UNIT- I OPERATING SYSTEMS OVERVIEW

Computer System Overview-Basic Elements, Instruction Execution, Interrupts, Memory Hierarchy, Cache Memory, Direct Memory Access, Multiprocessor and Multicore Organization. Operating system overview-objectives and functions, Evolution of Operating System.- Computer System Organization-Operating System Structure and Operations- System Calls, System Programs, OS Generation and System Boot.

COMPUTER SYSTEM OVERVIEW:

BASIC ELEMENTS OF A COMPUTER: A computer consists of processor, memory, I/O components and system bus.

Processor: It Controls the operation of the computer and performs its data processing functions. When there is only one processor, it is often referred to as the central processing unit.

Main memory: It Stores data and programs. This memory is typically volatile; that is, when the computer is shut down, the contents of the memory are lost. Main memory is also referred to as real memory or primary memory.

I/O modules: It moves data between the computer and its external environment. The external environment consists of a variety of devices, including secondary memory devices (e. g., disks), communications equipment, and terminals.

System bus: It provides the communication among processors, main memory, and I/O modules.

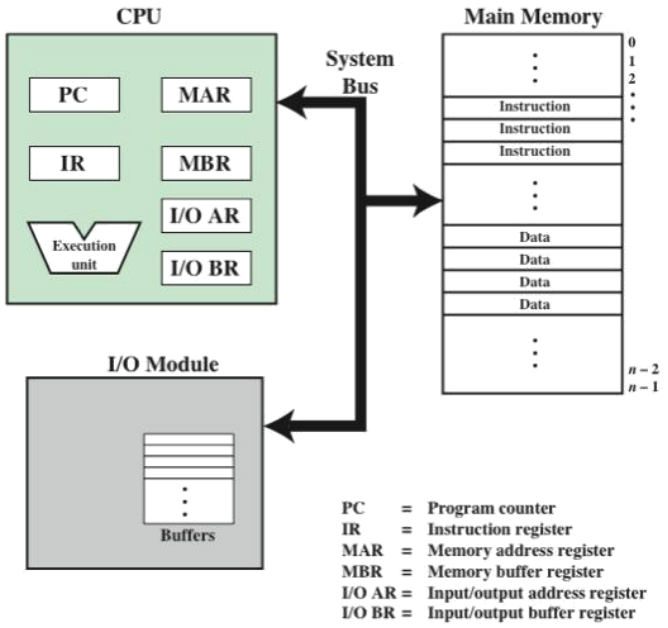


Figure 3.2 Computer Components: Top-Level View

One of the processor's functions is to exchange data with memory. For this purpose, it typically makes use of two internal registers

A memory address registers (MAR), which specifies the address in memory for the next read or write.

A memory buffer register (MBR), which contains the data to be written into memory or which receives the data read from memory.

An I/O address register (I/OAR) specifies a particular I/O device.

An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the processor.

A memory module consists of a set of locations, defined by sequentially numbered addresses.

An I/O module transfers data from external devices to processor and memory, and vice versa. It contains internal buffers for temporarily holding data until they can be sent on.

PROCESSOR REGISTERS:

A processor includes a set of registers that provide memory that is faster and smaller than main memory. Processor registers serve two functions:

User-visible registers: Enable the machine or assembly language programmer to minimize main memory references by optimizing register use.

Control and status registers: Used by the processor to control the operation of the processor and by privileged OS routines to control the execution of programs.

1. User-Visible Registers:

A user-visible register is generally available to all programs, including application programs as well as system programs. The types of User visible registers are

Data Registers

Address Registers

Data Registers can be used with any machine instruction that performs operations on data.

Control and status register:

A variety of processor registers are employed to control the operation of the processor. In addition to the MAR, MBR, I/OAR, and I/OBR register the following are essential to instruction execution:

Program counter (PC): Contains the address of the next instruction to be fetched.
Instruction register (IR): It contains the instruction most recently fetched.

All processor designs also include a register or set of registers, often known as the program status word (PSW) that contains status information. The PSW typically contains condition codes plus other status information, such as an interrupt enable/disable bit and a kernel/user mode bit, carry bit, auxiliary carry bit.

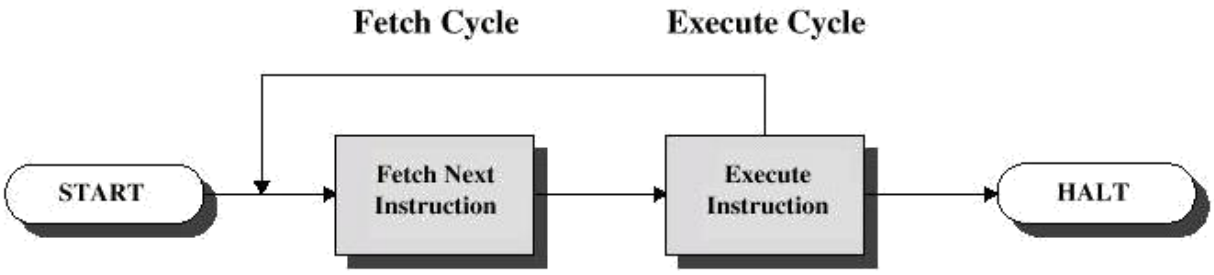
INSTRUCTION EXECUTION:

A program to be executed by a processor consists of a set of instructions stored in Memory. The instruction processing consists of two steps.

The processor reads (fetches) instructions from memory one at a time **(fetch stage)**

Execute the instruction.**(execute stage)**

Program execution consists of repeating the process of instruction fetch and instruction execution
The two steps are referred to as the fetch stage and the execute stage.
The processing required for a single instruction is called an **instruction cycle.**



Instruction Fetch and Execute:

At the beginning of each instruction cycle, the processor fetches an instruction from memory.

The instruction contains bits that specify the action the processor is to take. The processor interprets the instruction and performs the required action. In general, these actions fall into four categories, **Processor-memory:** Data may be transferred from processor to memory or from memory to processor. **Processor-I/O**: Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.

Data processing: The processor may perform some arithmetic or logic operation on data. **Control**: An instruction may specify that the sequence of execution be altered.

(a) Instruction format

(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

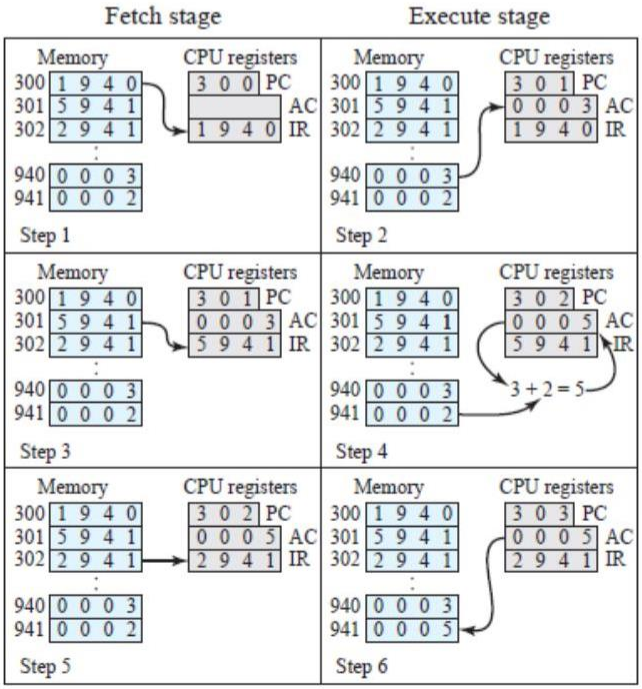


Figure 1.4 **Example of Program Execution** (contents of memory and registers in hexadecimal)

Example:

The processor contains a single data register, called the accumulator (AC).
The instruction format provides 4 bits for the opcode, allowing as many as 24 = 16 different opcodes.
The opcode defines the operation the processor is to perform. The remaining 12 bits of the can be directly addressed.
The program fragment adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the location 941.

The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the IR and the PC is incremented.

The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded from memory. The remaining bits (three hexadecimal digits) specify the address, which is 940.

The next instruction (5941) is fetched from location 301 and the PC is incremented.

The old contents of the AC and the contents of location 941 are added and the result is stored in the AC.

The next instruction (2941) is fetched from location 302 and the PC is incremented.

The contents of the AC are stored in location 941.

I/O Function:

Data can be exchanged directly between an I/O module and the processor.

Just as the processor can initiate a read or write with memory, specifying the address of a memory location, the processor can also read data from or write data to an I/O module.

The processor identifies a specific device that is controlled by a particular I/O module. In some cases, it is desirable to allow I/O exchanges to occur directly with main memory to relieve the processor of the I/O task.

In such a case, the processor grants to an I/O module the authority to read from or write to memory, so that the I/O memory transfer can occur without tying up the processor.

During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange. This operation is known as **direct memory access.**

An interrupt is defined as hardware or software generated event external to the currently executing process that affects the normal flow of the instruction execution.

Interrupts are provided primarily as a way to improve processor utilization

Program	Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, and reference outside a user's allowed memory space.
Timer	Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis.
I/O	Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

Example: Consider a processor that executes a user application. In figure (a) the user program performs a series of WRITE calls interleaved with processing.

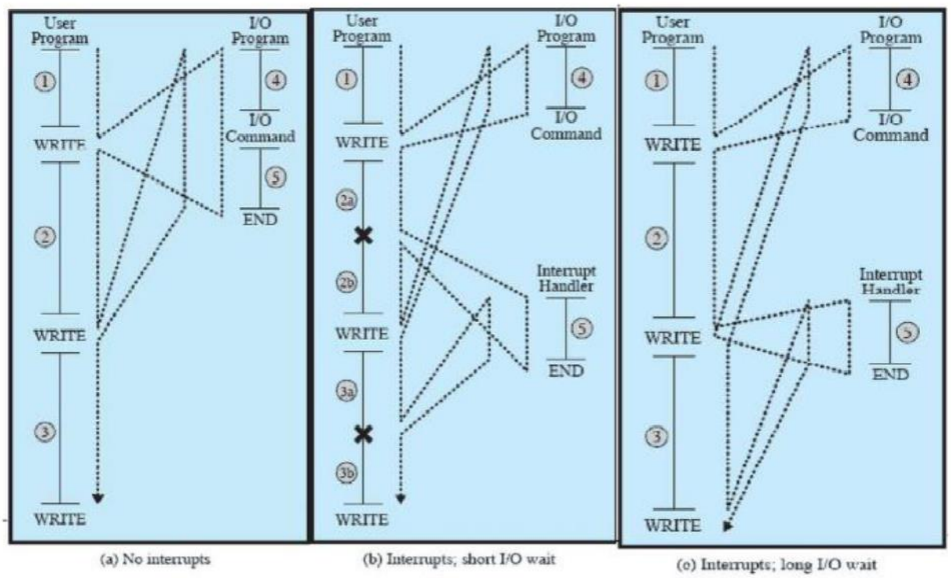
The WRITE calls are the call to an I/O routine that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

A sequence of instructions (4) to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.

The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function. The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.

A sequence of instructions (5) to complete the operation. This may include setting a flag indicating the success or failure of the operation.

- After the first WRITE instruction is encountered, the user program is interrupted and execution continues with the I/O program.
- After the I/O program execution is complete, execution resumes in the user program immediately following the WRITE instruction.



Program Flow of Control without and with Interrupts

Interrupts and the Instruction Cycle:

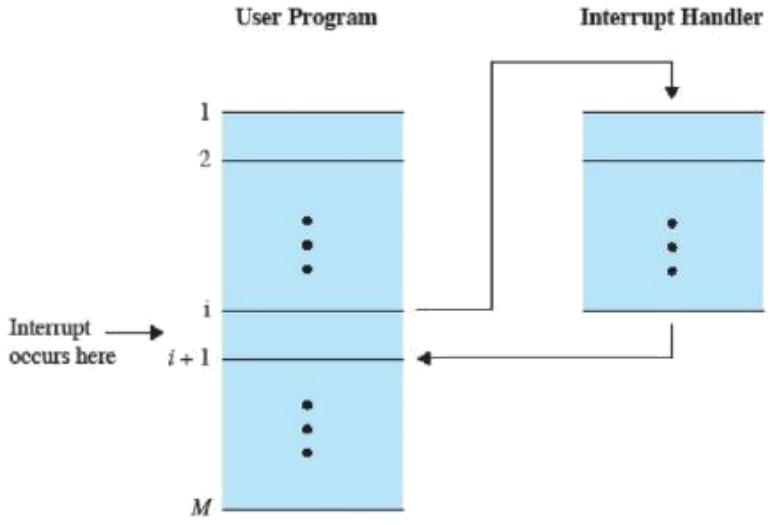
With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.

When the processor encounters the WRITE instruction the I/O program is invoked that consists only of the preparation code and the actual I/O command. After these few instructions have been executed, control returns to the user program.

Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user Program.

When the external device becomes ready to accept more data from the processor, the I/O module for that external device sends an interrupt request signal to the processor.

The processor responds by suspending operation of the current program. This process of branching off to a routine to service that particular I/O device is known as an **interrupt handler** and resuming the original execution after the device is serviced.



Transfer of control via Interrupts

To accommodate interrupts, an interrupt stage is added to the instruction cycle. In the interrupt stage, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal.
If no interrupts are pending, the processor proceeds to the fetch stage and fetches the next instruction of the current program.
If an interrupt is pending, the processor suspends execution of the current program and executes an interrupt-handler routine. This routine determines the nature of the interrupt and performs whatever actions are needed.

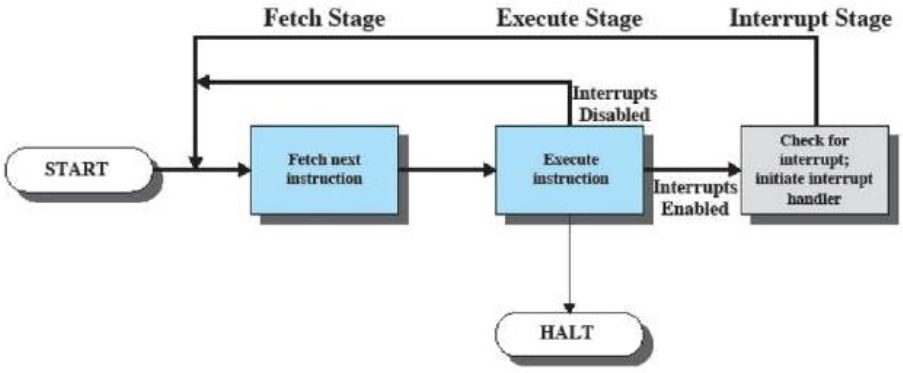


Figure 1.7 Instruction Cycle with Interrupts

Interrupt Processing:

An interrupt triggers a number of events, both in the processor hardware and in software. When an I/O device completes an I/O operation, the following hardware events occurs:

The device issues an interrupt signal to the processor.

The processor finishes execution of the current instruction before responding to the interrupt.

The processor tests for a pending interrupt request, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.

The processor next needs to prepare to transfer control to the interrupt routine. It saves the program status word (PSW) and the location of the next instruction to be executed, which is contained in the program counter. These can be pushed onto a control stack.

The processor then loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt.

Once the program counter has been loaded, the processor proceeds to the next instruction cycle, which begins with an instruction fetch. The contents of the processor registers need to be saved, because these registers may be used by the interrupt handler. So all of these values, plus any other state information, need to be saved.

The interrupt handler may now proceed to process the interrupt.

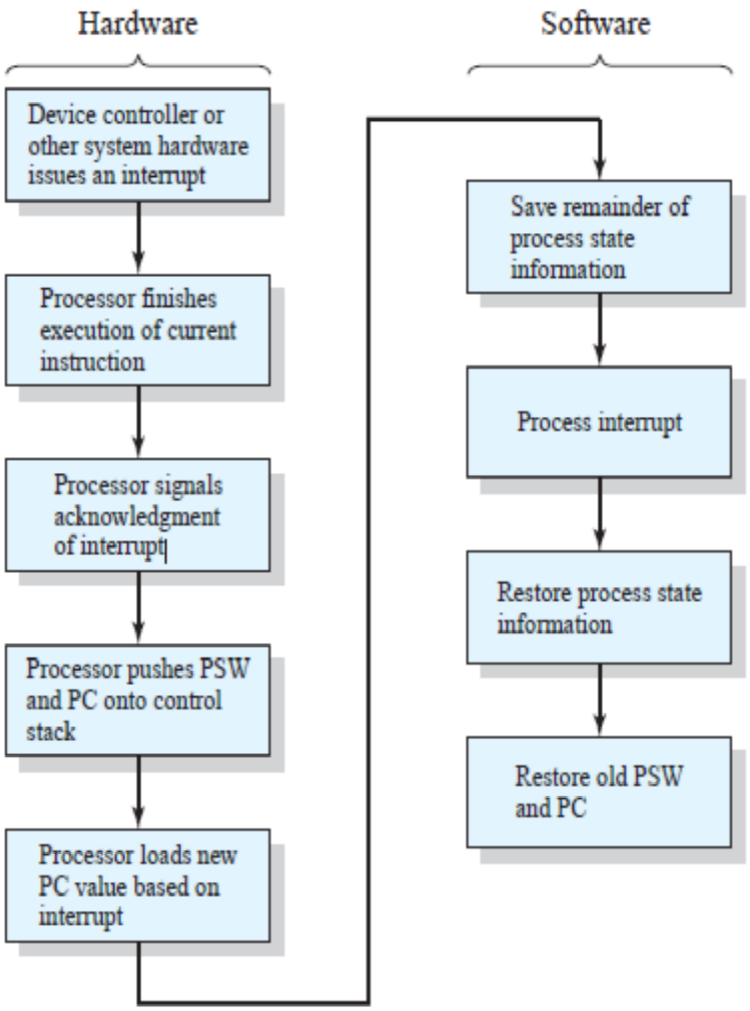
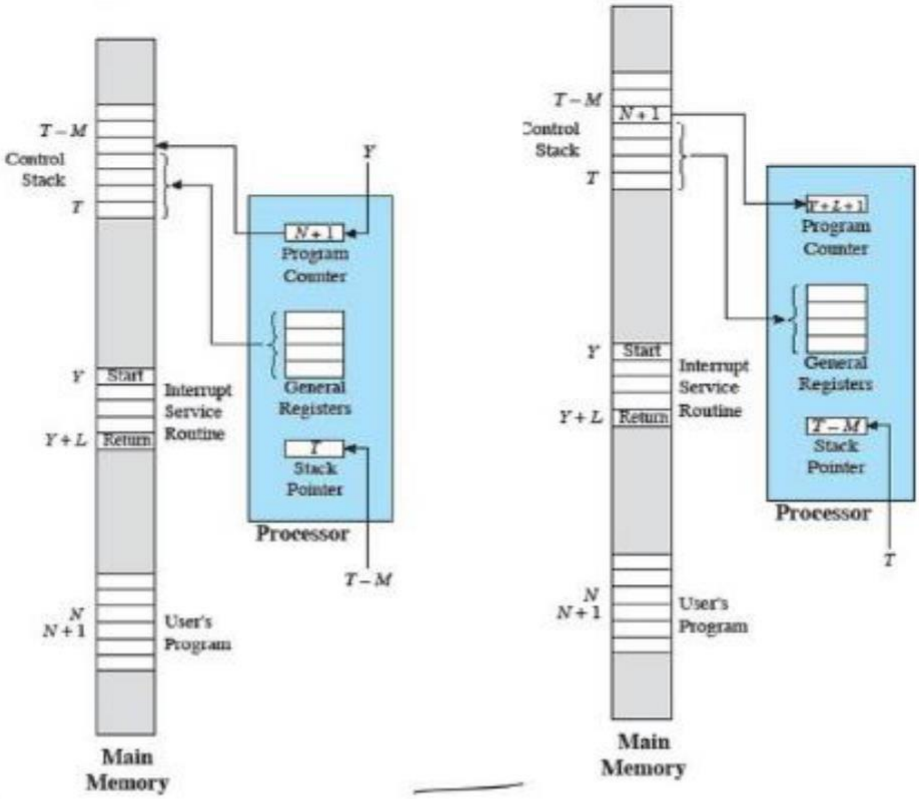


Figure 1.10 Simple Interrupt Processing

When interrupt processing is complete, the saved register values are retrieved from the stack and restored to the registers

The final act is to restore the PSW and program counter values from the stack. As a result, the next instruction to be executed will be from the previously interrupted program.

- The following is an example for a user program that is interrupted after the instruction at location N.
- The contents of all of the registers plus the address of the next instruction (N + 1), a total of M words, are pushed onto the control stack.
- The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.



(a) Interrupt occurs after instruction at location *N*

(b) Return from interrupt

Multiple Interrupts:

One or more interrupts can occur while an interrupt is being processed. This is called as Multiple Interrupts.

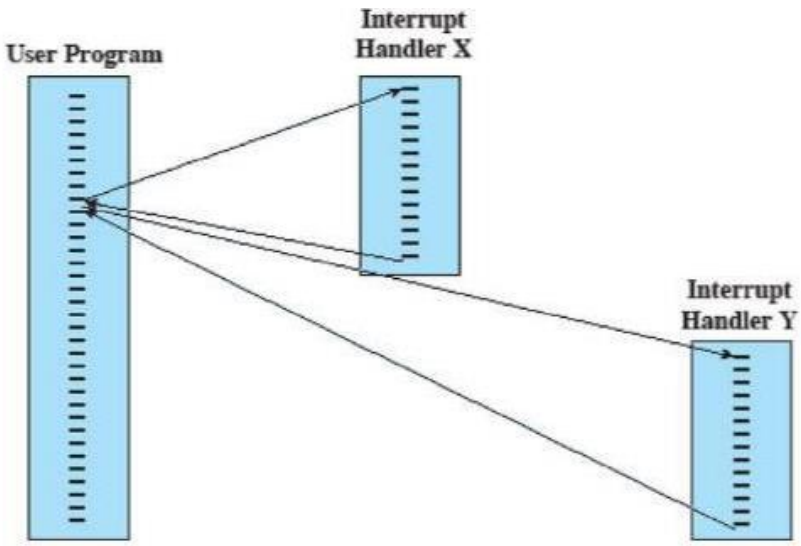
Two approaches can be taken to dealing with multiple interrupts.

Sequential interrupt processing

Nested interrupt processing

Sequential interrupt processing:

The first approach is to disable interrupts while an interrupt is being processed. A disabled interrupt simply means that the processor ignores any new interrupt request signal.



(a)Sequential Interrupt processing.

If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has reenabled interrupts.

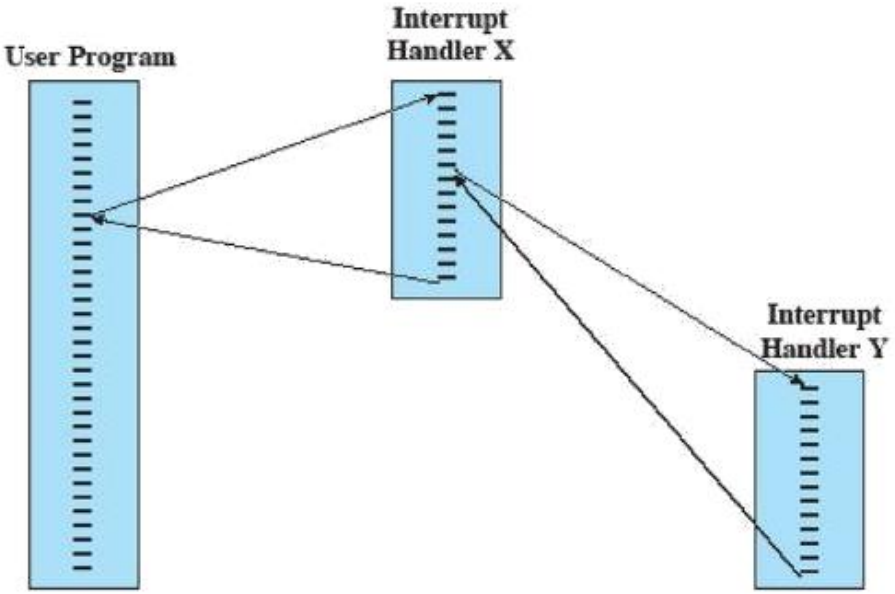
Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the interrupt-handler routine completes, interrupts are reenabled before resuming the user program and the processor checks to see if additional interrupts have occurred.

This approach is simple as interrupts are handled in strict sequential order.

The drawback to this approach is that it does not take into account relative priority or time-critical needs.

Nested interrupt processing:

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be interrupted.



Let us consider a system with three I/O devices. A printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. A user program begins at t=0. At t=10, a printer interrupt occurs.

While this routine is still executing, at t=15, a communications interrupts occur. Because the communications line has highest priority than the printer, the interrupt request is honored.

The printer ISR is interrupted, its state is pushed onto the stack and the execution continues at the communications ISR. While this routine is executing an interrupt occurs at t=20.This interrupt is of lower priority it is simply held and the communications ISR runs to the completion.

When the communications ISR is complete at t=25, the previous processor state is restored which the execution of the printer ISR. However, before even a single instruction in that routine can be executed the processor honors the higher priority disk interrupt and transfers control to the disk ISR. Only when that routine completes (t= 35) the printer ISR is resumed. When the Printer ISR completes at t=40then finally the control returns to the user program.

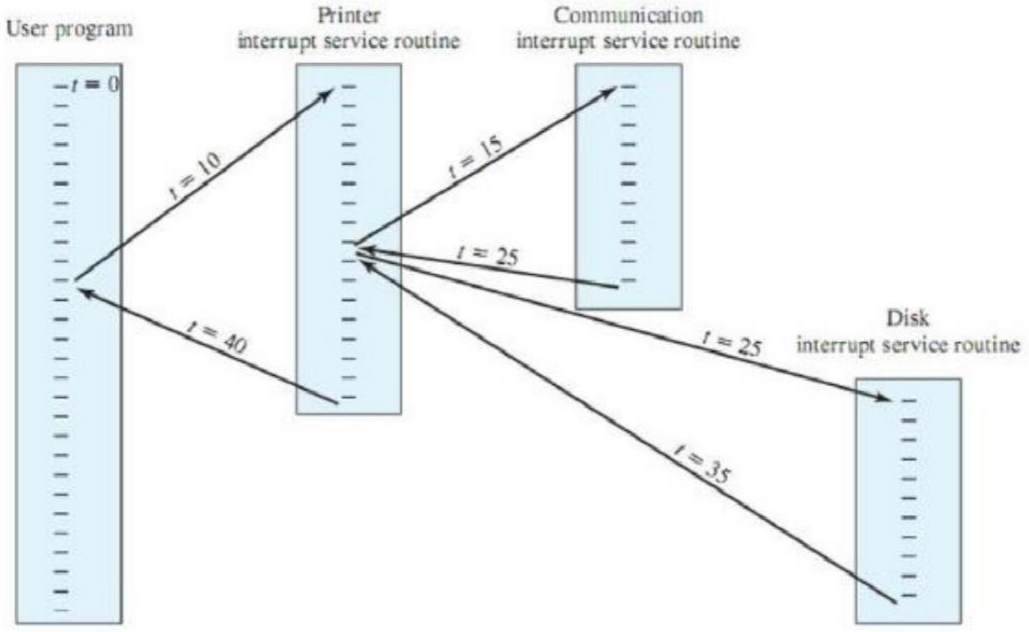


Figure 1.13 Example Time Sequence of Multiple Interrupts

Multiprogramming:

With the use of interrupts, a processor may not be used very efficiently. If the time required to complete an I/O operation is much greater than the user code between I/O calls then the processor will be idle much of the time.

A solution to this problem is to allow multiple user programs to be active at the same time. This approach is called as multiprogramming.

When a program has been interrupted, the control transfers to an interrupt handler, once the interrupt- handler routine has completed, control may not necessarily immediately be returned to the user program that was in execution at the time.

Instead, control may pass to some other pending program with a higher priority. This concept of multiple programs taking turns in execution is known as multiprogramming.

MEMORY HIERARCHY:

To achieve greatest performance, the memory must be able to keep up with the processor.

As the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands.

Thus the cost of memory must be reasonable in relationship to other components.

There is a tradeoff among the three key characteristics of memory: namely, capacity, access time, and cost.

Faster access time, greater cost per bit

Greater capacity, smaller cost per bit

Greater capacity, slower access speed

The designer would like to use memory technologies that provide for large-capacity memory. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with fast access times.

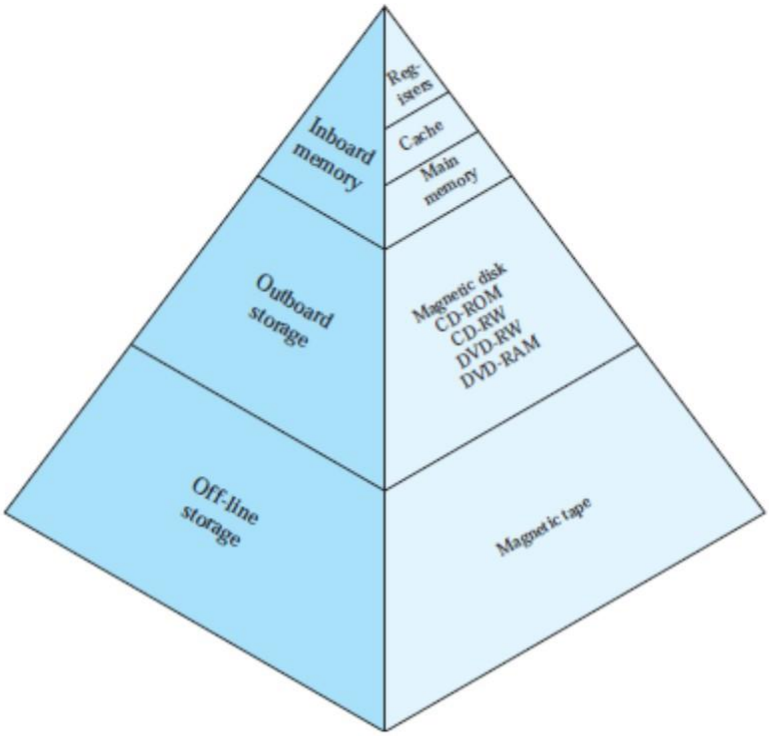
The idea is to not rely on a single memory component but to employ a memory hierarchy. As one goes down the hierarchy, the following occur:

Decreasing cost per bit

Increasing capacity

Increasing access time

Decreasing frequency of access to the memory by the processor



Suppose that the processor has access to two levels of memory. Level 1 contains 1000 bytes and has an access time of 0.1 μs; level 2 contains 100,000 bytes and has an access time of 1 μs.

Assume that if a byte to be accessed is in level 1, then the processor accesses it directly. If it is in level 2, then the byte is first transferred to level 1 and then accessed by the processor.

T1 is the access time to level 1, and T2 is the access time to level 2.

As can be seen, for high percentages of level 1 access, the average total access time is much closer to that of level 1 than that of level 2. Suppose 95% of the memory accesses are found in the cache (H =0.95).Then the average time to access a byte can be expressed as

(0.95) (0.1 μs) + (0.05) (0.1 μs + 1 μs) = 0.095 + 0.055 = 0.15 μs

Thus the result is close to the access time of the faster memory. So the strategy of using two memory levels works in principle.

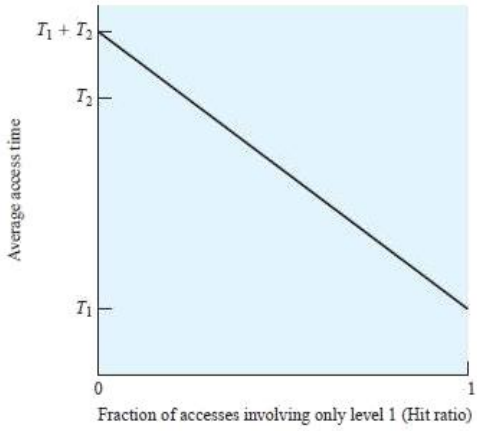


Figure 1.15 **Performance of a Simple Two-Level Memory**

The basis for the validity of condition (Decreasing frequency of access to the memory by the processor) is a principle known as **locality of reference.**

It is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is less than that of the level above.

The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor.

The cache is the next level of memory that is not usually visible to the programmer or, indeed, to the processor. Main memory is usually extended with a higher-speed, smaller cache.

Each location in main memory has a unique address, and most machine instructions refer to one or more main memory addresses. The three forms of memory just described are, typically, volatile and employ semiconductor technology.

External, nonvolatile memory is also referred to as **secondary memory** or **auxiliary memory**. These are used to store program and data files and are usually visible to the programmer only in terms of files and records.

CACHE MEMORY:

A **CPU cache** is a Cache used by the central processing unit (CPU) of a computer to reduce the average time to access data from the main memory.

The cache is a smaller, faster memory which stores copies of the data from main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc.)

The cache memory is small, fast memory between the processor and main memory.

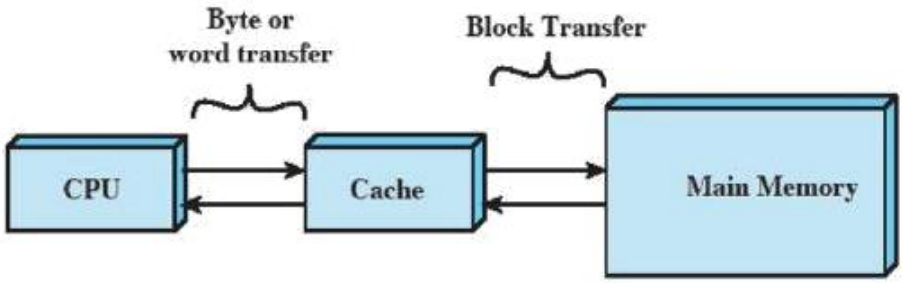


Figure 1.16 Cache and Main Memory

CACHE PRINCIPLES:

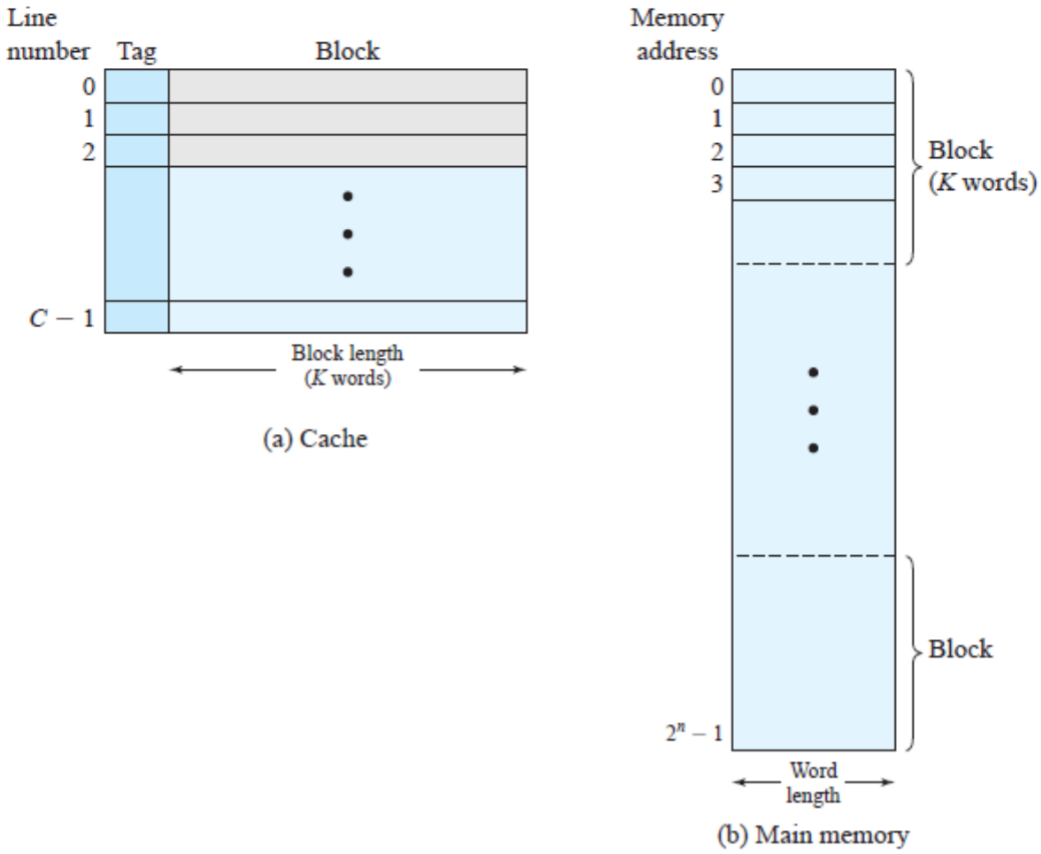
Cache memory provide memory access time similar to that of fastest memories available and at the same time support a large memory size that has the price of less expensive types of semiconductor memories.

The cache contains a copy of a portion of main memory.

When the processor attempts to read a byte or word of memory, a check is made to determine if the byte or word is in the cache.

If so, the byte or word is delivered to the processor.**(CACHE HIT)**

If not, a block of main memory, consisting of some fixed number of bytes, is read into the cache and then the byte or word is delivered to the processor. (**CACHE MISS**)



Cache / Main memory

In the above diagram the Main memory consists of up to 2n addressable words, with each word having a unique n-bit address. This memory is considered to consist of a number of fixed length blocks of K words each. That is, there are M= 2n/K blocks. Cache consists of C slots of K words each, and the number of slots is considerably less than the number of main memory blocks (C << M).

If a word in a block of memory that is not in the cache is read, that block is transferred to one of the slots of the cache.

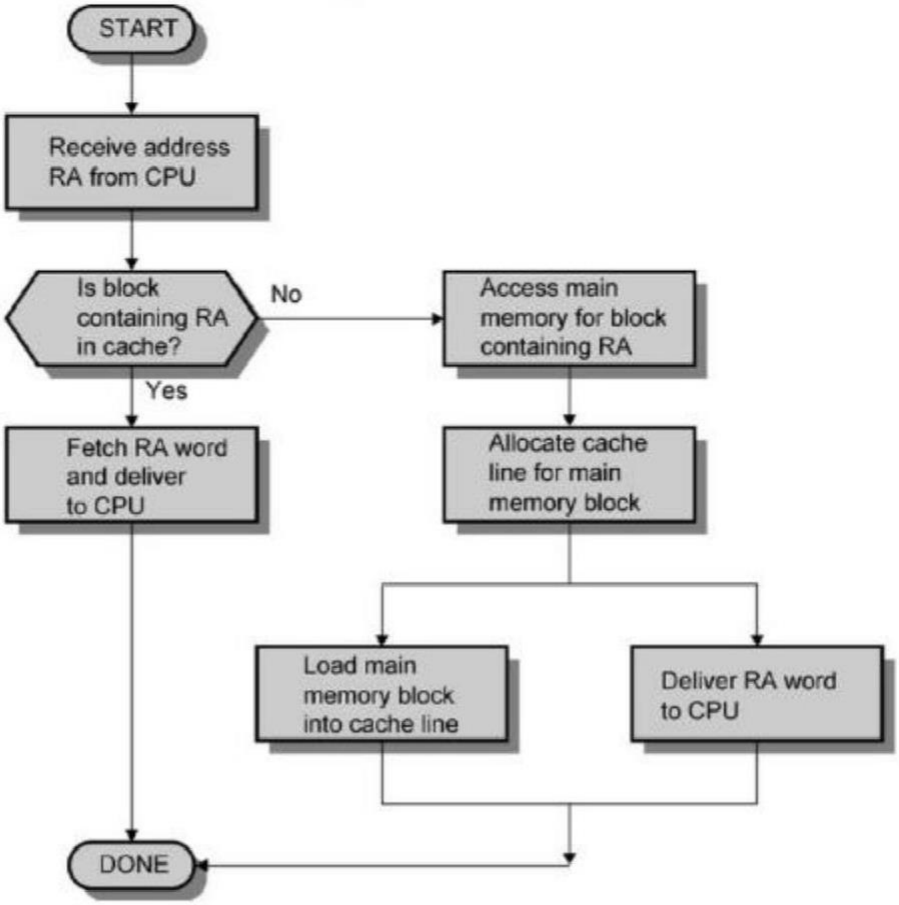
Each slot includes a tag that identifies which particular block is currently being stored. The tag is usually some number of higher-order bits of the address and refers to all addresses that begin with that sequence of bits.

Example: suppose that we have a 6-bit address and a 2-bit tag. The tag 01 refers to the block of locations with the following addresses: 010000, 010001,010010, 010011, 010100, 010101, 010110, 010111, 011000, 011001, 011010, 011011,011100, 011101, 011110, 011111.

READ OPERATION IN A CACHE:

The processor generates the real address, RA, of a word to be read.

If the word is contained in the cache, it is delivered to the processor. Otherwise, the block containing that word is loaded into the cache and the word is delivered to the processor.



CACHE READ OPERATION

CACHE DESIGN:

The key elements of cache design includes,

Cache size
Block Size
Mapping Function
Replacement algorithm
Write policy

The issue with cache size is that small caches can have a significant impact on performance.

Another size issue is that of block size: As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality:

As the block size increases, more useful data are brought into the cache.

The hit ratio will begin to decrease, however, as the block becomes even bigger and the probability of using the newly fetched data becomes less than the probability of reusing the data that have to be moved out of the cache to make room for the new block.

When a new block of data is read into the cache, the mapping function determines which cache location the block will occupy. When one block is read in, another may have to be replaced. The replacement algorithm chooses, within the constraints of the mapping function, which block to replace when a new block is to be loaded into the cache and the cache already has all slots filled with other blocks.

A block that is least likely to be needed again in the near future will be replaced. An effective strategy is to replace the block that has been in the cache longest with no reference to it. This policy is referred to as the **least-recently-used (LRU) algorithm**.

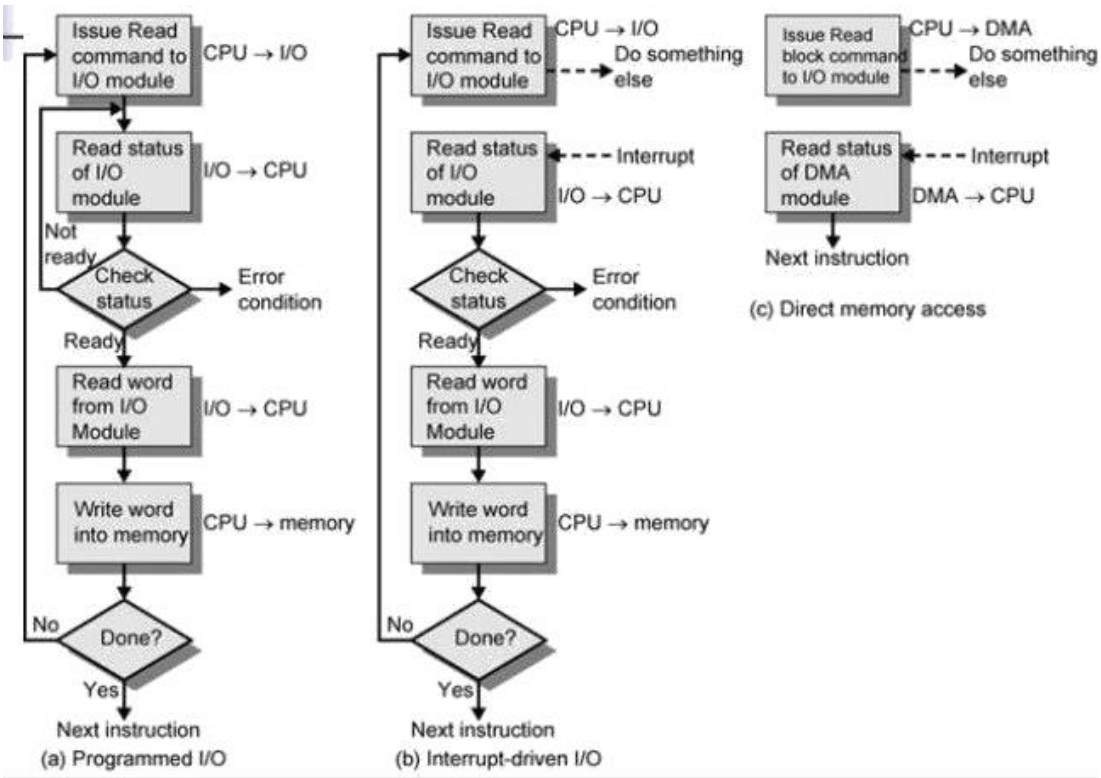
When a system writes data to cache, it must at some point write that data to the backing store as well. The timing of this write is controlled by what is known as the **write policy.**There are two basic writing approaches:

Write-through: write is done synchronously both to the cache and to the backing store.

Write-back (or write-behind): initially, writing is done only to the cache. The write to the backing store is postponed until the cache blocks containing the data are about to be modified/replaced by new content.

I/O COMMUNICATION TECHNIQUES

I/O Communication techniques determine the communication between the memory and the I/O devices.



Three techniques are possible for I/O operations:

Programmed I/O
Interrupt-driven I/O
Direct memory access (DMA)

Programmed I/O:

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module.

The I/O module performs the requested action and takes no action to alert the processor and it does not interrupt the processor.

The processor periodically checks the status of the I/O module until it finds that the operation is complete.

The processor is responsible for extracting data from main memory for output and storing data in main memory for input.

Control: Used to activate an external device and tell it what to do.

Status: Used to test various status conditions associated with an I/O module and its peripherals.

Transfer: Used to read and/or write data between processor registers and external devices.

Interrupt-Driven I/O:

- An alternative to Programmed I/O is for the processor to issue an I/O command to a module and then go on to do some other useful work.

The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor.
The processor then executes the data transfer and then resumes its former processing.
The processor issues a READ command. The I/O module receives a READ command from the processor and then proceeds to read data in from the device.
Once the data are in the I/O module's data register the module signals an interrupt to the processor over a control line.
When the interrupt from the I/O module occurs, the processor saves the context of the program it is currently executing and begins to execute an interrupt-handling program that processes the interrupt.
Interrupt-driven I/O is more efficient than programmed I/O because it eliminates needless waiting.

DIRECT MEMORY ACCESS

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module.
When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA).
The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module.

When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:

Whether a read or write is requested
The address of the I/O device involved
The starting location in memory to read data from or write data to
The number of words to be read or written

The processor then continues with other work. It has delegated this I/O operation to the DMA module, and that module will take care of it.
The DMA module transfers the entire block of data, one word at a time, directly to or from memory without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor.
Thus the processor is involved only at the beginning and end of the transfer.

MULTIPROCESSOR AND MULTICORE ORGANIZATION:

A processor executes programs by executing machine instructions in sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results).
In order to achieve performance and reliability, the concept of parallelism has been introduced in the computers which include symmetric multiprocessors, multicore computers and clusters.
The multiple-processor systems in use today are of two types.

Asymmetric multiprocessing, in which each processor is assigned a specific task.A boss processor, controls the system; the other processors either look to the boss for instruction or have predefined tasks. **This scheme defines a boss–worker relationship.** The boss processor schedules and allocates work to the worker processors.

Symmetric multiprocessing (SMP), in which each processor performs all tasks within the operating system. SMP means that all processors are peers; no boss–worker relationship exists between processors.

Symmetric Multiprocessors:

An SMP can be defined as a stand-alone computer system with the following characteristics:

There are two or more similar processors of comparable capability.

These processors share the same main memory and I/O facilities and are interconnected by a bus or other internal connection scheme, such that memory access time is approximately the same for each processor.

All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.

All processors can perform the same functions

The system is controlled by an integrated operating system that provides interaction between processors and their programs at the job, task, file, and data element levels.

In an SMP, individual data elements can constitute the level of interaction, and there can be a high degree of cooperation between processes.

Advantages of Symmetric multiprocessors:

Increased throughput.

By increasing the number of processors, we expect to get more work done in less time.

If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.

2. **Economy of scale**. Multiprocessor systems can cost less than equivalent multiple single-processor systems, because they can share peripherals, mass storage, and power supplies. If several programs operate on the same set of data, it is cheaper to store those data on one disk and to have all the processors share them than to have many computers with local disks and many copies of the data.
3. **Increased reliability**. If functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slow it down. If we have ten processors and one fails, then each of the remaining nine processors can pick up a share of the work of the failed processor. Thus, the entire system the entire system runs slower, rather than failing altogether. Increased reliability of a computer system is crucial in many applications.

The ability to continue providing service proportional to the level of surviving hardware is called **Graceful Degradation**. Some systems go beyond graceful degradation and are called fault tolerant, because they can suffer a failure of any single component and still continue operation.

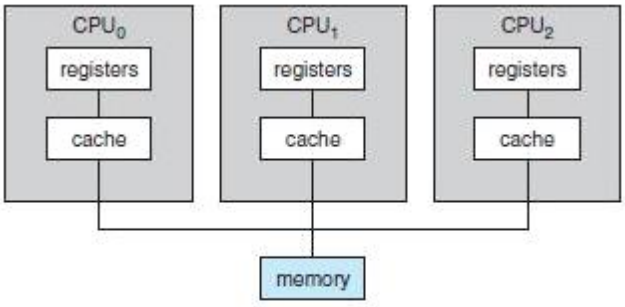


Figure 1.6 Symmetric multiprocessing architecture.

The Disadvantage of symmetric multiprocessor includes

If one processor fails then it will affect in the speed

Multiprocessor systems are expensive

Complex OS is required 4) Large main memory required.

An attractive feature of an SMP is that the existence of multiple processors is transparent to the user. The operating system takes care of scheduling of tasks on individual processors and of synchronization among processors.

There are multiple processors, each of which contains its own control unit, arithmetic logic unit, and registers.

Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism; a shared bus is a common facility.

The processors can communicate with each other through memory (messages and status information left in shared address spaces).

MULTICORE ORGANIZATION:

A dual-core design contains two cores on the same chip.

In this design, each core has its own register set as well as its own local cache. Other designs might use a shared cache or a combination of local and shared caches.

Performance has also been improved by the increased complexity of processor design to exploit parallelism in instruction execution and memory access.

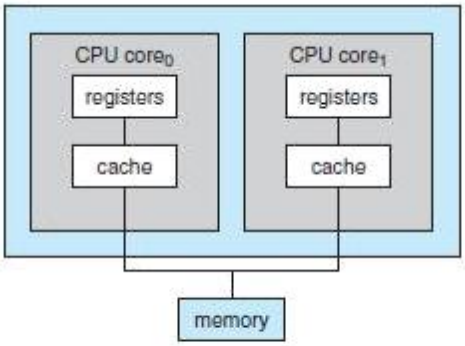


Figure 1.7 A dual-core design with two cores placed on the same chip.

An example of a multicore system is the Intel Core i7, which includes four x86 processors, each with a dedicated L2 cache, and with a shared L3 cache

OPERATING SYSTEM OVERVIEW:

An OS is defined as a System program that controls the execution of application programs and acts as an interface between applications and the computer hardware.

OPERATING SYSTEM OBJECTIVES AND FUNCTIONS:

An operating system is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. It can be thought of as having three objectives:

Convenience
Efficiency
Ability to evolve

The three other aspects of the operating system are

The operating system as a user or computer interface
The operating system as a resource manager
Ease of evolution of an operating system.

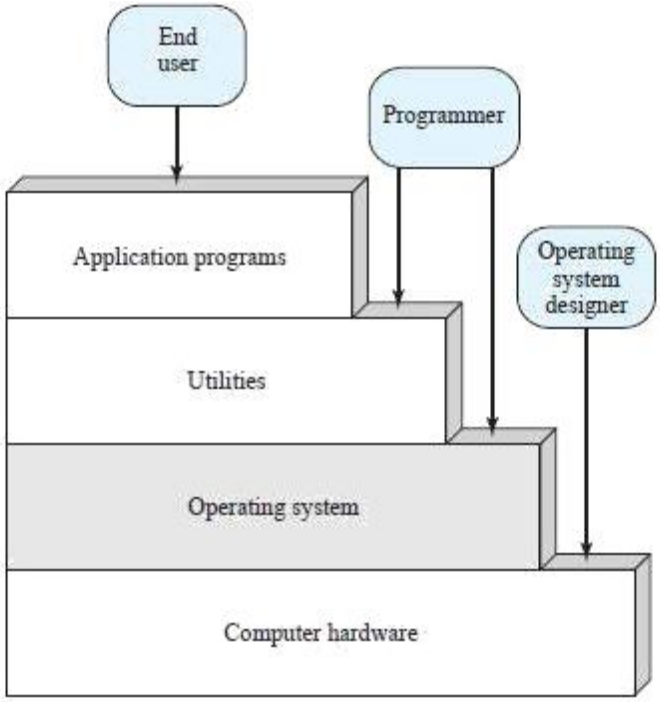
The Operating System as a User/Computer Interface

The user of those applications, the end user, generally is not concerned with the details of computer hardware.

An application can be expressed in a programming language and is developed by an application programmer.

A set of system programs referred to as utilities implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices.

The most important collection of system programs comprises the OS. The OS masks the details of the hardware from the programmer and provides the programmer with a convenient interface for using the system.



Briefly, the OS typically provides services in the following areas:

Program development
Program execution
Access to I/O devices
Controlled access to files
System access
Error detection and response
Accounting:

The Operating System as Resource Manager

A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions. The OS is responsible for managing these resources.

The OS functions in the same way as ordinary computer software; that is, it is a program or suite of programs executed by the processor.

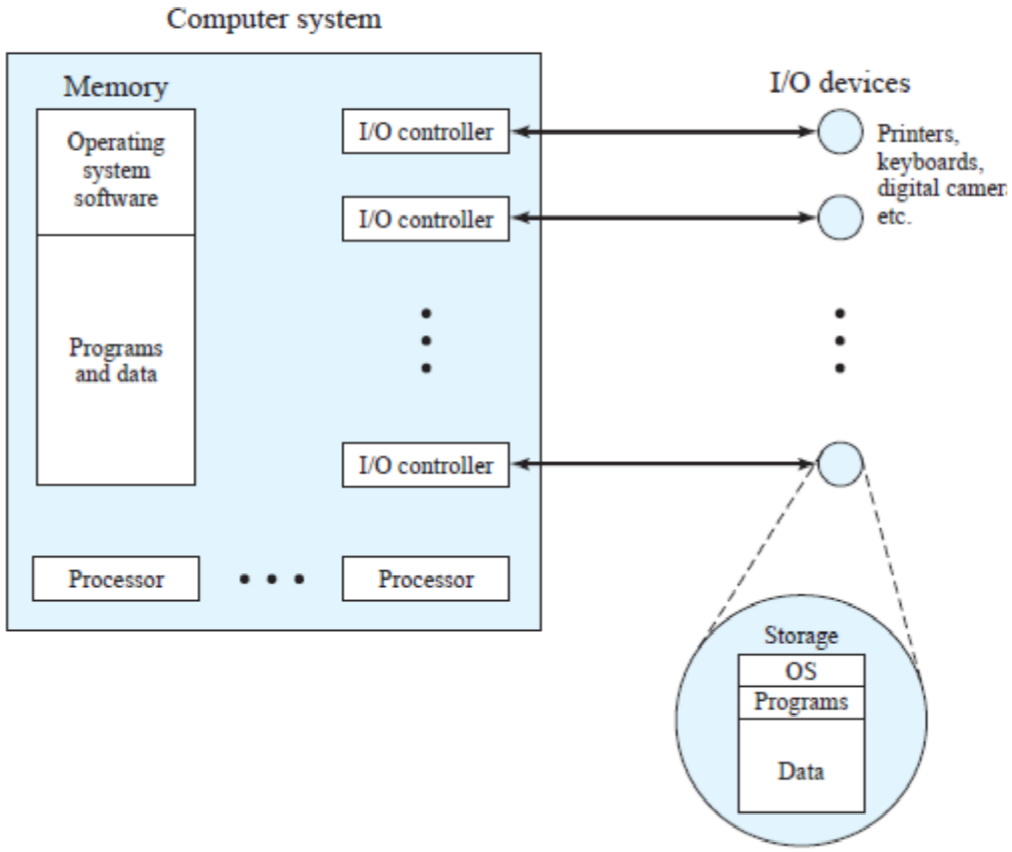
The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

The OS directs the processor in the use of the other system resources and in the timing of its execution of other programs.

A portion of the OS is in main memory. This includes the kernel, or nucleus, which contains the most frequently, used functions in the OS.The remainder of main memory contains user programs and data.

The allocation of this resource (main memory) is controlled jointly by the OS and memory management hardware in the processor.

The OS decides when an I/O device can be used by a program in execution and controls access to and use of files.



The processor itself is a resource, and the OS must determine how much processor time is to be devoted to the execution of a particular user program. In the case of a multiple-processor system, this decision must span all of the processors.

Ease of Evolution of an Operating System

A major operating system will evolve over time for a number of reasons:

Hardware upgrades plus new types of hardware
New services: OS expands to offer new services in response to user demands.
Fixes: Any OS has faults.

The functions of operating system includes,

Process management
Memory management
File management
I/O management
Storage management.

EVOLUTION OF OPERATING SYSTEM:

An operating system acts as an intermediary between the user of a computer and the computer hardware. The evolution of operating system is explained at various stages.

Serial Processing
Simple Batch Systems
Multiprogrammed batch systems.
Time sharing systems

Serial processing

During 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS.

Programs in machine code were loaded via the input device (e.g., a card reader).
If an error halted the program, the error condition was indicated by the lights.
If the program proceeded to a normal completion, the output appeared on the printer.

Scheduling: Most installations used a hardcopy sign-up sheet to reserve computer time. A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer processing time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.

Setup time: A single program, called a job, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program) and then loading and linking together the object program and common functions. Thus, a considerable amount of time was spent just in setting up the program to run.

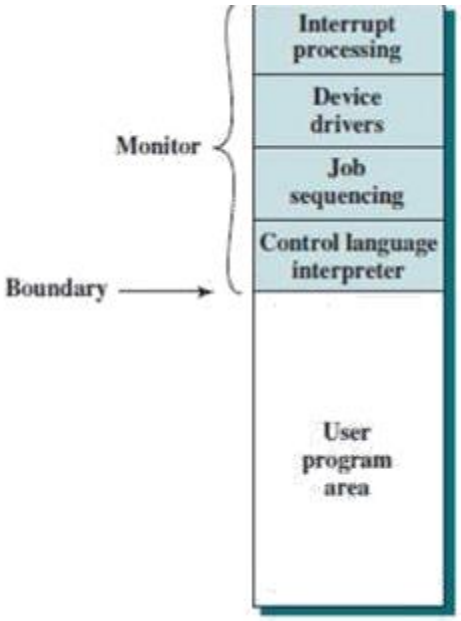
This mode of operation could be termed serial processing, reflecting the fact that users have access to the computer in series

Simple Batch Systems

The central idea behind the simple batch-processing scheme is the use of a piece of software known as the **monitor.**

With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor.

Each program is constructed to branch back to the monitor when it completes processing, and the monitor automatically begins loading the next program.



Memory Layout for a Resident Monitor

The monitor controls the sequence of events. For this the monitor must always be in main memory and available for execution. That portion is referred to as the resident monitor.

The monitor reads in jobs one at a time from the input device .As it is read in, the current job is placed in the user program area, and control is passed to this job.

Once a job has been read in, the processor will encounter a branch instruction in the monitor that instructs the processor to continue execution at the start of the user program. The processor will then execute the instructions in the user program until it encounters an ending or error condition.

When the job is completed, it returns control to the monitor, which immediately reads in the next job. The results of each job are sent to an output device, such as a printer, for delivery to the user.

The monitor performs a scheduling function: A batch of jobs is queued up, and jobs are executed as rapidly as possible, with no intervening idle time.

With each job, instructions are included in a form of job control language (JCL) which are denoted by the beginning $. This is a special type of programming language used to provide instructions to the monitor. The overall format of the job is given as

```
$JOB
$FTN
•
•          } FORTRAN instructions
•
$LOAD
$RUN
•
•          } Data
•
$END
```

The hardware features that are added as a part of simple batch systems include,

Memory protection
Timer
Privileged instructions
Interrupts.

The memory protection leads to the concept of dual mode operation.

User Mode
Kernel Mode.

Thus the simple batch system improves utilization of the computer

Multiprogrammed Batch Systems:

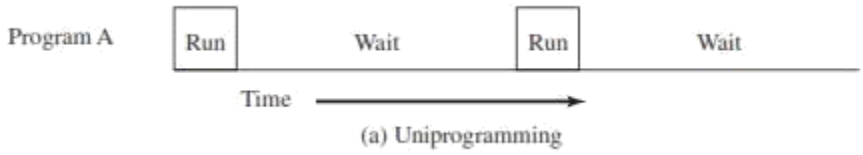
Even in simple batch operating system, the processor is often idle. The problem is that I/O devices are slow compared to the processor.

Let us consider a program that processes a file of records and performs, on average, 100 machine instructions per record. The computer spends over 96% of its time waiting for I/O devices to finish transferring data to and from the file.

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
Total	31 μs

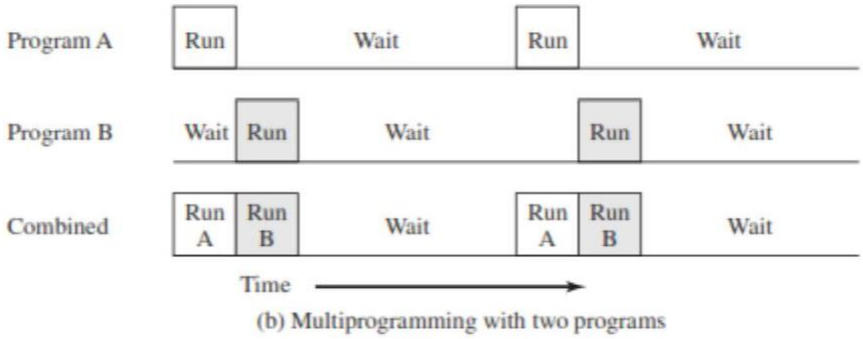
$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

In uniprogramming we will have a single program in the main memory. The processor spends a certain amount of time executing, until it reaches an I/O instruction. It must then wait until that I/O instruction concludes before proceeding. This inefficiency is not necessary.

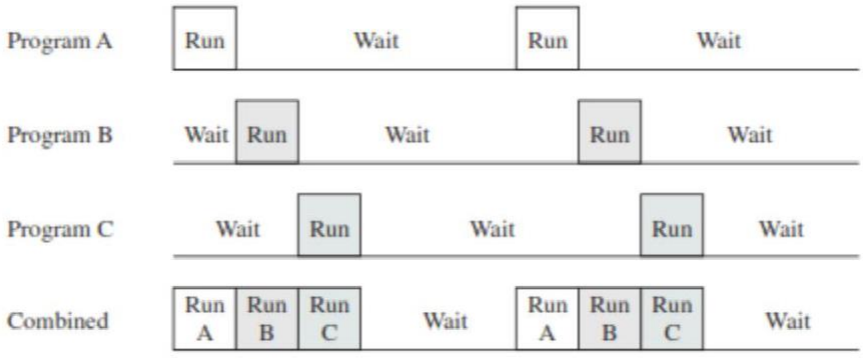


(a) Uniprogramming

In Multiprogramming we will have OS and more user programs. When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O.This approach is known as **multiprogramming, or multitasking**.



(b) Multiprogramming with two programs



(c) Multiprogramming with three programs

Figure 2.5 Multiprogramming Example

The most notable feature that is useful for multiprogramming is the hardware that supports I/O interrupts and DMA (direct memory access).

With interrupt-driven I/O or DMA, the processor can issue an I/O command for one job and proceed with the execution of another job while the I/O is carried out by the device controller.

When the I/O operation is complete, the processor is interrupted and control is passed to an interrupt-handling program in the OS.The OS will then passes control to another job.

Multiprogramming operating systems are fairly sophisticated compared to single-program, or uniprogramming, systems. To have several jobs ready to run, they must be kept in main memory, requiring some form of memory management.

In addition, if several jobs are ready to run, the processor must decide which one to run, this decision requires an algorithm for scheduling.

Time-Sharing Systems:

In time sharing systems the processor time is shared among multiple users.

In a time-sharing system, multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation.

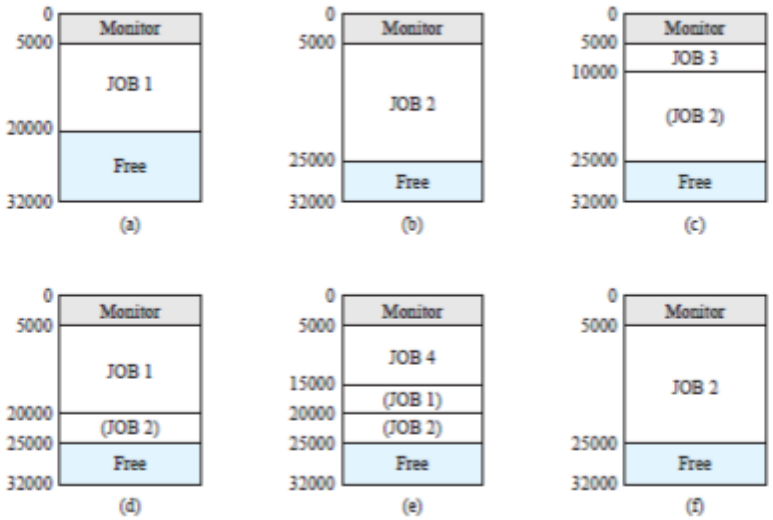
If there are n users actively requesting service at one time, each user will only see on the average 1/n of the effective computer capacity.

Batch Multiprogramming Vs Time Sharing systems

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

One of the first time-sharing operating systems to be developed was the Compatible Time-Sharing System (CTSS)
The system ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming
5000 of that. When control was to be assigned to an interactive user, the user's program and data were loaded into the remaining 27,000 words of main memory.
A program was always loaded to start at the location of the 5000th word
A system clock generated interrupts at a rate of approximately one every 0.2 seconds.
At each clock interrupt, the OS regained control and could assign the processor to another user. This technique is known as **time slicing.**

Example: Assume that there are four interactive users with the following memory requirements, in words:
JOB1: 15,000
JOB2: 20,000
JOB3: 5000
JOB4: 10,000



Initially, the monitor loads JOB1 and transfers control to it.

Later, the monitor decides to transfer control to JOB2. Because JOB2 requires more memory than JOB1, JOB1 must be written out first, and then JOB2 can be loaded.

Next, JOB3 is loaded in to be run. However, because JOB3 is smaller than JOB2, a portion of

Later, the monitor decides to transfer control back to JOB1.An additional portion of JOB2 must be written out when JOB1 is loaded back into memory.

When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained.

At this point, if either JOB1 or JOB2 is activated, only a partial load will be required. In this example, it is JOB2 that runs next. This requires that JOB4 and the remaining resident portion of JOB1 be written out and that the missing portion of JOB2 be read in.

COMPUTER SYSTEM ORGANIZATION:

Computer system organization deals with the structure of the computer system.

Computer system operation:

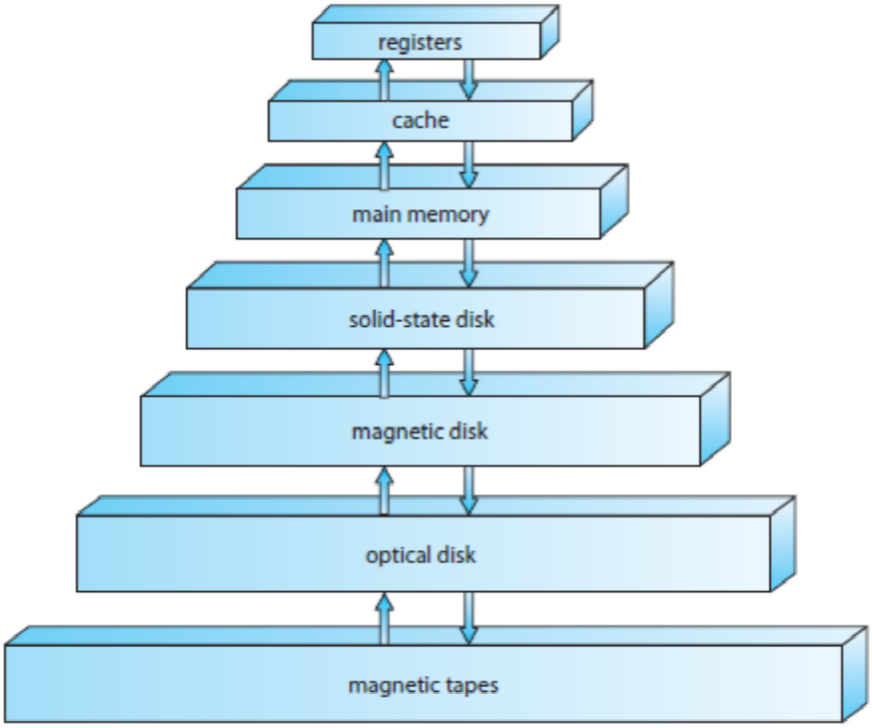
A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory.
For a computer to start running when it is powered up or rebooted—it needs to have an initial program to run. This initial program is called as the Bootstrap program.
It is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), known by the general term **firmware.**

The bootstrap loader It initializes all aspects of the system, from CPU registers to device controllers to memory contents.
The bootstrap program loads the operating system and start executing that system.
Once the kernel is loaded and executing, it can start providing services to the system and its users. When is the system is booted it waits for some event to occur.
The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software.
When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. That contains the starting address of the service routine for the interrupt.
The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

Storage structure:

The CPU can load instructions only from memory, so any programs to run must be stored in main memory.
Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory**
ROM is a read only memory that is used to store the static programs such as bootstrap loader.
All forms of memory provide an array of bytes. Each byte has its own address. The operations are done through load or store instructions.
The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory.
Ideally, we want the programs and data to reside in main memory permanently.

Main memory is usually too small to store all needed programs and data permanently
Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.
Most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.
The wide variety of storage systems can be organized in a hierarchy according to speed and cost.
The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases
Volatile storage loses its contents when the power to the device is removed so that the data must be written to **nonvolatile storage** for safekeeping.
Caches can be installed to improve performance where a large difference in access time or transfer rate exists between two components.



I/O Structure:

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system.

A general-purpose computer system consists of CPUs and multiple device controllers that are connected through a common bus. Each device controller is in charge of a specific type of device.

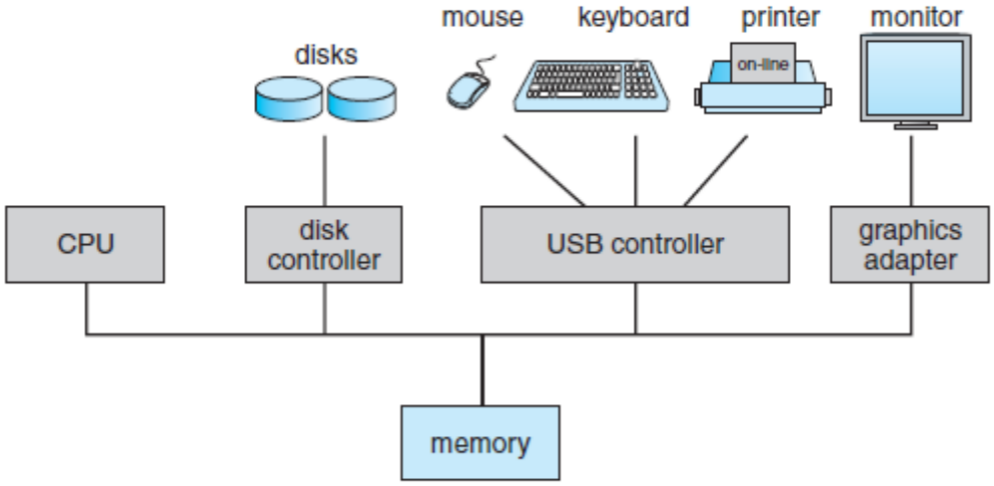
The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage

Operating systems have a **device driver** for each device controller. This device driver understands the device controller and provides the rest of the operating system with a uniform interface to the device.

To start an I/O operation, the device driver loads the appropriate registers within the device controller.

The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. This is called as interrupt driven I/O.

The direct memory access I/O technique transfers a block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed,



OPERATING SYSTEM STRUCTURE:

The operating systems are large and complex. A common approach is to partition the task into small components, or modules, rather than have one **monolithic** system.

The structure of an operating system can be defined the following structures.

- Simple structure
- Layered approach
- Microkernels
- Modules
- Hybrid systems

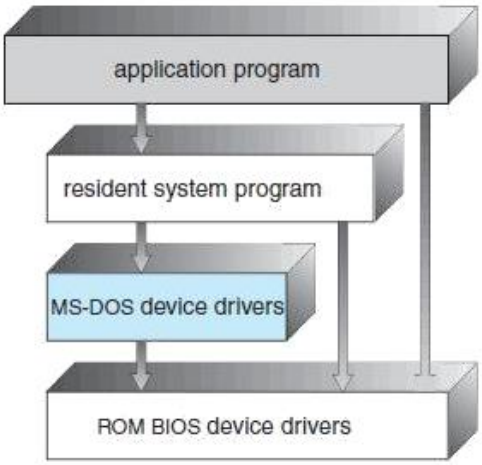
Simple structure:

The Simple structured operating systems do not have a well-defined structure. These systems will be simple, small and limited systems.

Example: MS-DOS.

In MS-DOS, the interfaces and levels of functionality are not well separated.

In MS-DOS application programs are able to access the basic I/O routines. This causes the entire systems to be crashed when user programs fail.



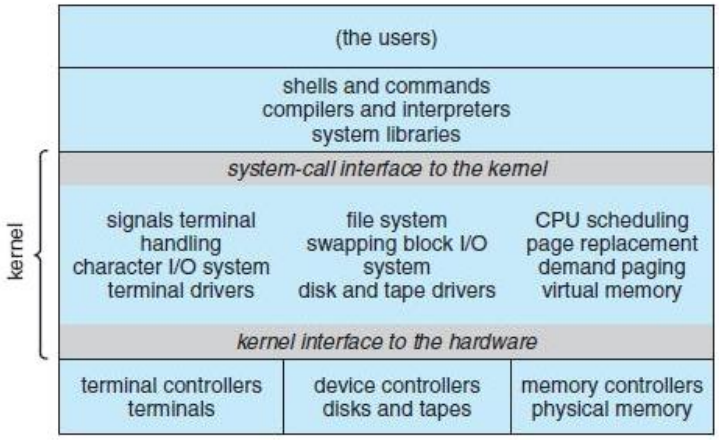
Example: Traditional UNIX OS

It consists of two separable parts: the kernel and the system programs.

The kernel is further separated into a series of interfaces and device drivers

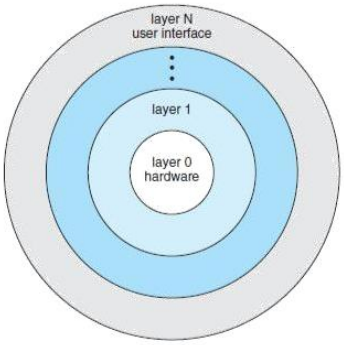
The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

This monolithic structure was difficult to implement and maintain.



Layered approach:

A system can be made modular in many ways. One method is the **layered approach**, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer is the user interface.



An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data.

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions (operations) and services of only lower-level layers.

Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do.

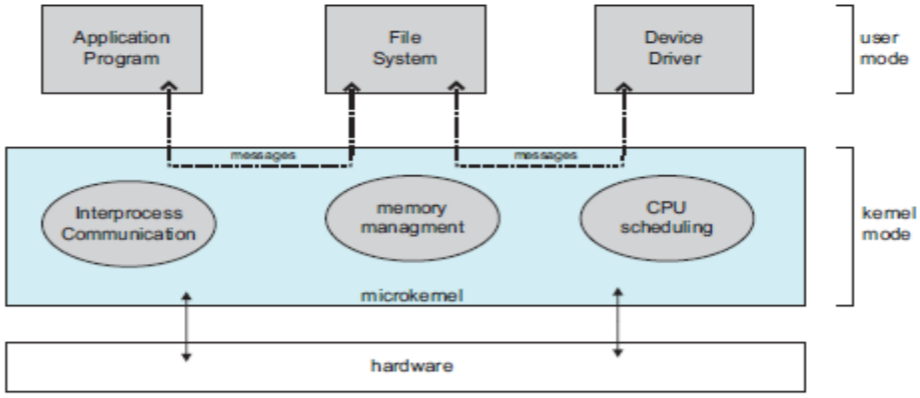
The major difficulty with the layered approach involves appropriately defining the various layers because a layer can use only lower-level layers.

A problem with layered implementations is that they tend to be less efficient than other types.

Microkernels:

In the mid-1980s, researchers at Carnegie Mellon University developed an operating system called **Mach** that modularized the kernel using the **microkernel** approach.

This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs.



Microkernel provide minimal process and memory management, in addition to a communication facility.

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.

The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.

The performance of microkernel can suffer due to increased system-function overhead.

Modules:

The best current methodology for operating-system design involves using **loadable kernel modules**

The kernel has a set of core components and links in additional services via modules, either at boot time or during run time.

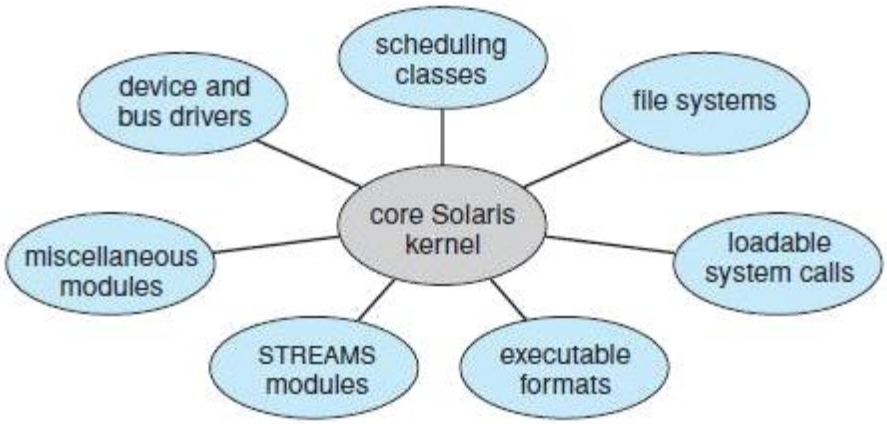
The kernel provides core services while other services are implemented dynamically, as the kernel is running.

Linking services dynamically is more comfortable than adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.

Example: Solaris OS

The Solaris operating system structure is organized around a core kernel with seven types of loadable kernel modules:

Scheduling classes
File systems
Loadable system calls



Executable formats
STREAMS modules
Miscellaneous
Device and bus drivers

Hybrid Systems:

The Operating System combines different structures, resulting in hybrid systems that address performance, security, and usability issues.

They are monolithic, because having the operating system in a single address space provides very efficient performance.

However, they are also modular, so that new functionality can be dynamically added to the kernel.

Example: Linux and Solaris are monolithic (simple) and also modular, IOS.
Apple IOS Structure

OPERATING SYSTEM OPERATIONS:

The operating system and the users share the hardware and software resources of the computer system, so we need to make sure that an error in a user program could cause problems only for the one program running.

Without protection against these sorts of errors, either one erroneous program might modify another program, the data of another program, or even the operating system itself.

Dual-Mode and Multimode Operation:

In order to ensure the proper execution of the operating system, we must be able to distinguish between the execution of operating-system code and user defined code.

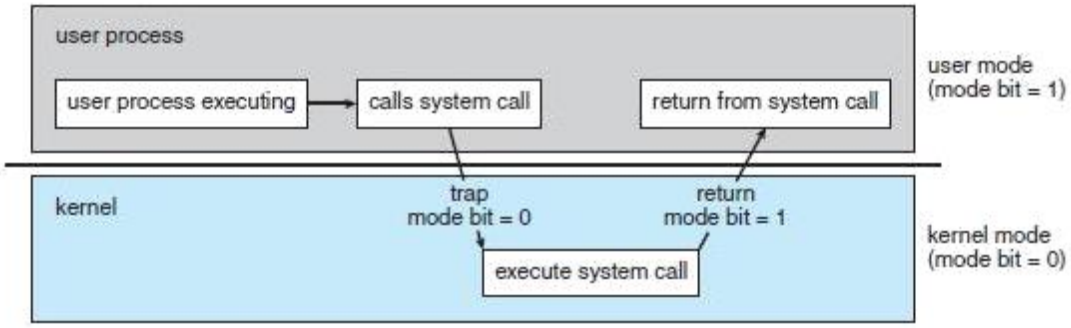
The computer systems provide hardware support that allows us to differentiate among various modes of execution.

User mode
Kernel mode(Supervisor mode or system mode or privileged mode)

A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1)

The mode bit, can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

When the computer system is executing on behalf of a user application, the system is in user mode and when a user application requests a service from the operating system the system must make a transition from user to kernel mode



At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode.

Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0).

The dual mode of operation provides us with the means for protecting the operating system from errant users— and errant users from one another

The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system

System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.

The lack of a hardware-supported dual mode can cause serious shortcomings in an operating system.MS-DOS was written for the Intel 8088 architecture, which has no mode bit and therefore no dual mode

Once hardware protection is in place, it detects errors that violate modes. These errors are normally handled by the operating system.

Timer:

The operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system.

A timer can be set to interrupt the computer after a specified period. A **variable timer** is generally implemented by a fixed-rate clock and a counter.

The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs

SYSTEM CALLS:

The system call provides an interface to the operating system services.

Application developers often do not have direct access to the system calls, but can access them through an application programming interface (API). The functions that are included in the API invoke the actual system calls.

Systems execute thousands of system calls per second. Application developers design programs according to an **application programming interface (API)**.

For most programming languages, the Application Program Interface provides a **system call interface** that serves as the link to system calls made available by the operating system

The system-call interface intercepts function calls in the API and invokes the necessary system calls within the Operating system.

Example: System calls for writing a simple program to read data from one file and copy them to another file

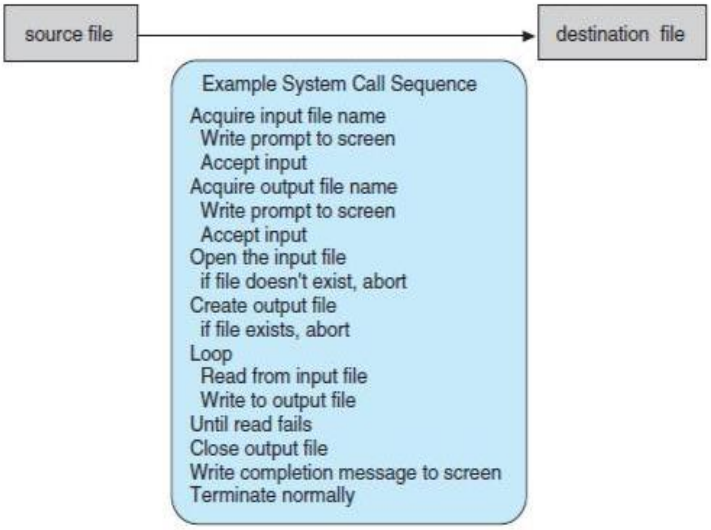


Figure 2.5 Example of how system calls are used.

The caller of the system call need know nothing about how the system call is implemented or what it does during execution.

The caller need only obey the API and understand what the operating system will do as a result of the execution of that system call.

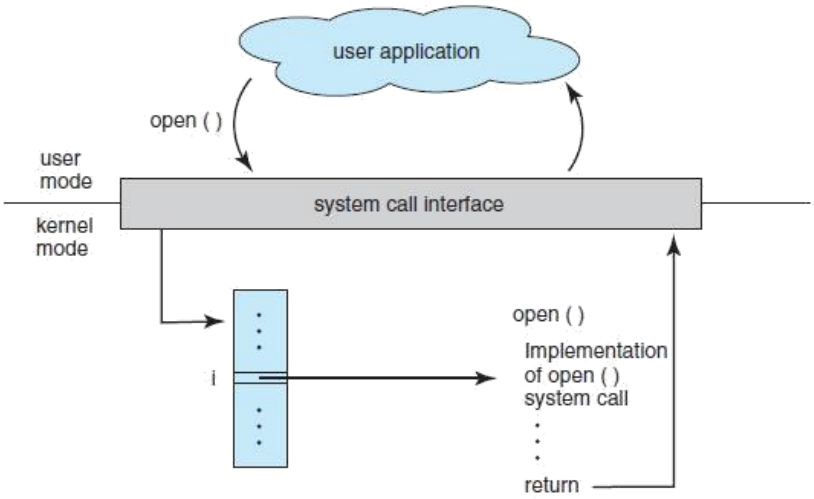


Figure 2.6 The handling of a user application invoking the `open()` system call.

Three general methods are used to pass parameters to the operating system

pass the parameters in registers
parameters are generally stored in a block, or table, in memory, and the address of the block is passed as a parameter in a register
Parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system.

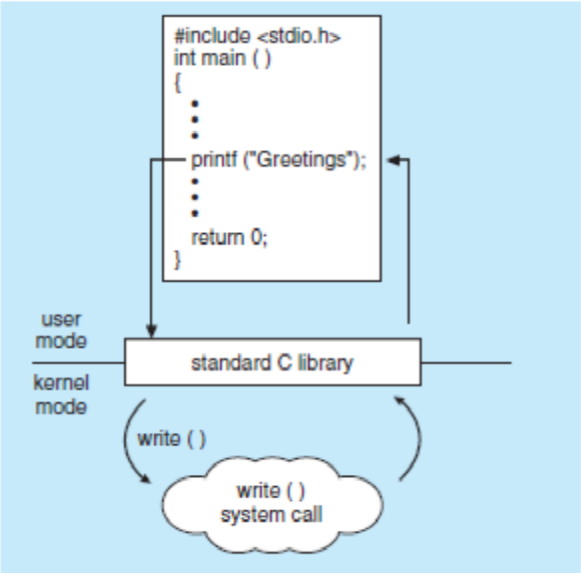
Types of System Calls:

System calls can be grouped roughly into six major categories

Process control,
File manipulation
Device manipulation,
Information maintenance,
C**ommunications**,
Protection.

PROCESS CONTROL:

A Running program needs to be able to halt its execution either normally (end ()) or abnormally (abort()).
Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter.
A process or job executing one program may want to load() and execute() another program. This feature allows the command interpreter to execute a program as directed by, for example, a user command, the click of a mouse, or a batch command.



If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution that requires to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (get process attributes() and set process attributes()).

We may also want to terminate a job or process that we created (terminate process()) if we find that it is incorrect or is no longer needed.

The System calls associated with process control includes

end, abort
load, execute
create process, terminate process
get process attributes, set process attributes
Wait for time
wait event, signal event
allocate and free memory

When a process has been created We may want to wait for a certain amount of time to pass (wait time()) or we will want to wait for a specific event to occur (wait event()).

The jobs or processes should then signal when that event has occurred (signal event()

To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed

When the process is done, it executes an exit() system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code.

In order to work with files We first need to be able to create () and delete () files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open() it and to use it.

We may also read (), write (), or reposition ().Finally, we need to close () the file, indicating that we are no longer using it.

In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary.

File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, get file attributes () and set file attributes (), are required for this function.

The System calls associated with File management includes

File management
create file, delete file
open, close
read, write, reposition

get file attributes, set file attributes

DEVICE MANAGEMENT:

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

A system with multiple users may require us to first request() a device, to ensure exclusive use of it.

After we are finished with the device, we release() it. These functions are similar to the open() and close() system calls for files.

Once the device has been requested (and allocated to us), we can read(), write(), and (possibly) reposition() the device, just as we can with files.

I/O devices are identified by special file names, directory placement, or file attributes.

The System calls associated with Device management includes

request device, release device
read, write, reposition
get device attributes, set device attributes
logically attach or detach devices

INFORMATION MAINTENANCE:

Many system calls exist simply for the purpose of transferring information between the user program and the operating system.

Example, most systems have a system call to return the current time() and date().

Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

Many systems provide system calls to dump() memory. This provision is useful for debugging.

Many operating systems provide a time profile of a program to indicate the amount of time that the program executes at a particular location or set of locations.

The operating system keeps information about all its processes, and system calls are used to access this information.

Generally, calls are also used to reset the process information (get process attributes() and set process attributes()).

The System calls associated with Device management includes

get time or date, set time or date
get system data, set system data
get process, file, or device attributes
set process, file, or device attributes

COMMUNICATION:

There are two common models of Interprocess communication: the message passing model and the shared-memory model.

In the message-passing model, the communicating processes exchange messages with one another to transfer information.

Messages can be exchanged between the processes either directly or indirectly through a common mailbox.

Each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process. The get hostid() and get processid() system calls do this translation.

The recipient process usually must give its permission for communication to take place with an accept connection () call.

The source of the communication, known as the client, and the receiving daemon, known as a server, then exchange messages by using read message() and write message() system calls.

The close connection() call terminates the communication

In the shared-memory model, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes.

The system calls associated with communication includes,

create, delete communication connection
send, receive messages

Transfer status information
attach or detach remote devices

PROTECTION:

Protection provides a mechanism for controlling access to the resources provided by a computer system.
System calls providing protection include set permission () and get permission (), which manipulate the permission settings of resources such as files and disks.
The allow user () and deny user () system calls specify whether particular users can—or cannot—be allowed access to certain resources.

System programs, also known as system utilities, provide a convenient environment for program development and execution.
They can be divided into these categories:

File management
Status information
File modification.
Programming-language support
Program loading and execution
Communications
Background services

File Management: These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

Status Information: Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information.
Others are more complex, providing detailed performance, logging, and debugging information.

File Modification: Several text editors may be available to create and modify the content of files stored on disk or other storage devices
There may also be special commands to search contents of files or perform transformations of the text.

Programming Language support: Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system.

Program Loading and Execution: Once a program is assembled or compiled, it must be loaded into memory to be executed.
The system may provide absolute loaders, relocatable loader.

Communication: These programs provide the mechanism for creating virtual connections among processes, users, and computer systems.
They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.

Background Services: All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as services, subsystems, or daemons.

Along with system programs, most operating systems are supplied with programs that are useful in solving common problems or performing common operations.

Such application programs include Web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

The operating systems are designed to run on any of a class of machines at a variety of sites with a variety of peripheral configurations.

The Computer system must then be configured or generated for each specific computer site, a process sometimes known as system generation SYSGEN.

The operating system is normally distributed on disk, on CD-ROM or DVD-ROM, or as an ―ISO‖ image, which is a file in the format of a CD-ROM or DVD-ROM.

To generate a system, the special program called SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system.

The following kinds of information must be determined.

- What CPU is to be used?
- How will the boot disk be formatted?
- How much memory is available?
- What devices are available?
- What operating-system options are desired, or what parameter values are to be used?

A system administrator can use this information to modify a copy of the source code of the operating system. The operating system then is completely compiled.

The system description can lead to the creation of tables and the selection of modules from a precompiled library. These modules are linked together to form the generated operating system

It is also possible to construct a system that is completely table driven. All the code is always part of the system, and selection occurs at execution time, rather than at compile or link time.

SYSTEM BOOT:

The procedure of starting a computer by loading the kernel is known as booting the system.

A small piece of code known as the bootstrap program or bootstrap loader locates the kernel,loads it into main memory, and starts its execution.

First a simple bootstrap loader fetches a more complex boot program from disk

A complex boot program loads the OS

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine.

It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory and then it starts the Operating system.

All forms of ROM are also known as **firmware**, since their characteristics fall somewhere between those of hardware and those of software.

A problem with firmware in general is that executing code there is slower than executing code in RAM.

Some systems store the operating system in firmware and copy it to RAM for fast execution.

A final issue with firmware is that it is relatively expensive, so usually only small amounts are available.

For large operating systems the bootstrap loader is stored in firmware, and the operating system is on disk.

The Bootstrap program has a piece of code that can read a single block at a fixed location from disk into memory and execute the code from that Boot block.

The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution.

A disk that has a Boot partition is called as a Boot Disk.

GRUB is an example of an open-source bootstrap program for Linux systems.