furthermore, rather than treating character-device I/O as an unstructured byte stream, STREAMS allows support for message boundaries and control information when communicating between modules. Most UNIX variants support STREAMS, and it is the preferred method for writing protocols and device drivers. For example, System V UNIX and Solaris implement the socket mechanism using STREAMS.

Lecture 14: Performance

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O also exposes any inefficiencies in the interrupt-handling mechanisms in the kernel. In addition, I/O loads down the memory bus during data copies between controllers and physical memory and again during copies between kernel buffers and application data space. Coping gracefully with all these demands is one of the major concerns of a computer architect.

Although modern computers can handle many thousands of interrupts per second, interrupt handling is a relatively expensive task. Each interrupt causes the system to perform a state change, to execute the interrupt handler, and then to restore state. Programmed I/O can be more efficient than interrupt-driven I/O, if the number of cycles spent in busy waiting is not excessive. An I/O completion typically unblocks a process, leading to the full overhead of a context switch.

Network traffic can also cause a high context-switch rate. Consider, for instance, a remote login from one machine to another. Each character typed on the local machine must be transported to the remote machine. On the local machine, the character is typed; a keyboard interrupt is generated; and the character is passed through the interrupt handler to the device driver, to the kernel, and then to the user process. The user process issues a network I/O system call to send the character to the remote machine. The character then flows into the local kernel, through the network layers that construct a network packet, and into the network device driver. The network device driver transfers the packet to the network controller, which sends the character and generates an interrupt. The interrupt is passed back up through the kernel to cause the network I/O system call to complete.

Now, the remote system’s network hardware receives the packet, and an interrupt is generated. The character is unpacked from the network protocols and is given to the appropriate network daemon. The network daemon identifies which remote login session is involved and passes the packet to the appropriate sub daemon for that session.

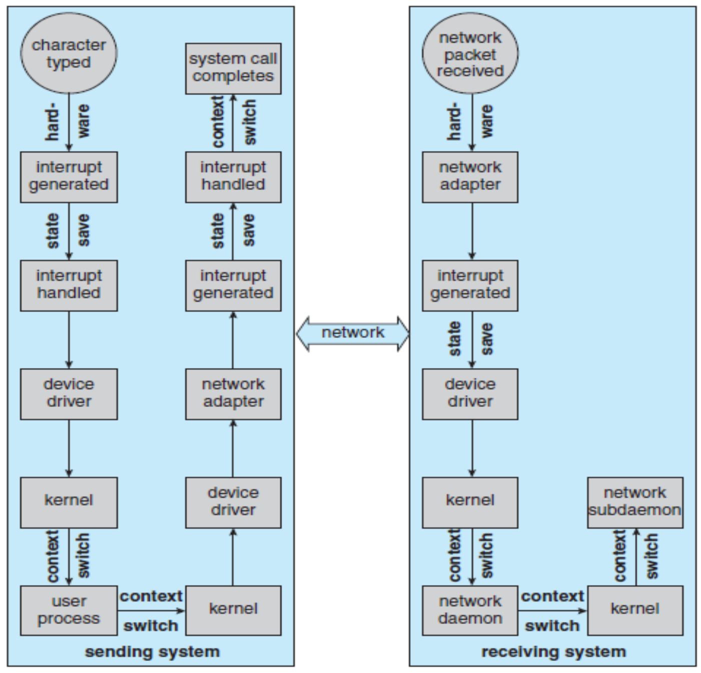


Figure 13.15 Intercomputer communications.

Throughout this flow, there are context switches and state switches (Figure 13.15). Usually, the receiver echoes the character back to the sender; that approach doubles the work.

To eliminate the context switches involved in moving each character between daemons and the kernel, the Solaris developers reimplemented the telnet daemon using in-kernel threads. Sun estimated that this improvement increased the maximum number of network logins from a few hundred to a few thousand on a large server.

Other systems use separate **front-end processors** for terminal I/O to reduce the interrupt burden on the main CPU. For instance, a **terminal concentrator** can multiplex the traffic from hundreds of remote terminals into one port on a large computer. An **I/O channel** is a dedicated, special-purpose CPU found in mainframes and in other high-end systems. The job of a channel is to offload I/O work from the main CPU. The idea is that the channels keep the data flowing smoothly, while the main CPU remains free to process the data. Like the device controllers and DMA controllers found in smaller computers, a channel can process more general and sophisticated programs, so channels can be tuned for particular workloads.

We can employ several principles to improve the efficiency of I/O:

- Reduce the number of context switches.
- Reduce the number of times that data must be copied in memory while passing between device and application.

• Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).
• Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
• Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.
• Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

I/O devices vary greatly in complexity. For instance, a mouse is simple. The mouse movements and button clicks are converted into numeric values that are passed from hardware, through the mouse device driver, to the application.
By contrast, the functionality provided by the Windows disk device driver is complex. It not only manages individual disks but also implements RAID arrays (Section 10.7). To do so, it converts an application's read or write request into a coordinated set of disk I/O operations. Moreover, it implements sophisticated error-handling and data-recovery algorithms and takes many steps to optimize disk performance. Where should the I/O functionality be implemented—in the device hardware, in the device driver, or in application software? Sometimes we observe the progression depicted in Figure 13.16.

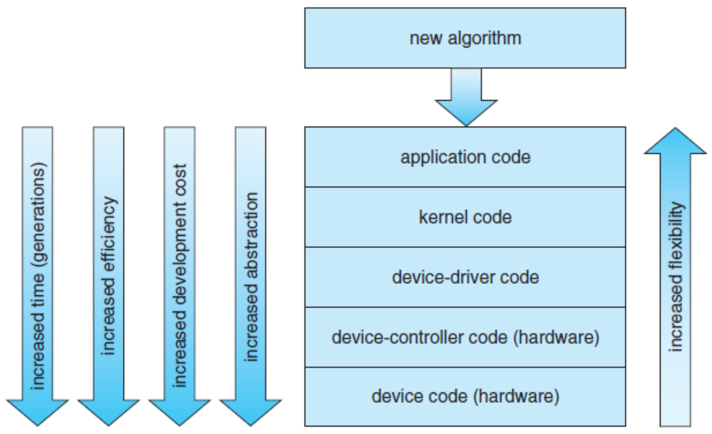


Figure 13.16 Device functionality progression.

• Initially, we implement experimental I/O algorithms at the application level, because application code is flexible and application bugs are unlikely to cause system crashes. Furthermore, by developing code at the application level, we avoid the need to reboot or reload device drivers after every change to the code. An application-level implementation can be inefficient, however, because of the overhead of context switches and because the application cannot take advantage of internal kernel data structures and kernel functionality (such as efficient in-kernel messaging, threading, and locking).

• When an application-level algorithm has demonstrated its worth, we may reimplement it in the kernel. This can improve performance, but the development effort is more challenging, because an operating-system kernel is a large, complex software system. Moreover, an in-kernel implementation must be thoroughly debugged to avoid data corruption and system crashes.

• The highest performance may be obtained through a specialized implementation in hardware, either in the device or in the controller. The disadvantages of a hardware implementation include the difficulty and expense of making further improvements or of fixing bugs, the increased development time (months rather than days), and the decreased flexibility. For instance, a hardware RAID controller may not provide any means for the kernel to influence the order or location of individual block reads and writes, even if the kernel has special information about the workload that would enable it to improve the I/O performance.