DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING &
DEPARTMENT OF INFORMATION TECHNOLOGY
COMMON COURSE (REGULATIONS 2013)

CS6401 – OPERATING SYSTEMS - Notes for Quick Reference

UNIT - I - OPERATING SYSTEMS OVERVIEW

Operating System

- ✓ An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware.
- ✓ An OS act as an intermediary between user and hardware.
- ✓ An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users.
- ✓ The OS also manages secondary memory and I/O (input/output) devices on behalf of its users.

BASIC ELEMENTS

A computer consists of processor, memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs.
There are four main structural elements:

• Processor
- 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** (CPU).

• Main memory
- 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
- Move 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**
  - Provides for communication among processors, main memory, and I/O modules.

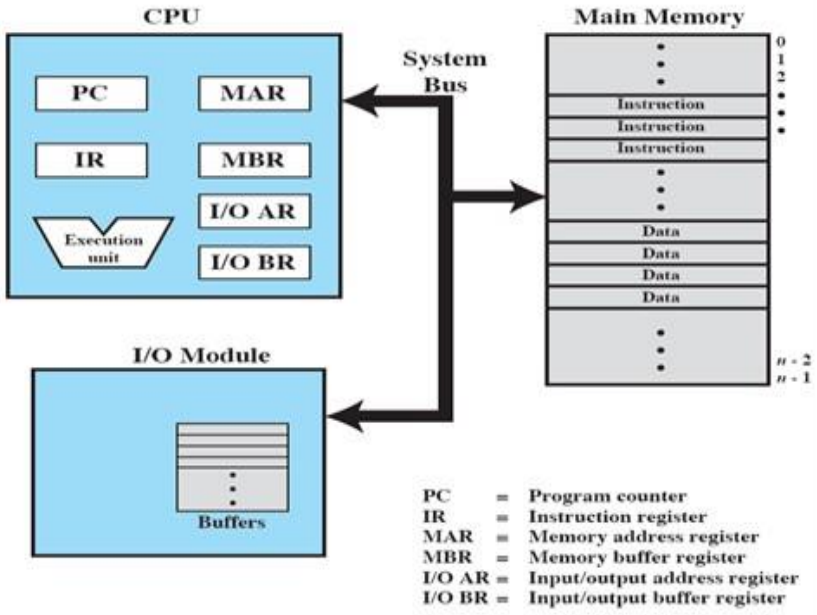
The processor's functions is to exchange data with memory. For this purpose, it makes use of two internal (to the processor) registers:

- ✓ Memory address register (MAR), which specifies the address in memory for the next read or write;
- ✓ Memory buffer register (MBR), which contains the data to be written into memory or which receives the data read from memory.

- Similarly, 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. Each location contains a bit pattern that can be interpreted as either an instruction or data.

-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.

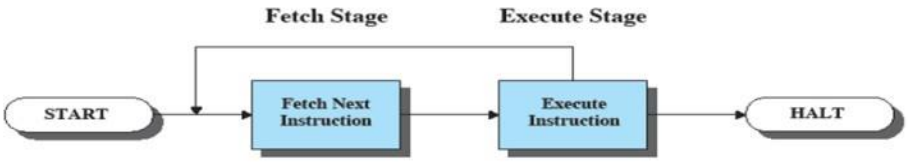


Computer Components: Top-Level View

INSTRUCTION EXECUTION

A program to be executed by a processor consists of a set of instructions stored in memory. Instruction processing consists of two steps:
- The processor reads (*fetches*) instructions from memory one at a time
- Executes each instruction.

The processing required for a single instruction is called an *instruction cycle,* which is shown in the figure below.



The two steps are referred to as the *fetch stage* and the *execute stage*. Program execution halts only if the processor is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the processor is encountered.

- At the beginning of each instruction cycle, the processor fetches an instruction from memory. Typically, the program counter (PC) holds the address of the next instruction to be fetched.
- The fetched instruction is loaded into the instruction register (IR). 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.

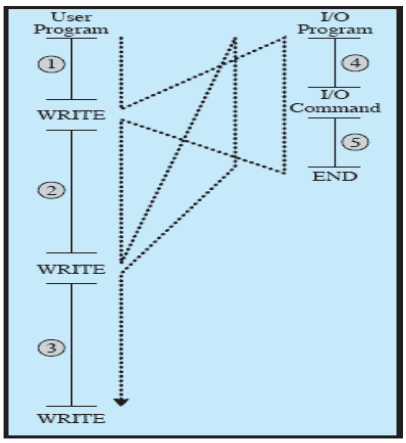
INTERRUPTS

All computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor. The most common classes of interrupts are

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.

Interrupts are provided primarily as a way to improve processor utilization.
Most I/O devices are much slower than the processor. Suppose that the processor is transferring data to a printer.. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many thousands or even

millions of instruction cycles. Clearly, this is a very wasteful use of the processor. This is depicted in the following figure.



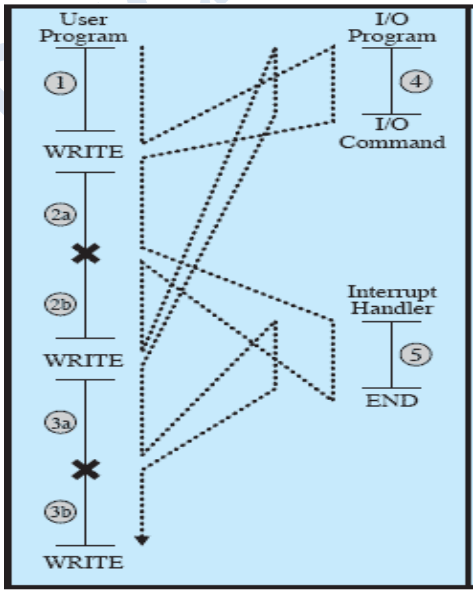
(a) No interrupts

The I/O program consists of three sections:

• A sequence of instructions, 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.
• A sequence of instructions to complete the operation. This may include setting a flag indicating the success or failure of the operation.

Interrupts and the Instruction Cycle

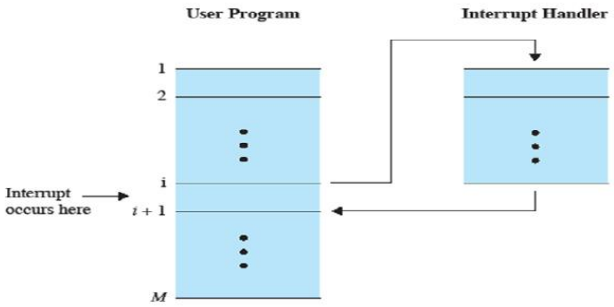
With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in the following figure



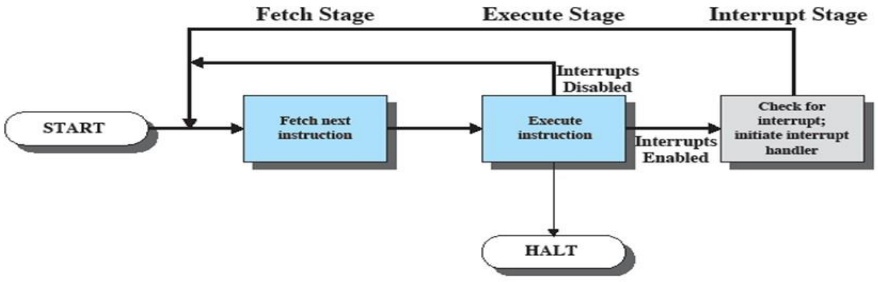
(b) Interrupts; short I/O wait

- As before, the user program reaches a point at which it makes a system call in the form of a WRITE call.
- The I/O program that is invoked in this case 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 be serviced, that is, when it is 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; branching off to a routine to service that particular I/O device, known as an interrupt handler; and resuming the original execution after the device is serviced.

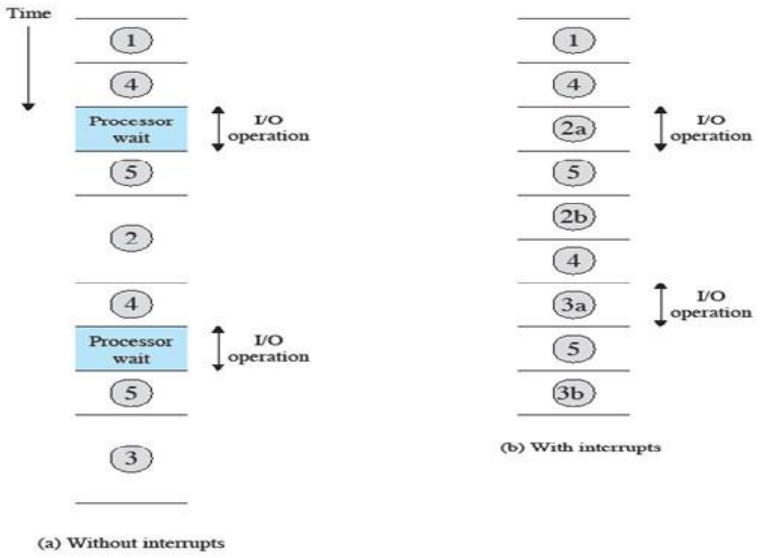
When the interrupt processing is completed, execution resumes. Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the OS are responsible for suspending the user program and then resuming it at the same point.



- To accommodate interrupts, an *interrupt stage* is added to the instruction cycle, as shown in the following Figure.
- 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. The interrupt-handler routine is generally part of the OS.



To appreciate the gain in efficiency, the timing diagram based on the flow of control is depicted in the following figure.



Interrupt Processing

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

1. The device issues an interrupt signal to the processor.

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

3. 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.

4. The processor next needs to prepare to transfer control to the interrupt routine. To begin, it saves information needed to resume the current program at the point of interrupt. The minimum information required is the program status word (PSW) and the location of the next instruction to be executed, which is contained in the program counter (PC). These can be pushed onto a control stack

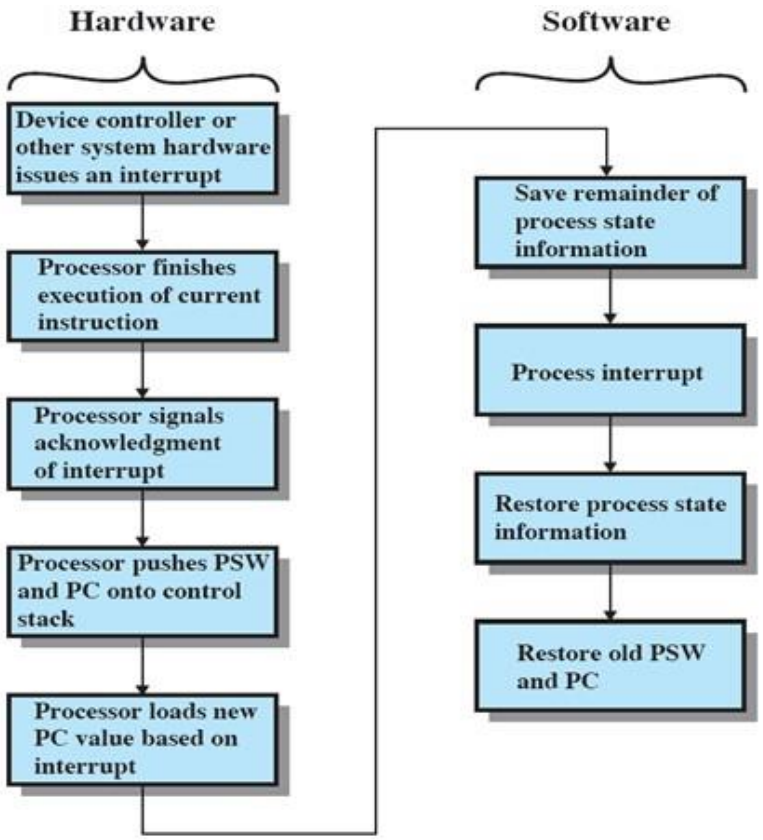
5. The processor then loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt.

6. 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. Typically, the interrupt handler will begin by saving the contents of all registers on the stack

7. The interrupt handler may now proceed to process the interrupt. This includes an examination of status information relating to the I/O operation or other event that caused an interrupt. It may also involve sending additional commands or acknowledgments to the I/O device.

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

9. 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.



Multiple Interrupts

One or more interrupts can occur while an interrupt is being processed. For example, a program may be receiving data from a communications line and printing results at the same time. The printer will generate an interrupt every time that it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives.

Two approaches can be taken to dealing with multiple interrupts.

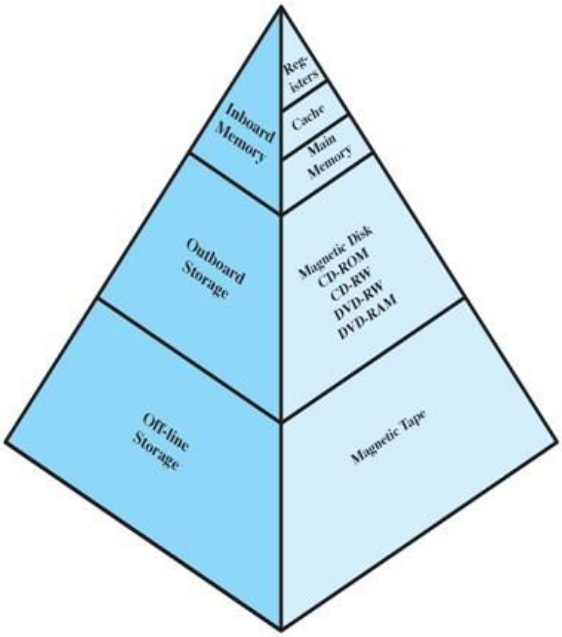
- The first is to disable interrupts while an interrupt is being processed. A *disabled interrupt* simply means that the processor ignores any new interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has re-enabled interrupts. This approach is simple, as interrupts are handled in strict sequential order.
- 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.

THE MEMORY HIERARCHY

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

There is a trade-off among the three key characteristics of memory: namely, capacity, access time, and cost. The following relationships hold between the constraints:
• Faster access time, greater cost per bit
• Greater capacity, smaller cost per bit
• Greater capacity, slower access speed

A typical memory hierarchy is illustrated in the following Figure.



As one goes down the hierarchy, the following occur:
a. Decreasing cost per bit
b. Increasing capacity
c. Increasing access time
d. Decreasing frequency of access to the memory by the processor

The basis for the validity of condition (d) is a principle known as **locality of reference**.

- ✓ During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster.
- ✓ Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.
- ✓ Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above.
- ✓ This principle can be applied across more than two levels of memory.

The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Main memory is usually extended with a higher-speed, smaller cache. The three forms of memory just described are, typically, volatile and employ semiconductor technology.

Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable disk, tape, and optical storage. 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, as opposed to individual bytes or words. A hard disk is also used to provide an extension to main memory known as virtual memory.

CACHE MEMORY

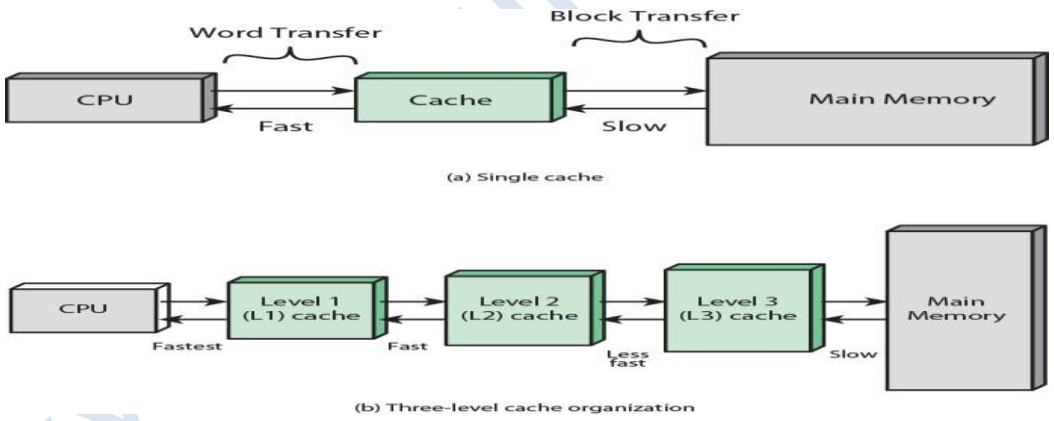
The cache is a device for staging the movement of data between main memory and processor registers to improve performance.

Motivation:

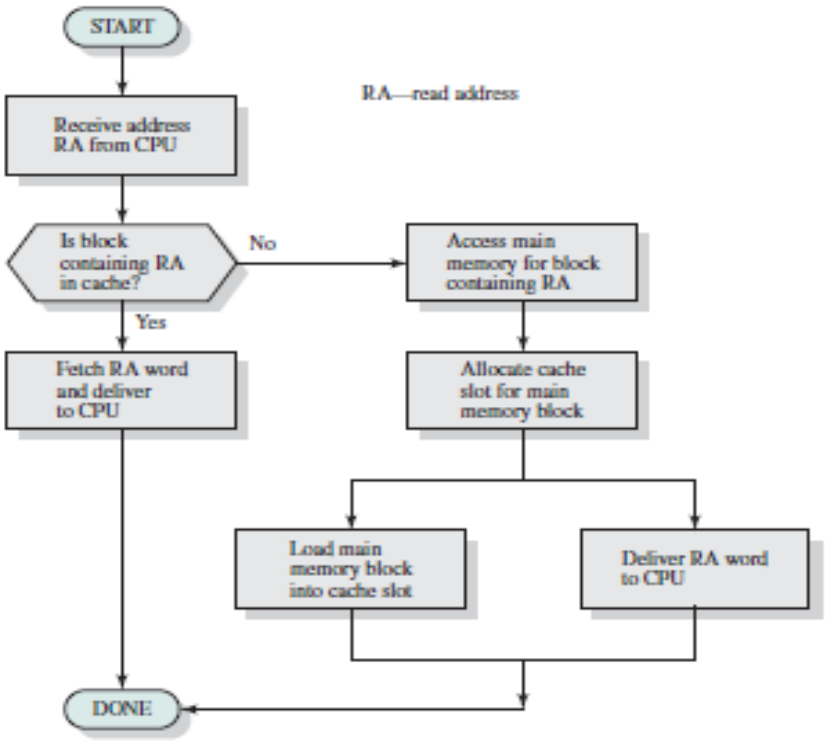
- The processor speed has consistently increased more rapidly than memory access speed. There is a trade-off among speed, cost, and size. Ideally, main memory should be built with the same technology as that of the processor registers, giving memory cycle times comparable to processor cycle times. This has always been too expensive a strategy.
- The solution is to exploit the principle of locality by providing a small, fast memory between the processor and main memory, namely the cache.

Cache Principles

Cache memory is intended to provide memory access time approaching that of the fastest memories available and at the same time support a large memory size that has the price of less expensive types of semiconductor memories. There is a relatively large and slow main memory together with a smaller, faster cache memory. The concept is illustrated in Figure.



- 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.
- 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.



The above figure illustrates the read operation. The processor generates the 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 Design

Key elements fall into the following categories:

- Cache size
- Block size - the unit of data exchanged between cache and main memory.
- Mapping function – determines which cache location the block will occupy
- 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.
- Write policy - dictates when the memory write operation takes place
- Number of cache levels

DIRECT MEMORY ACCESS

Three techniques are possible for I/O operations:

- programmed I/O
- interrupt-driven I/O
- Direct memory access (DMA).

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.

Programmed I/O

- The I/O module performs the requested action and then sets the appropriate bits in the I/O status register but takes no further action to alert the processor. In particular, it does not interrupt the processor.
- For this purpose, the processor periodically checks the status of the I/O module until it finds that the operation is complete.
- The processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of more data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the performance level of the entire system is severely degraded.

Interrupt-driven I/O

- The processor issues 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, as before, and then resumes its former processing.

Both of these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

DIRECT MEMORY ACCESS (DMA)

DMA is involved when large volumes of data are to be moved.
The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module.

Technique works as follows:

- 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.
- The overall effect is to cause the processor to execute more slowly during a DMA transfer when processor access to the bus is required.
- Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

MULTIPROCESSOR AND MULTICORE ORGANISATION

As computer technology has evolved and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to improve performance and, in some cases, to improve reliability.

The three most popular approaches to provide parallelism by replicating processors:

- ✓ symmetric multiprocessors (SMPs)
- ✓ multicore computers
- ✓ clusters.

Symmetric Multiprocessors

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

1. There are two or more similar processors of comparable capability.
2. 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.
3. All processors share access to I/O devices, either through the same channels or through different channels that provide paths to the same device.
4. All processors can perform the same functions (hence the term *symmetric*).
5. 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.

Advantages

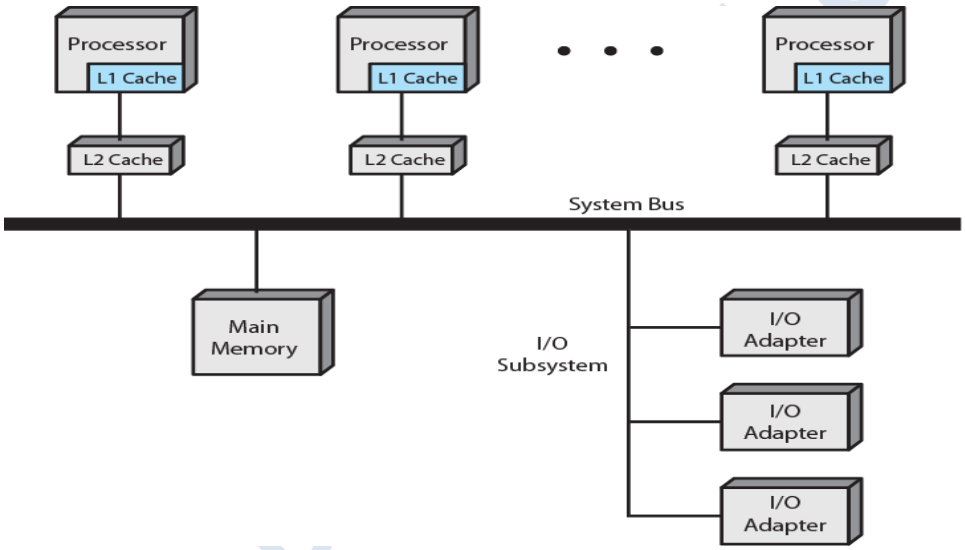
- **Performance:** If the work to be done by a computer can be organized so that some portions of the work can be done in parallel, then a system with multiple processors will yield greater performance than one with a single processor of the same type.
- **Availability:** In a symmetric multiprocessor, because all processors can perform the same functions, the failure of a single processor does not halt the machine. Instead, the system can continue to function at reduced performance.
- **Incremental growth:** A user can enhance the performance of a system by adding an additional processor.
- **Scaling:** Vendors can offer a range of products with different price and performance characteristics based on the number of processors configured in the system.

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.

Organization

Figure illustrates the general organization of an SMP.

- 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). It may also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.



Cache coherence problem

- In modern computers, processors generally have at least one level of cache memory that is private to the processor.
- This use of cache introduces some new design considerations. Because each local cache contains an image of a portion of main memory, if a word is altered in one cache, it could conceivably invalidate a word in another cache.
- To prevent this, the other processors must be alerted that an update has taken place.
- This problem is known as the cache coherence problem and is typically addressed in hardware rather than by the OS.

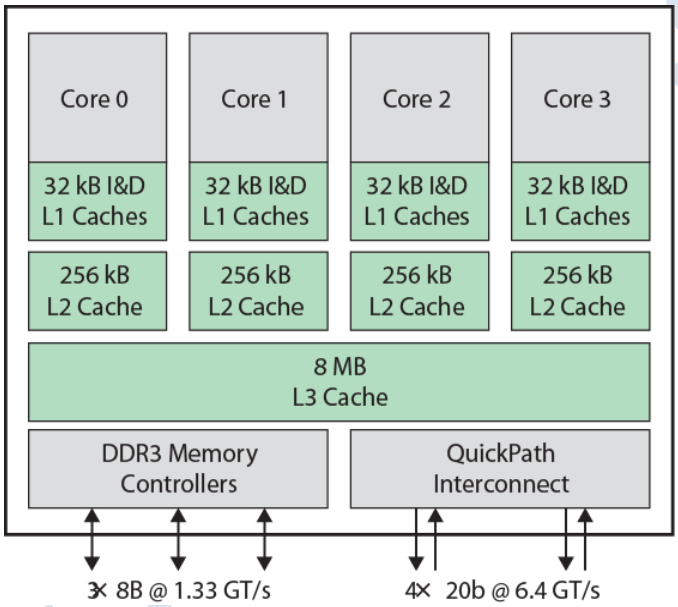
Multicore Computers

- A **multicore** computer, also known as a **chip multiprocessor**, combines two or more processors (called cores) on a single piece of silicon (called a die).

- Each core consists of all of the components of an independent processor, such as registers, ALU, pipeline hardware, and control unit, plus L1 instruction and data caches. In addition to the multiple cores, contemporary multicore chips also include L2 cache and, in some cases, L3 cache.

Motivation

- Microprocessor systems have experienced a steady, usually exponential, increase in performance.
- Performance has also been improved by the increased complexity of processor design to exploit parallelism in instruction execution and memory access.
- So,the best way to improve performance is to take advantage of advances in hardware is to put multiple processors and a substantial amount of cache memory on a single 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 as in the below figure.



One mechanism Intel uses to make its caches more effective is **prefetching**, in which the hardware examines memory access patterns and attempts to fill the caches speculatively with data that are likely to be requested soon.

The Core i7 chip supports two forms of external communications to other chips.

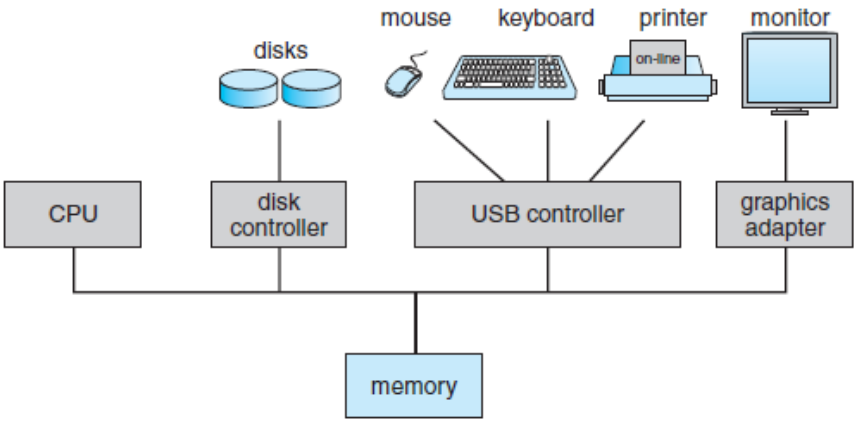
- The **DDR3 memory controller**
  - This brings the memory controller for the DDR (double data rate) main memory onto the chip.
  - The interface supports three channels that are 8 bytes wide for a total bus width of 192 bits, for an aggregate data rate of up to 32 GB/s.
  - With the memory controller on the chip, the Front Side Bus is eliminated.

- The **Quick Path Interconnect (QPI)**
  - It is a point-to-point link electrical interconnect specification.
  - It enables high-speed communications among connected processor chips.
  - The QPI link operates at 6.4 GT/s (transfers per second).

COMPUTER-SYSTEM ORGANIZATION

1. 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.
- Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles.
- To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.



Bootstrap program

- 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, or **bootstrap program**, tends to be simple.
- It is stored within the computer hardware in read-only memory (**ROM**) or electrically erasable programmable read-only memory (**EEPROM**), also known as **firmware**.
- It initializes all aspects of the system, from CPU registers to device controllers to memory contents.
- The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locatethe operating-system kernel and load it into memory.
- Once the kernel is loaded and executing, it can start providing services to the system and its users and the system waits for some event to occur.

Interrupt

- The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software.
- Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.
- Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).
- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.
- The interrupt must transfer control to the appropriate interrupt service routine. The routine, in turn, would call the interrupt-specific handler.
- Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed.
- Generally, the table of pointers is stored in low memory. These locations hold the addresses of the interrupt service routines for the various devices.
- This array, or **interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device..
- The interrupt architecture must also save the address of the interrupted instruction.
- After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

2. Storage Structure

- The CPU can load instructions only from main memory (also called **random-access memory**, or **RAM**), so any programs to run must be stored here..
- Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM)**.
- The other forms of memory are read-only memory, (ROM) and electrically erasable programmable read-only memory, (EEPROM).
- Because ROM cannot be changed, only static programs, such as the bootstrap program, are stored there.
- EEPROM can be changed but cannot be changed frequently and so contains mostly static programs. For example, smartphones have EEPROM to store their factory-installed programs.

Instruction Execution

- A typical instruction–execution cycle, as executed on a system with a **von Neumann architecture**, first fetches an instruction from memory and stores that instruction in the **instruction register**.
- The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register.
- After the instruction on the operands has been executed, the result may be stored back in memory.

- The programs and data must reside in main memory permanently for a program to run. This arrangement usually is not possible for the following two reasons:
- 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.
- So, 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 most common secondary-storage device is a **magnetic disk**, which provides storage for both programs and data.
- Others include cache memory, CD-ROM, magnetic tapes, and so on.
- Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time.
- The main differences among the various storage systems lie in speed, cost, size, and volatility.
- The wide variety of storage systems can be organized in a hierarchy (Figure) 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.

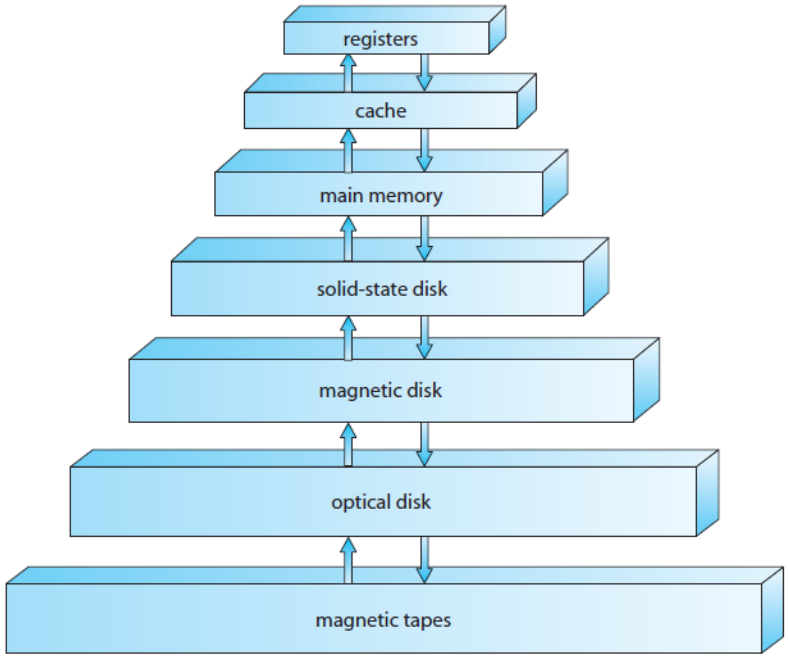


Figure - Storage-device hierarchy

- The top four levels of memory in may be constructed using semiconductor memory.
- In the hierarchy shown, the storage systems above the solid-state disk are volatile, whereas those including the solid-state disk and below are nonvolatile.

Solid-state

- **Solid-state disks** have several variants but in general are faster than magnetic disks and are nonvolatile. One type of solid-state disk stores data in a large DRAM array during normal operation but also contains a hidden magnetic hard disk and a battery for backup power.
- Another form of solid-state disk is flash memory, which is popular in cameras and **personal digital assistants (PDAs)**, in robots, and increasingly for storage on general-purpose computers.

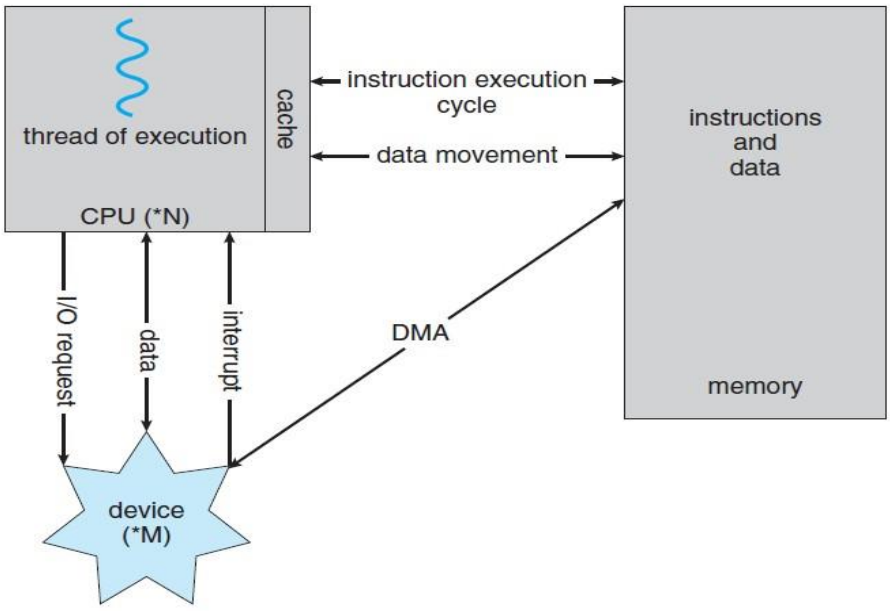
3. I/O Structure

- 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.
- 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 device controller, in turn, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard").
- 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.
- The device driver then returns control to the operating system. For other operations, the device driver returns status information.

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, **direct memory access (DMA)** is used.

- After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire 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, rather than the one interrupt per byte generated for low-speed devices.

The figure shows the interplay of all components of a computer system.



OPERATING SYSTEMS- OBJECTIVES & FUNCTIONS

An OS is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. It can be thought of as having three objectives.

- **Convenience:** An OS makes a computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

1) The Operating System as a User/Computer Interface

- The hardware and software used in providing applications to a user can be viewed in a layered or hierarchical fashion.
- The user of those applications, the end user, generally is not concerned with the details of computer hardware.
- Thus, the end user views a computer system in terms of a set of applications.
- An application can be expressed in a programming language and is developed by an application programmer. To ease this chore, a set of system programs is provided. Some of these programs are referred to as utilities, or library programs.
- These implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices.

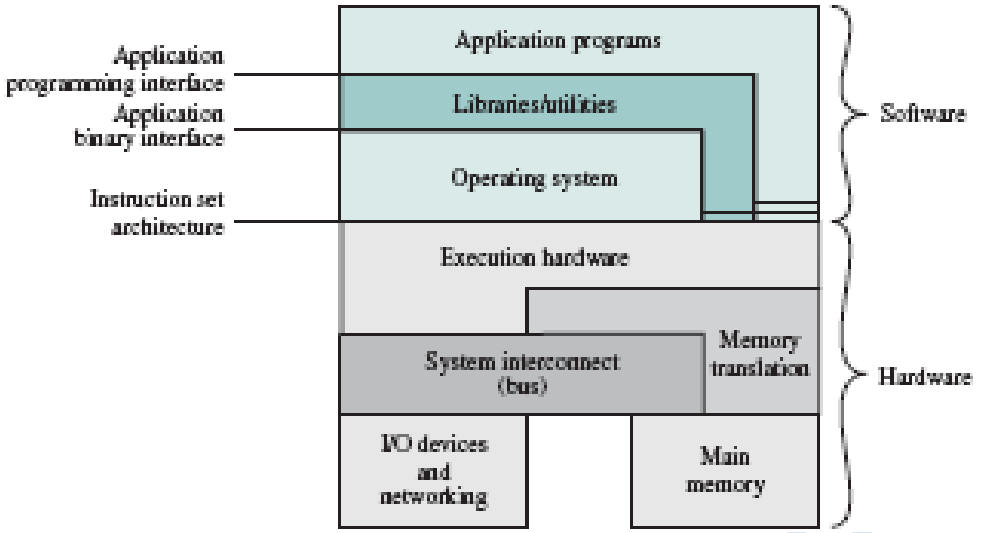


Figure - Computer Hardware and Software Structure

- 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.
- It acts as mediator, making it easier for the programmer and for application programs to access and use those facilities and services.

The OS typically provides services in the following areas:

- **Program development:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs.
- **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles these scheduling duties for the user.
- **Access to I/O devices:** Each I/O device requires its own peculiar set of instructions or control signals for operation. The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.
- **Controlled access to files:** For file access, the OS must reflect a detailed understanding of not only the nature of the I/O device (disk drive, tape drive) but also the structure of the data contained in the files on the storage medium.
- **System access:** For shared or public systems, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.
- **Error detection and response:** A variety of errors can occur while a computer system is running. The OS must provide a response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error, to retrying the operation, to simply reporting the error to the application.

- **Accounting:** A good OS will collect usage statistics for various resources and monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance.

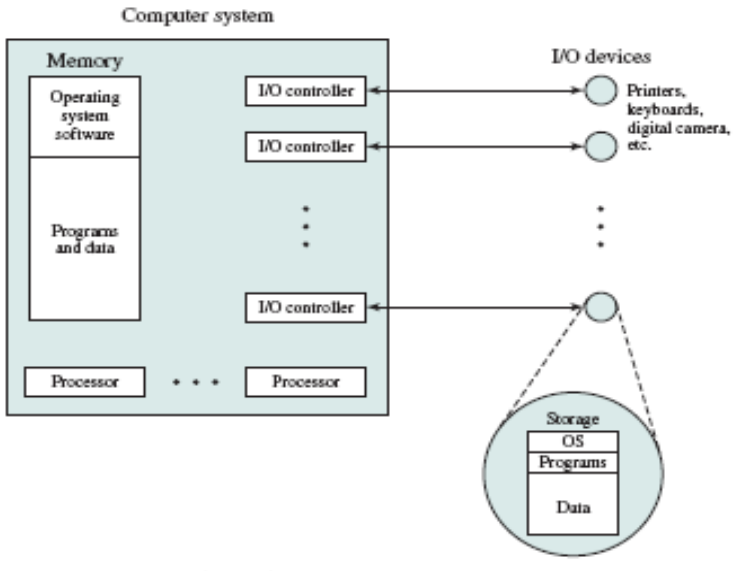
The Figure also indicates three key interfaces in a typical computer system.

- **Instruction set architecture (ISA):** The ISA defines the repertoire of machine language instructions that a computer can follow. This interface is the boundary between hardware and software.
- **Application binary interface (ABI) :** The ABI defines a standard for binary portability across programs. The ABI defines the system call interface to the operating system and the hardware resources and services available in a system through the user ISA.
- **Application programming interface (API) :** The API gives a program access to the hardware resources and services available in a system through the user ISA supplemented with high-level language (HLL) library calls. Using an API enables application software to be ported easily, to other systems that support the same API.

2) 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.



- Like other computer programs, the OS provides instructions for the processor. 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 and, at a given time, other portions of the OS currently in use.
- The remainder of main memory contains user programs and data. The memory management hardware in the processor and the OS jointly control the allocation of main memory. 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.

3) Ease of Evolution of an Operating System

A major OS will evolve over time for a number of reasons.

- **Hardware upgrades plus new types of hardware:** For example, early versions of UNIX and the Macintosh OS did not employ a paging mechanism because they were run on processors without paging hardware. Subsequent versions of these operating systems were modified to exploit paging capabilities.

- **New services:** In response to user demand or in response to the needs of system managers, the OS expands to offer new services.

- **Fixes:** Any OS has faults. These are discovered over the course of time and fixes are made. Of course, the fix may introduce new faults. The need to change an OS regularly places certain requirements on its design. **2.2**

EVOLUTION OF OPERATING SYSTEMS

The stages in the evolution of operating systems are

- Serial Processing
- Simple Batch Systems
- Multiprogrammed Batch Systems
- Time Sharing Systems

EVOLUTION OF OPERATING SYSTEMS

Serial Processing

- With the earliest computers, the programmer interacted directly with the computer hardware; there was no OS.
- These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer.
- 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.

Two main problems

- **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.

✓ Each of these steps could involve mounting or dismounting tapes or setting up card decks.
✓ If an error occurred, the hapless user typically had to go back to the beginning of the setup sequence.
✓ 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.

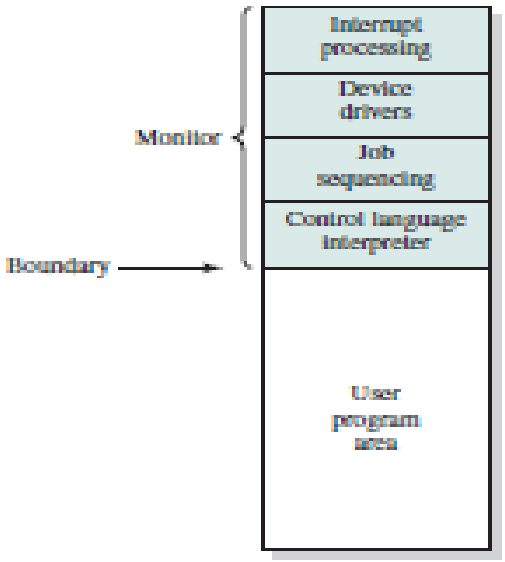
Simple Batch Systems

- Early computers were very expensive, and therefore it was important to maximize processor utilization. To improve utilization, the concept of a batch OS was developed.

Idea:

- To use 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, at which point the monitor automatically begins loading the next program.
- The two points of view:
    - the monitor
    - the processor.

- **Monitor point of view:**
    - The monitor controls the sequence of events.
    - Monitor must always be in main memory and available for execution. That portion is referred of as the **resident monitor**. The rest of the monitor consists of utilities and common functions that are loaded as subroutines to the user program at the beginning of any job that requires them.



- 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.
- 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.
-

• **Processor point of view**

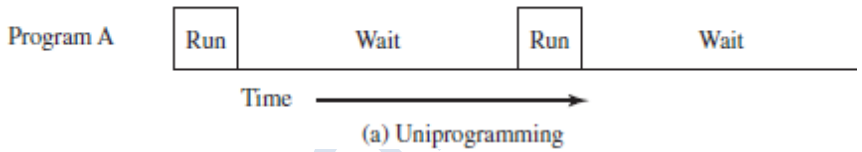
- The processor is executing instructions from the portion of main memory containing the monitor. These instructions cause the next job to be read into another portion of main memory.
- Once a job has been read in, the processor will encounter a branch instruction in the monitor that instructs the processor 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.
- Either event causes the processor to fetch its next instruction from the monitor program.
- 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.
- The monitor improves job setup time as well. With each job, instructions are included in a primitive form of **job control language (JCL)**.
- This is a special type of programming language used to provide instructions to the monitor.

The monitor, or batch OS, is simply a computer program. It relies on the ability of the processor to fetch instructions from various portions of main memory to alternately size and relinquish control. Certain other hardware features are also desirable:

- **Memory protection:** While the user program is executing, it must not alter the memory area containing the monitor. If such an attempt is made, the processor hardware should detect an error and transfer control to the monitor.
- **Timer:** A timer is used to prevent a single job from monopolizing the system. If the timer expires, the user program is stopped, and control returns to the monitor.
- **Privileged instructions:** Certain machine level instructions are designated privileged and can be executed only by the monitor. If the processor encounters such an instruction while executing a user program, an error occurs causing control to be transferred to the monitor
- **Interrupts:** This feature gives the OS more flexibility in relinquishing control to and regaining control from user programs.
    - A user program executes in a **user mode**, in which certain areas of memory are protected from the user's use and in which certain instructions may not be executed.
    - The monitor executes in a system mode, or what has come to be called **kernel mode**, in which privileged instructions may be executed and in which protected areas of memory may be accessed.

Multiprogrammed Batch Systems

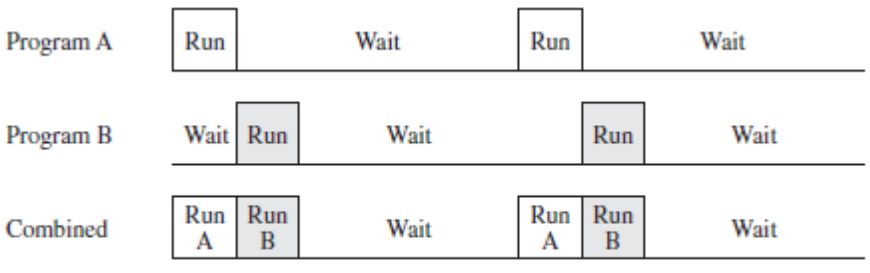
Even with the automatic job sequencing provided by a simple batch OS, the processor is often idle. The problem is that I/O devices are slow compared to the processor.
Figure (a) below illustrates this situation, where we have a single program, referred to as uniprogramming.



(a) Uniprogramming

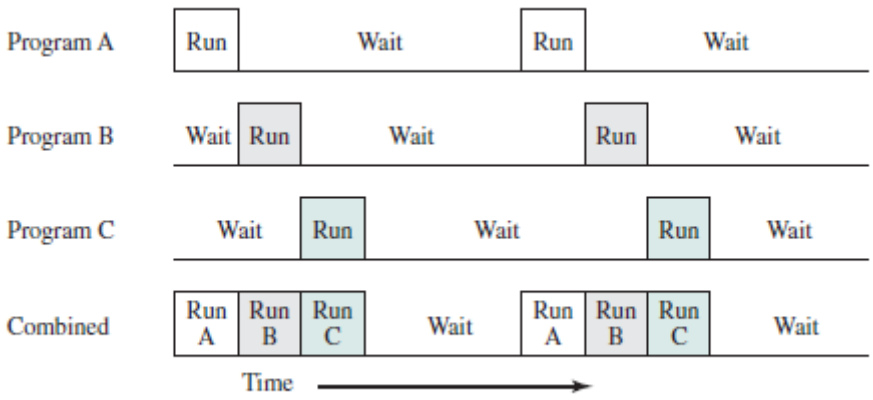
- The processor spends a certain amount of time executing, until it reaches an I/O instruction.
- It must then wait until that I/O instruction concludes before proceeding. This inefficiency is not necessary.

The memory can hold the OS (resident monitor) and one user program. Suppose that there is room for the OS and two 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 as shown in the figure (b).



(b) Multiprogramming with two programs

If the memory is expanded to hold three, four, or more programs and switch among all of them (Figure c). The approach is known as **multiprogramming**, or **multitasking**. It is the central theme of modern operating systems.



(c) Multiprogramming with three programs

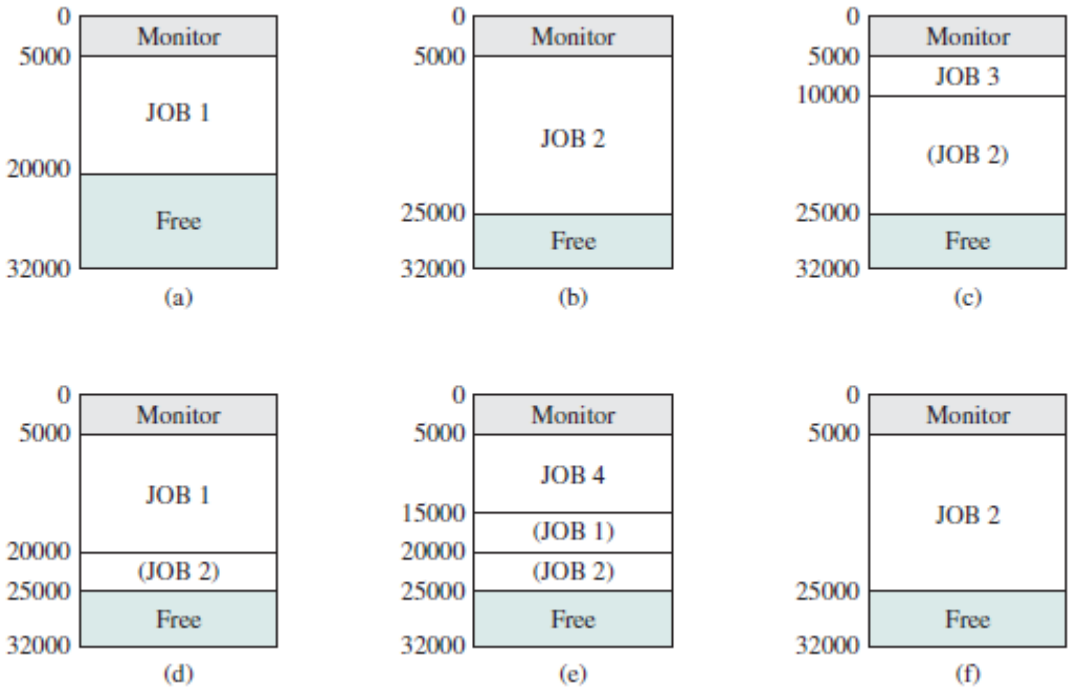
- As with a simple batch system, a multiprogramming batch system must rely on certain computer hardware features.
- The most notable additional feature that is useful for multiprogramming is the hardware that supports I/O interrupts and DMA (direct memory access).
- With interrupt-driven I/O or DMA, 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 pass control to another job.
- Multiprogramming operating systems are fairly sophisticated compared to single-program, or uniprogramming systems.

Time-Sharing Systems

- Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as **time sharing**, because 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, not counting OS overhead.
- The key differences between batch processing and time sharing use multiprogramming are listed in Table.

	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; this simplified both the monitor and memory management.
- 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**.
- Thus, at regular time intervals, the current user would be preempted and another user loaded in. To preserve the old user program status for later resumption, the old user programs and data were written out to disk before the new user programs and data were read in.
- This principle is illustrated in the Figure.
- 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 (a). 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 (b).Next, JOB3 is loaded in to be run. However, because JOB3 is smaller than JOB2, a portion of JOB2 can remain in memory, reducing disk write time (c).



- 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 (d). When JOB4 is loaded, part of JOB1 and the portion of JOB2 remaining in memory are retained (e). 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 (f).

The CTSS approach is primitive compared to present-day time sharing, but it was effective. It was extremely simple, which minimized the size of the monitor. Because a job was always loaded into the same locations in memory, there was no need for relocation techniques at load time. The technique of only writing out, what was necessary, minimized disk activity.

If multiple jobs are in memory, then they must be protected from interfering with each other by, for example, modifying each other's data. With multiple interactive users, the file system must be protected so that only authorized users have access to a particular file. The contention for resources, such as printers and mass storage devices, must be handled.

OPERATING-SYSTEM STRUCTURE

One of the most important aspects of operating systems is the ability to multiprogram. **Multiprogramming** increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

Idea:

- The operating system keeps several jobs in memory simultaneously.
- The jobs are kept initially on the disk in the **job pool**. This pool consists of all processes residing on disk awaiting allocation of main memory.
- The set of jobs in memory can be a subset of the jobs kept in the job pool.
- The operating system picks and begins to execute one of the jobs in memory.
- Eventually, the job may have to wait for some task, such as an I/O operation, to complete.
- In a multiprogrammed system, the operating system simply switches to, and executes, another job. When ***that*** job needs to wait, the CPU switches to ***another*** job, and so on.
- The first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle.

Multiprogrammed systems provide an environment in which the various system are utilized effectively, but they do not provide for user interaction with the computer system.
Time sharing (or **multitasking**)

- It is a logical extension of multiprogramming.
- In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.
- Time sharing requires an **interactive** computer system, which provides direct communication between the user and the system.
- The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. The **response time** should be short—typically less than one second.
- A time-shared operating system allows many users to share the computer simultaneously.
- As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.
- A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.

Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves **job scheduling**.

When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires some form of memory management.

In addition, if several jobs are ready to run at the same time, the system must choose which job will run first. Making this decision is **CPU scheduling.**

In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through **swapping**, whereby processes are swapped in and out of main memory to the disk. A more common method for ensuring reasonable response time is **virtual memory**, a technique that allows the execution of a process that is not completely in memory. The main advantage of the virtual-memory scheme is that it enables users to run programs that are larger than actual **physical memory**.

A time-sharing system must also provide a file system. In addition, a time-sharing system provides a mechanism for protecting resources from inappropriate use.

OPERATING-SYSTEM OPERATIONS

- Operating systems are **interrupt driven.** Events are almost always signaled by the occurrence of an interrupt or a trap.
- A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed.
- For each type of interrupt, separate segments of code in the operating system determine what action should be taken.
- An interrupt service routine is provided to deal with the interrupt.
- Since the operating system and the users share the hardware and software resources of the computer system, we need to make sure that an error in a user program could cause problems only for the one program running.
- A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

1. Dual-Mode and Multimode Operation

Hardware support is provided to differentiate among various modes of execution.

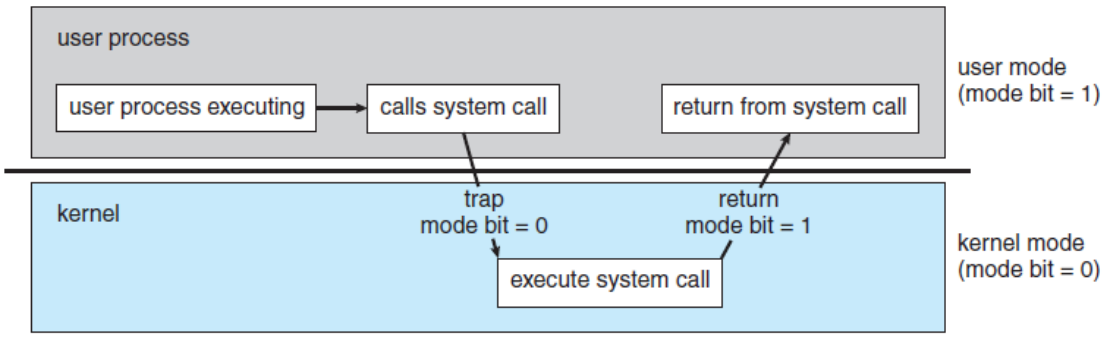


Figure Transition from user to kernel mode

- The two separate ***modes*** of operation:
  - ✓ **user mode** and
  - ✓ **kernel mode** (also called **supervisor mode, 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).
- When the computer system is executing on behalf of a user application, the system is in user mode.
- When a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request. This is shown in Figure.
- At system boot time, the hardware starts in kernelmode. 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). Thus, whenever the operating system gains control of the computer, it is in kernel mode.
- The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.
- 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.
- The instruction to switch to kernel mode is an example of a privileged instruction.

CPUs that support virtualization frequently have a separate mode to indicate when the **virtual machine manager (VMM)**—and the virtualization management software—is in control of the system.
In this mode, the VMM has more privileges than user processes but fewer than the kernel.

Life cycle of instruction execution

- Initial control resides in the operating system, where instructions are executed in kernel mode.
- When control is given to a user application, the mode is set to user mode.
- The control is switched back to the operating system via an interrupt, a trap, or a system call.
- 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.

2. Timer

- The operating system maintains control over the CPU. A user program cannot be allowed to get stuck in an infinite loop. To accomplish this goal, a **timer** is used.
- 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.

- Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time.
- The timer is used to prevent a user program from running too long.

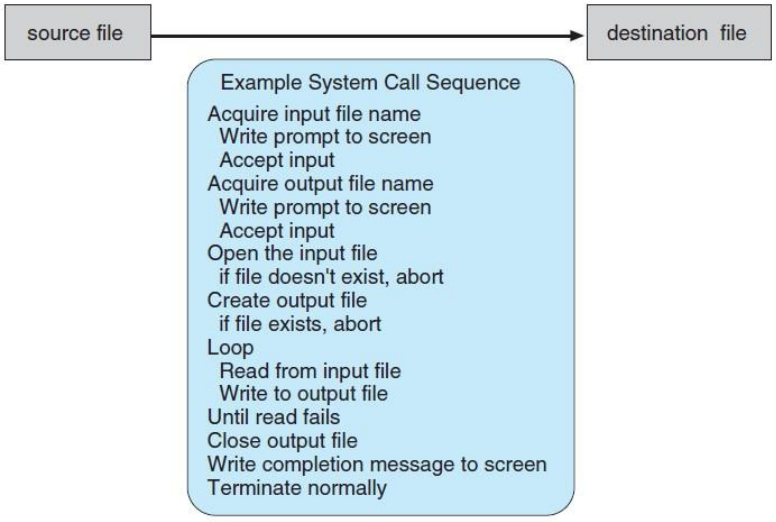
SYSTEM CALLS

System calls provide an interface to the services made available by an operating system. These calls are generally available as routines written in C and C++.

For example, simple program to read data from one file and copy them to another file.

- The first input that the program will need is the names of the two files: the input file and the output file.
- Once the two file names have been obtained, the program must open the input file and create the output file. Each of these operations requires another system call.
- Possible error conditions for each operation can require additional system calls.
- When both files are set up, we enter a loop that reads from the input file (a system call) and writes to the output file (another system call).
- Each read and write must return status information regarding various possible error conditions.
- Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window (more system calls), and finally terminate normally (the final system call).

This system-call sequence is shown in the Figure. Example system calls are shown here.



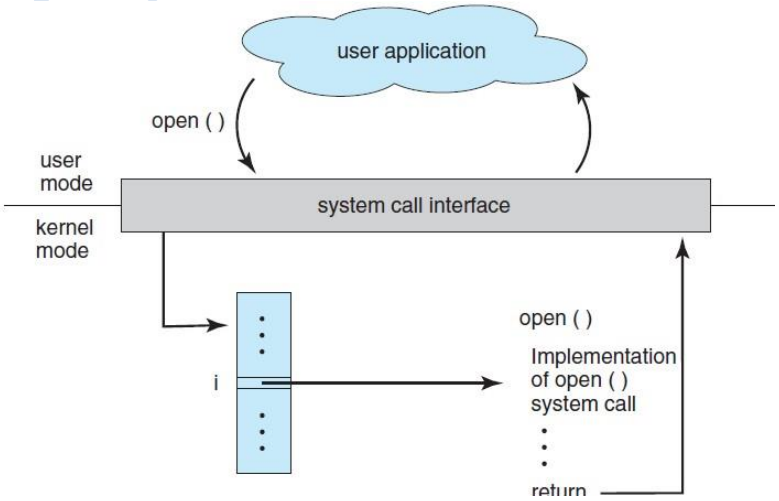
- Application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application

programmer, including the parameters that are passed to each function and the return values the programmer can expect.

- Three of the common APIs available to application programmers are
- Windows API for Windows systems,
- POSIX API for POSIX-based systems (virtually all versions of UNIX, Linux, and Mac OSX),
- Java API for programs that run on the Java virtual machine.
- A programmer accesses an API via a library of code provided by the operating system.
- An application programmer prefer programming according to an API rather than invoking actual system calls. One benefit concerns program portability.

System call Implementation

- The run-time support system 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.
- A number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.
- The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values.
- The caller need know nothing about how the system call is implemented or what it does during execution. Rather, 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.
- Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.
- The relationship between an API, the system-call interface, and the operating system is shown in Figure, which illustrates how the operating system handles a user application invoking the open() system call.

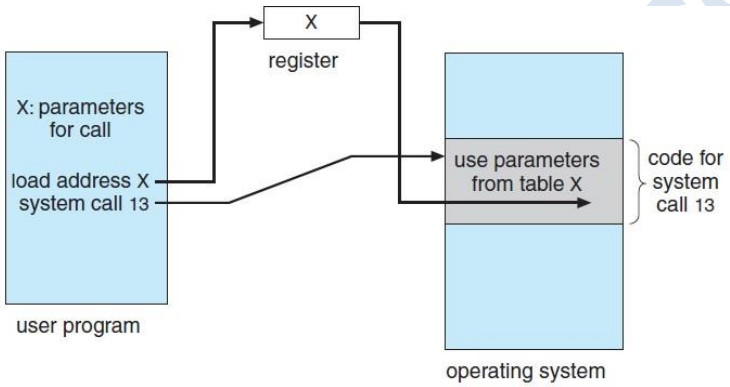


System calls – Parameter passing

- System calls occur in different ways, depending on the computer in use. Often, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call.

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

- The simplest approach is to pass the parameters in registers.
- If more parameters are present than registers, then the 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 (Figure). This is the approach taken by Linux and Solaris.

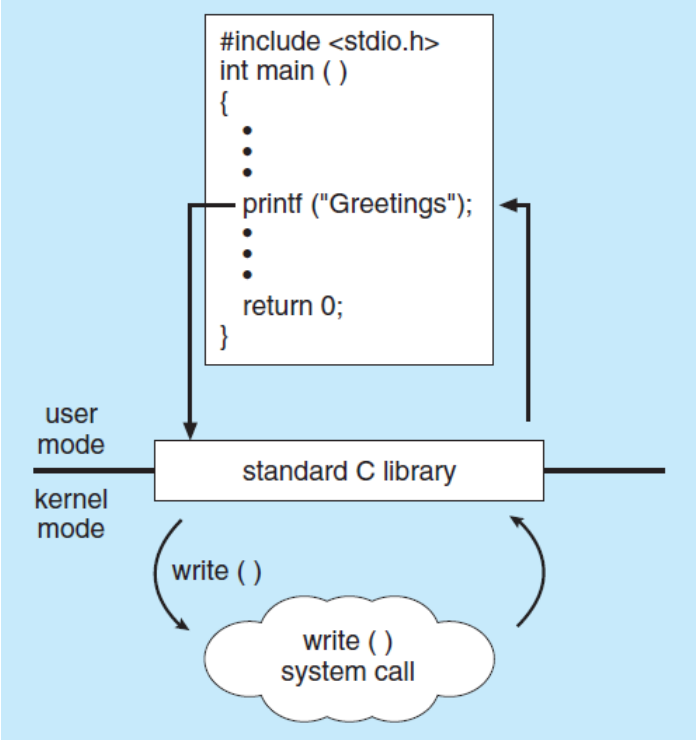


Passing parameters as Table

- Parameters also can be placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system.

Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

Example System call



TYPES OF SYSTEM CALLS

System calls can be grouped roughly into six major categories.

1. Process control
2. File manipulation
3. Device manipulation
4. Information maintenance,
5. Communications,
6. Protection

1. Process Control

- A running program needs to be able to halt its execution either normally (**end**()) or abnormally (**abort**()).
- If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated.
- Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command.
- In an interactive system, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error.

- In a GUI system, a pop-up window might alert the user to the error and ask for guidance.
- In a batch system, the command interpreter usually terminates the entire job and continues with the next job.
- A process or job executing one program may want to **load**() and **execute**() another program.
- If more programs continue concurrently, a new job or process created is to be multiprogrammed. Often, there is a system call specifically for this purpose(**create process**() or **submit job**()).
- 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. This control requires the ability 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**())
- To terminate a job or process that we created (**terminate process**()) if it is incorrect or is no longer needed.
- Having created new jobs or processes, we need to wait for them to finish their execution. We may want to wait for a certain amount of time to pass (**wait time**()).
- More probably, 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**()).
- The system calls of Process control are
    - 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

2. File Management

- It is needed 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() (rewind or skip to the end of the file, for example).
- Finally, we need to close() the file, indicating that we are no longer using it.
- To determine the values of various attributes and to reset them if necessary, two system calls, get file attributes() and set file attributes(), are required. File attributes include the file name, file type, protection codes, accounting information, and so on.
- The system calls of File management are
    - create file, delete file
    - open, close
    - read, write, reposition
    - get file attributes, set file attributes

3 Device Management

- The various resources—main memory, disk drives, access to files, and so on, controlled by the operating system can be thought of as devices.
- Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).
- 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.
- We can read(), write(), and (possibly) reposition() the device.
- The system calls of Device management are
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices

4 Information Maintenance

- Many system calls exist simply for the purpose of transferring information between the user program and the operating system.
- For 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.
- Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump() memory. This provision is useful for debugging.
- 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 of Information maintenance
    - 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

5 Communication

There are two common models of interprocess communication:

- message passing model and
- shared-memory model.

- **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.
    - Before communication can take place, a connection must be opened.

- They execute a wait for connection() call and are awakened when a connection is made.
- The source **client**, and the receiving daemon **server**, then exchange messages by using read message() and write message() system calls.
- The close connection() call terminates the communication.

➢ **Shared-memory model**

- The 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 of Communications
- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

6 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.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

SYSTEM PROGRAMS

System programs, also known as **system utilities**, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex.
They can be divided into these categories.

- **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.
- **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices.
- **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 or available as a separate download.
- **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 loaders, linkage editors, and overlay loaders.
- **Communications**. 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.

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.

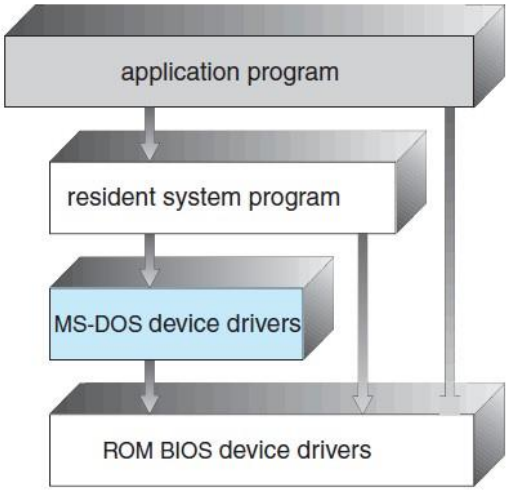
OPERATING-SYSTEM STRUCTURE

- A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily.
- A common approach is to partition the task into small components, or modules, rather than have one **monolithic** system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.

1. Simple Structure

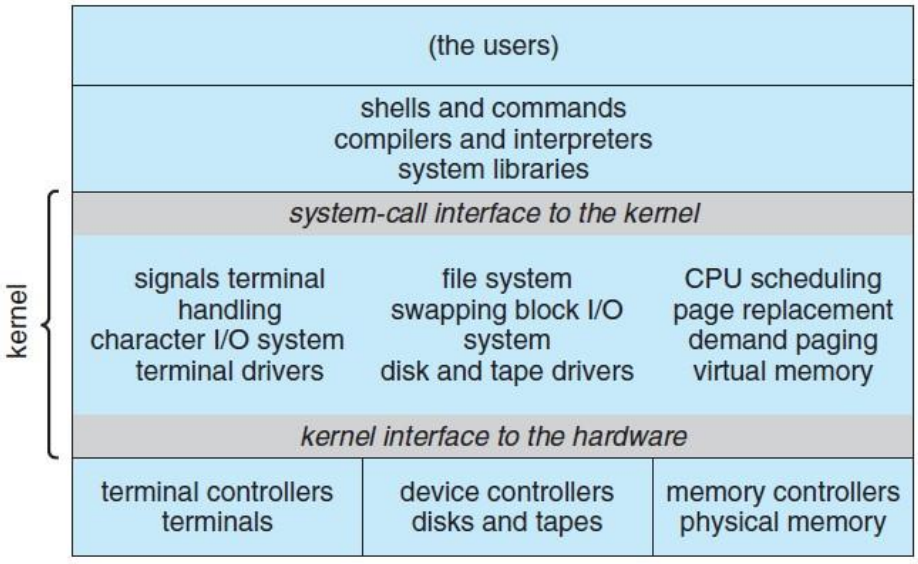
- Many operating systems do not have well-defined structures.
- MS-DOS is an example of such a system. In MS-DOS, the interfaces and levels of functionality are not well separated.
- Application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious)

programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era.



MS-DOS Layers

- Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited by hardware functionality.
- It consists of two separable parts:
  - Kernel and
  - System programs.
- The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved.
- The traditional UNIX operating system is layered to some extent, as shown in the Figure



Traditional UNIX system structure.

- Everything below the system-call interface and above the physical hardware is the kernel.
- The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.
- Taken in sum, that is an enormous amount of functionality to be combined into one level.
- This monolithic structure was difficult to implement and maintain. It had a distinct performance advantage, however: there is very little overhead in the system call interface or in communication within the kernel.

2 Layered Approach

In **layered approach**, the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer *N*) is the user interface. This layering structure is depicted in Figure.

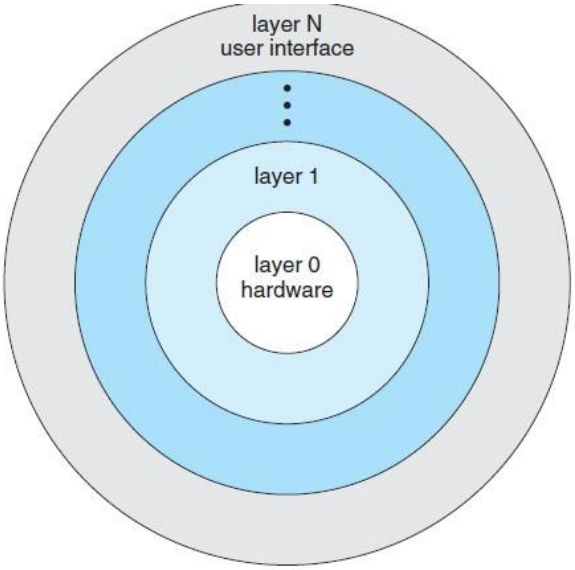


Figure: A layered operating system.

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

- A typical operating-system layer—say, layer *M*—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M*, in turn, can invoke operations on lower-level layers.
- 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.
- This approach simplifies debugging and system verification.
- Each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.
- A final problem with layered implementations is that they tend to be less efficient than other types.

3 Microkernels

- The **microkernel** approach structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.
- Microkernels provide minimal process and memory management, in addition to a communication facility. The Figure illustrates the architecture of a typical microkernel.

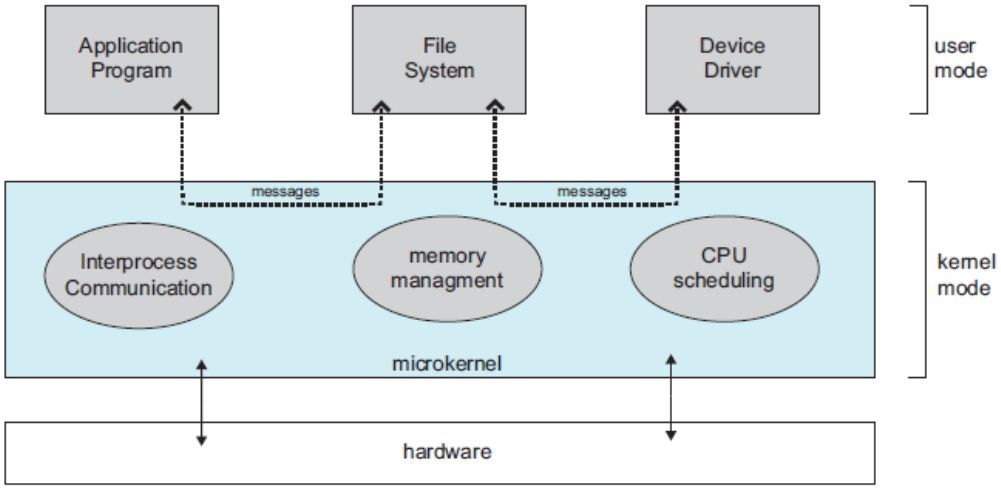


Figure- Architecture of a typical microkernel.

- 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.
- Communication is provided through **message passing.** For example, if the client program wishes to access a file, it must interact with the file server. 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. When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel. The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

4. 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 idea of the design is for the kernel to provide core services while other services are implemented dynamically, as the kernel is running.
- Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.
- Thus, for example, we might build CPU scheduling and memory management algorithms directly into the kernel and then add support for different file systems by way of loadable modules.

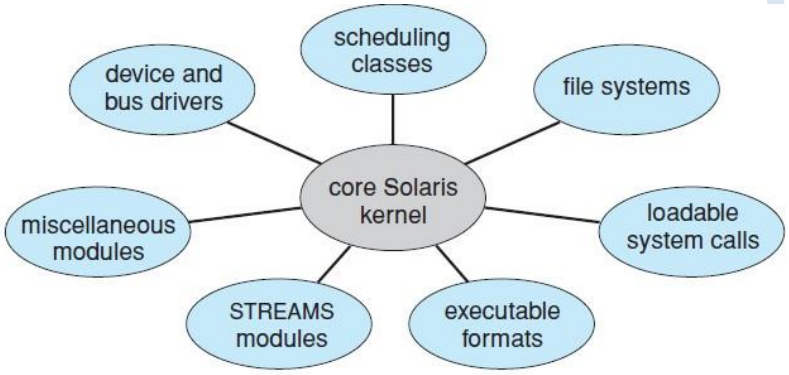


Figure- Solaris loadable modules

The Solaris operating system structure, shown in Figure, 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

Linux also uses loadable kernel modules, primarily for supporting device drivers and file systems.

5 Hybrid Systems

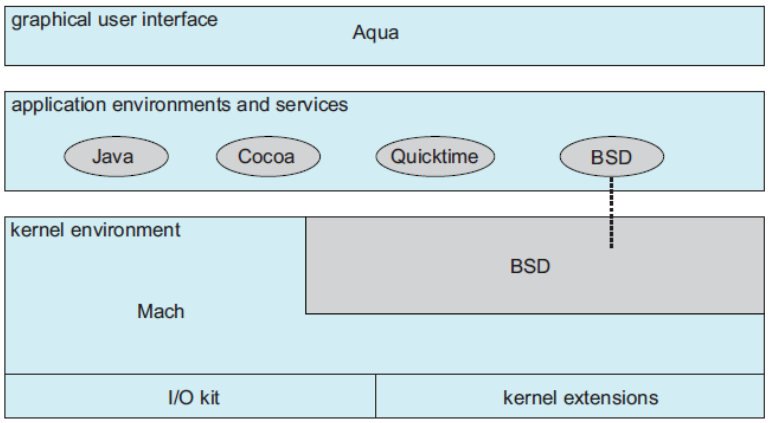
Very few operating systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. Three hybrid systems are as follows.

- Apple Mac OS X operating system

- -Two most prominent mobile operating systems—iOS and Android.

i) Mac OS X

- The Apple Mac OS X operating system uses a hybrid structure.

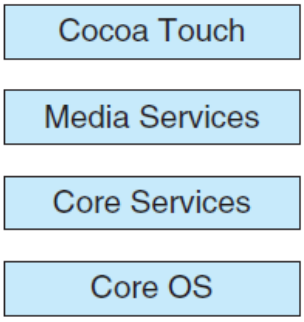


- It is a layered system. The top layers include the ***Aqua*** user interface and a set of application environments and services.
- The **Cocoa** environment specifies an API for the Objective-C programming language, which is used for writing Mac OS X applications.
- Below these layers is the ***kernel environment***, which consists primarily of the Mach microkernel and the BSD UNIX kernel.
- Mach provides memory management; support for remote procedure calls (RPCs) and interprocess communication (IPC) facilities, including message passing; and thread scheduling. The BSD component provides a BSD command-line interface, support for networking and file systems, and an implementation of POSIX APIs, including Pthreads.
- In addition to Mach and BSD, the kernel environment provides an I/O kit for development of device drivers and dynamically loadable modules

ii) iOS

- iOS is a mobile operating system designed by Apple to run its smartphone, the ***iPhone***, as well as its tablet computer, the ***iPad***.
- iOS is structured on the MacOS X operating system, with added functionality pertinent to mobile devices,but does not directly run Mac OS X applications.
- **Cocoa Touch** is an API for Objective-C that provides several frameworks for developing applications that run on iOS devices.
- Cocoa Touch provides support for hardware features unique to mobile devices, such as touch screens.
- The **media services** layer provides services for graphics, audio, and video.
- The **core services** layer provides a variety of features, including support for cloud computing and databases.

- The bottom layer represents the core operating system, which is based on the kernel environment
- The structure of iOS appears in Figure.



iii) Android

- The Android operating system was designed by the Open Handset Alliance (led primarily by Google) and was developed for Android smartphones and tablet computers.
- Android runs on a variety of mobile platforms and is open-sourced, partly explaining its rapid rise in popularity.
- Android is similar to iOS in that it is a layered stack of software that provides a rich set of frameworks for developing mobile applications.
- The structure of Android appears in Figure

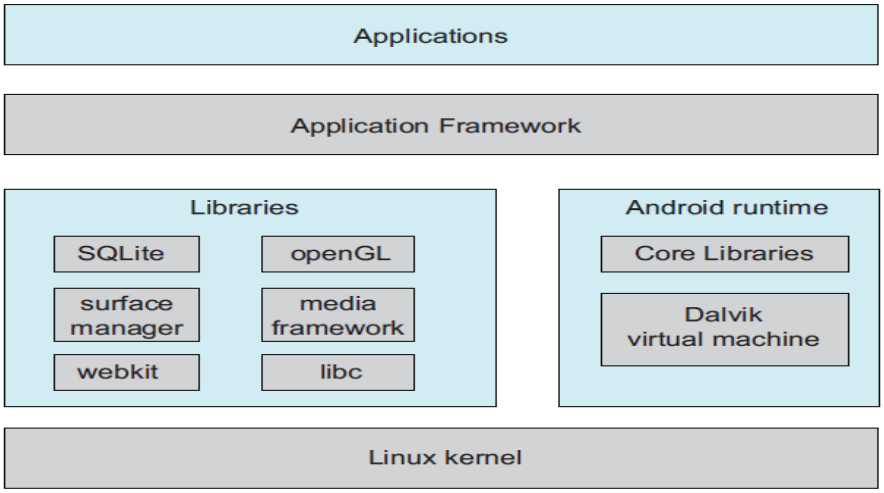


Figure - Architecture of Google's Android

- At the bottom of this software stack is the Linux kernel, although it has been modified by Google and is currently outside the normal distribution of Linux releases. Linux is used primarily for process, memory, and device-driver support for hardware and has been expanded to include power management.

- The Android runtime environment includes a core set of libraries as well as the Dalvik virtual machine. Software designers for Android devices develop applications in the Java language. The Java class files are first compiled to Java bytecode and then translated into an executable file that runs on the Dalvik virtual machine. The Dalvik virtual machine was designed for Android and is optimized for mobile devices with limited memory and CPU processing capabilities.
- The set of libraries available for Android applications includes frameworks for developing web browsers (webkit), database support (SQLite), and multimedia.
- The libc library is similar to the standard C library but is much smaller and has been designed for the slower CPUs that characterize mobile devices.

OPERATING-SYSTEM GENERATION

- 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 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.
- This SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there.
- The following kinds of information must be determined.
    - What CPU is to be used? What options are installed?
    - How will the boot disk be formatted? How many sections, or "partitions," will it be separated into, and what will go into each partition?
    - How much memory is available?
    - What devices are available? The system will need to know how to address each device (the device number), the device interrupt number, the device's type and model, and any special device characteristics.
    - What operating-system options are desired, or what parameter values are to be used?
- Once this information is determined, it can be used in several ways. At one extreme, a system administrator can use it to modify a copy of the source code of the operating system.
- 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. Selection allows the library to contain the device drivers for all supported I/O devices, but only those needed are linked into the operating system.
- Because the system is not recompiled, system generation is faster, but the resulting system may be overly general.
- At the other extreme, it is 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 generation involves simply creating the appropriate tables to describe the system.

SYSTEM BOOT

- After an operating system is generated, it must be made available for use by the hardware.
- The procedure of starting a computer by loading the kernel is known as **booting** the system. On most computer systems, 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.
- This program is in the form of **read-only memory (ROM)**, because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.
- The bootstrap program can perform a variety of tasks. Usually, one taskis to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.
- For large operating systems, the bootstrap loader is stored in firmware, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) 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.
- **GRUB** is an example of an open-source bootstrap program for Linux systems
- A disk that has a boot partition is called a **boot disk** or **system disk**.
- When the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be **running**.