UNIT- I OPERATING SYSTEMS OVERVIEW

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

COMPUTER SYSTEM OVERVIEW:

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

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

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

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

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

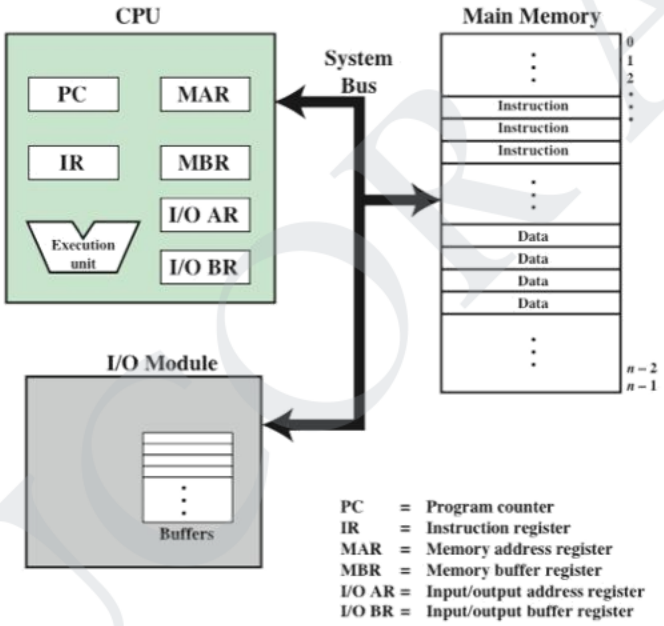


Figure 3.2 Computer Components: Top-Level View

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

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

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

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

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

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

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

PROCESSOR REGISTERS:

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

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

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

1. User-Visible Registers:

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

Data Registers

Address Registers

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

Address registers contain main memory addresses of data and instructions. Examples of address registers include the following:

□ Ā □ Ā □ Ā □ Ā □ Ā

ndex register.

□ Ā □ Ā □ Ā □ Ā □ Ā □

egment pointer

□ Ā □ Ā □ Ā □ Ā □ Ā □

tack pointer

Control and status register:

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

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

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

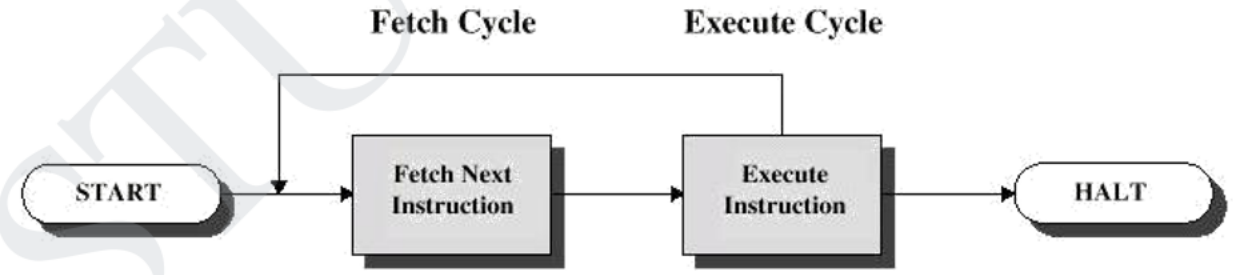
INSTRUCTION EXECUTION:

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

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

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

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



Instruction Fetch and Execute:

At the beginning of each instruction cycle, the processor fetches an instruction from memory.
The instruction contains bits that specify the action the processor is to take. The processor interprets the

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

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

(a) Instruction format

(b) Integer format

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

(c) Internal CPU registers

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

(d) Partial list of opcodes

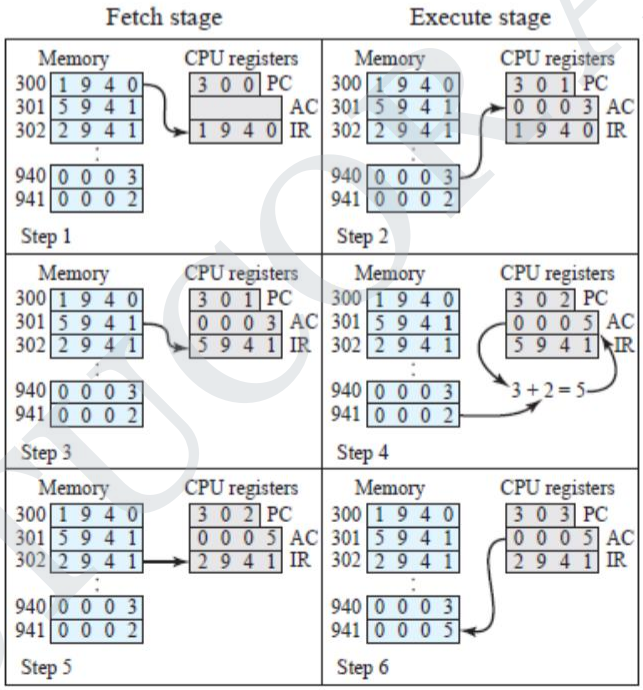


Figure 1.4 Example of Program Execution (contents of memory and registers in hexadecimal)

Example:

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

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

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

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

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

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

The contents of the AC are stored in location 941.

I/O Function:

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

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

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

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

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

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

Interrupts are provided primarily as a way to improve processor utilization

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

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

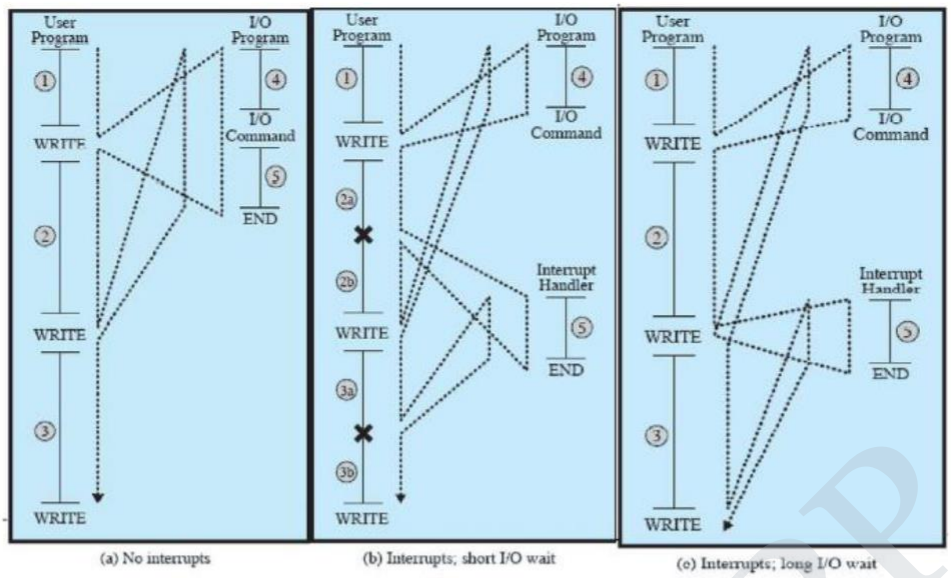
The WRITE calls are the call to an I/O routine that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:

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

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

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

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



Program Flow of Control without and with Interrupts

Interrupts and the Instruction Cycle:

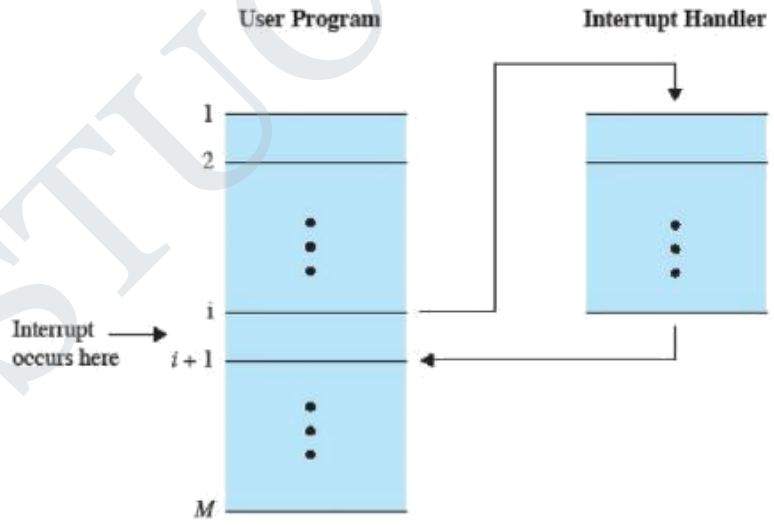
With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.

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

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

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

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



Transfer of control via Interrupts

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

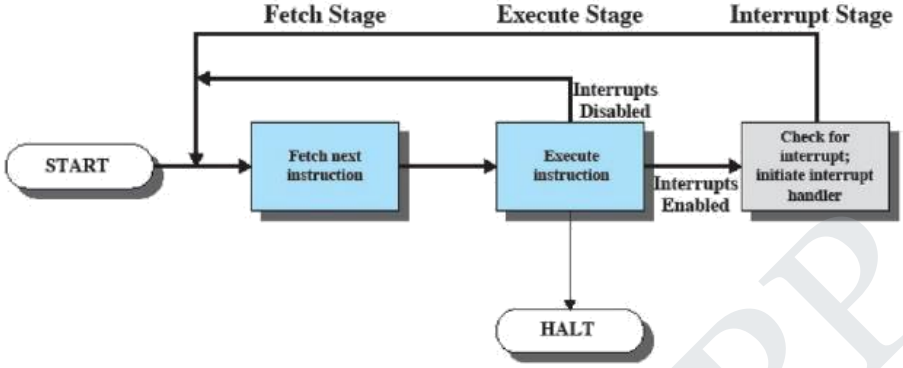


Figure 1.7 Instruction Cycle with Interrupts

Interrupt Processing:

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

The device issues an interrupt signal to the processor.

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

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

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

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

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

The interrupt handler may now proceed to process the interrupt.

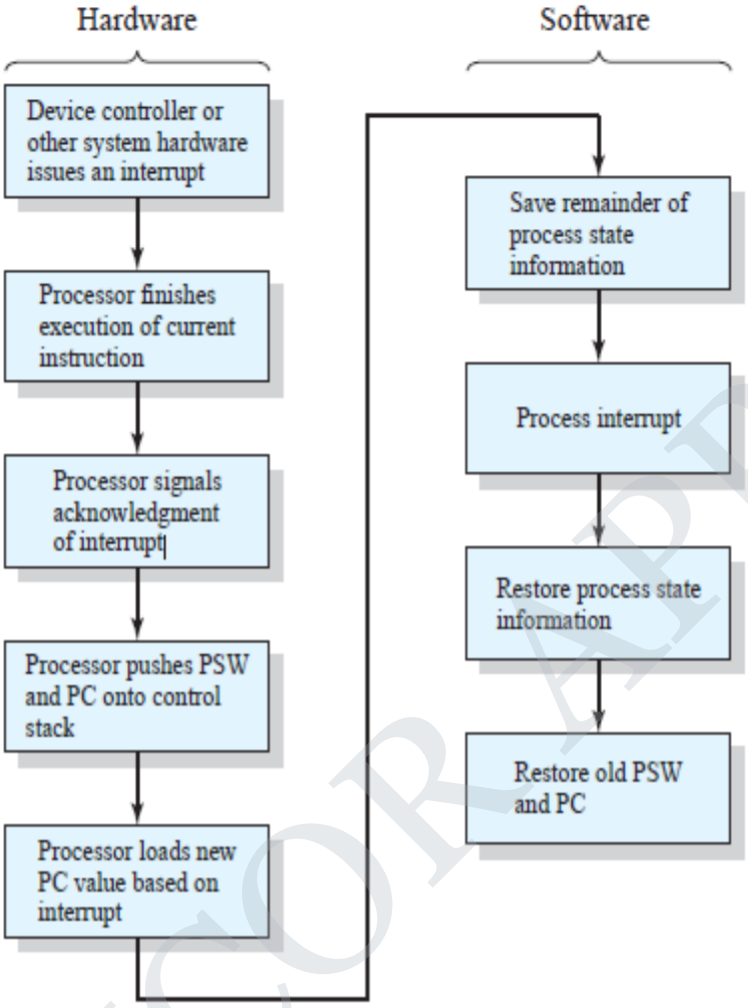
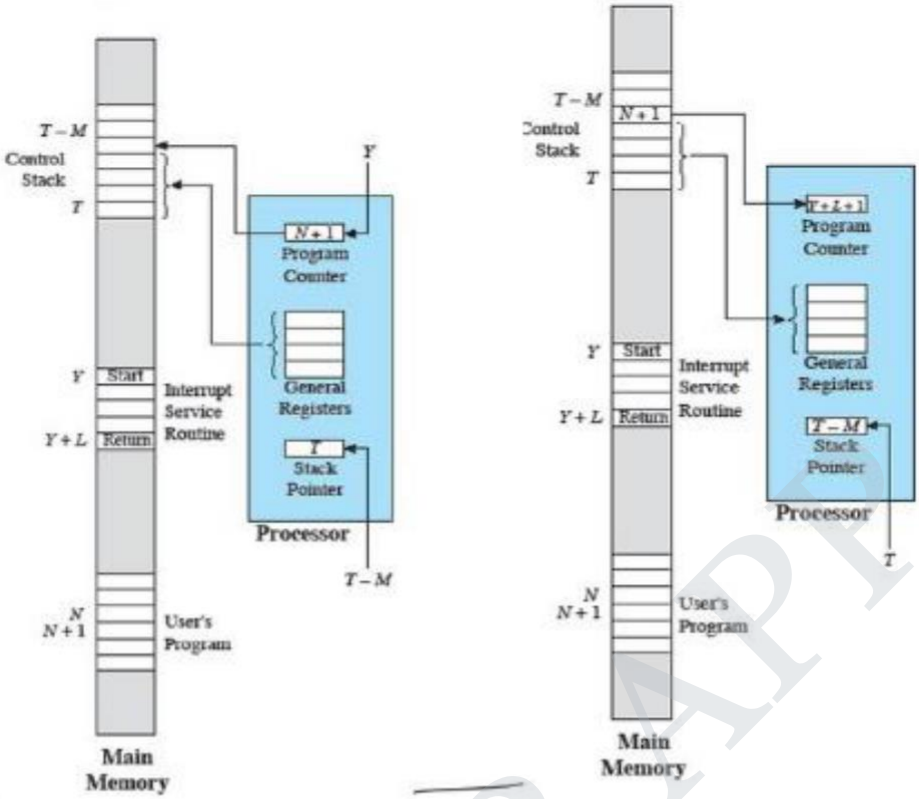


Figure 1.10 Simple Interrupt Processing

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

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

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



(a) Interrupt occurs after instruction at location N (b) Return from interrupt

Multiple Interrupts:

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

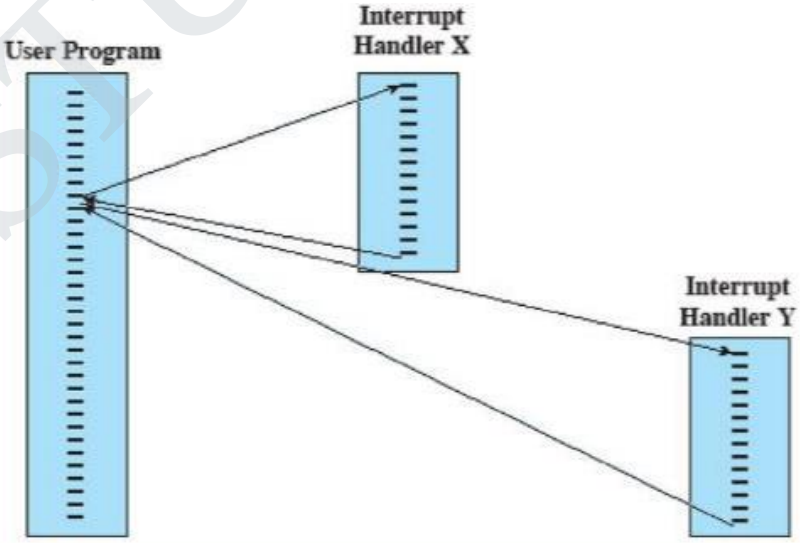
Two approaches can be taken to dealing with multiple interrupts.

Sequential interrupt processing

Nested interrupt processing

Sequential interrupt processing:

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



(a)Sequential Interrupt processing.

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

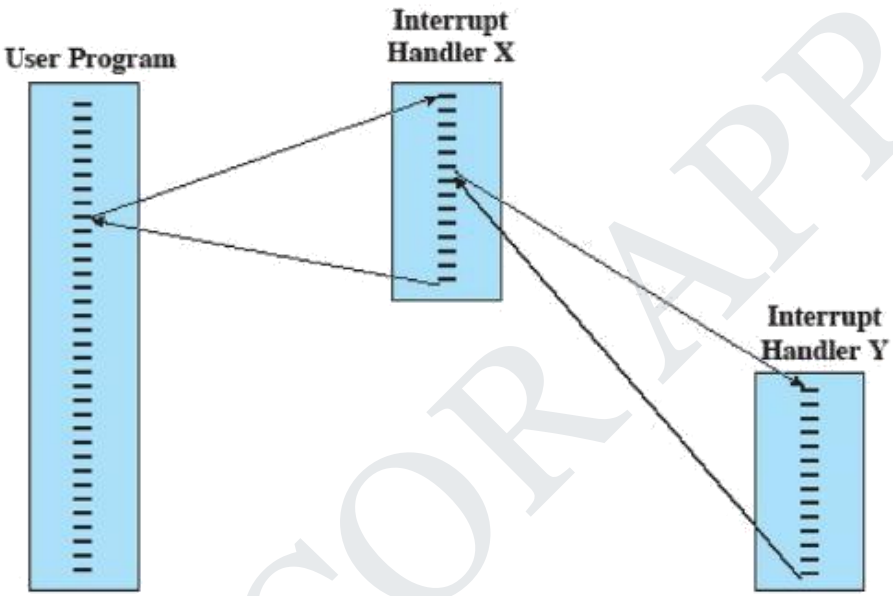
Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the interrupt-handler routine completes, interrupts are reenabled before resuming the user program and the processor checks to see if additional interrupts have occurred.

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

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

Nested interrupt processing:

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



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

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

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

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

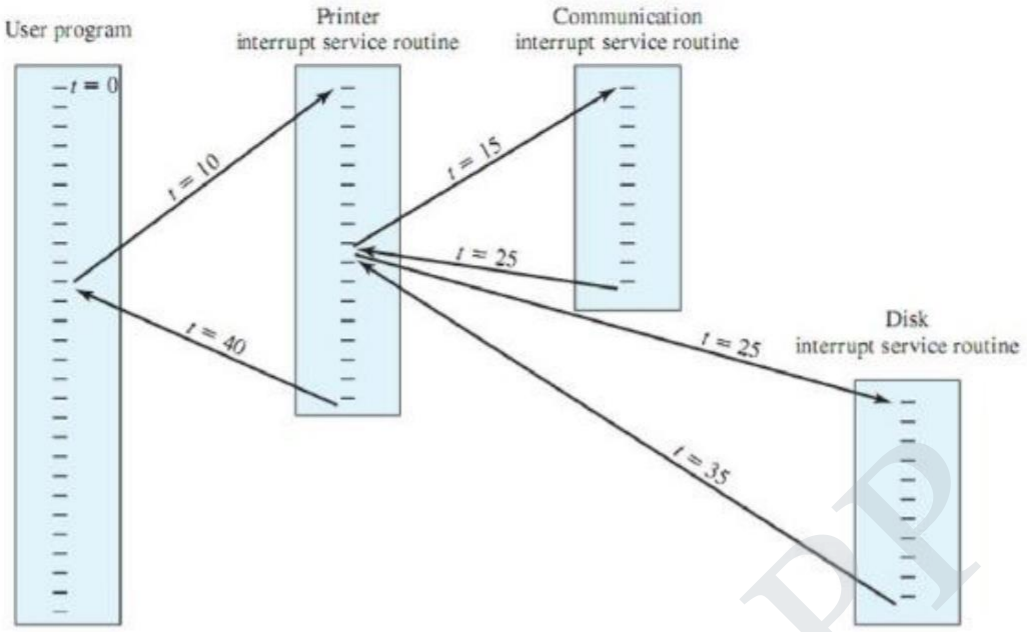


Figure 1.13 Example Time Sequence of Multiple Interrupts

Multiprogramming:

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

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

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

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

MEMORY HIERARCHY:

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

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

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

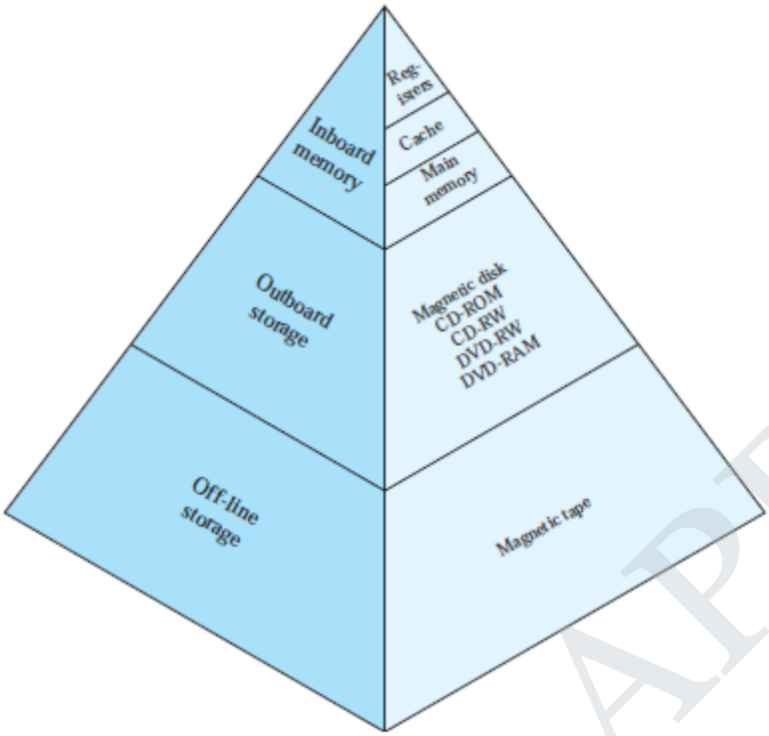
There is a tradeoff among the three key characteristics of memory: namely, capacity, access time, and cost.

Faster access time, greater cost per bit
Greater capacity, smaller cost per bit
Greater capacity, slower access speed

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

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

Decreasing cost per bit
Increasing capacity
Increasing access time
Decreasing frequency of access to the memory by the processor



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

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

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

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

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

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

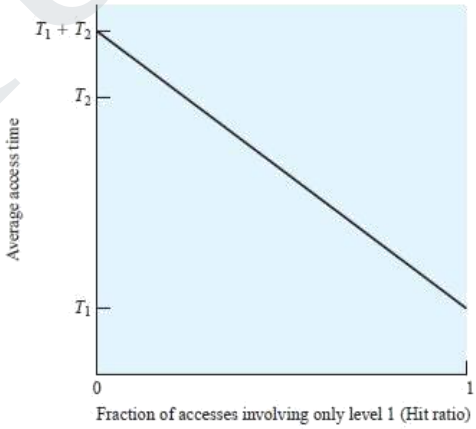


Figure 1.15 Performance of a Simple Two-Level Memory

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

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

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

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

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

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

CACHE MEMORY:

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

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

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

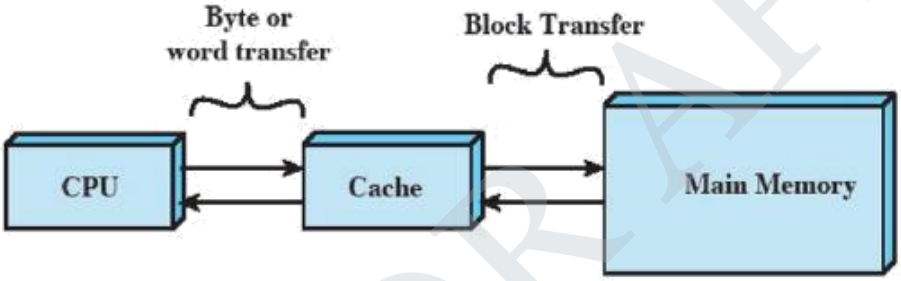


Figure 1.16 Cache and Main Memory

CACHE PRINCIPLES:

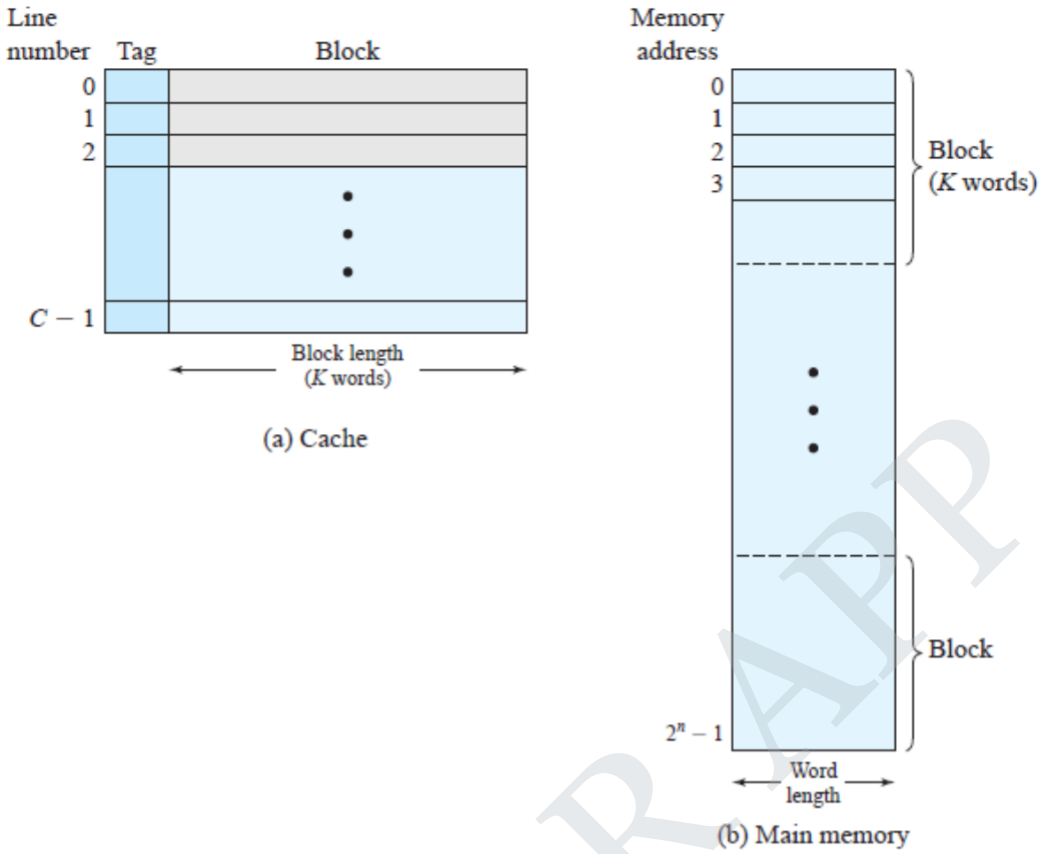
Cache memory provide memory access time similar to that of fastest memories available and at the same time support a large memory size that has the price of less expensive types of semiconductor memories.

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

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

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

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



Cache / Main memory

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

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

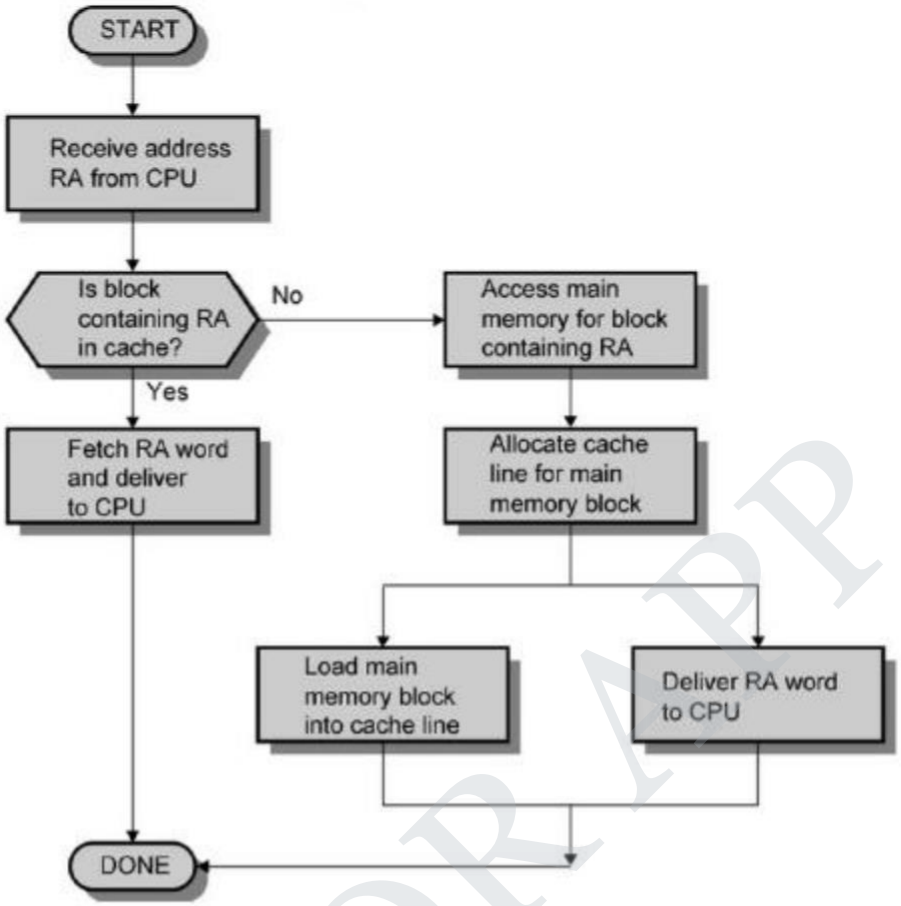
Each slot includes a tag that identifies which particular block is currently being stored. The tag is usually some number of higher-order bits of the address and refers to all addresses that begin with that sequence of bits.

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

READ OPERATION IN A CACHE:

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

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



CACHE READ OPERATION

CACHE DESIGN:

The key elements of cache design includes,

- Cache size
- Block Size
- Mapping Function
- Replacement algorithm
- Write policy

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

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

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

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

When a new block of data is read into the cache, the mapping function determines which cache location the block will occupy.

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

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

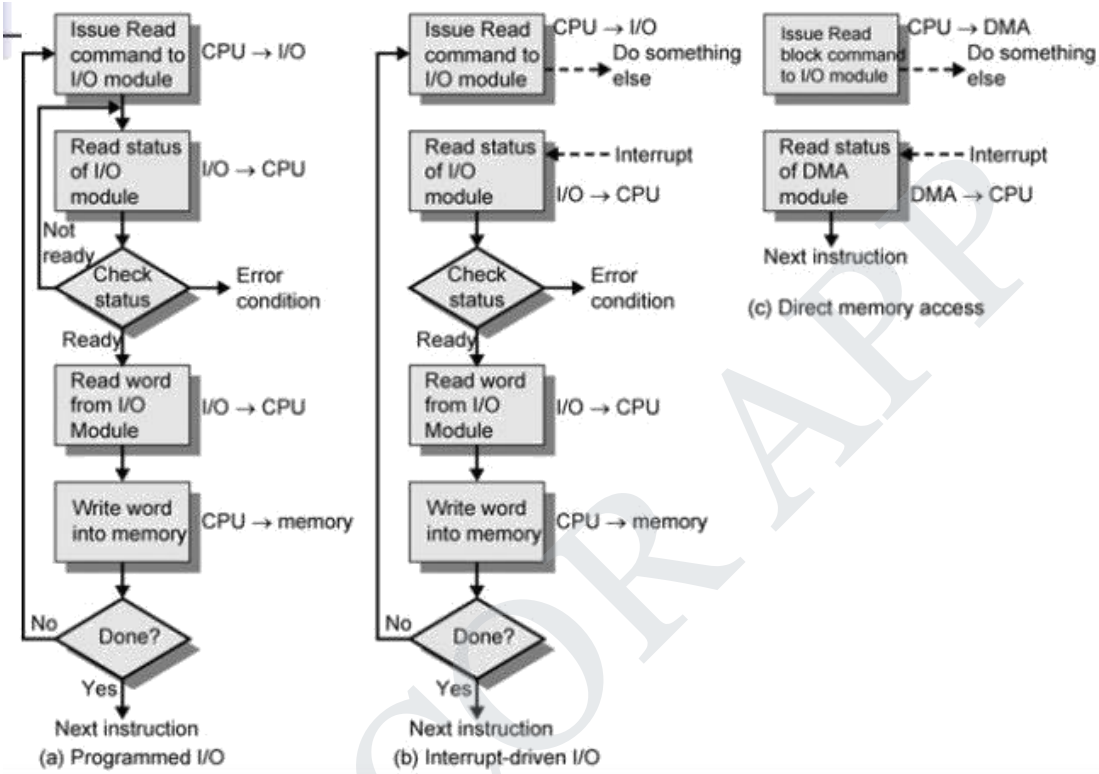
When a system writes data to cache, it must at some point write that data to the backing store as well. The timing of this write is controlled by what is known as the **write policy.**There are two basic writing approaches:

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

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

I/O COMMUNICATION TECHNIQUES

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



Three techniques are possible for I/O operations:

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

Programmed I/O:

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

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

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

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

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

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

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

Interrupt-Driven I/O:

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

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

DIRECT MEMORY ACCESS

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

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

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

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

MULTIPROCESSOR AND MULTICORE ORGANIZATION:

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

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

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

Symmetric Multiprocessors:

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

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

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

All processors can perform the same functions

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

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

Advantages of Symmetric multiprocessors:

Increased throughput.

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

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

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

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

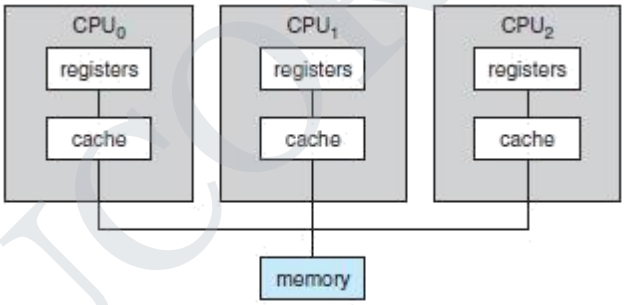


Figure 1.6 Symmetric multiprocessing architecture.

The Disadvantage of symmetric multiprocessor includes

If one processor fails then it will affect in the speed
Multiprocessor systems are expensive
Complex OS is required 4) Large main memory required.

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

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

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

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

MULTICORE ORGANIZATION:

A dual-core design contains two cores on the same chip.
In this design, each core has its own register set as well as its own local cache. Other designs might use a shared cache or a combination of local and shared caches.
Performance has also been improved by the increased complexity of processor design to exploit parallelism in instruction execution and memory access.

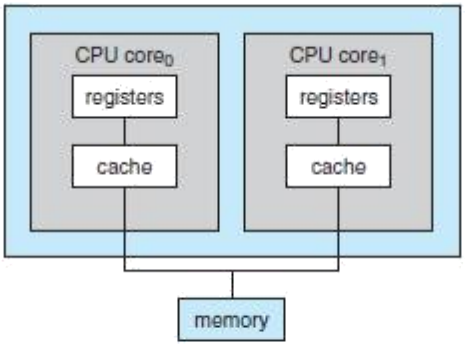


Figure 1.7 A dual-core design with two cores placed on the same chip.

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

OPERATING SYSTEM OVERVIEW:

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

OPERATING SYSTEM OBJECTIVES AND FUNCTIONS:

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

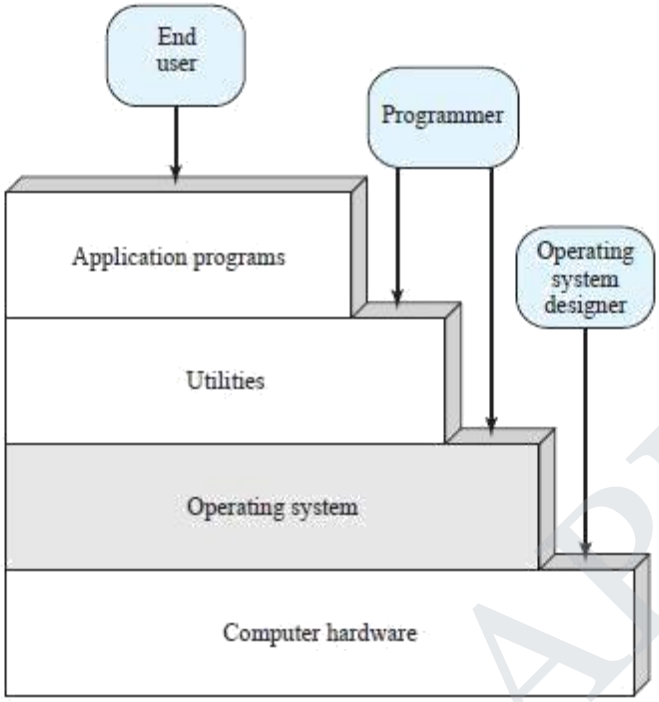
Convenience
Efficiency
Ability to evolve

The three other aspects of the operating system are
The operating system as a user or computer interface
The operating system as a resource manager
Ease of evolution of an operating system.

The Operating System as a User/Computer Interface

The user of those applications, the end user, generally is not concerned with the details of computer hardware.
An application can be expressed in a programming language and is developed by an application programmer.
A set of system programs referred to as utilities implement frequently used functions that assist in program creation, the management of files, and the control of I/O devices.

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



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

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

The Operating System as Resource Manager

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

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

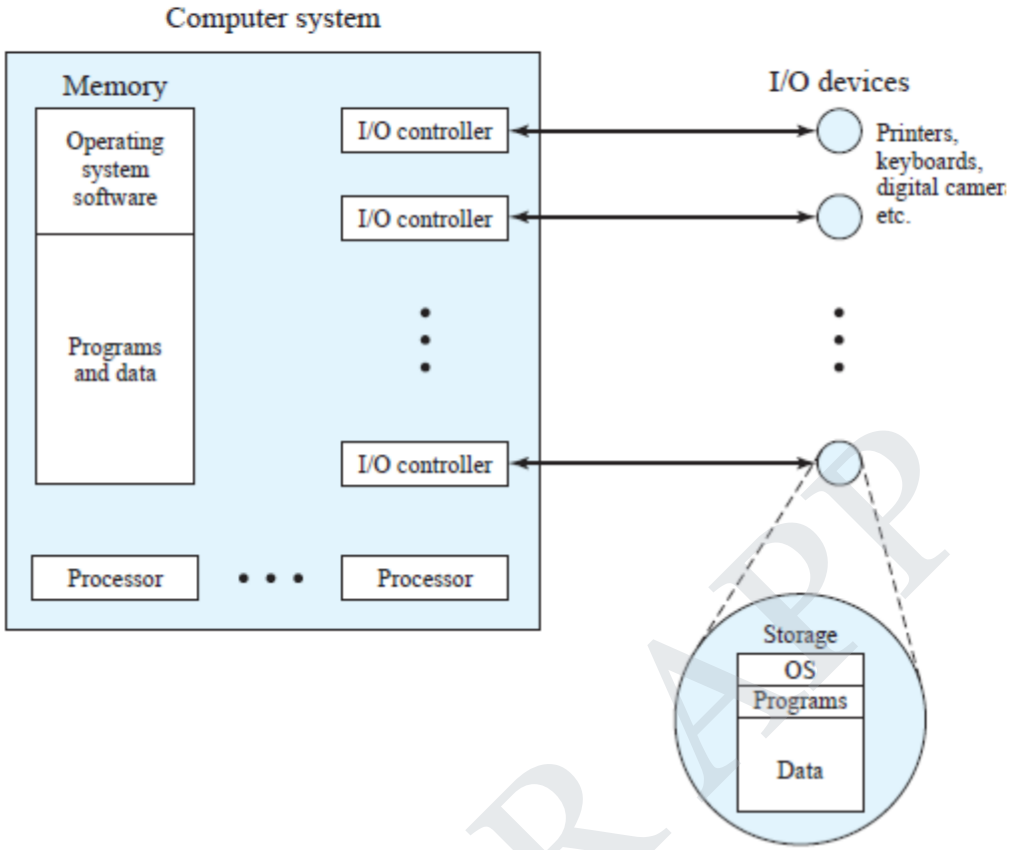
The OS frequently relinquishes control and must depend on the processor to allow it to regain control.

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

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

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

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



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

Ease of Evolution of an Operating System

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

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

The functions of operating system includes,

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

EVOLUTION OF OPERATING SYSTEM:

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

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

Serial processing

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

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

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

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

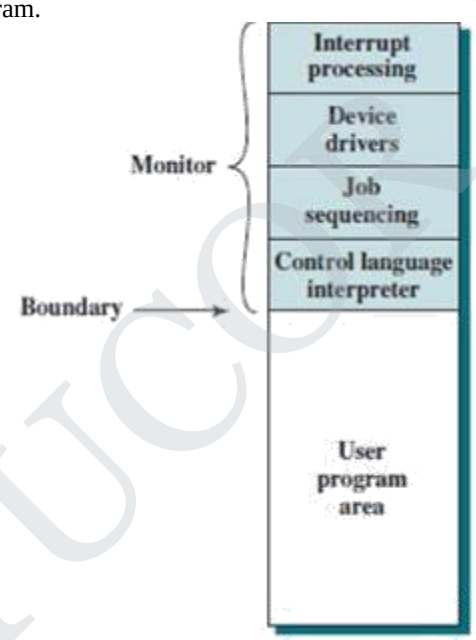
This mode of operation could be termed serial processing, reflecting the fact that users have access to the computer in series

Simple Batch Systems

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

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

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



Memory Layout for a Resident Monitor

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

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

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

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

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

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

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

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

Memory protection
Timer
Privileged instructions
Interrupts.

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

User Mode
Kernel Mode.

Thus the simple batch system improves utilization of the computer

Multiprogrammed Batch Systems:

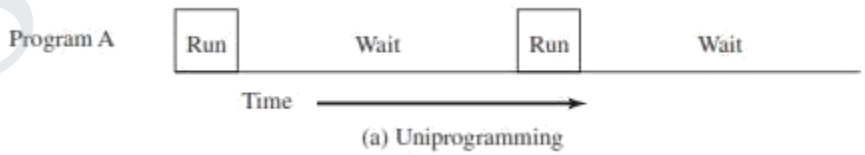
Even in simple batch operating system, the processor is often idle. The problem is that I/O devices are slow compared to the processor.

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

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

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

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



(a) Uniprogramming

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

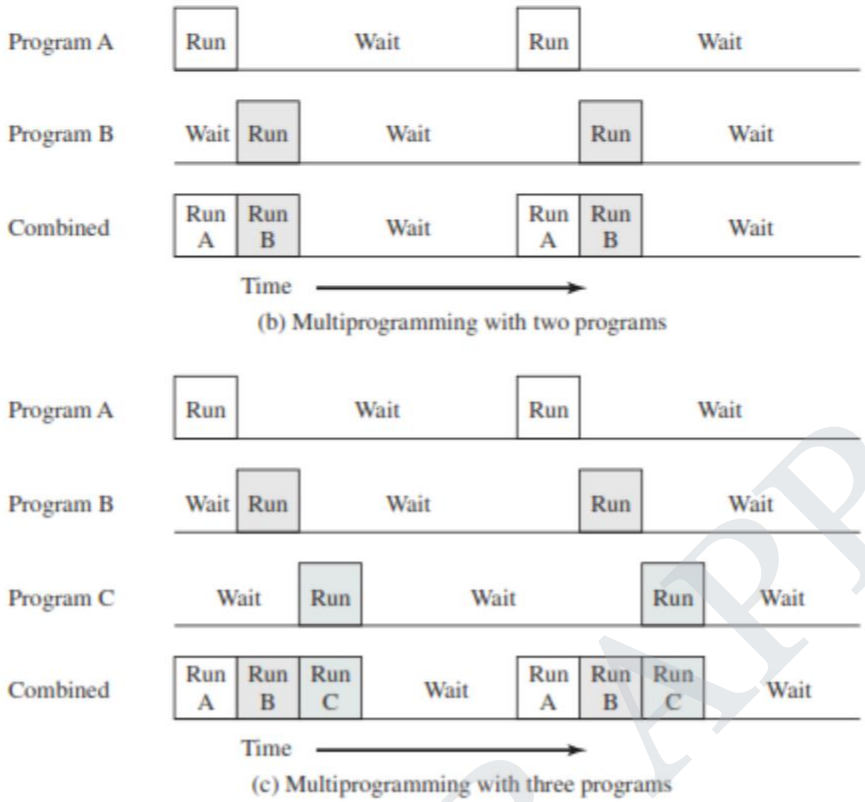


Figure 2.5 Multiprogramming Example

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

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

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

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

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

Time-Sharing Systems:

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

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

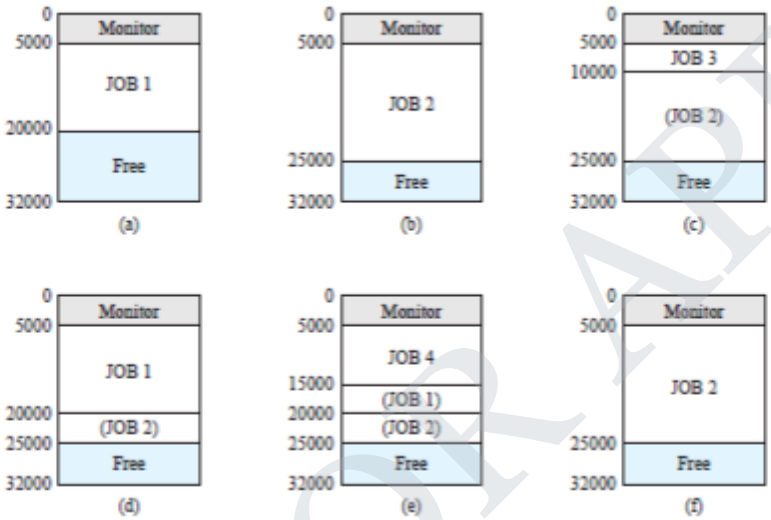
If there are n users actively requesting service at one time, each user will only see on the average 1/n of the effective computer capacity.

Batch Multiprogramming Vs Time Sharing systems

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

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

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



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

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

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

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

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

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

COMPUTER SYSTEM ORGANIZATION:

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

Computer system operation:

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

The bootstrap loader It initializes all aspects of the system, from CPU registers to device controllers to memory contents.

The bootstrap program loads the operating system and start executing that system.

Once the kernel is loaded and executing, it can start providing services to the system and its users. When is the system is booted it waits for some event to occur.

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software.

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. That contains the starting address of the service routine for the interrupt.

The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

Storage structure:

The CPU can load instructions only from memory, so any programs to run must be stored in main memory.

Main memory commonly is implemented in a semiconductor technology called **dynamic random-access memory**

ROM is a read only memory that is used to store the static programs such as bootstrap loader.

All forms of memory provide an array of bytes. Each byte has its own address. The operations are done through load or store instructions.

The load instruction moves a byte or word from main memory to an internal register within the CPU, whereas the store instruction moves the content of a register to main memory.

Ideally, we want the programs and data to reside in main memory permanently.

Main memory is usually too small to store all needed programs and data permanently

Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.

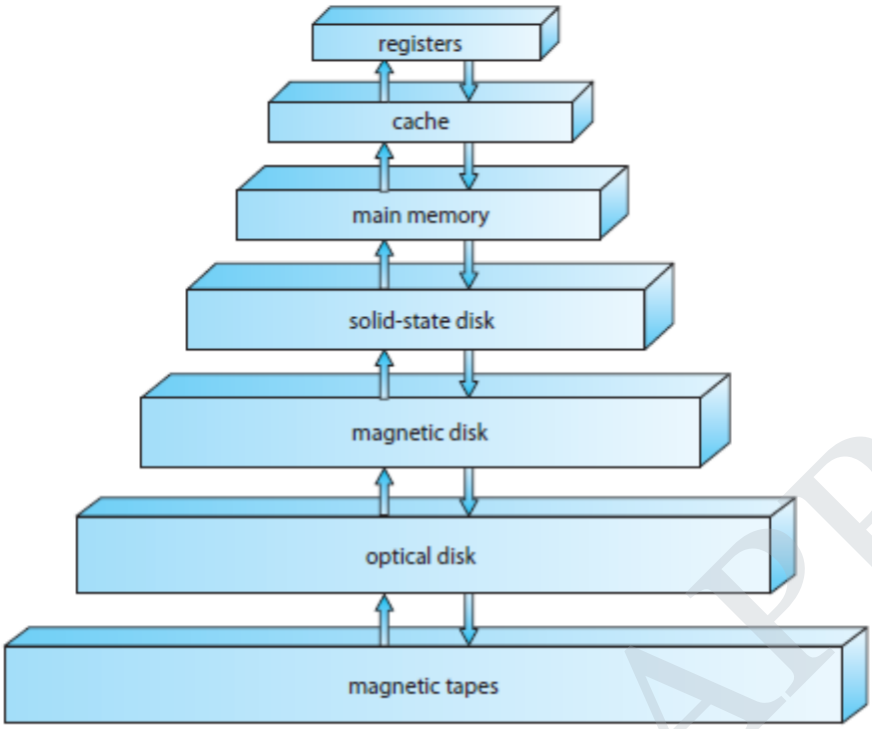
Most computer systems provide **secondary storage** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently.

The wide variety of storage systems can be organized in a hierarchy according to speed and cost.

The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases

Volatile storage loses its contents when the power to the device is removed so that the data must be written to **nonvolatile storage** for safekeeping.

Caches can be installed to improve performance where a large difference in access time or transfer rate exists between two components.



I/O Structure:

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

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

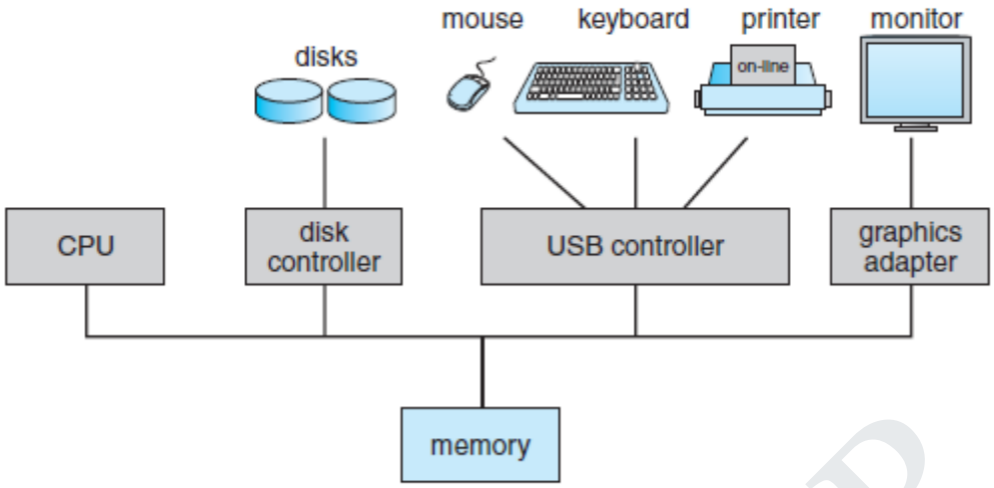
The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage

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

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

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

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



OPERATING SYSTEM STRUCTURE:

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

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

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

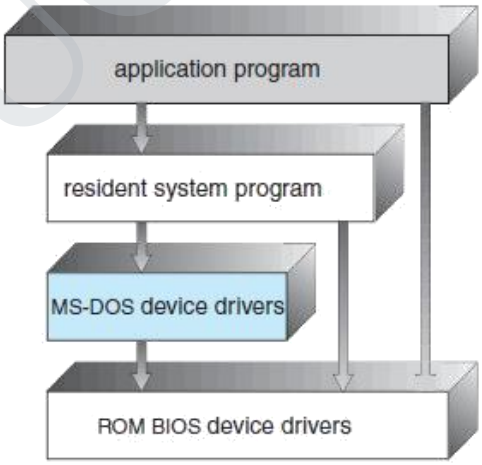
Simple structure:

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

Example: MS-DOS.

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

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



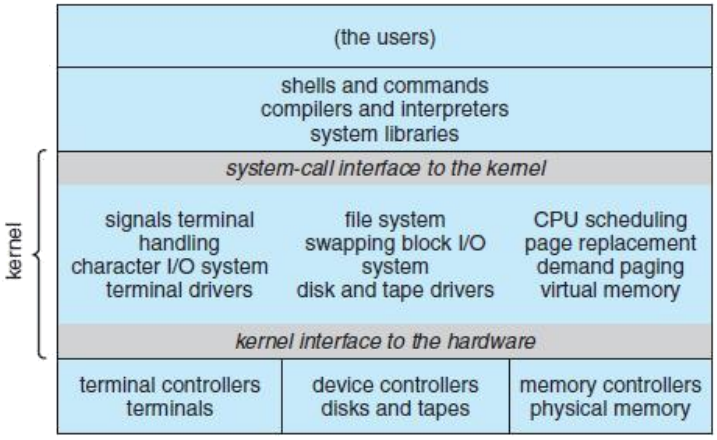
Example: Traditional UNIX OS

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

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

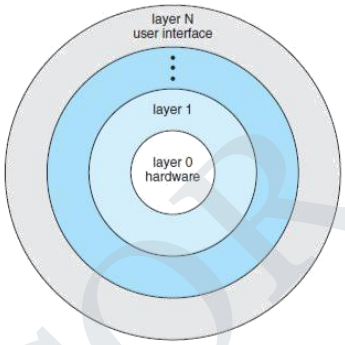
The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls.

This monolithic structure was difficult to implement and maintain.



Layered approach:

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



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

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

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

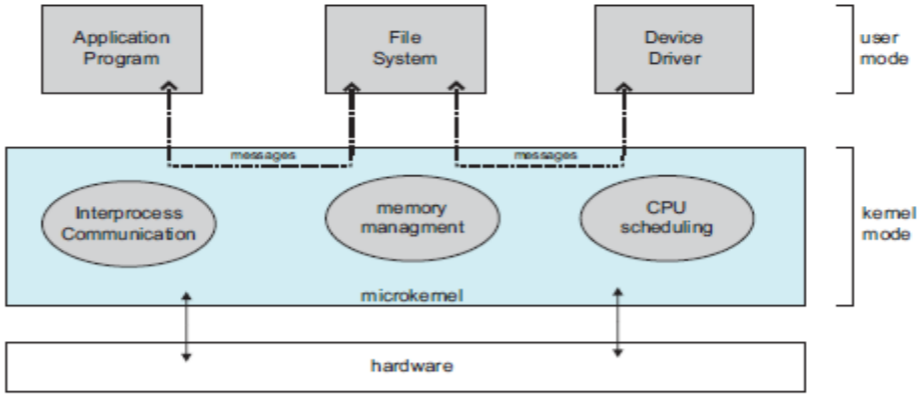
The major difficulty with the layered approach involves appropriately defining the various layers because a layer can use only lower-level layers.

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

Microkernels:

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

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



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

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

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

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

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

Modules:

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

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

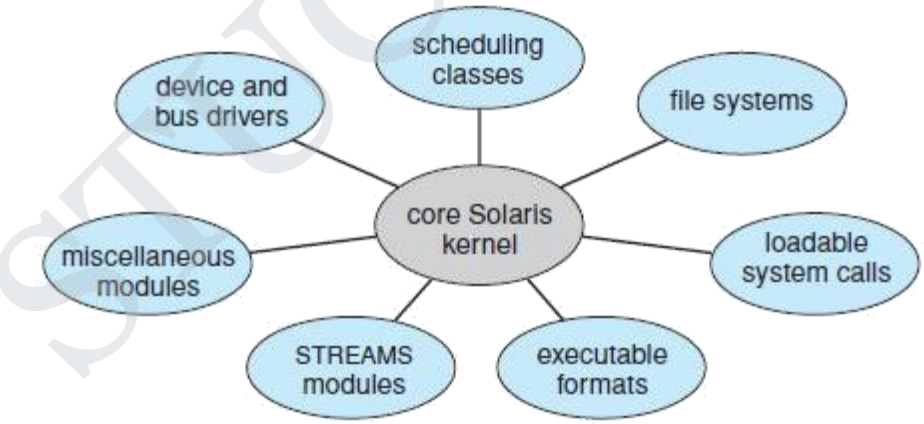
The kernel provides core services while other services are implemented dynamically, as the kernel is running.

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

Example: Solaris OS

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

Scheduling classes
File systems
Loadable system calls



Executable formats
STREAMS modules
Miscellaneous
Device and bus drivers

Hybrid Systems:

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

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

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

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

OPERATING SYSTEM OPERATIONS:

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

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

Dual-Mode and Multimode Operation:

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

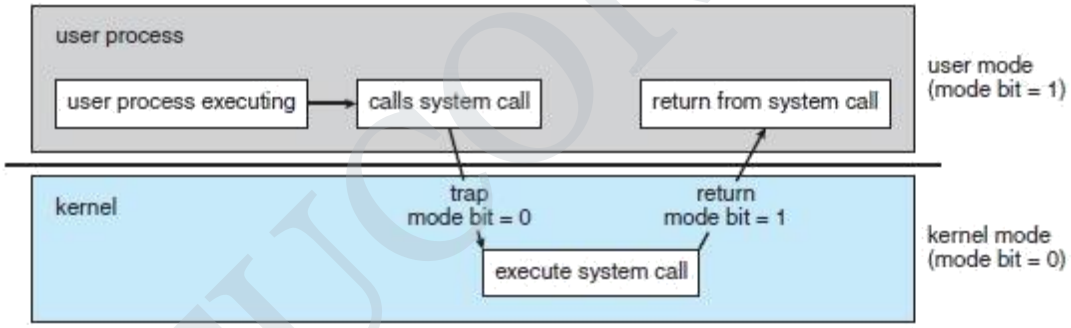
The computer systems provide hardware support that allows us to differentiate among various modes of execution.

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

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

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

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



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

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

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

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

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

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

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

Timer:

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

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

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

SYSTEM CALLS:

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

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

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

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

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

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

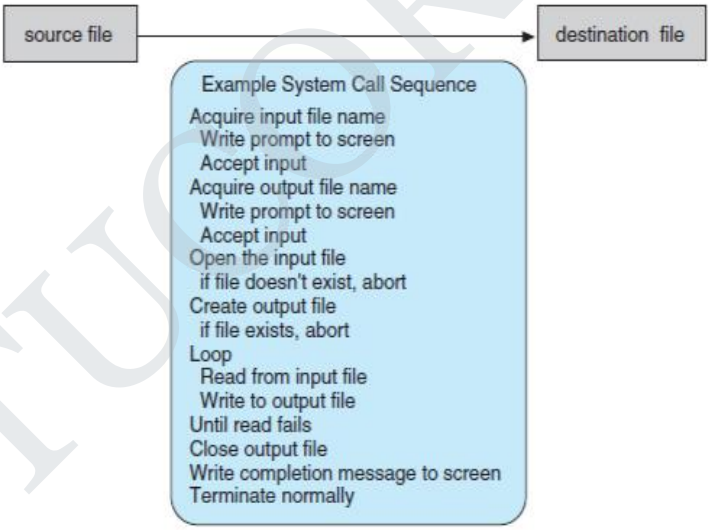


Figure 2.5 Example of how system calls are used.

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

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

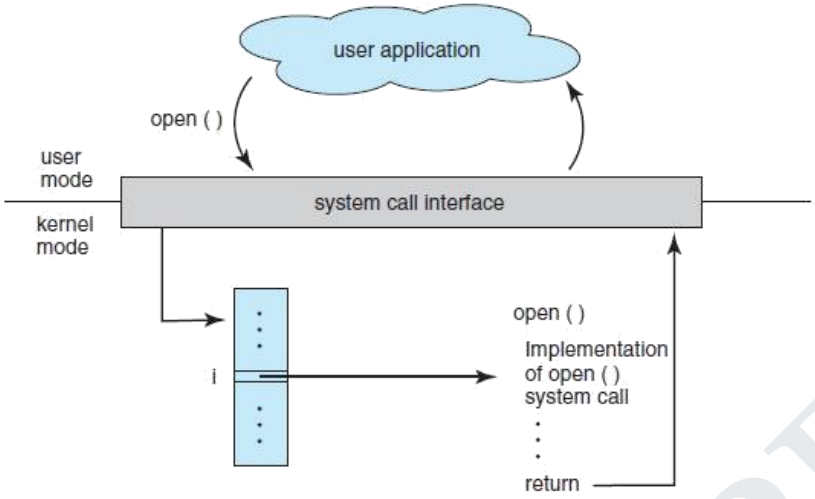


Figure 2.6 The handling of a user application invoking the `open()` system call.

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

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

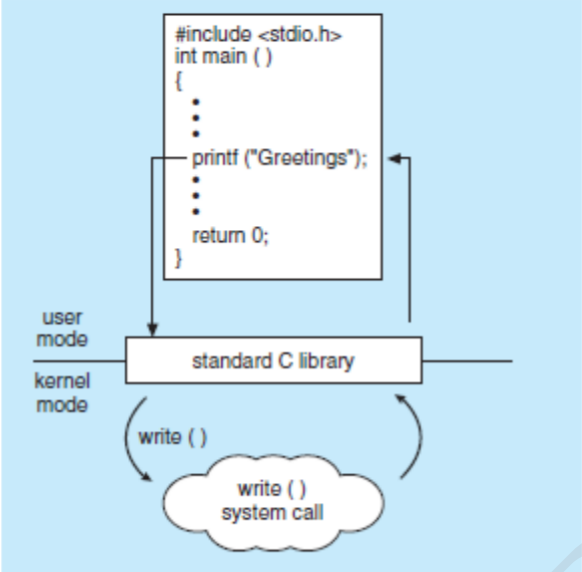
Types of System Calls:

System calls can be grouped roughly into six major categories

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

PROCESS CONTROL:

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



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

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

The System calls associated with process control includes

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

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

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

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

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

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

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

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

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

The System calls associated with File management includes

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

get file attributes, set file attributes

DEVICE MANAGEMENT:

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

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

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

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

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

The System calls associated with Device management includes

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

INFORMATION MAINTENANCE:

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

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

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

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

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

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

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

The System calls associated with Device management includes

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

COMMUNICATION:

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

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

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

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

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

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

The close connection() call terminates the communication

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

The system calls associated with communication includes,

create, delete communication connection
send, receive messages

Transfer status information
attach or detach remote devices

PROTECTION:

Protection provides a mechanism for controlling access to the resources provided by a computer system.

System calls providing protection include set permission () and get permission (), which manipulate the permission settings of resources such as files and disks.

The allow user () and deny user () system calls specify whether particular users can—or cannot—be allowed access to certain resources.

System programs, also known as system utilities, provide a convenient environment for program development and execution.

They can be divided into these categories:

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

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

Status Information: Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information.

Others are more complex, providing detailed performance, logging, and debugging information.

File Modification: Several text editors may be available to create and modify the content of files stored on disk or other storage devices

There may also be special commands to search contents of files or perform transformations of the text.

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

Program Loading and Execution: Once a program is assembled or compiled, it must be loaded into memory to be executed.

The system may provide absolute loaders, relocatable loader.

Communication: These programs provide the mechanism for creating virtual connections among processes, users, and computer systems.

They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.

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

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

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

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

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

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

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

The following kinds of information must be determined.

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

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

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

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

SYSTEM BOOT:

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

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

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

A complex boot program loads the OS

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

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

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

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

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

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

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

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

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

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

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

UNIT- II PROCESS MANAGEMENT

Processes - Process Concept, Process Scheduling, Operations on Processes, Inter-process Communication; CPU Scheduling - Scheduling criteria, Scheduling algorithms, Multiple-processor scheduling, Real time scheduling; Threads- Overview, Multithreading models, Threading issues; Process Synchronization - The critical-section problem, Synchronization hardware, Mutex locks, Semaphores, Classic problems of synchronization, Critical regions, Monitors; Deadlock - System model, Deadlock characterization, Methods for handling deadlocks, Deadlock prevention, Deadlock avoidance, Deadlock detection, Recovery from deadlock.

PROCESS:

A Process is defined as a program in execution.

A program is a passive entity, such as a file containing a list of instructions stored on disk

A process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.

A program becomes a process when an executable file is loaded into memory.

A process is more than the program code, which is sometimes known as the text section.

It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers.

A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables)

It contains a data section, which contains global variables.

A Process may also include a heap which is a memory that is dynamically allocated during process run time.

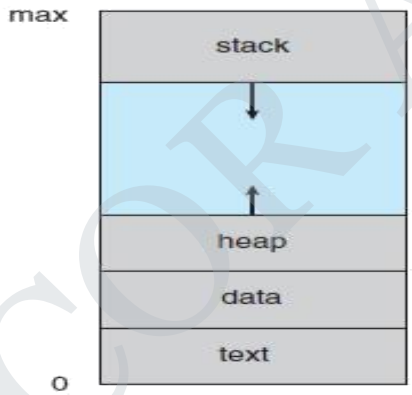


Figure 3.1 Process in memory.

Process State:

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

New. The process is being created.
Running. Instructions are being executed.
Waiting. The process is waiting for some event to occur
Ready. The process is waiting to be assigned to a processor.
Terminated. The process has finished execution.

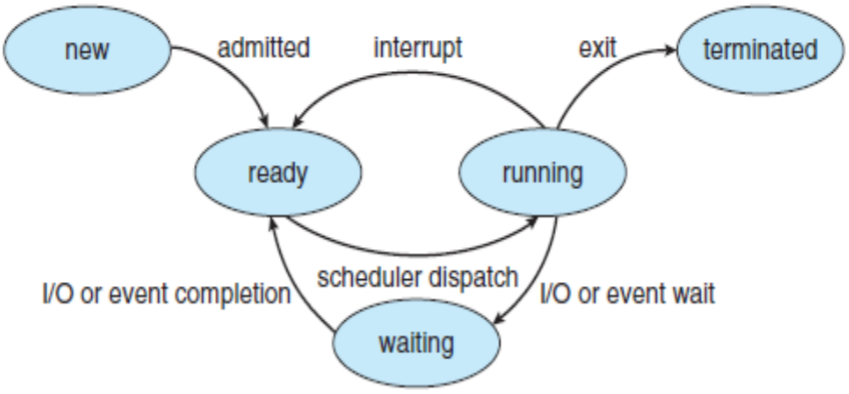


Figure 3.2 Diagram of process state.

Process Control Block:

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**.

It contains many pieces of information associated with a specific process

Process state. The state may be new, ready, running, waiting, halted, and so on.

Program counter. The counter indicates the address of the next instruction to be executed for this process.

CPU registers. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers

CPU-scheduling information. This information includes a process priority, pointers to scheduling queues

Memory-management information. This information may include such items as the value of the base and limit registers and the page tables

Accounting information. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers,

I/O status information. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

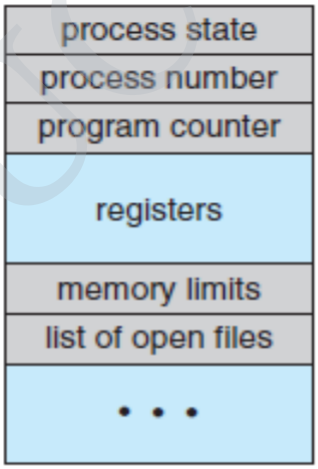


Figure 3.3 Process control block (PCB).

PROCESS SCHEDULING:

The **process scheduler** selects an available process for program execution on the CPU.

For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

Scheduling Queues:

The Scheduling Queues are of three types

Job Queue

Ready Queue

Device Queue

As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.

The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**

A ready-queue header contains pointers to the first and final PCBs in the list.

Each process that requires I/O Operation may have to wait for the device. The list of processes waiting for a particular I/O device is called a **device queue**.

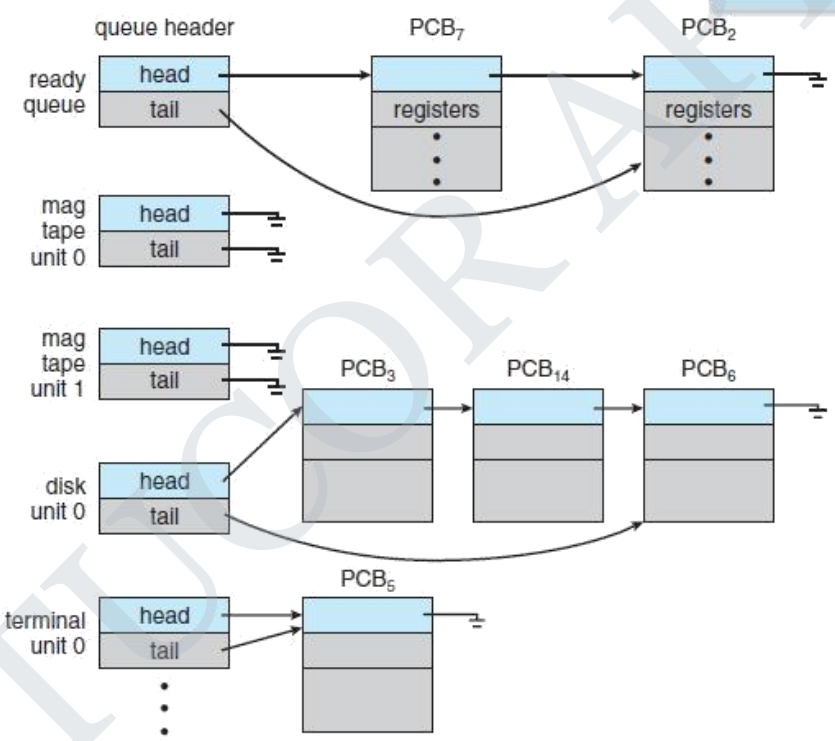


Figure 3.5 The ready queue and various I/O device queues.

A common representation of process scheduling is a **queuing diagram**

Two types of queues are present: **Ready queue and a set of device queues.** The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

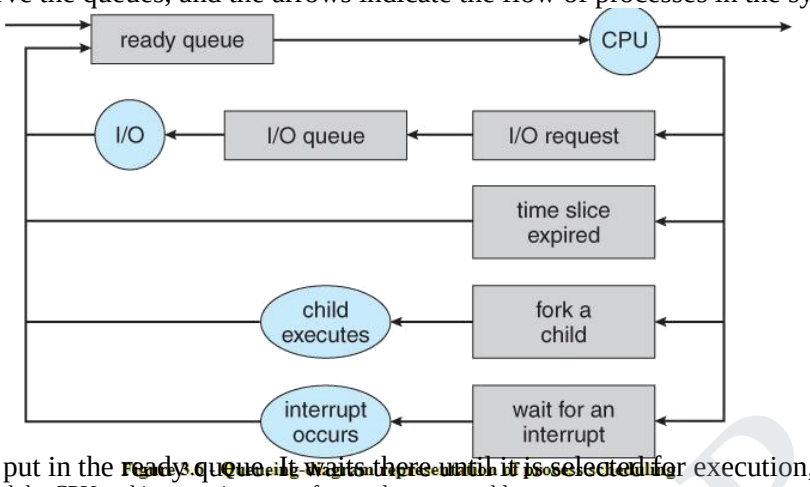


Figure 3.6 - Queueing diagram representation of process scheduling

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched.**

Once the process is allocated the CPU and is executing, one of several events could occur:

The process could issue an I/O request and then be placed in an I/O queue.

The process could create a new child process and wait for the child's termination.

The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Schedulers:

The operating system must select, for scheduling purposes, processes from the queues in some approach. The selection process is carried out by the appropriate **scheduler**.

It makes use of two types of schedulers

Long term scheduler or job scheduler
Short term scheduler or CPU scheduler.
Medium term scheduler

The **long-term scheduler**, or **job scheduler**, selects processes from the job queue and loads them into memory for execution.

The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

The Long term scheduler must have a careful selection of both I/O Bound and CPU Bound process.

An **I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations.

A **CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.

If all processes are I/O bound, the ready queue will almost always be empty, If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused.

The medium-term scheduler is used to remove a process from memory to reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.

The process is swapped out, and is later swapped in, by the medium-term scheduler.

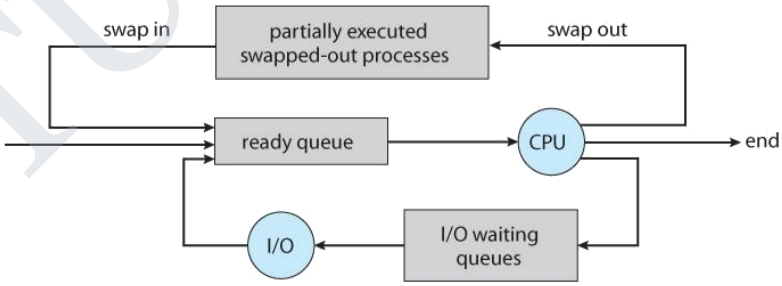
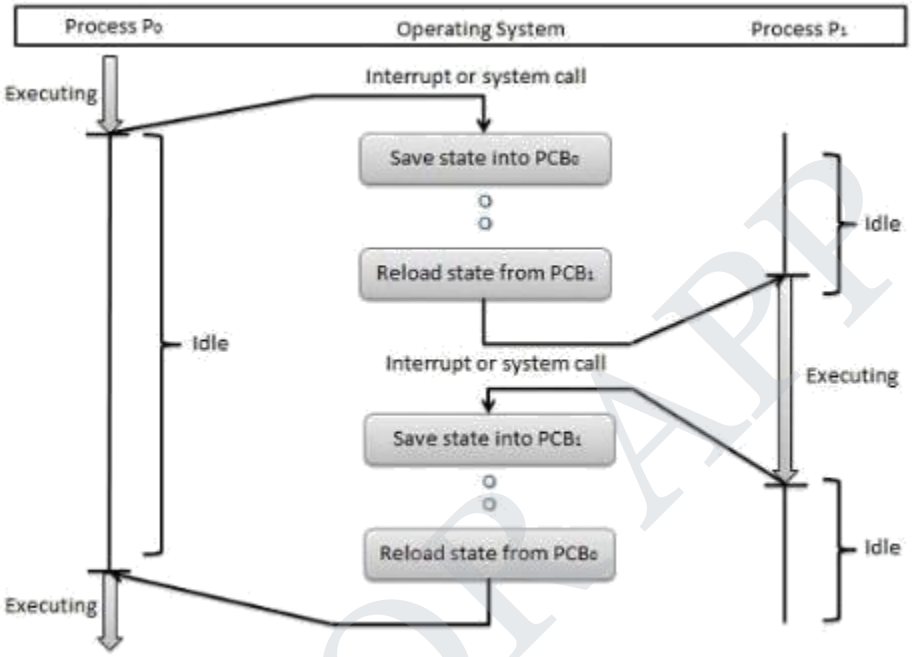


Figure 3.7 - Addition of a medium-term scheduling to the queueing diagram

Context Switch:

The process of switching the CPU from one process to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a **context switch.**
When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU so that it can restore that context when its processing is done.
The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory management information.



OPERATIONS ON PROCESSES:

The operating system must provide a mechanism for process creation and termination. The process can be created and deleted dynamically by the operating system.
The Operations on the process includes
Process creation
Process Termination

Process Creation:

During Execution a process may create several new processes.
The creating process is called as the **parent process** and the newly created process is called as the **child process**.
The operating systems identify the processes according to their unique process identifier.

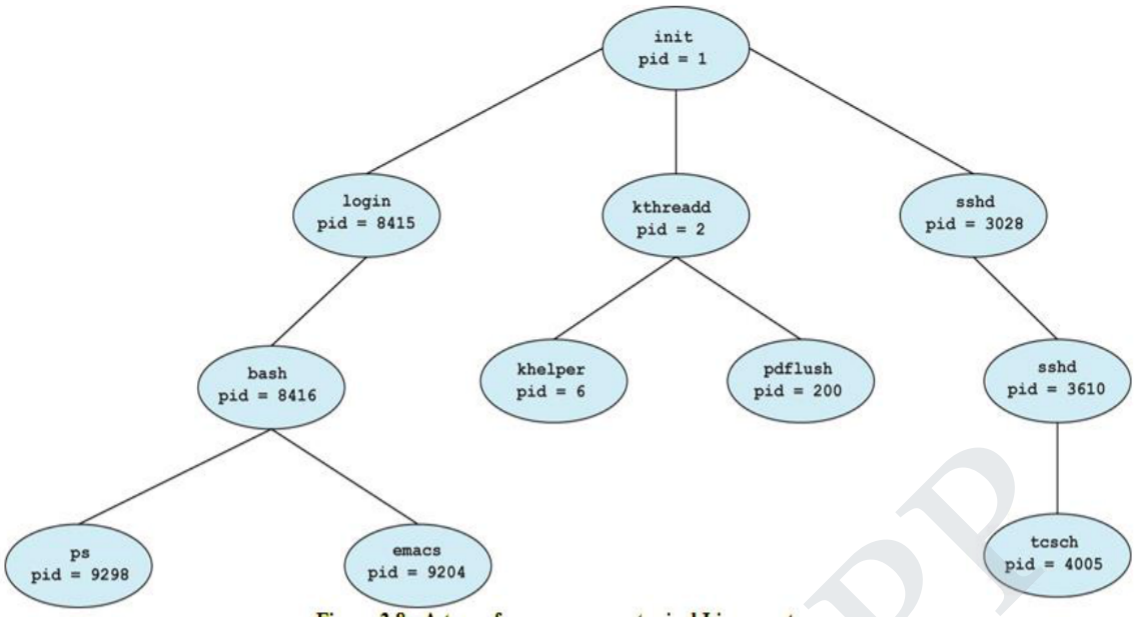


Figure 3.8 - A tree of processes on a typical Linux system

Example: Process tree for Linux operating system.

The init process serves as the root parent process for all the user process.

Once the system has booted, the init process can also create various user processes, such as a web or print server, an ssh server.

The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel

The sshd process is responsible for managing clients that connect to the system by using ssh(Secure shell)

The login process is responsible for managing clients that directly log onto the system

The command ps –el will list complete information for all processes currently active in the system.

When a process creates a new process, two possibilities for execution exist:

The parent continues to execute concurrently with its children.

The parent waits until some or all of its children have terminated

There are also two address-space possibilities for the new process:

The child process is a duplicate of the parent process (it has the same program as the parent).

The child process has a new program loaded into it.

The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program.

A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.

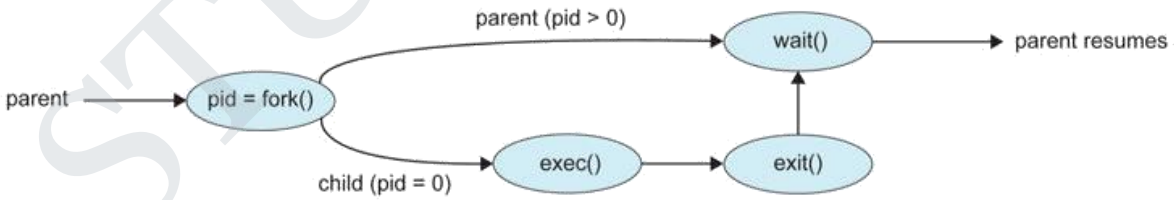


Figure 3.10 - Process creation using the fork( ) system call

Creating a separate process using the UNIX fork() system call:

The parent and child are concurrent processes running the same code instructions. Because the child is a copy of the parent, each process has its own copy of any data.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
Pid_t pid;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { /* child process */
execlp("/bin/ls","ls",NULL);
}
else { /* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
}
return 0;
}
```

Process Termination:

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.

At that point, the process may return a status value (typically an integer) to its parent process.

All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system

A parent may terminate the execution of one of its children for a variety of reasons, such as

- The child has exceeded its usage of some of the resources that it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon is referred to as **cascading termination.**

A parent process may wait for the termination of a child process by using the wait() system call

This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid t pid;
int status;
pid = wait(&status);
```

A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie** process.

INTERPROCESS COMMUNICATION:

A process can be either an independent process or a cooperating process.

A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Advantages of cooperating process i) Information sharing ii) Computation speedup iii) Modularity iv) Convenience**.**

DEFINITION:

An Interprocess communication is a mechanism that allows the cooperating process to exchange data and communication among each other.

There are two fundamental models of Interprocess communication

Shared Memory model

Message passing model

In shared memory model a region of memory is shared by the cooperating process. Processes can then exchange information by reading and writing data to the shared region.

In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Message passing is also easier to implement in a distributed system than shared memory.

The shared memory is faster than that of message passing.

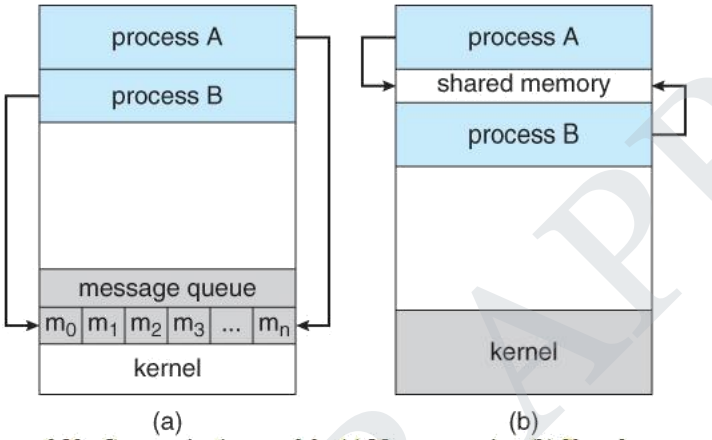


Figure 3.12 - Communications models: (a) Message passing. (b) Shared memory.

Shared-Memory Systems:

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.

Shared-memory region resides in the address space of the process creating the shared-memory segment.

Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

They can then exchange information by reading and writing data in the shared areas.

EXAMPLE: PRODUCER – CONSUMER PROCESS:

A **producer** process produces information that is consumed by a **consumer** process.

One solution to the producer–consumer problem uses shared memory

To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item.

Two types of buffers can be used.

Bounded Buffer.

Unbounded Buffer.

The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.

The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER SIZE 10
typedef struct {
. . .
}item;
item buffer[BUFFER SIZE];
int in = 0;
int out = 0;
```

The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFER SIZE) == out.

CODE FOR PRODUCER PROCESS:

```
item next produced;
while (true) {
/* produce an item in next produced */
while (((in + 1) % BUFFER SIZE) == out)
  /* do nothing */ buffer[in]
= next produced;
in = (in + 1) % BUFFER SIZE;
}
```

The producer process has local variable next produced in which the new item to be produced is stored.
The consumer process has a local variable next consumed in which the item to be consumed is stored.
This scheme allows at most BUFFER SIZE − 1 items in the buffer at the same time.

CODE FOR CONSUMER PROCESS

```
item next consumed;
while (true) {
while (in == out)
; /* do nothing */ 10
```

Direct or indirect communication

Direct Communication:

Each process that wants to communicate must explicitly name the recipient or sender of the communication.
Direct communication can be done in two ways symmetric addressing and asymmetric addressing.

In Symmetric addressing both the sender process and the receiver process must name the other to communicate.

send(P, message)—Send a message to process P.

receive(Q, message)—Receive a message from process Q.

In Asymmetric addressing only the sender names the recipient; the recipient is not required to name the sender.

send(P, message)—Send a message to process P.

receive(id, message)—Receive a message from any process

A communication link in this scheme has the following properties:

A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.

A link is associated with exactly two processes.

Between each pair of processes, there exists exactly one link.

With **indirect communication**, the messages are sent to and received from **mailboxes**, or **ports**.

A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification

The send() and receive() primitives are defined as follows:

send(A, message)—Send a message to mailbox A.

receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

A link is established between a pair of processes only if both members of the pair have a shared mailbox.

A link may be associated with more than two processes.
Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.
A mailbox may be owned either by a process or by the operating system.

Synchronous or asynchronous communication

Communication between processes takes place through calls to send() and receive() primitives.
Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous
Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox
Nonblocking send. The sending process sends the message and resumes Operation
Blocking receive. The receiver blocks until a message is available.
Nonblocking receive. The receiver retrieves either a valid message or a null.
When both send() and receive() are blocking, we have a rendezvous between the sender and the receiver.
Automatic or explicit buffering:
Messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

- o Zero capacity.
- o Bounded capacity
- o unbounded capacity

Zero capacity. The queue has a maximum length of zero In this case, the sender must block until the recipient receives the message.

Bounded capacity. The queue has finite length n; thus, at most n messages can reside in it.

Unbounded capacity. The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

```
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
/* consume the item in next consumed */
}
```

Message-Passing Systems:

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space
A message-passing facility provides at least two operations:

Receive(message)

If processes P and want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them.
There are several methods for logically implementing a link and the send()/receive() operations:
Direct or indirect communication
Synchronous or asynchronous communication

Automatic or explicit buffering

THREADS-OVERVIEW:

A thread is a basic unit of CPU utilization. It is a smallest unit of execution of a program that determines the flow of control of execution.
It comprises a thread ID, a program counter, a register set, and a stack.
It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files.

Within a process, there may be one or more threads, each with the following:

A thread execution state (Running, Ready, etc.).

A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process.

An execution stack

Access to the memory and resources of its process

A process with a single flow of control is called as heavy weighted process.

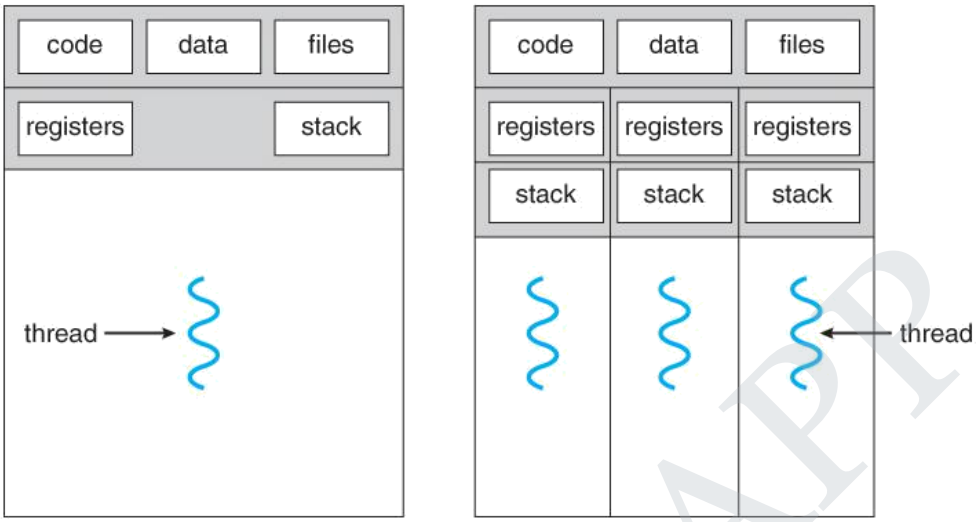


Figure 4.1 - Single-threaded and multithreaded processes

MULTITHREADING: Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process.

Example: Multithreaded server architecture:

A web server accepts client requests for web pages, images, sound, and so forth.

A Busy web server may have several clients concurrently accessing it.

If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

One solution is to have the server run as a single process that accepts requests.

When the server receives a request, it creates a separate process to service that request.Here Process creation is time consuming and resource intensive

Rather, if the web-server process is multithreaded, the server will create a separate thread that listens for client requests.

When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.

Threads also play a vital role in remote procedure call (RPC) systems.

ADVANTAGES OF MULTITHREADING:

Responsiveness: Multithreading an interactive application may allow a program to continue running even if part of it is blocked.

Resource sharing: threads share the memory and the resources of the process to which they belong by default.

Economy of scale: Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

Effective multiprocessor utilization: The benefits of multithreading can be even greater in a multiprocessor architecture

The process of placing multiple computing cores on a single chip is called as multicore programming.

Each core appears as a separate processor to the operating system, whether the cores appear across CPU chips or within CPU chips.

Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency

On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time because the processing core is capable of executing only one thread at a time.

On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core

A system is parallel if it can perform more than one task simultaneously.

A concurrent system supports more than one task by allowing all the tasks to make progress. Thus, it is possible to have concurrency without parallelism.

There are two types of parallelism

- o **Data Parallelism**
- o **Task parallelism.**

Data parallelism :

This focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

This focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

➢

Task parallelism:

This involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data**.**

Thus data parallelism involves the distribution of data across multiple cores and task parallelism on the distribution of tasks across multiple cores.

This involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

Thus data parallelism involves the distribution of data across multiple cores and task parallelism on the distribution of tasks across multiple cores

The challenges in the design of programming for multicore systems includes

Identifying tasks. This involves examining applications to find areas that can be divided into separate, concurrent tasks.

Balance. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.

Data splitting. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

Data dependency. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency

Testing and debugging. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult .

User level threads(ULTs)

Kernel-level threads (KLTs).

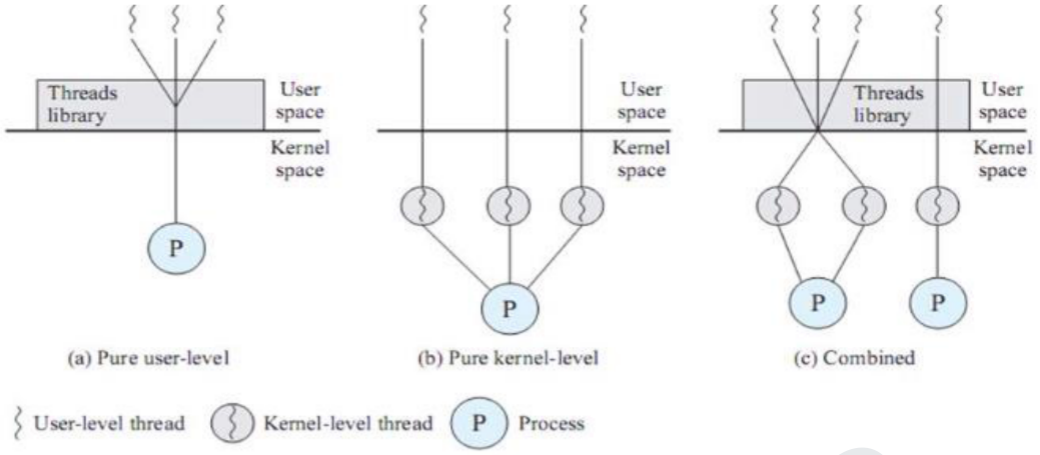


Figure 4.6 User-Level and Kernel-Level Threads

User level thread:

All of the thread management is done by the application and the kernel is not aware of the existence of threads.
The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.
At any time that the application is running the application may create a new thread to run within the same process.

Advantages of user level threads:

There are a number of advantages to the use of ULTs instead of KLTs
Thread switching does not require kernel mode privileges
Scheduling can be application specific
ULTs can run on any OS.

Disadvantages of user level threads:

When a ULT executes a blocking system call, not only is that thread blocked, but also all of the threads within the process are blocked.
A Multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time.

Kernel level threads:

All of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility.
The kernel maintains context information for the process as a whole and for individual threads within the process.

Advantages of Kernel level threads:

The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
If one thread in a process is blocked, the kernel can schedule another thread of the same process.

Disadvantages of kernel level threads:

The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

Combined approach:

Some operating systems provide a combined ULT/KLT facility
In a combined system, thread creation is done completely in user space
The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs.
In a combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process.
Relationship between user threads and kernel threads:

Many-to-one model,
One-to-one model,
Many-to many model.

MANY TO ONE MODEL:

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient.
The entire process will block if a thread makes a blocking system call.
Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. Example: Green threads in Solaris Operating system

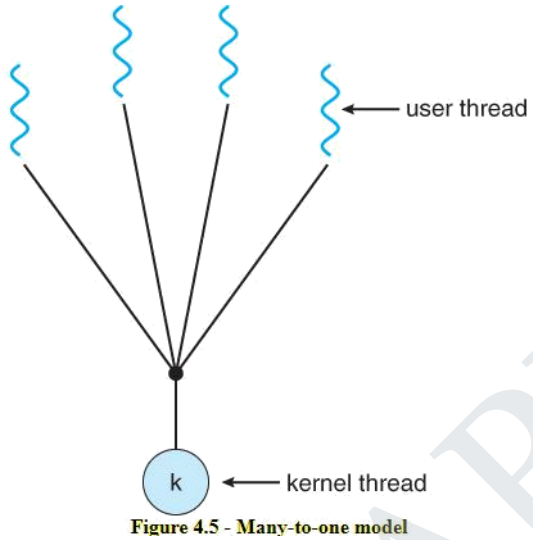


Figure 4.5 - Many-to-one model

ONE TO ONE MODEL:

The one-to-one model maps each user thread to a kernel thread.
It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
It also allows multiple threads to run in parallel on multiprocessors.
The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
Example: Linux and Windows operating system

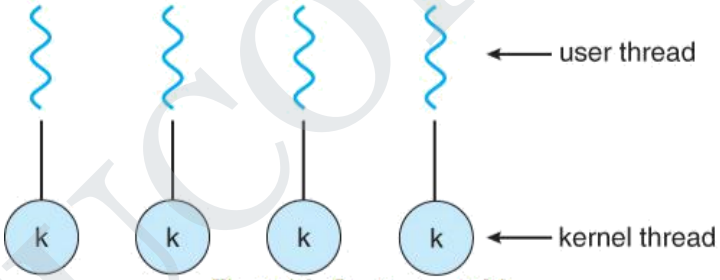


Figure 4.6 - One-to-one model

MANY TO MANY MODEL:

The many-to-many model (Figure 4.7) multiplexes many user-level threads to a smaller or equal number of kernel threads.
The number of kernel threads may be specific to either a particular application or a particular machine developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.

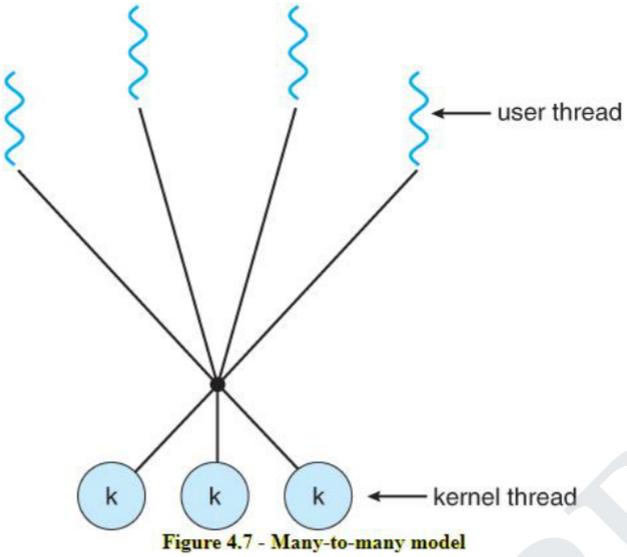


Figure 4.7 - Many-to-many model

TWO LEVEL MODELS:

Many-to-many model multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread.
This variation is referred to as the two-level model.

WINDOWS 7 AND SMP MANAGEMENT:

The important characteristics of Windows processes are,
i) Windows processes are implemented as objects
ii) An executable process may contain one or more threads.
iii) Both process and thread objects have built-in synchronization capabilities

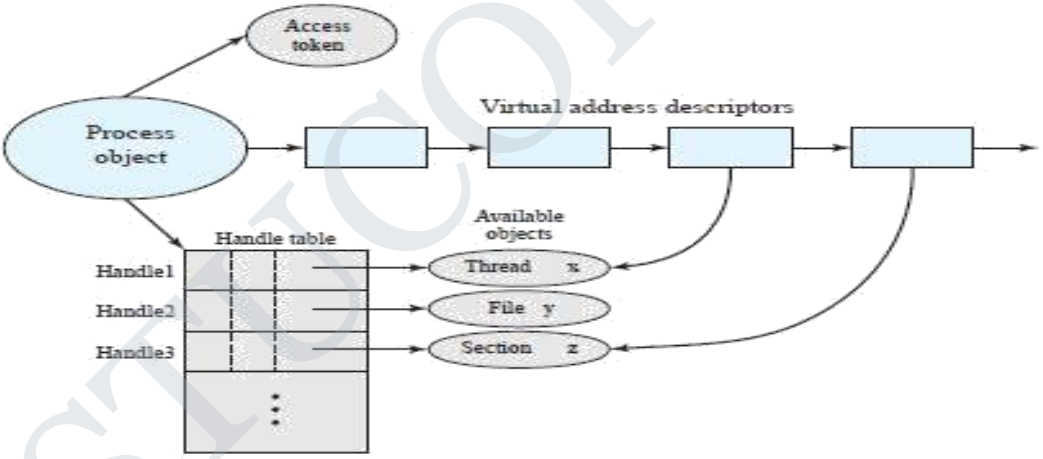


Figure 4.12 A Windows Process and Its Resources

Each process is assigned a security access token, called the primary token of the process When a user first logs on,
Windows creates an access token that includes the security ID for the user.
Every process that is created by or runs on behalf of this user has a copy of this access token.
Windows uses the token to validate the user's ability to access secured objects or to perform restricted functions on the system
Process contains a series of blocks that define the virtual address space currently assigned to this process.
The process includes an object table, with handles to other objects such as threads, files and data known to this process.

Process and Thread Objects:

Windows makes use of two types of process-related objects:

processes

Threads

A process is an entity corresponding to a user job or application that owns resources, such as memory, and opens files.

A thread is a dispatchable unit of work that executes sequentially and is interruptible, so that the processor can turn to another thread.

Each process is defined by a number of attributes and encapsulates a number of actions, or services, that it may perform.

Process ID	A unique value that identifies the process to the operating system.
Security Descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

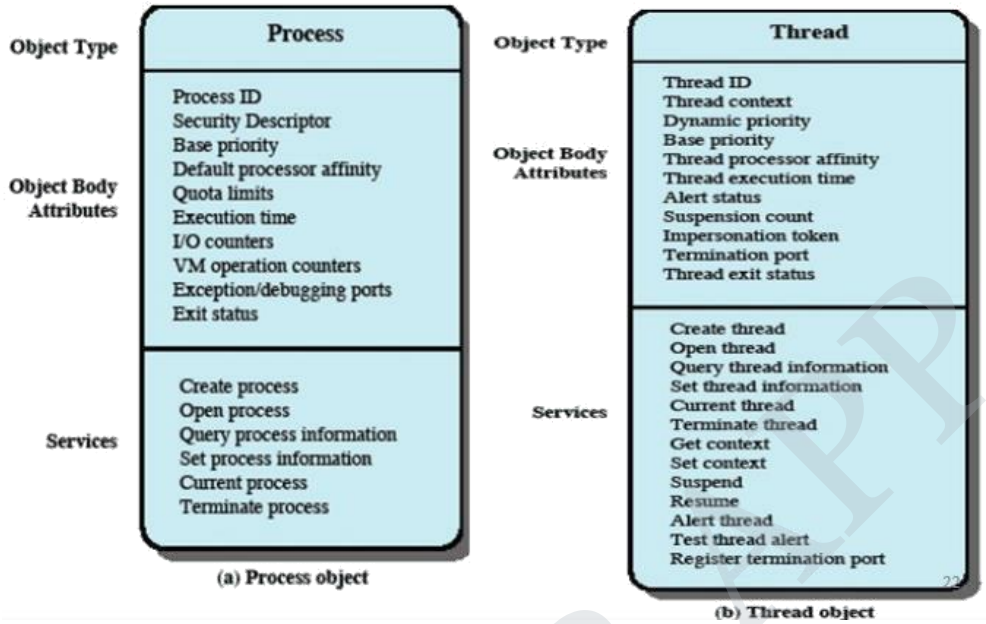
Each thread is defined by a number of attributes and encapsulates a number of actions, or services, that it may perform.

Process ID	A unique value that identifies the process to the operating system.
Security Descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

A Windows process must contain at least one thread to execute. That thread may then create other threads.

In a multiprocessor system, multiple threads from the same process may execute in parallel

An attribute called thread processor affinity is the set of processors in a multiprocessor system that may execute this thread; this set is equal to or a subset of the process processor affinity.



(a) Process object

(b) Thread object

Multithreading:

Windows supports concurrency among processes because threads in different processes may execute concurrently.

Multiple threads within the same process may be allocated to separate processors and execute simultaneously.

Threads within the same process can exchange information through their common address space and have access to the shared resources of the process.

Threads in different processes can exchange information through shared memory that has been set up between the two processes.

Thread States:

An existing Windows thread is in one of six states

Ready: May be scheduled for execution. The Kernel dispatcher keeps track of all ready threads and schedules them in priority order.

Standby: A standby thread has been selected to run next on a particular Processor. Once the Kernel dispatcher performs a thread switch, the standby thread enters the Running state and begins execution and continues execution until it is preempted by a higher priority thread

Waiting: A thread enters the Waiting state when (1) it is blocked on an event such as I/O operation or it voluntarily waits for synchronization purposes

Transition: A thread enters this state after waiting if it is ready to run but the resources are not available.

Terminated: A thread can be terminated by itself, by another thread, or when its parent process terminates.

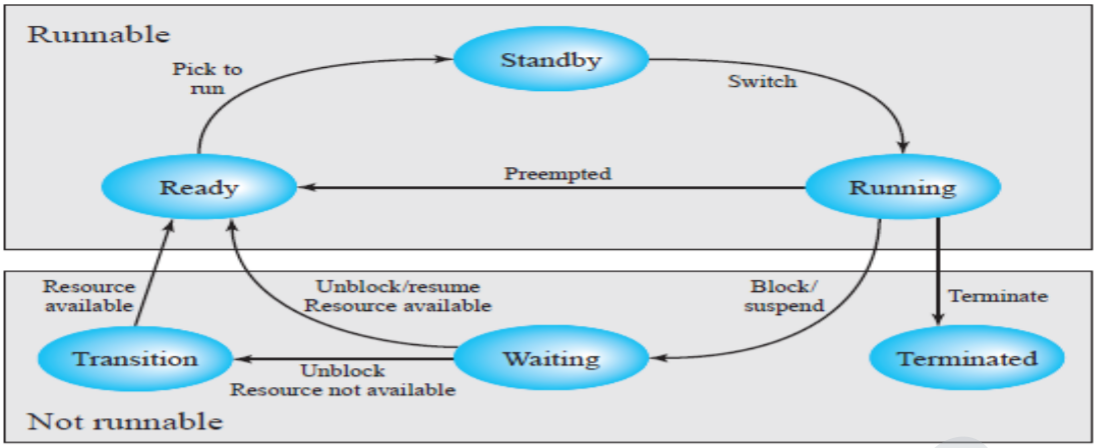


Figure 4.14 Windows Thread States

SYMMETRIC MULTIPROCESSOR SUPPORT:

The threads of any process can run on any processor.
In the absence of processor affinity the microkernel assigns a ready thread to the next available processor.
This assures that no processor is idle or is executing a lower-priority thread when a higher-priority thread is ready.
As a default, the microkernel uses the policy of soft affinity in assigning threads to processors.
The dispatcher tries to assign a ready thread to the same processor it last ran on. This helps reuse data still in that processor's memory caches from the previous execution of the thread. This is called as soft affinity.
It is possible for an application to restrict its thread execution to certain processors (hard affinity).

PROCESS SYNCHRONIZATION:

Process Synchronization is defined as the process of sharing system resources by cooperating processes in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data.

EXAMPLE:

Consider the producer consumer process that contains a variable called counter.
Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
The code for the producer process is

```
while (true) {
/* produce an item in next produced */
while (counter == BUFFER SIZE)
  /* do nothing */ 20 buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE;
counter++;
}
```

The code for the consumer process is **while**

```
(true) {
while (counter == 0)
  /* do nothing */
next consumed = buffer[out];
out = (out + 1) % BUFFER SIZE;
counter--;
/* consume the item in next consumed */
}
```

Suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements ―counter++‖ and ―counter--‖.
Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

When several processes access and manipulate the same data concurrently the outcome of the execution depends **on the particular order in which the access takes place, is called a race condition.**
To guard against the race condition we need to ensure that only one process at a time can be manipulating the variable counter.

CRITICAL SECTION PROBLEM:

The **critical-section problem** is to design a protocol that the processes can use to cooperate.
Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.
When one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

The structure of critical section problem is

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**.
The critical section may be followed by an **exit section**.
The remaining code is the **remainder section**.

A solution to the critical-section problem must satisfy the following three requirements:

1**. Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections

Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next.

Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

SOLUTIONS TO CRITICAL SECTION PROBLEM:

PETERSON'S SOLUTION:

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
The processes are numbered P0 and P1. For convenience, when presenting Pi, we use Pj to denote the other process.
Peterson's solution requires the two processes to share two data items:

int turn;
boolean flag[2];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process Pi is allowed to execute in its critical section.
The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that Pi is ready to enter its critical section.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
        remainder section
} while (true);
```

To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby checking that if the other process wishes to enter the critical section, it can do so.

Similarly to enter the critical section, process Pj first sets flag[j] to be true and then sets turn to the value i, thereby checking that if the other process wishes to enter the critical section.

The solution is correct and thus provides the following.

Mutual exclusion is preserved.

The progress requirement is satisfied.

The bounded-waiting requirement is met.

MUTEX LOCKS:

Mutex locks are used to protect critical regions and thus prevent race conditions

A process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (true);
```

The acquire() function acquires the lock, and the release() function releases the lock.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not.

If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.

A process that attempts to acquire an unavailable lock is blocked until the lock is released.

The definition of acquire() is as follows:

```
acquire()
{
while (!available)
  /* busy wait */
available = false;;
}
```

The definition of release() is as follows:

```
release()
{
available = true;
}
```

The main disadvantage of the implementation given here is that it requires **busy waiting**.
While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
This type of mutex lock is also called a **spinlock** because the process ―spins‖ while waiting for the lock to become available.
Busy waiting wastes CPU cycles that some other process might be able to use productively.
Spinlocks do have an advantage, however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time

SEMAPHORES:
A **semaphore** S is an integer variable that, is accessed only through two standard atomic operations: wait() and signal().
The wait() operation was originally termed P and the meaning is to test, the signal() was originally called V and the meaning is to increment.

The definition of wait() is as follows:

```
wait(S) {
while (S <= 0)
  // busy
wait S--;
}
```

The definition of signal() is as follows:

```
signal(S) {
S++;
}
```

Operating systems often distinguish between counting and binary semaphores.
The value of a **counting semaphore** can range over an unrestricted domain.
The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks.
Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
In this case the semaphore is initialized to the number of resources available.
Each process that wishes to use a resource performs a wait() operation on the
semaphore When a process releases a resource, it performs a signal() operation.
When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

SEMAPHORE IMPLEMENTATION:
The main disadvantage of semaphore is that it requires busy waiting. When one process is in its critical section any other process that tries to enter the critical section must loop continuously in the entry code.
The mutual exclusion implementation with semaphores is given
by **do { wait(mutex);**
//critical section signal(mutex);
//remainder section }while(TRUE);

To overcome the need for busy waiting, we can modify the wait() and signal() operations.
When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.
The block operation places a process into a waiting queue associated with the semaphore.
Then control is transferred to the CPU scheduler, which selects another process to execute.
A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
The process is restarted by a wakeup() operation, which changes the process fromthe waiting state to the ready state.
To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
int value;
struct process *list;
} semaphore
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes.
A signal() operation removes one process from the list of waiting processes and awakens that process.

The wait() semaphore operation can be defined as

```
wait(semaphore *S) {
S->value--;
if (S->value < 0) {
add this process to S->list;
block();
}
}
```

The signal() semaphore operation can be defined as

```
signal(semaphore *S) {
S->value++;
if (S->value <= 0) {
remove a process P from S->list;
wakeup(P);
}
}
```

The Block() and wakeup() operations are provided by the operating system as system calls.

DEADLOCKS AND STARVATION:

The implementation of a semaphore with waiting queue may result in a situation where two or more process are waiting indefinitely for an event that can be caused only by one of the waiting process**. This situation is called as deadlock**.
Example: Consider a system consisting of two process p0 and p1, each accessing two semaphores that is set to value 1.

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

P0 executes wait(S) and p1 executes Wait(Q).
When P0 executes wait(Q), it must wait until P1 executes signal(Q).
Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S).
Here the signal() operations cannot be executed, P0 and P1 are deadlocked.

PRIORITY INVERSION:

Assume we have three processes—L, M, and H—whose priorities follow the order L < M < H.
The process H requires resource R, which is currently being accessed by process L.
Process H would wait for L to finish using resource R. Suppose that process M becomes runnable, thereby preempting process L.
Indirectly, a process with a lower priority—process M—has affected process H that is waiting for L to release resource R.**This problem is known as priority inversion**
Priority-inheritance can solve the problem of priority inversion.
According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources that are requested.
When they are finished, their priorities revert to their original values.

CLASSIC PROBLEMS OF SYNCHRONIZATION:

1. Bounded Buffer problem:

In this problem, the producer process produces the data and the consumer processes consumes the data. Both of the process share the following data structures:

```
int n;
Semaphore mutex = 1;
Semaphore empty = n;
Semaphore full = 0
```

Assume that the pool consists of n buffers, each capable of holding one item.
The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers.
The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

Code for producer process:

```
do{
/* produce an item in next_produced */
……
wait(empty);
wait(mutex);
…..
/ * add next produced to the buffer
*/ signal(mutex);
signal(full); }
while(true);
```

Code for Consumer process:

```
do{
wait(full);
wait(mutex);
……
/* remove an item from buffer to next_consumed */
signal(mutex);
signal(empty);
……..
/* consume the item in next_consumed */
} while(true);
```

THE DINING-PHILOSOPHERS PROBLEM:

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
When a philosopher gets hungry she tries to pick up the two chopsticks that are closest to her.
A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.

When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks.
One simple solution is to represent each chopstick with a semaphore.
A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal () operation on the appropriate semaphores.
The shared data is semaphore chopstick [5]; where all the elements of the chopsticks are initialized to

```
1. do {
        wait(chopstick[i]);
        wait(chopstick[(i+1)%5]);
        ……
        /* eat for awhile */
        signal(chopstick[i]);
        signal(chopstick[(i+1)%5]);
        …..
        /* think for awhile */
 } while (true);
```

Several possible remedies to the deadlock problem are replaced by:

Allow at most four philosophers to be sitting simultaneously at the table.
Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
Use an asymmetric solution—that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick

THE READERS WRITERS PROBLEM:

A database is to be shared among several concurrent processes.
Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers**.
If two readers access the shared data simultaneously, no effects will result.

However, if a writer and some other process (either a reader or a writer) access the database simultaneously, problems may occur.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem**.

In the solution to the first readers–writers problem, the reader processes share the following data structures:

Semaphore rwmutex = 1;
Semaphore mutex = 1;
int read count = 0;

The semaphores mutex and rwmutex are initialized to 1; read count is initialized to 0. The semaphore rwmutex is common to both reader and writer

Code for writer process:

```
do {
wait(rw mutex);
. . .
/* writing is performed */
. . .
signal(rw mutex);
} while (true);
```

The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.
The read count variable keeps track of how many processes are currently reading the object.
The semaphore rwmutex functions as a mutual exclusion semaphore for the writers.

Code for readers process:

```
do {
wait(mutex);
read count++;
if (read count == 1)
wait(rw mutex);
signal(mutex);
. . .
/* reading is performed */
. . .
wait(mutex);
read count--;
if (read count == 0)
signal(rw mutex);
signal(mutex);
} while (true);
```

MONITORS:

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect.

EXAMPLE: Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution: signal(mutex);

```
...
critical section
...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.

Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
...
critical section
...
wait(mutex);
```

In this case, a deadlock will occur. To deal with such errors one fundamental high-level synchronization constructs called the **monitor** type is used.

A **monitor type** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.

The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables.

```
monitor monitor name
{/* shared variable declarations */
        function P1(. . . .){
        ... .
        }
        function P2(. . . .){
        ....
        }
                .
                .
        function Pn(. . . .){
        ....
        }
        initialization_code(. . . .) {
        ....
        }
}
```

Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

The monitor construct ensures that only one process at a time is active within the monitor.

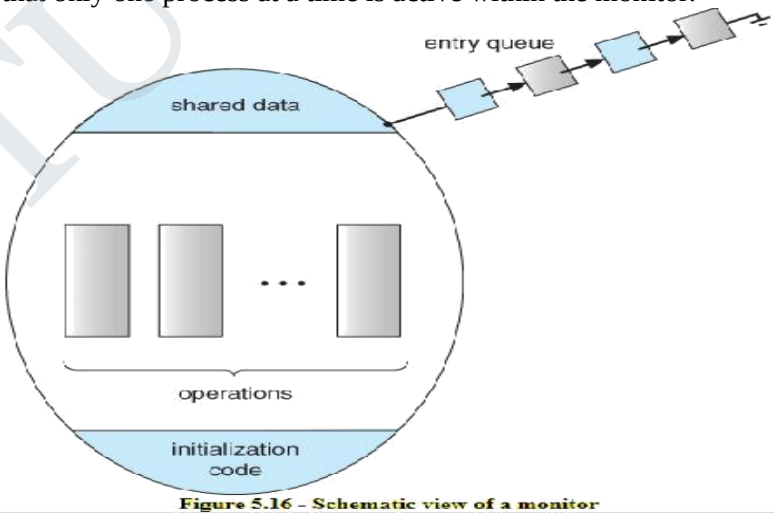


Figure 5.16 - Schematic view of a monitor

The monitors also provide mechanisms of synchronization by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition: Condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal().
The operation x.wait(); means that the process invoking this operation is suspended until another process invokes x.signal();
The x.signal() operation resumes exactly one suspended process
Now suppose that, when the x.signal() operation is invoked by a process P, there exists a suspended process associated with condition x.
Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor.
Note, however, that conceptually both processes can continuewith their execution. Two possibilities exist:

Signal and wait. P either waits until Q leaves the monitor or waits for another condition.
Signal and continue. Q either waits until P leaves the monitor or waits for another condition.

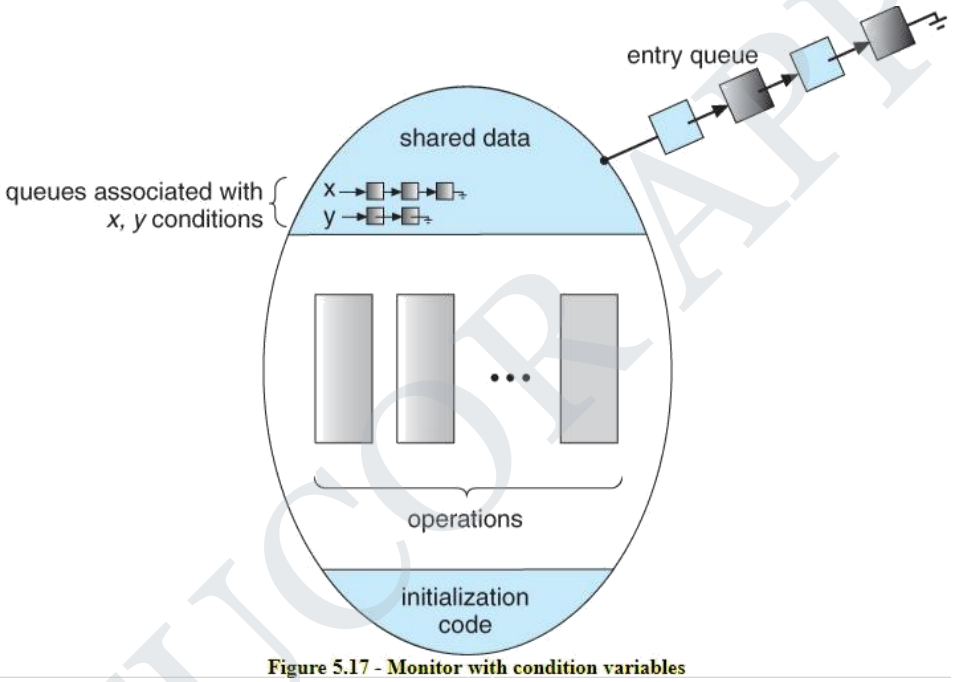


Figure 5.17 - Monitor with condition variables

Dining-Philosophers Solution Using Monitors:

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
When a philosopher gets hungry she tries to pick up the two chopsticks that are closest to her.
A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks.
The solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

```
monitor DiningPhilsophers
{
        enum {THINKING, HUNGRY, EATING} state[5];
        condition self[5];

        void pickup(int i) {
                state[i] = HUNGRY;
                test(i);
                if(state[i]!= EATING)
                        self[i].wait();
        }

        void putdown(int i){
                state[i] = THINKING;
                test((i+4)%5);
                test((i+1)%5);
        }

        void test(int i) {
                if((state[(i+4)%5] != EATING) &&
                  (state[i] == HUNGRY) &&
                  (state[(i+1)%5] != EATING)){
                        state[i] = EATING;
                        self[i].signal();
                  }
        }

        initialization_code() {
                for(int i=0; i < 5; i++)
                        state[i] = THINKING;
        }
}
```

To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

enum {THINKING, HUNGRY, EATING} state[5];

Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating:

(state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).

Also declare condition self[5]; that allows philosopher i to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Each philosopher, before starting to eat, must invoke the operation pickup().

This may result in the suspension of the philosopher process.

After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation.

Thus, philosopher i must invoke the operations pickup() and putdown() in the following sequence:

DiningPhilosophers.pickup(i);

eat

DiningPhilosophers.putdown(i);

This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

CPU SCHEDULING ALGORITHM:

CPU–I/O Burst Cycle:

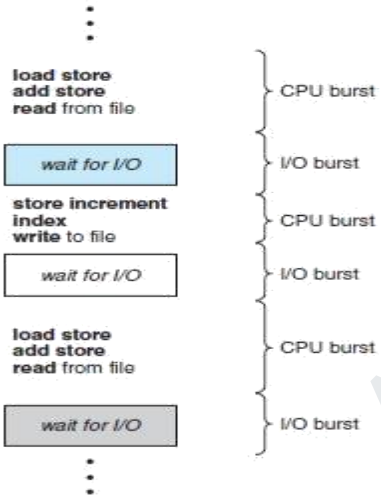
The process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states.

Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on.

Eventually, the final CPU burst ends with a system request to terminate execution.

CPU–I/O Burst Cycle:

The process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states.



Preemptive Scheduling and Non Preemptive scheduling:

Preemptive scheduling allows a process to be interrupted in the midst of its execution, taking the CPU away and allocating it to another process.

Nonpreemptive scheduling ensures that a process relinquishes control of the CPU only when it finishes with its current CPU burst.

CPU-scheduling decisions may take place under the following four circumstances:

When a process switches from the running state to the waiting state
When a process switches from the running state to the ready state
When a process switches from the waiting state to the ready state
When a process terminates

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative**. Otherwise, it is **preemptive**.

Scheduling Criteria:

The criteria include the following:

CPU utilization. The CPU Should be kept as busy as possible for effective CPU utilization.

Throughput: The total number of process completed per unit time is called as throughput.

Turnaround time: The interval from the time of submission of a process to the time of completion is the turnaround time.

Turnaround time = Waiting time + Burst time

Waiting time: Waiting time is the sum of the periods spent waiting in the ready queue.

Response time: The time from the submission of a request until the first response is produced.

Scheduling Algorithms:

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms.

First-Come, First-Served Scheduling

Shortest-Job-First Scheduling
Priority Scheduling
Round-Robin Scheduling

FIRST COME FIRST SERVE SCHEDULING:

With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue.

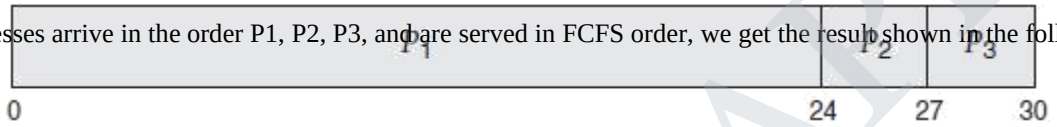
When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue.

The average waiting time under the FCFS policy is often quite long.

EXAMPLE: Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

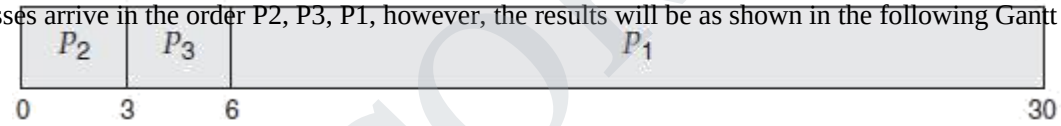
If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart**.



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3.

Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds.

If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. Thus, the average waiting time under an FCFS policy is generally not minimal.

Assume that we have one CPU-bound process and many I/O-bound processes. The CPU-bound process will get and hold the CPU.

During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU.

While the processes wait in the ready queue, the I/O devices are idle.

Now the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. Now the CPU sits idle. This is called as CONVOY EFFECT which results in lower CPU and device utilization.

The FCFS scheduling algorithm is non preemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

ADVANTAGES:

Better for long processes
Simple method (i.e., minimum overhead on processor)
No starvation

DISADVANTAGES:

Waiting time can be large if short requests wait behind the long ones.

It is not suitable for time sharing systems where it is important that each user should get the CPU for an equal amount of time interval.

A proper mix of jobs is needed to achieve good results from FCFS scheduling.

SHORTEST-JOB-FIRST SCHEDULING:

With this algorithm the process that comes with the shortest job will be allocated the CPU first.

This algorithm includes with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

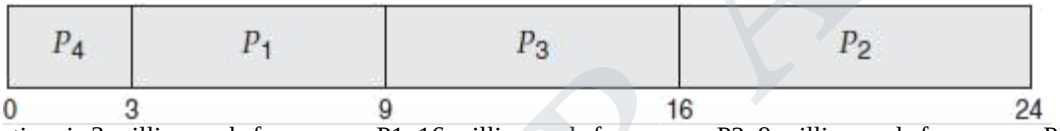
If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

This is also called as shortest next CPU burst algorithm.

EXAMPLE: consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4.

Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.

The SJF algorithm can be either preemptive or nonpreemptive.

When a new process arrives at the ready queue while a previous process is still executing and the next CPU burst of the newly arrived process is shorter than what is left of the currently executing process, then a preemptive or non preemptive approach can be chosen.

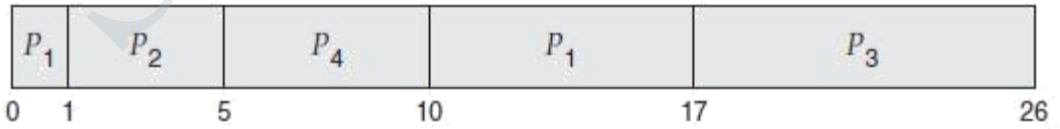
A preemptive SJF algorithm will preempt the currently executing process.

A nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

EXAMPLE: consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

The preemptive SJF schedule is as depicted in the following Gantt chart:



Process P1 is started at time 0, since it is the only process in queue. Process P2 arrives at time 1.

The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

The average waiting time is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5 milliseconds.
Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds

ADVNATAGES:

The SJF scheduling algorithm has minimum average waiting time for a given set of processes.
Gives superior turnaround time performance to shortest process next because a short job is given immediate preference to a running longer job.
Throughput is high.

DISADVANTAGES:

The difficulty with the SJF algorithm is knowing the length of the next CPU request
Starvation may be possible for the longer processes.

PRIORITY SCHEDULING:

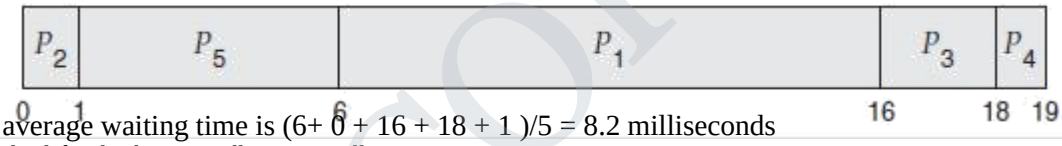
Apriority is associated with each process, and the CPU is allocated to the process with the highest priority.
Equal-priority processes are scheduled in FCFS order.

An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:.



Thus, the average waiting time is (6+ 0 + 16 + 18 + 1)/5 = 8.2 milliseconds
Priorities can be defined either internally or externally
Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. Hence the higher-priority processes can prevent a low-priority process from ever getting the CPU. This problem with priority scheduling algorithms is **indefinite blocking**, or **starvation**.
A solution to the problem of indefinite blockage of low-priority processes is **aging**.
Aging involves gradually increasing the priority of processes that wait in the system for a long time.

ADVANTAGES:

1.Simplicity.
Reasonable support for priority.
Suitable for applications with varying time and resource requirements.

DISADVANTAGES

Indefinite blocking or starvation.

A priority scheduling can leave some low priority waiting processes indefinitely for CPU.

If the system eventually crashes then all unfinished low priority processes gets lost.

ROUND ROBIN SCHEDULING:

The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems.

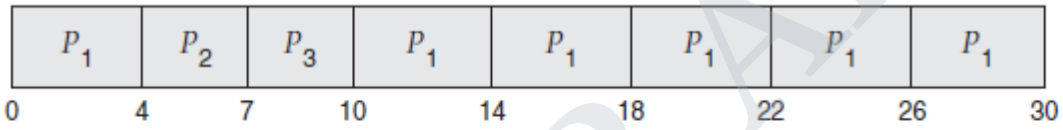
A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length.

The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.

EXAMPLE: Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds

Process	Burst Time
P_1	24
P_2	3
P_3	3



If we use a time quantum of 4 milliseconds, then process *P*1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P*2

Process *P*2 does not need 4 milliseconds, so it quits before its time quantum expires.

The CPU is then given to the next process, process *P*3. Once each process has received 1 time quantum, the CPU is returned to process *P*1 for an additional time quantum.

*P*1 waits for 6 milliseconds (10 - 4), *P*2 waits for 4 milliseconds, and *P*3 waits for 7 milliseconds.

Thus, the average waiting time is 17/3 = 5.66 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum.

If the time quantum is extremely large, the RR policy is the same as the FCFS policy.

If the time quantum is extremely small the RR approach can result in a large number of context switches.

ADVANTAGES:

Does not suffer by starvation.

There is Low throughput.

There are Context Switches.

MULTI LEVEL QUEUE SCHEDULING:

The processes are divided into foreground process and background process.

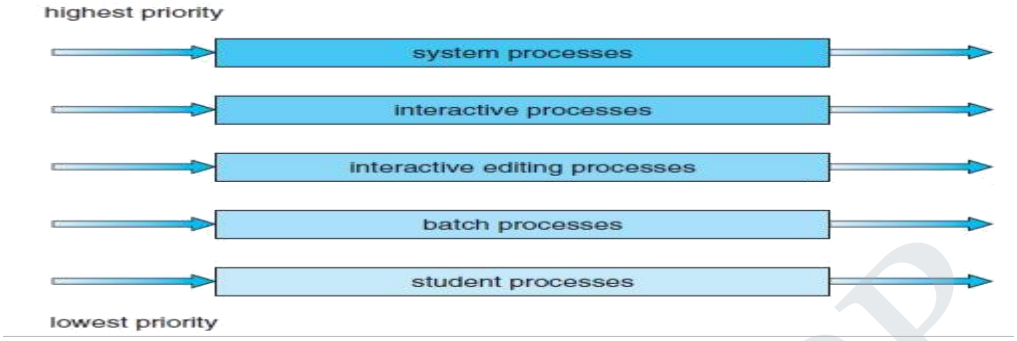
These two types of processes have different response-time requirements and have different scheduling needs.

The foreground processes may have priority over background processes.

A **multilevel queue** scheduling algorithm partitions the ready queue into several separate queues.

The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.

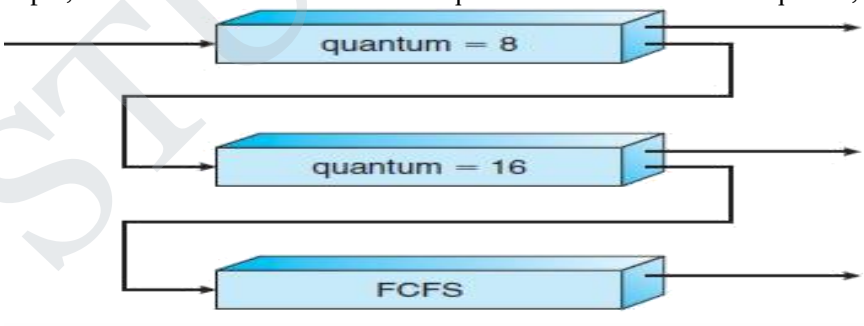
Example of a multilevel queue scheduling algorithm with five queues, in order of priority:
System processes
Interactive processes
Interactive editing processes
Batch processes
Student processes



Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes.
The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.
There must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.
No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.
If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

MULTILEVEL FEEDBACK QUEUE SCHEDULING:

The **multilevel feedback queue** scheduling algorithm, allows a process to move between queues.
The idea is to separate processes according to the characteristics of their CPU bursts.
If a process uses too much CPU time, it will be moved to a lower-priority queue.
A process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.
EXAMPLE: For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2



The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.
A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.
A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds.
If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty. a multilevel feedback queue

DEADLOCKS:

A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

The resources of a computer system may be partitioned into several types such as CPU cycles, files, and I/O devices (such as printers and DVD drives)

A process must request a resource before using it and must release the resource after using it.

A process may utilize a resource in only the following sequence.

Request. The process requests the resource. If the request cannot be granted immediately then the requesting process must wait until it can get the resource.

Use. The process can operate on the resource

Release. The process releases the resource.

NECESSARY CONDITIONS FOR DEADLOCKS:

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

Mutual exclusion. At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource.

Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

No preemption. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait. A set *{P0, P1, ..., Pn}* of waiting processes must exist such that *P0* is waiting for a resource held by *P1*, *P1* is waiting for a resource held by *P2*, ..., *Pn*−1 is waiting for a resource held by *Pn*, and *Pn* is waiting for a resource held by *P0*.

RESOURCE ALLOCATION GRAPH:

Deadlocks can be described more clearly in terms of a directed graph called a **system resource-allocation graph**.

This graph consists of a set of vertices *V* and a set of edges *E*.

The set of vertices *V* is partitioned into two different types of nodes *P = {P1, P2, ..., Pn}*, the set consisting of all the active processes in the system, and *R = {R1, R2, ..., Rm}*, the set consisting of all resource types in the system.

A directed edge *Pi → Rj* is called a **request edge**; a directed edge *Rj → Pi* is called an **assignment edge**

Represent each process *Pi* as a circle and each resource type *Rj* as a rectangle.

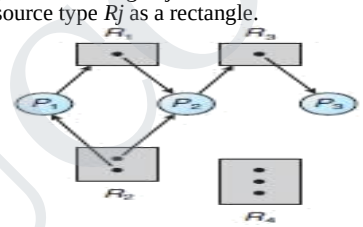


Figure 7.1 Resource-allocation graph.

The sets *P*, *R*, and *E*:

P = {P1, P2, P3}

R = {R1, R2, R3, R4}

E={P1→R1,P2→R3,R1→P2,R2→P2,R2→P1,R3→P3}

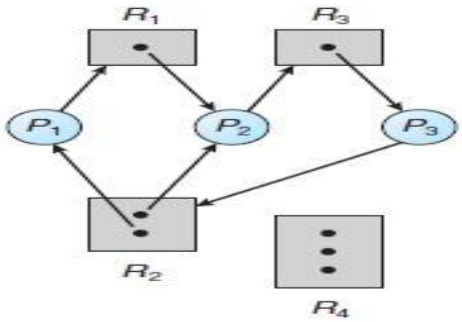
Process states:

Process *P1* is holding an instance of resource type *R2* and is waiting for an instance of resource type *R1*.

Process *P2* is holding an instance of *R1* and an instance of *R2* and is waiting for an instance of *R3*.

Process *P3* is holding an instance of *R3*.

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist



Consider a process *P*3 requests an instance of resource type *R*2. Since no resource instance is currently available, we add a request edge *P*3→ *R*2 to the graph.

Two minimal cycles exist in the system:

***P*1→*R*1→*P*2→*R*3→*P*3→*R*2→*P*1**

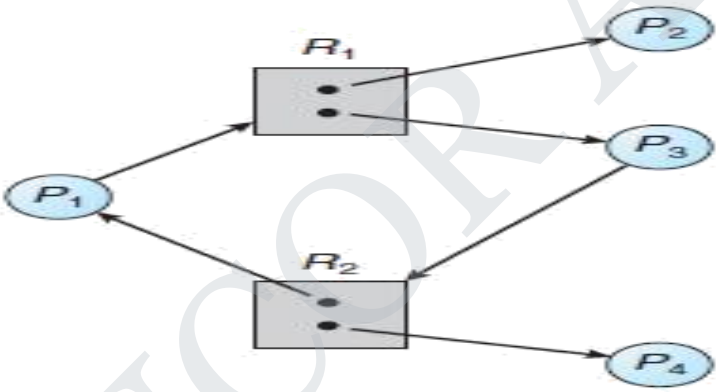
***P*2→*R*3→*P*3→*R*2→*P*2**

Processes *P*1, *P*2, and *P*3 are deadlocked.

Process *P*2 is waiting for the resource *R*3, which is held by process *P*3. Process *P*3 is waiting for either process *P*1 or process *P*2 to release resource *R*2. In addition, process *P*1 is waiting for process *P*2 to release resource *R*1.

If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

In the following diagram there is a cycle *P*1 → *R*1 → *P*3 → *R*2 → *P*1.But there is no deadlock.



Resource allocation graph with a cycle but no deadlock.

*P*4 may release its instance of resource type *R*2. That resource can then be allocated to *P*3, breaking the cycle.

If a resource-allocation graph does not have a cycle, then the system is ***not*** in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.

METHODS FOR HANDLING DEADLOCKS:

The method for handling deadlocks include

Deadlock avoidance or Deadlock Prevention.

Deadlock Detection

Deadlock Recovery.

DEADLOCK PREVENTION:

Deadlock prevention provides a set of methods to ensure that at least one of the four necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.

MUTUAL EXCLUSION:

The mutual exclusion condition must hold. That is, at least one resource must be non-sharable

Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.

Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

We cannot prevent deadlocks by denying the mutual-exclusion condition for the non-sharable resources .

HOLD AND WAIT:

Whenever a process requests a resource, it does not hold any other resources.

This allows a process to request resources only when it has none. A process may request some resources and use them. Before it can request any additional resources, it must release all the resources that it is currently allocated.

NO PREEMPTION:

If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted.

The preempted resources are added to the list of resources for which the process is waiting.

The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

CIRCULAR WAIT:

One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

Let $R = \{R1, R2, ..., Rm\}$ be the set of resource types.

Assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.

If the set of resource types *R* includes tape drives, disk drives, and printers, then the function *F* might be defined as follows:

***F*(tape drive) = 1**

***F*(disk drive) = 5**

***F*(printer) = 12**

Each process can request resources only in an increasing order of enumeration.

That is, a process can initially request any number of instances of a resource type —say, *Ri* .

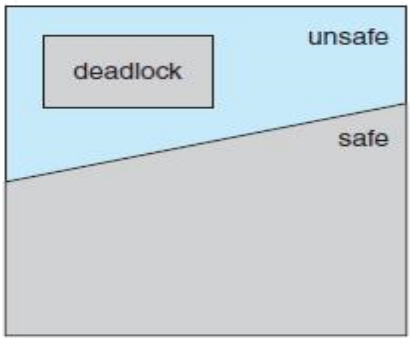
After that, the process can request instances of resource type *Rj* if and only if $F(Rj) > F(Ri)$.

Example, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

DEADLOCK AVOIDANCE:

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.

SAFE STATE:



A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock.

A system is in a safe state only if there exists a **safe sequence**.

A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state.

Not all unsafe states are deadlocks; however an unsafe state ***may*** lead to a deadlock.

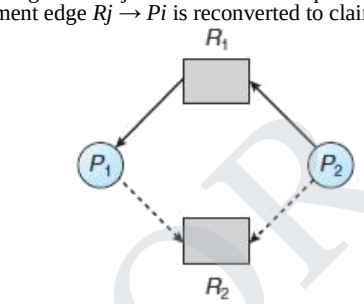
A sequence of processes <*P*1, *P*2, ..., *Pn*> is a safe sequence for the current allocation state if,
the resource requests that *Pi* make can be satisfied by the currently available resources plus the resources held by all *Pj*, with *j* < *i*.
Example: consider a system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2
Process *P*0 requires ten tape drives, process *P*1 may need as many as four tape drives, and process *P*2 may need up to nine tape drives.
Suppose that, at time *t*0, process *P*0 is holding five tape drives, process *P*1 is holding two tape drives, and process *P*2 is holding two tape drives .

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

At time *t*0, the system is in a safe state. The sequence <*P*1, *P*0, *P*2> satisfies the safety condition.
Process *P*1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives);
Then process *P*0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process *P*2 can get all its tape drives and return them (the system will then have all twelve tape drives available).

RESOURCE ALLOCATION GRAPH:

A new type of edge, called a **claim edge** is used in resource allocation graph.
A claim edge *Pi* → *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future.
When process *Pi* requests resource *Rj* , the claim edge *Pi* → *Rj* is converted to a request edge.
When a resource *Rj* is released by *Pi*, the assignment edge *Rj* → *Pi* is reconverted to claim edge Pi ->Rj



Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi* → *Rj* to an assignment edge *Rj* → *Pi* does not result in the formation of a cycle in the resource-allocation graph.
If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

BANKERS ALGORITHM:

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
This number may not exceed the total number of resources in the system.
When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
The following data structures are needed to implement bankers algorithm, where *n* is the number of processes in the system and *m* is the number of resource types:

Available: Length *m* indicates the number of available resources of each type.
Max. An *n* × *m* matrix defines the maximum demand of each process.
Allocation. An *n* × *m* matrix defines the number of resources of each type currently allocated.
Need. An *n* × *m* matrix indicates the remaining resource need of each process.

SAFETY ALGORITHM:

It is an algorithm for finding out whether or not a system is in a safe state

Let***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize ***Work*** = ***Available*** and ***Finish***[*i*] = ***false*** for *i* = 0, 1, ..., *n* − 1.

Find an index *i* such that both a. ***Finish***[*i*] == ***false***
Need*i* ≤***Work***

Work =***Work*** + ***Allocation****i*
Finish[*i*] = ***true***
Go to step 2.

If ***Finish***[*i*] == ***true*** for all *i*, then the system is in a safe state.
This algorithm may require an order of *m* × *n*2 operations to determine whether a state is safe.

RESOURCE REQUEST ALGORITHM: Let
Request*i* be the request vector for process *Pi.*

If ***Request****i* ≤***Need****i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.

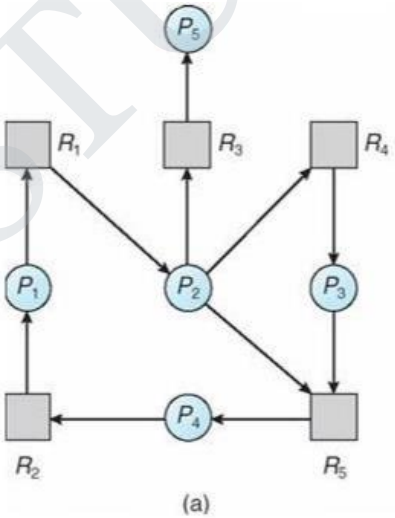
Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows: ***Available*** = ***Available***–***Request****i* ;
Allocation*i* = ***Allocation****i* + ***Request****i*
; ***Need****i* = ***Need****i* –***Request****i* ;
If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources. However, if the new state is unsafe, then *Pi* must wait for ***Request****i* **,** and the old resource-allocation state is restored.

DEADLOCK DETECTION:

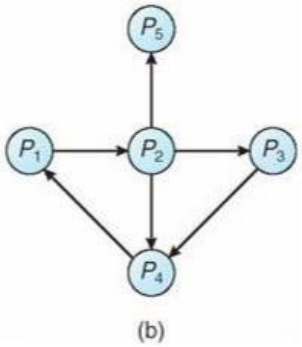
A Deadlock detection algorithm examines the state of the system to determine whether a deadlock has occurred

SINGLE INSTANCE OF EACH RESOURCE TYPE:

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a **wait-for** graph.



Resource-Allocation Graph



Corresponding wait-for graph

This graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
A deadlock exists in the system if and only if the wait-for graph contains a cycle.
To detect deadlocks, the system needs to ***maintain*** the wait for graph and periodically ***invoke an algorithm*** that searches for a cycle in the graph.

SEVERAL INSTANCE OF RESOURCE TYPE:

The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type
This algorithm uses the following data structure,

Available. A vector of length *m* indicates the number of available resources of each type.

Allocation. An *n* × *m* matrix defines the number of resources currently allocated to each process.

Request. An *n* × *m* matrix indicates the current request of each process.

Let***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize ***Work*** = ***Available.*** For *i* = 0, 1, ..., *n–1*, if ***Allocation****i* _= 0, then ***Finish***[*i*] = ***false.*** Otherwise, ***Finish***[*i*] = ***true.***

Find an index *i* such that both
Finish[*i*] == ***false***
Request*i* ≤***Work***

Work =***Work*** + ***Allocation****i*
Finish[*i*] = ***true***
Go to step 2.

If ***Finish***[*i*] ==***false*** for some *i*, 0≤*i*<*n*, then the system is in a deadlocked state.
Moreover, if ***Finish***[*i*] == ***false,*** then process *Pi* is deadlocked.
This algorithm requires an order of *m* × *n*2 operations to detect whether the system is in a deadlocked state.

DEADLOCK RECOVERY:

When a detection algorithm determines that a deadlock exists ,the possibility is to let the system **recover** from the deadlock automatically
There are two options for breaking a deadlock.

1. PROCESS TERMINATION:

It is the process of eliminating the deadlock by aborting a process. It involves two methods

Abort all deadlocked process: This method breaks the deadlock cycle but with a greater expense.

Abort one process at a time until the deadlock cycle is eliminated: This method aborts the deadlocked process one by one and after each process is aborted, it checks whether any process are still deadlocked.

2. RESOURCE PREEMPTION:

To eliminate deadlocks Preempt some resources from the process and give the resource to other process until the deadlock cycle is broken.

Selecting an victim: It is the process of selecting which resource and which process are to be preempted.

Rollback: If we preempt a resource from a process, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

Starvation: The resources will not always be preempted from the same process, else it leads to starvation.

UNIT - III STORAGE MANAGEMENT

Main Memory – Background, Swapping, Contiguous Memory Allocation, Paging, Segmentation, Segmentation with paging, 32 and 64 bit architecture Examples; Virtual Memory – Background, Demand Paging, Page Replacement, Allocation, Thrashing; Allocating Kernel Memory, OS Examples.

MAIN MEMORY:

MEMORY HARDWARE:

Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly.

Registers that are built into the CPU are generally accessible within one cycle of the CPU clock.

A memory access may take many cycles of the CPU clock. In such cases, the processor normally needs to **stall**, since it does not have the data required to complete the instruction.

The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for fast access called as CACHE.

MEMORY PROTECTION:

For proper system operation we must protect the operating system from access by user processes.

Each process has a separate memory space. Separate per-process memory space protects the processes from each other.

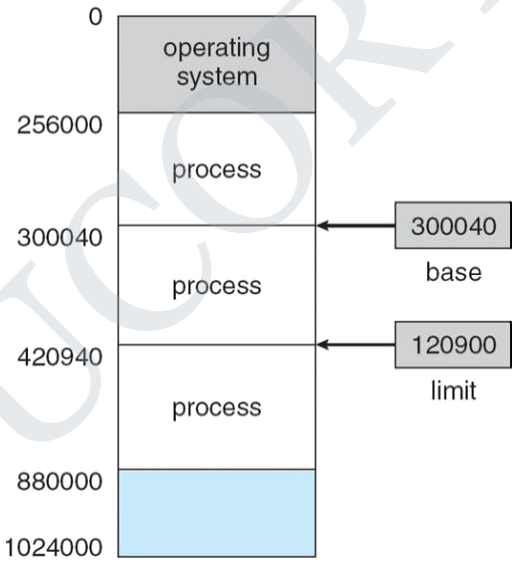
The hardware protection of memory is provided by two registers

- Base Register
- Limit Register

The base register holds the smallest legal physical memory address, called the starting address of the process.

The Limit register specifies the size of range of the process.

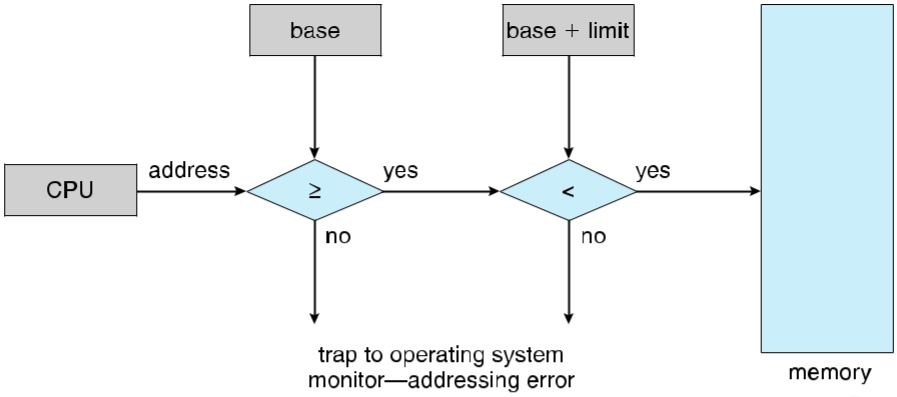
If the base register holds300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939



Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.

Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, resulting in addressing error.

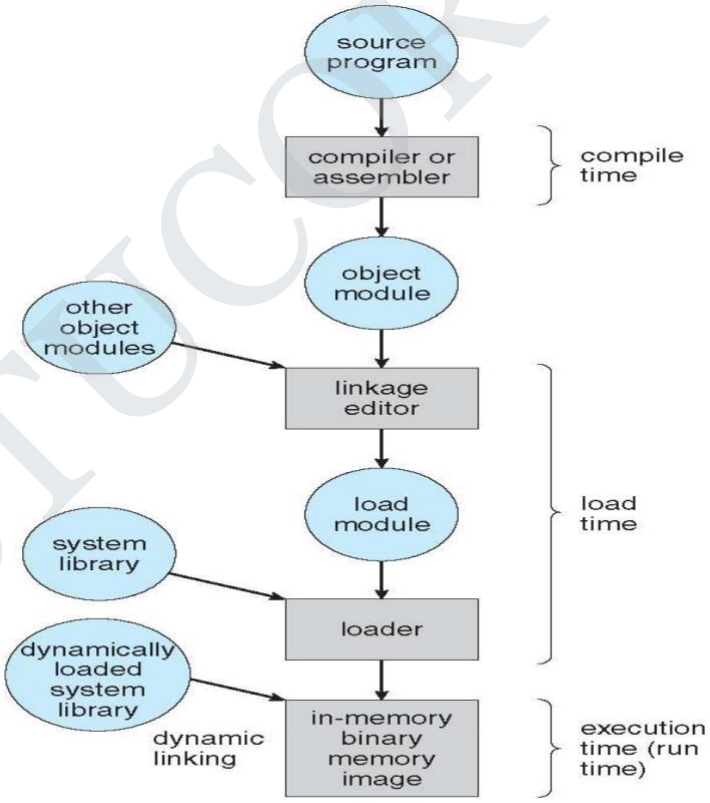
The base and limit registers can be loaded only by the operating system into the CPU hardware.



This scheme prevents a user program from modifying the code or data structures of either the operating system or other users.
The address generated by the CPU for a process should lie between the Base address of the process and base + Limit of the process, Else the hardware sends an interrupt to the OS.

ADDRESS BINDING:

Address binding is the process of mapping the program's logical or virtual addresses to corresponding physical or main memory addresses.
Addresses in the source program are generally symbolic.
A compiler typically **binds** these symbolic addresses to relocatable addresses.
The linkage editor or loader in turn binds the relocatable addresses to absolute addresses
Each binding is a mapping from one address space to another .



The binding of instructions and data to memory addresses can be done in three ways.

Compile time. If you know at compile time where the process will reside in memory, then **absolute code** can be generated.

Load time. If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code**.

Execution time. If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

LOGICAL VERSUS PHYSICAL ADDRESS SPACE:

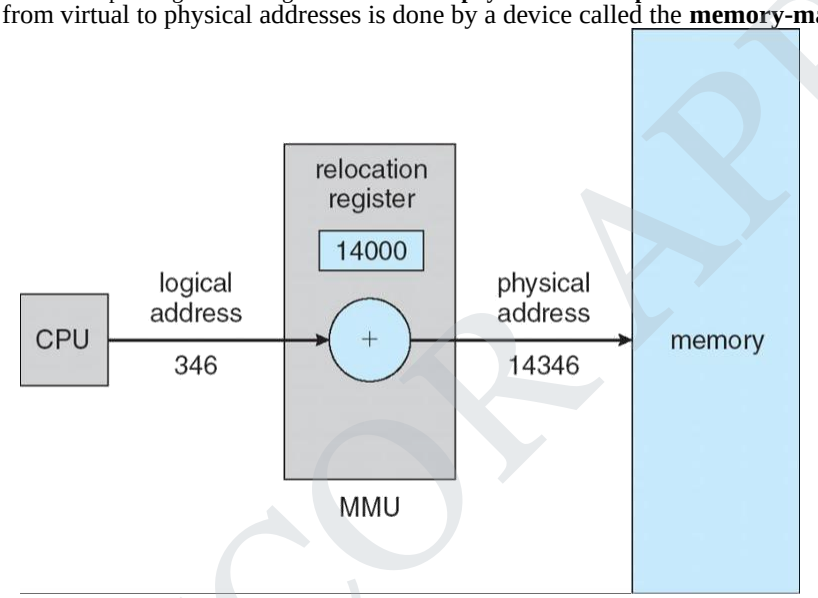
An address generated by the CPU is commonly referred to as a **logical address**. which is also called as virtual address

The set of all logical addresses generated by a program is a **logical address space**.

An address seen by the memory unit—that is, the one loaded into the **memory-address register** of the memory—is commonly referred to as a **physical address**.

The set of all physical addresses corresponding to these logical addresses is a **physical address space**.

The run-time mapping from virtual to physical addresses is done by a device called the **memory-management unit (MMU)**.



The base register is also called as **relocation register**.

The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory.

For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346

DYNAMIC LOADING:

Dynamic Loading is the process of loading a routine only when it is called or needed during runtime.

Initially all routines are kept on disk in a relocatable load format.

The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If it has not, the relocatable linking loader is called to load the desired routine into memory.

The advantage of dynamic loading is that a routine is loaded only when it is needed.

This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

DYNAMIC LINKING AND SHAREDLIBRARIES:

Dynamically linked libraries are system libraries that are linked to user programs when the programs are in execution.

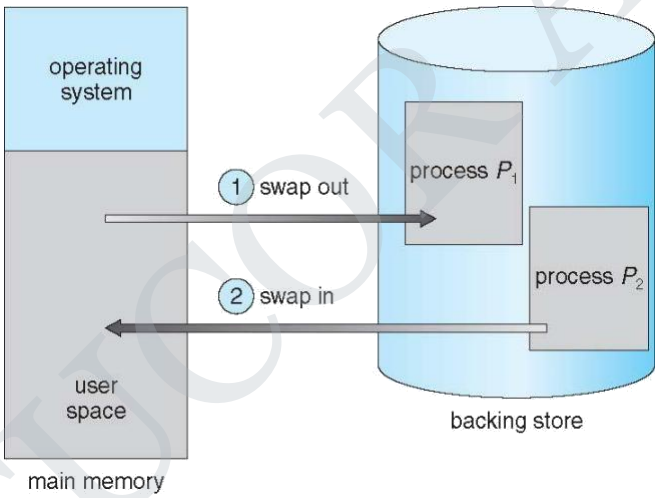
In Dynamic linking the linking of system libraries are postponed until execution time.
Static Linking combines the system libraries to the user program at the time of compilation**.**
Dynamic linking saves both the disk space and the main memory space.
The libraries can be replaced by a new version and all the programs that reference the library will use the new version. **This system is called as Shared libraries** which can be done with dynamic linking.

SWAPPING:

A process must be in memory to be executed.
A process can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. This process is called as Swapping.
Swapping allows the total physical address space of all processes to exceed the real physical memory of the system, and increases the degree of multiprogramming in a system.
Swapping involves moving processes between main memory and a backing store. The backing store is commonly a fast disk.

EXAMPLE: Consider a multiprogramming environment with Round robin Scheduling**.** When the quantum time of a process expires the memory manager will swap out the process just finished and swap another process into the memory space.

The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory and are ready to run.
Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory.
If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.



The context-switch time in such a swapping system is fairly high.
The major part of the swap time is transfer time. The total transfer time is directly proportional to the amount of memory swapped.
If we want to swap a process, we must be sure that it is completely idle. A process that is waiting for any event such as I/O Operation to occur should not be swapped.
A variant of swapping policy is used for priority based scheduling algorithms. If a higher priority process arrives and want service the memory manager can then swap the lower priority process and then load and execute the higher priority process.
When the higher priority process finishes then the lower priority process can be swapped back in. This is also called as **Roll in and Roll out.**

CONTIGUOUS MEMORY ALLOCATION:

The main memory must accommodate both the operating system and the various user processes

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.

In Multiprogramming several user processes to reside in memory at the same time.

The OS need to decide how to allocate available memory to the processes that are in the input queue waiting to be brought into memory.

In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

MEMORY PROTECTION:

We can prevent a process from accessing memory of other process.

If we have a system with a relocation register together with a limit register we accomplish our goal.

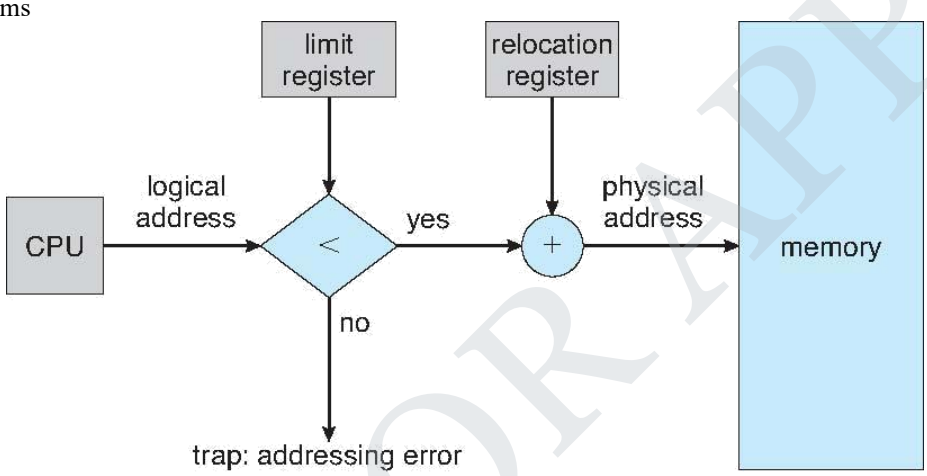
The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses

The MMU maps the logical address dynamically by adding the value in the relocation register.

This mapped address is sent to memory.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch.

Every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs



MEMORY ALLOCATION:

In Contiguous memory allocation the memory can be allocated in two ways

Fixed partition scheme

Variable partition scheme

Fixed partition scheme:

One of the simplest methods for allocating memory is to divide memory into several fixed-sized Partitions. Each partition may contain exactly one process.

Thus, the degree of multiprogramming is bound by the number of partitions.

In this **multiple partition method**, when a partition is free, a process is selected from the input queue and is loaded into the free partition.

When the process terminates, the partition becomes available for another process.

INTERNAL FRAGMENTATION: In fixed size partitions, each process is allocated with a partition, irrespective of its size. **The allocated memory for a process may be slightly larger than requested memory; this memory that is wasted internal to a partition, is called as internal fragmentation.**

Variable Partition scheme:

In the **variable-partition** scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied.

Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**.

When a process is allocated space, it is loaded into memory, and it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then fill with another process from the input queue.

Os will have a list of available block sizes and an input queue. The operating system can order the input queue according to a scheduling algorithm.

Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied—that is, no available block of memory (or hole) is large enough to hold that process.

When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.

If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.

When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

The system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

This procedure is a particular instance of the general **dynamic storage allocation problem**

First fit: Allocate the first hole that is big enough.

Best fit: Allocate the smallest hole that is big enough.

Worst fit: Allocate the largest hole.

Both the first-fit and best-fit strategies suffer from **external fragmentation.**

EXTERNAL FRAGMENTATION: As processes are loaded and removed from memory, the free memory space is broken into little pieces. **External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous**, **so that the memory cannot be allocated to the process.**

COMPACTION: One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block.

50 PERCENT RULE: The analysis of first fit, reveals that given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation. That is, one-third of memory may be unusable! This property is known as the **50-percent rule**.

NON CONTIGUOUS MEMORY ALLOCATION:

The solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever memory is available. Two complementary techniques achieve this solution:

segmentation
paging

SEGMENTATION:

Segmentation is a memory-management scheme that supports the programmer view of memory.

A logical address space is a collection of segments.

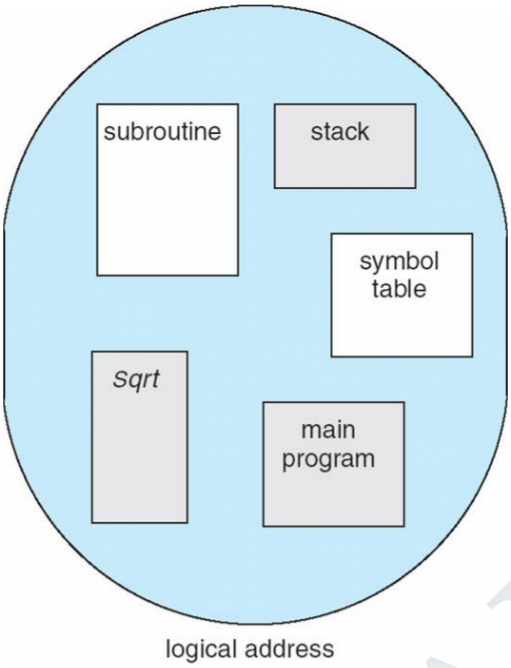
Each segment has a name and a length.

The logical addresses specify both the segment name and the offset within the segment. Each address is specified by two quantities: a segment name and an offset

Segment-number, offset>.

A C compiler might create separate segments for the following:

The code
Global variables
The heap, from which memory is allocated
The stacks used by each thread
The standard C library



SEGMENTATION HARDWARE:

The programmer can refer to objects in the program by a two-dimensional address (segment number and offset); the actual physical memory a one dimensional sequence of bytes.

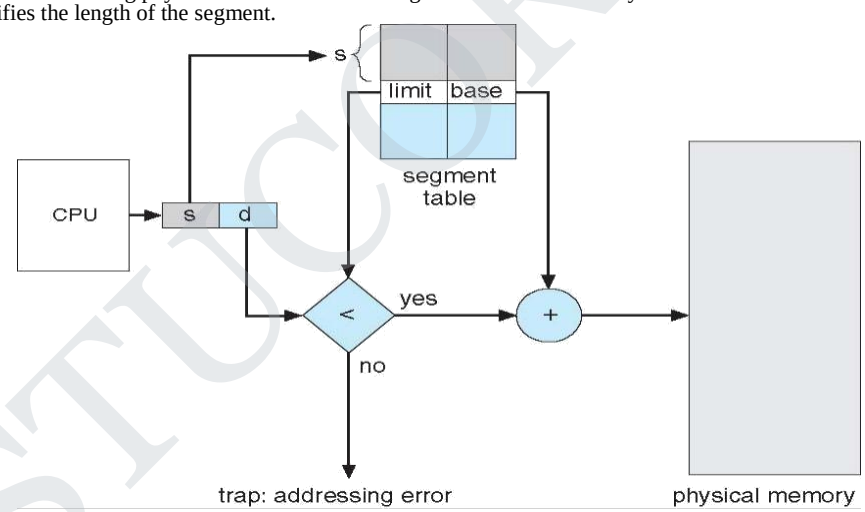
The two-dimensional user-defined addresses should be mapped into one-dimensional physical addresses.

The mapping of logical address to physical address is done by a table called segment table.

Each entry in the segment table has a **segment base** and a **segment limit**.

The segment base contains the starting physical address where the segment resides in memory

The segment limit specifies the length of the segment.



A logical address consists of two parts: a segment number, *s*, and an offset into that segment, *d.*

The segment number is used as an index to the segment table. The offset *d* of the logical address must be between 0 and the segment limit.

If it is not between 0 and limit then hardware trap to the operating system (logical addressing attempt beyond end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.

The segment table is an array of base–limit register pairs.

Segmentation can be combined with paging.

Example: Consider five segments numbered from 0 through 4. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).

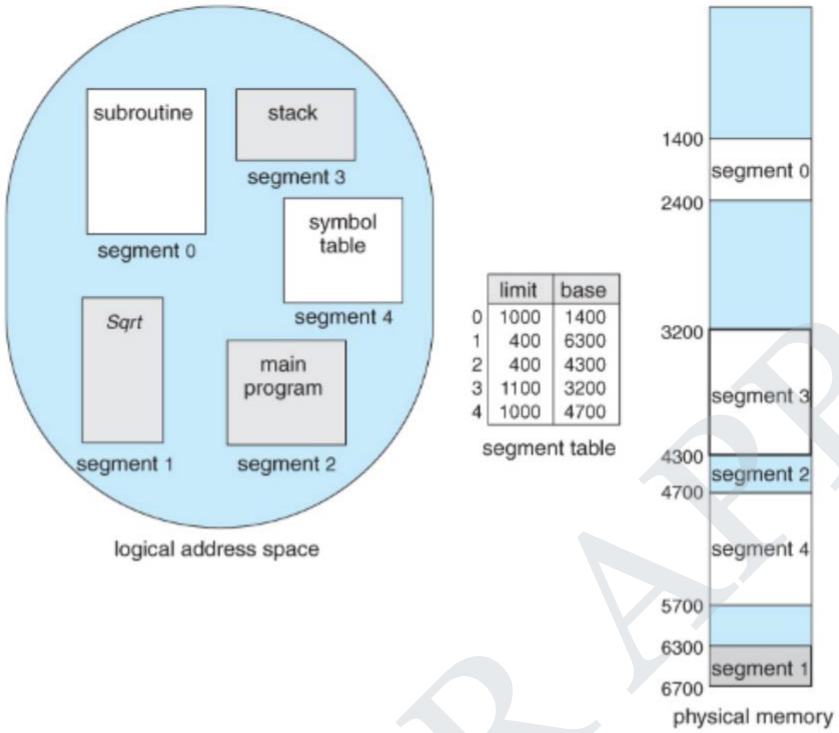


Figure 8.9 - Example of segmentation

The Segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353.
A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.
A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

PAGING:

Paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.
Paging avoids external fragmentation and the need for compaction, whereas segmentation does not.
When a process is to be executed, its pages are loaded into any available memory frames

PAGING HARDWARE:

Every address generated by the CPU is divided into two parts: a **page number (p)** and a **page offset (d)**.
The page number is used as an index into a **page table.**
The page table contains the base address of each page in physical memory.
This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

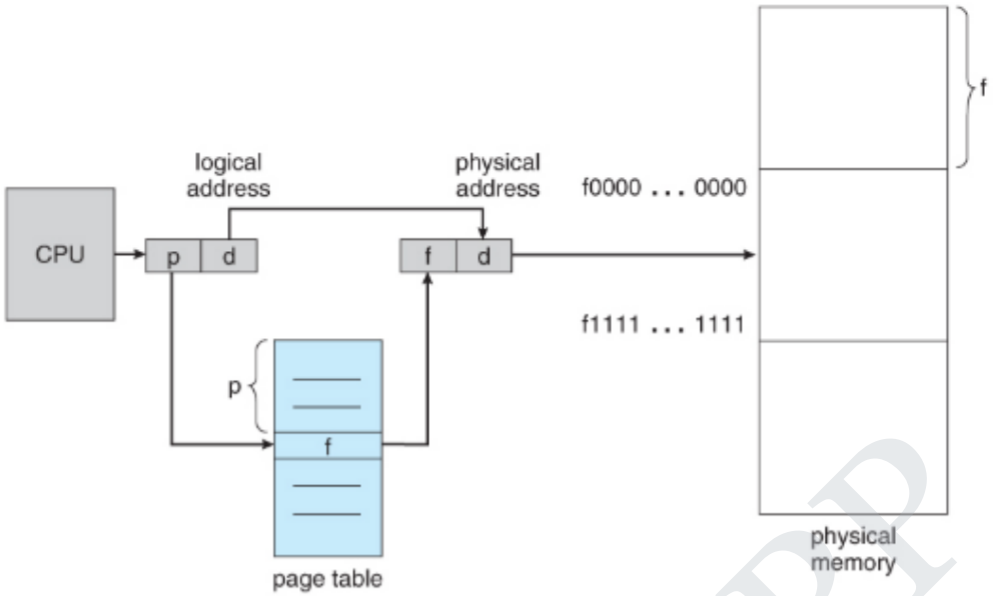
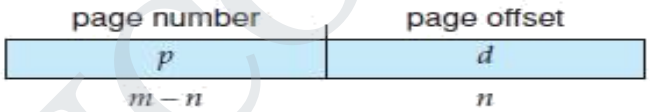


Figure 8.10 - Paging hardware

The page size is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page.
If the size of the logical address space is 2*m*, and a page size is 2*n* bytes, then the high-order *m*− *n* bits of a logical address designate the page number, and the *n* low-order bits designate the page offset. The logical address is given by



Here p is an index into the page table and d is the displacement within the page.

PAGING MODEL:

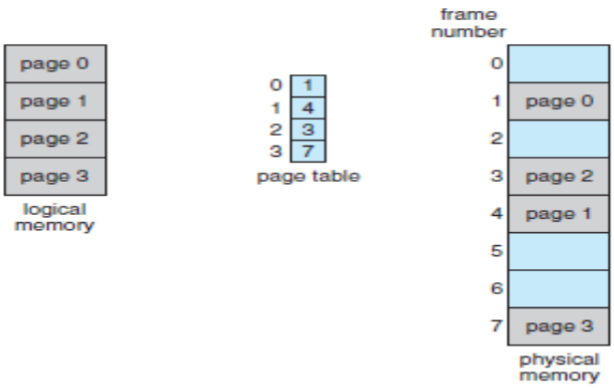


Figure 8.11 Paging model of logical and physical memory.

PAGING EXAMPLE:

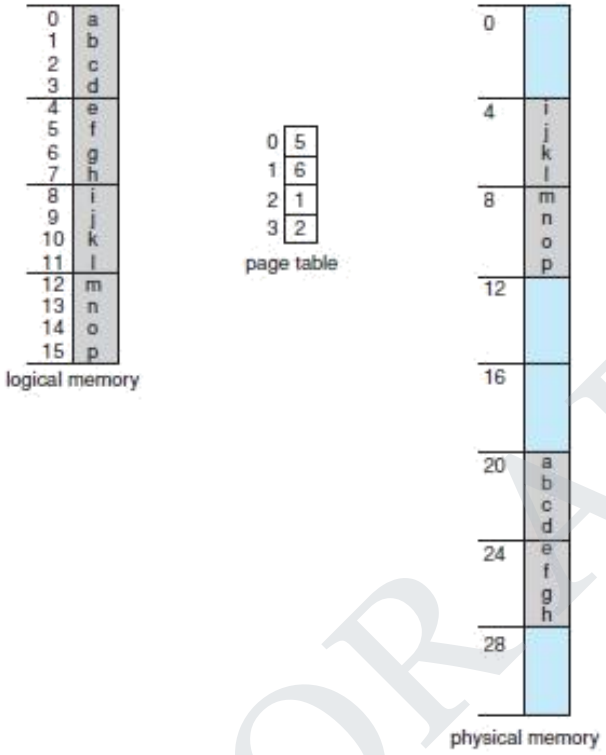


Figure 8.12 Paging example for a 32-byte memory with 4-byte pages.

Consider the memory with the logical address, n= 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes

Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) + 0].

Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].

Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.

Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].

FREE FRAME LIST:

Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory.

If n frames are available, they are allocated to this arriving process.

The operating system is managing physical memory and knows the allocation details of physical memory— which frames are allocated, which frames are available, how many total frames there are, and so on.

This information is generally kept in a data structure called a frame table.

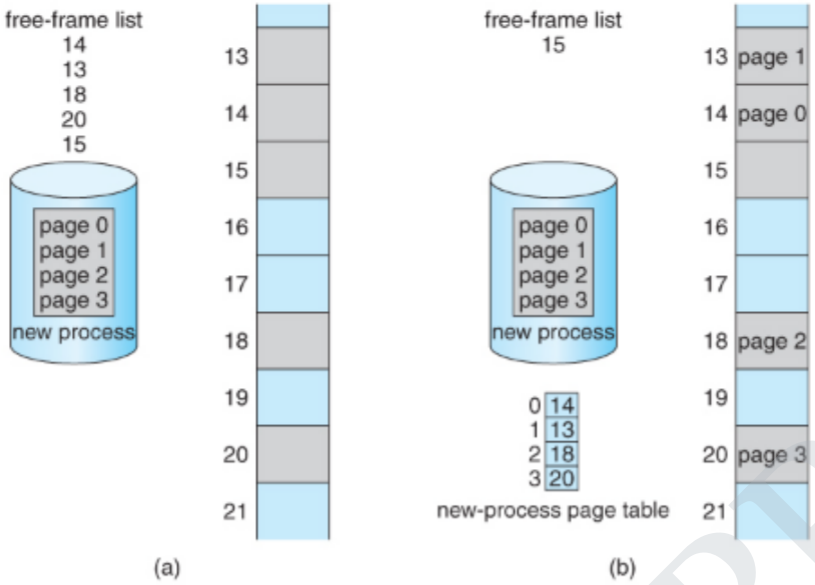


Figure 8.13 - Free frames (a) before allocation and (b) after allocation

HARDWARE SUPPORT:

The hardware implementation of the page table can be done in several ways. The page table is implemented as a set of dedicated registers if the size of the page table is too small.

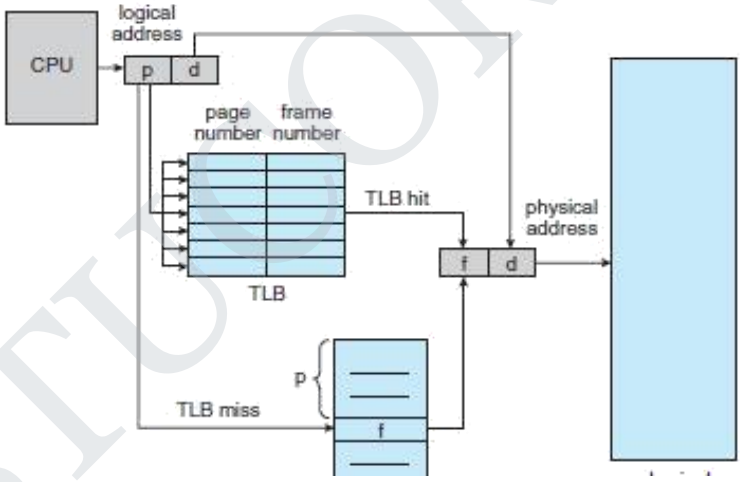
If the size of the page table is too large then the page table is kept in main memory and a page table base register is used to point to the page table.

When the page table is kept in main memory then two memory accesses are required to access a byte.

One for accessing the page table entry, another one for accessing the byte.

Thus the overhead of accessing the main memory increases.

The standard solution to this problem is to use a special, small, fast lookup hardware cache called a translation look-aside buffer (TLB).



Each entry in the TLB consists of two parts: a key (or tag) and a value.

The TLB contains only a few of the page-table entries.

When a logical address is generated by the CPU, its page number is presented to the TLB. If the page number is found **(TLB HIT),** its frame number is immediately available and is used to access memory.

If the page number is not in the TLB (TLB **miss**), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory.

The percentage of times that the page number of interest is found in the TLB is called the hit ratio.

The access time of a byte is said to be effective when the TLB hit ratio is high.

Thus the effective access time is given by

Effective access time = TLB hit ratio* Memory access time +TLB miss ratio* (2*memory access time)

PROTECTION:

Memory protection in a paged environment is accomplished by protection bits associated with each frame.

One bit can define a page to be read–write or read-only. When the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page

One additional bit is generally attached to each entry in the page table: a valid–invalid bit. When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal.

When the bit is set to invalid, the page is not in the process's logical address space.

Page-table length register (PTLR),is used to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process

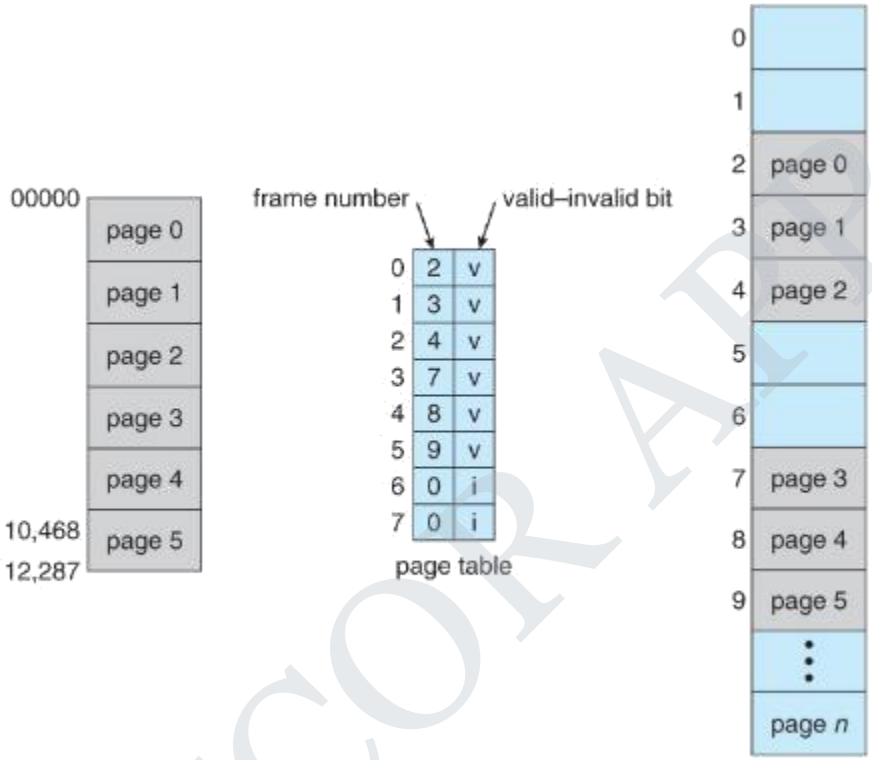


Figure 8.15 - Valid (v) or invalid (i) bit in page table

SHARED PAGES:

An advantage of paging is the possibility of sharing common code.

If the code is reentrant code (or pure code), however, it can be shared.

Reentrant code is non-self-modifying code: it never changes during execution. Thus, two or more processes can execute the same code at the same time

EXAMPLE: Consider three processes that share a page editor which is of three pages. Each process has its own data page.

Only one copy of the editor need be kept in physical memory. Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.

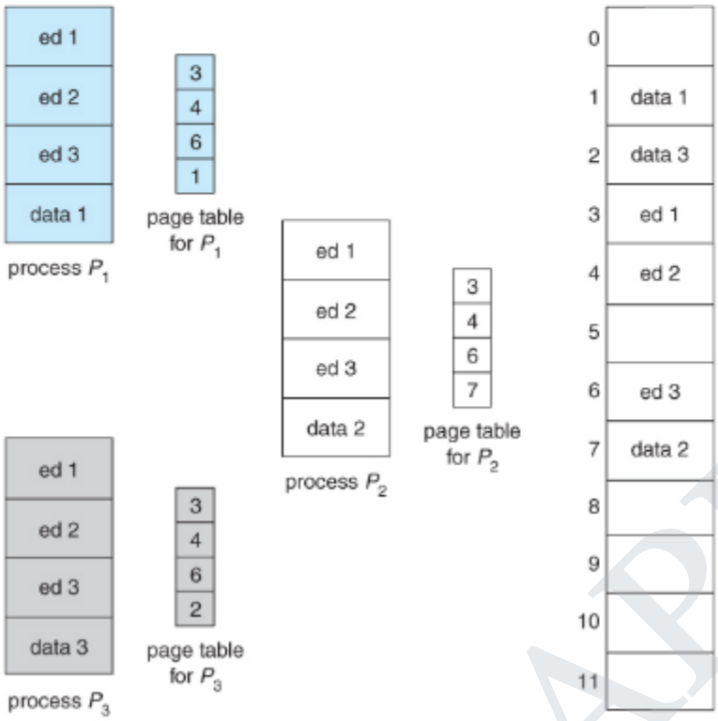


Figure 8.16 - Sharing of code in a paging environment

STRUCTURE OF PAGE TABLE:

The structure of page table includes

- Hierarchical paging
- Hashed page table
- Inverted page table.

HIERARCHIAL PAGING:

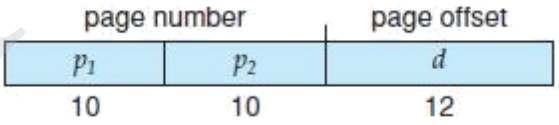
In Computer a system that has a 32 bit logical address space the page table becomes too large. Each process may need upto 4MB of Physical address space for the page table alone.

One simple solution to this problem is to divide the page table into smaller pieces. One way is to use a two-level paging algorithm, in which the page table itself is also paged.

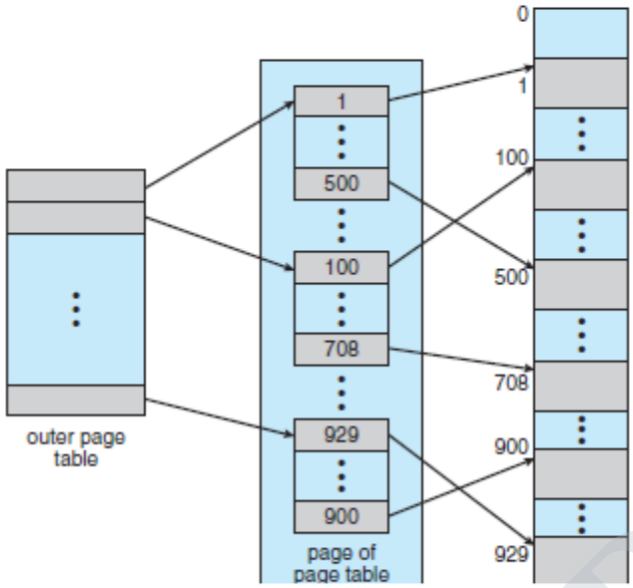
Consider the system with a 32-bit logical address space and a page size of (2 12)4 KB.

A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits.

The page number is further divided into a 10-bit page number and a 10-bit page offset.



Here p1 is an index into the outer page table and p2 is the displacement within the page of the inner page table.



Address translation works from the outer page table inward, this scheme is also known as a forward mapped page table.

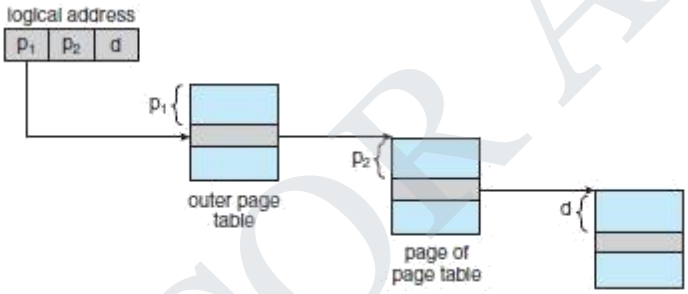


Figure 8.18 Address translation for a two-level 32-bit paging architecture.

HASHED PAGE TABLES:

A common approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual page number.

Each entry in the hash table contains a linked list of elements that hash to the same location

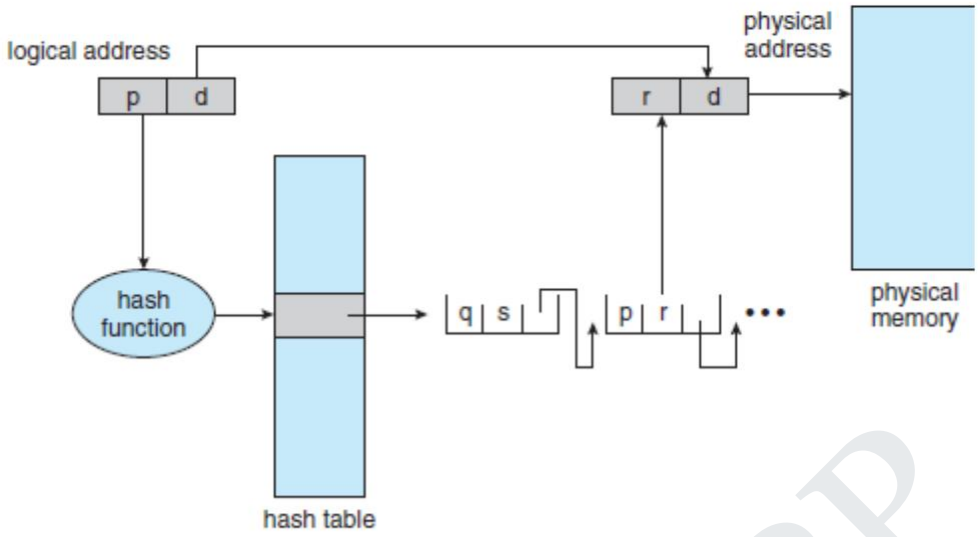
Each element consists of three fields:

virtual page number,
value of the mapped page frame
pointer to the next element in the linked list

The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list.

If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.

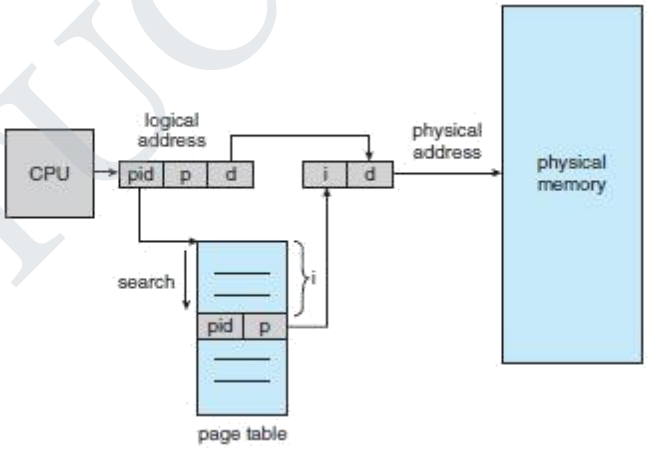
If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.



A variation to hashed page table is **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page.

INVERTED PAGE TABLES:

Each process has an associated page table. The page table has one entry for each page that the process is using.
The table is sorted by virtual address, the operating system calculate where in the table the associated physical address entry is located and to use that value directly.
One of the drawbacks of this method is that each page table may consist of millions of entries.
To solve this problem, we can use an inverted page table.
An inverted page table has one entry for each frame of memory. Each entry consists of the virtual address of the page stored in that real memory location; with information about the process that owns the page.
Thus, only one page table is in the system, and it has only one entry for each page of physical memory.
Each Logical address in the system consists of a triple :< **process-id, page-number, offset>.**
Each inverted page-table entry is a pair **<process-id, page-number**>
When a memory reference occurs, part of the virtual address, consisting of <process-id,page number>,
is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry i—then the physical address <i, offset> is generated..



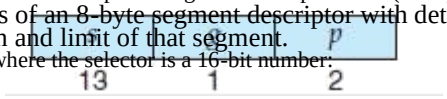
INTEL 32 AND 64-BIT ARCHITECTURES [SEGMENTATION WITH PAGING]

Memory management in IA-32 systems is divided into two components segmentation and paging.

The CPU generates logical addresses, which are given to the segmentation unit.
The segmentation unit produces a linear address for each logical address.
The linear address is then given to the paging unit, which in turn generates the physical address in main memory.
Thus, the segmentation and paging units form the equivalent of the memory-management unit

IA-32 Segmentation:

The IA-32 architecture allows a segment to be as large as 4 GB, and the maximum number of segments per process is 16 K.
The logical address space of a process is divided into two partitions. Information about the process that are private to the process is kept in the local descriptor table (LDT);
The information that is shared among other processes is kept in the global descriptor table (GDT).
Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the base location and limit of that segment.
The logical address is a pair (selector, offset), where the selector is a 16-bit number:



s designates the segment number, g indicates whether the segment is in the GDT or LDT, and p deals with protection.
The offset is a 32-bit number specifying the location of the byte.
The segment registers points to the appropriate entry in the LDT or GDT.
The base and limit information about the segment is used to generate a linear address.

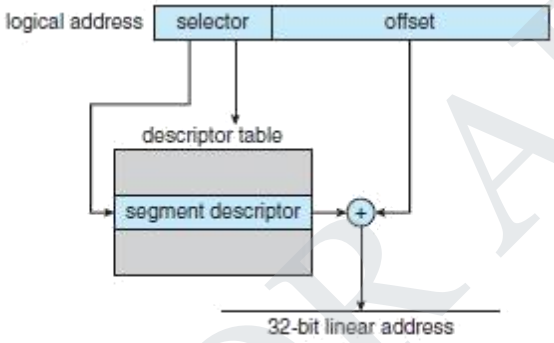
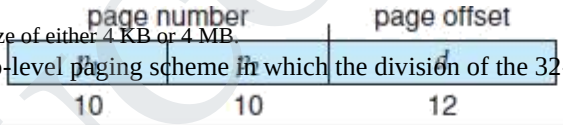


Figure 8.22 IA-32 segmentation.

IA -32 PAGING:

The IA-32 architecture allows a page size of either 4 KB or 4 MB.
For 4-KB pages, IA-32 uses a two-level paging scheme in which the division of the 32-bit linear address is as follows.



The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the page directory.
The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.
Finally, the low-order 12 bits refer to the offset in the 4-KB page pointed to in the page table.
Intel adopted a page address extension (PAE), which allows 32-bit processors to access a physical address space larger than 4 GB

PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to extend from 20 to 24 bits

The 10 high-order bits reference an entry in the outermost page table, which IA-32 terms the page directory.

The page directory entry points to an inner page table that is indexed by the contents of the innermost 10 bits in the linear address.

Finally, the low-order 12 bits refer to the offset in the 4-KB page pointed to in the page table.

Intel adopted a page address extension (PAE), which allows 32-bit processors to access a physical address space larger than 4 GB

PAE also increased the page-directory and page-table entries from 32 to 64 bits in size, which allowed the base address of page tables and page frames to extend from 20 to 24 bits

VIRTUAL MEMORY:

Virtual memory is a memory management technique that allows the execution of processes that are not completely in memory.

In some cases during the execution of the program the entire program may not be needed, such as error conditions, menu selection options etc.

The virtual address space of a process refers to the logical view of how a process is stored in memory.

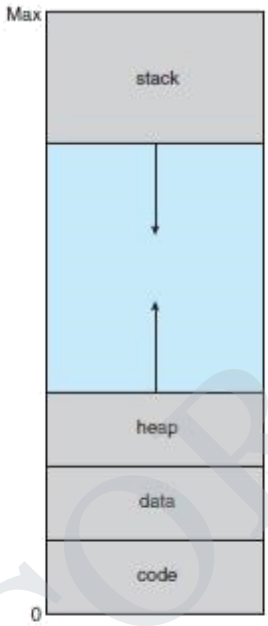


Figure 9.2 Virtual address space.

The heap will grow upward in memory as it is used for dynamic memory allocation.

The stack will grow downward in memory through successive function calls .

The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.

Virtual address spaces that include holes are known as sparse address spaces.

Sparse address space can be filled as the stack or heap segments grow or if we wish to dynamically link libraries

Virtual memory allows files and memory to be shared by two or more processes through page sharing

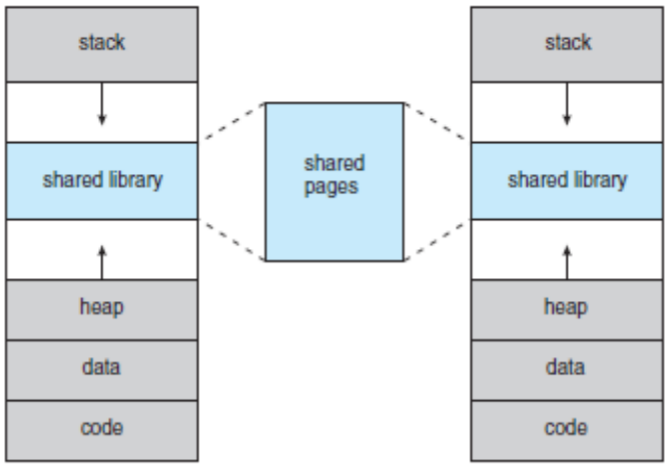


Figure 9.3 Shared library using virtual memory.

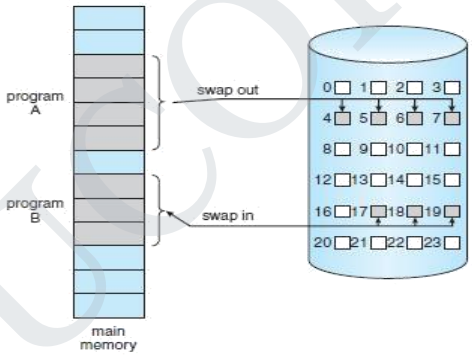
ADVANTAGES:

One major advantage of this scheme is that programs can be larger than physical memory
Virtual memory also allows processes to share files easily and to implement shared memory.
Increase in CPU utilization and throughput.
Less I/O would be needed to load or swap user programs into memory

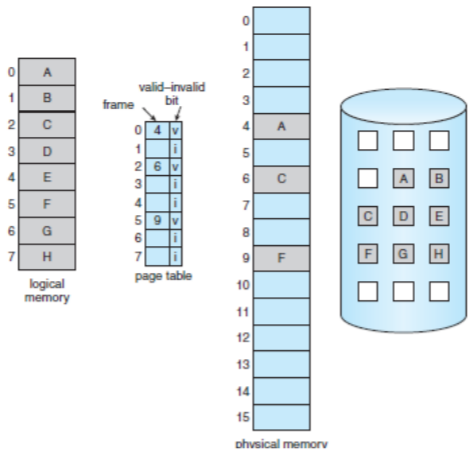
DEMAND PAGING:

Demand paging is the process of loading the pages only when they are demanded by the process during execution. Pages that are never accessed are thus never loaded into physical memory.

A demand-paging system is similar to a paging system with swapping where processes reside in secondary memory



When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory we use a **lazy swapper** that never swaps a page into memory unless that page will be needed.
Lazy swapper is termed to as pager in demand paging.
When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those pages into memory.
Os need the hardware support to distinguish between the pages that are in memory and the pages that are on the disk. The valid–invalid bit scheme can be used for this purpose.
If the bit is set to —valid,‖ the associated page is both legal and in memory.
If the bit is set to —invalid,‖ the page either is not valid or is valid but is currently on the disk.



PAGE FAULT: If the process tries to access a page that was not brought into memory, then it is called as a page fault. Access to a page marked invalid causes a page fault.
The paging hardware, will notice that the invalid bit is set, causing a trap to the operating system.

PROCEDURE FOR HANDLING THE PAGE FAULT:

Check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.

If the reference was invalid, we terminate the process. If it was valid but we have not yet brought in that page, we now page it in.

Find a free frame

Schedule a disk operation to read the desired page into the newly allocated frame.

When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

Restart the instruction that was interrupt. Though it had always been in memory.

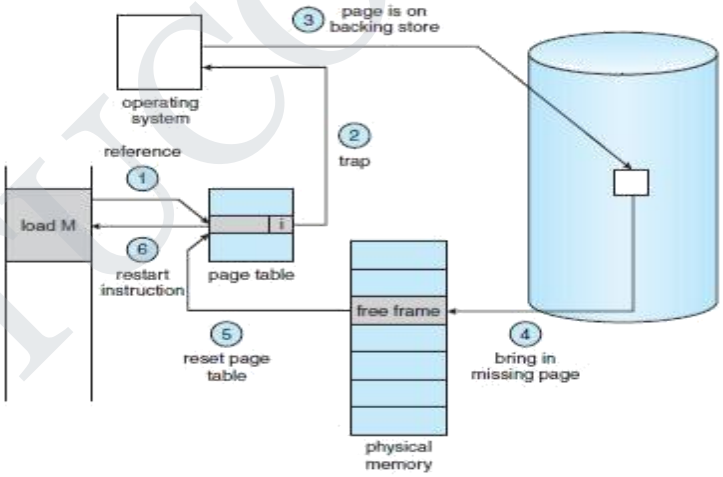


Figure 9.6 Steps in handling a page fault.

PURE DEMANDPAGING: The process of executing a program with no pages in main memory is called as pure demand paging. This never brings a page into memory until it is required.

The hardware to support demand paging is the same as the hardware for paging and swapping:

Page table. This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.

Secondary memory. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as **swap space.**

PERFORMANCE OF DEMAND PAGING:

Demand paging can affect the performance of a computer system.
The effective access time for a demand-paged memory is given by
effective access time = (1 − p) × ma + p × page fault time.
The memory-access time, denoted *ma*, ranges from 10 to 200 nanoseconds.
If there is no page fault then the effective access time is equal to the memory access time.
If a page fault occurs, we must first read the relevant page from disk and then access the desired word.
There are three major components of the page-fault service time:

Service the page-fault interrupt.
Read in the page.
Restart the process

With an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is

Effective access time = (1 − p) × (200) + p (8 milliseconds)
(1 − p) × 200 + p × 8,000,000
200 + 7,999,800 × p.

Effective access time is directly proportional to the page-fault rate.

Page replacement is basic to demand paging

If a page requested by a process is in memory, then the process can access it. If the requested page is not in main memory, then it is page fault.

When there is a page fault the OS decides to load the pages from the secondary memory to the main memory. It looks for the free frame. If there is no free frame then the pages that are not currently in use will be swapped out of the main memory, and the desired page will be swapped into the main memory.

The process of swapping a page out of main memory to the swap space and swapping in the desired page into the main memory for execution is called as Page Replacement.

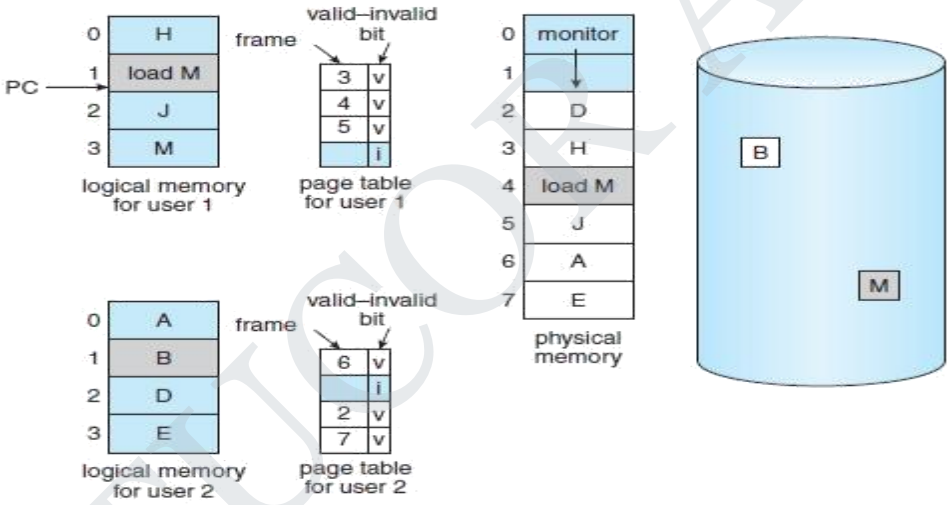


Figure 9.9 Need for page replacement.

STEPS IN PAGE REPLACEMENT:

Find the location of the desired page on the disk.
Find a free frame:
a. If there is a free frame, use it.
b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
c. Write the victim frame to the disk; change the page and frame tables accordingly.
Read the desired page into the newly freed frame; change the page and frame tables.
Continue the user process from where the page fault occurred.

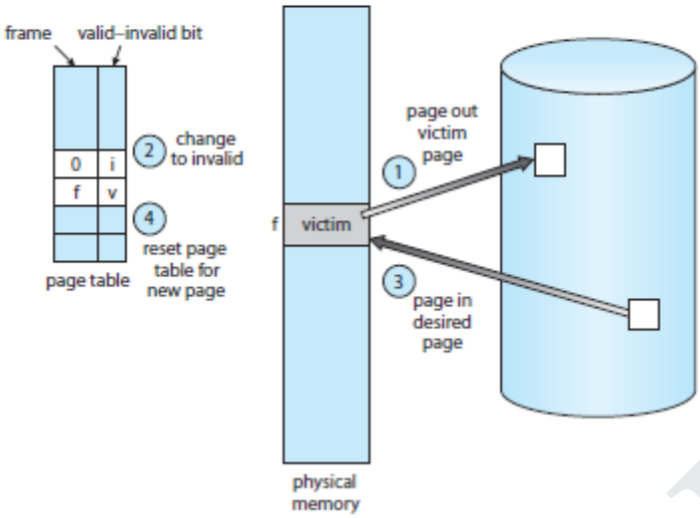


Figure 9.10 Page replacement.

If no frames are free, two page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.
This overhead can be reduced by using a modify bit (or dirty bit).
When this scheme is used, each page or frame has a modify bit associated with it in the hardware.
MODIFY BIT: The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified**.**
When we select a page for replacement, we examine its modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk.
If the modify bit is not set, however, the page has not been modified since it was read into memory. In this case, we need not write the memory page to the disk: it is already there.

PAGE REPLACEMENT ALGORITHMS:

If we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced.
The string of memory references made by a process is called a **reference string**.

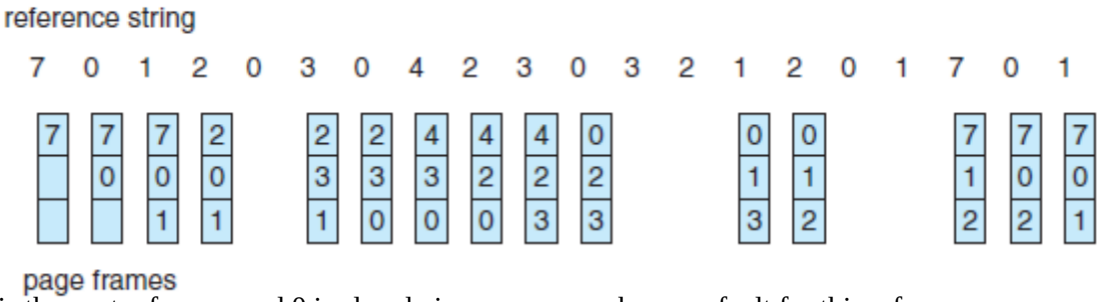
FIFO page Replacement
Optimal Page Replacement
LRU Page Replacement
LRU Approximation page Replacement algorithm
Counting Based Page Replacement Algorithm
Page Buffering Algorithm

FIFO PAGE REPLACEMENT:

The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm.
A FIFO replacement algorithm replaces the oldest page that was brought into main memory.

EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.

The three frames are empty initially.
The first three references (7, 0, 1) cause page faults and are brought into these empty frames.
The algorithm has 15 faults.



Page 0 is the next reference and 0 is already in memory, we have no fault for this reference.
The first reference to 3 results in replacement of page 0, since it is now first in line.
Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. The process continues until all the pages are referenced.

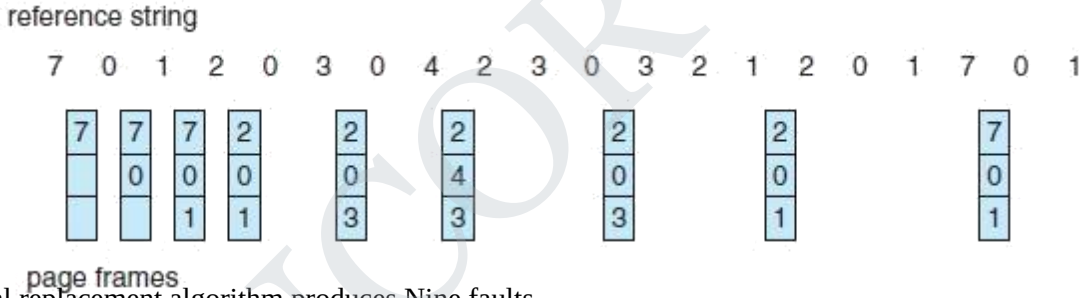
Advantages:
The FIFO page-replacement algorithm is easy to understand and program

Disadvantages:
The Performance is not always good.
It Suffers from Belady's Anomaly.

BELADY'S ANOMALY: The page fault increases as the number of allocated memory frame increases. This unexpected result is called as Belady's Anomaly.

OPTIMAL PAGE REPLACEMENT
This algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly
Optimal page replacement algorithm Replace the page that will not be used for the longest period of time

EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.



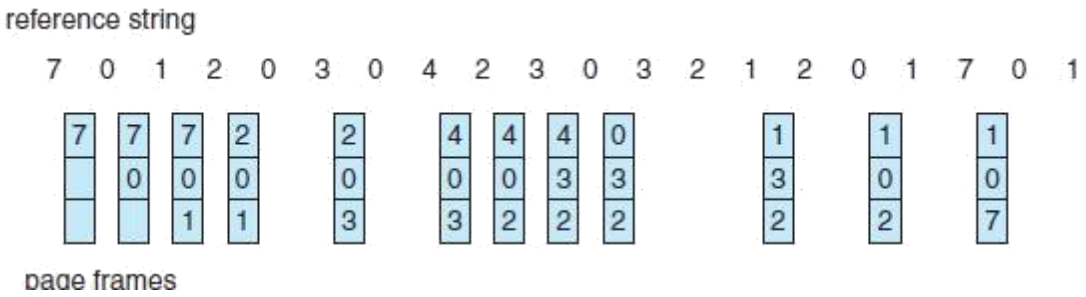
The Optimal replacement algorithm produces Nine faults
The first three references cause faults that fill the three empty frames.
The reference to page 2 replaces page 7, because page 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14.
The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again.

Advantages:
optimal replacement is much better than a FIFO algorithm

Disadvantage:
The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU PAGE REPLACEMENT
The Least Recently used algorithm replaces a page that has not been used for a longest period of time.
LRU replacement associates with each page the time of that page's last use.
It is similar to that of Optimal page Replacement looking backward in time, rather than forward.

EXAMPLE: Consider the Reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.



The LRU algorithm produces twelve faults

The first three references cause faults that fill the three empty frames.

The reference to page 2 replaces page 7, because page 7 has not been used for a longest period of time, when we look backward.

The Reference to page 3 replaces page 1, because page 1 has not been used for a longest period of time.

When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.

Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used.

When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory.

Advantages:

The LRU policy is often used as a page-replacement algorithm and is considered to be good.

LRU replacement does not suffer from Belady's anomaly.

Disadvantage:

The problem is to determine an order for the frames defined by the time of last use.

Two implementations are feasible:

Counters. We associate with each page-table a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. So we can find the ―time‖ of the last reference to each page.

Stack. Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom

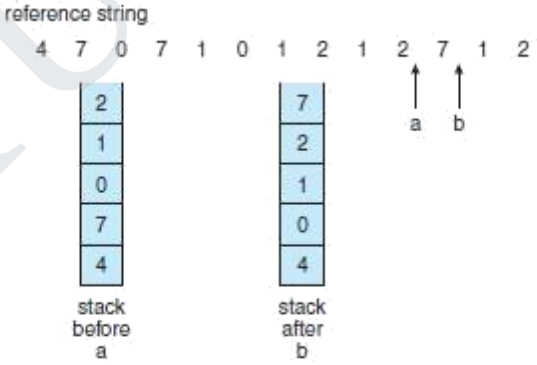


Figure 9.16 Use of a stack to record the most recent page references.

STACK ALGORITHM:

A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for *n* frames is always a ***subset*** of the set of pages that would be in memory with *n* + 1 frames.

LRU APPROXIMATION PAGE REPLACEMENT ALGORITHM:

The system provides support to the LRU algorithm in the form of a bit called Reference bit.

REFERENCE BIT:

The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page).

Reference bits are associated with each entry in the page table.

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

After some time, we can determine which pages have been used and which have not been used by examining the reference bits.

This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Additional-reference-bits algorithm

The additional ordering information can be gained by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory.

At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system.

These 8-bit shift registers contain the history of page use for the last eight time periods.

If the shift register contains 00000000, for example, then the page has not been used for eight time periods.

A page that is used at least once in each period has a shift register value of 11111111.

A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

Thus the page with the lowest number is the LRU page, and it can be replaced.

Second-Chance Algorithm:

The basic algorithm of second-chance replacement is a FIFO replacement algorithm.

When a page has been selected, however, we inspect its reference bit.

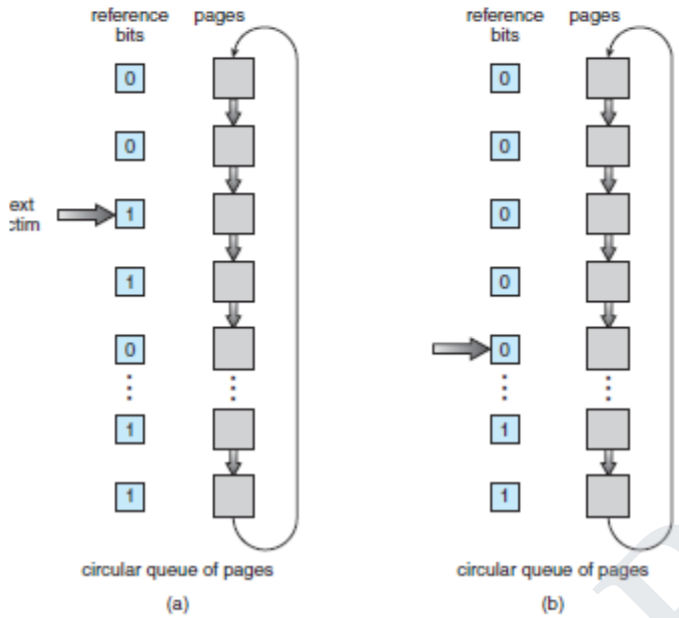
If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page.

When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced

One way to implement the second-chance algorithm is as a circular queue.

A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit.

Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position



Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify as an ordered pair

The order is {Reference bit, Modify bit}

we have the following four possible classes:

0, 0) neither recently used nor modified—best page to replace

0, 1) not recently used but modified—not quite as good

1, 0) recently used but clean—probably will be used again soon

1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

Here we give preference to those pages that have been modified in order to reduce the number of I/Os required. Thus the modified pages will not be replaced before writing it to the disk.

COUNTING-BASED PAGE REPLACEMENT

We can keep a counter of the number of references that have been made to each page.

This method includes two schemes

Least frequently used (LFU) page-replacement: The **least frequently used (LFU)** page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.

Most frequently used (MFU) page-replacement algorithm: The **most frequently used (MFU)** page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

PAGE BUFFERING ALGORITHM:

systems commonly keep a pool of free frames

When a page fault occurs, a victim frame is chosen as and the desired page is read into a free frame from the pool before the victim is written out.

This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset.

This scheme increases the probability that a page will be clean when it is selected for replacement

ALLOCATION OF FRAMES:

Allocation of frames deals with how the operating system allocates the fixed amount of free memory among the various processes.

Consider a single-user system with 128 KB of memory composed of pages 1 KB in size.

This system has 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process.

Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults.

The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on.

When the process terminated, the 93 frames would once again be placed on the free-frame list.

Operating system allocates all its buffer and table space from the free-frame list.

When this space is not in use by the operating system, it can be used to support user paging.

Minimum Number of Frames

OS cannot allocate more than the total number of available frames (unless there is page sharing). It must also allocate at least a minimum number of frames as required by the process.

The reason for allocating at least a minimum number of frames involves performance. As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.

Consider a machine in which all memory-reference instructions may reference only one memory address. We need at least one frame for the instruction and one frame for the memory reference.

If one-level indirect addressing is allowed (for example, a load instruction on page 16 can refer to an address on page 0, which is an indirect reference to page 23), then paging requires at least three frames per process.

The worst-case scenario occurs in computer architectures that allow multiple levels of indirection.

To overcome this difficulty, we must place a limit on the levels of indirection.

When the first indirection occurs, a counter is set to the maximum number of indirections allowed; the counter is then decremented for each successive indirection for this instruction. If the counter is decremented to 0, a trap occurs.

The minimum number of frames is defined by the computer architecture and the maximum number is defined by the amount of available physical memory.

Allocation Algorithms

The Operating system makes use of various allocation algorithms to allocate the frames to the user process.

Equal Allocation

Proportional Allocation

EQUAL ALLOCATION:

The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m*/*n* frames.

If there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool. This scheme is called **equal allocation**.

Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, each process will be allocated with 31 frames.

The student process does not need more than 10 frames, so the other 21 are wasted.

The Os allocates the available memory to each process according to its size.

Let the size of the virtual memory for process *pi* be *si* , given by

If the total number of available frames is *m*, we allocate *ai* frames to process *pi*, where *ai* is approximately ***ai = si/S × m.***

Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames.

we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since $10/137 \times 62 \approx 4$, and $127/137 \times 62 \approx 57$.

Thus both processes share the available frames according to their ―needs,‖ rather than equally.

A high-priority process is treated the same as a low-priority process.
we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.
One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

Global versus Local Allocation

Another important factor in the way frames are allocated to the various processes is page replacement.

Page-replacement algorithms are divided into two broad categories:

- **global replacement**
- **local replacement**.

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process;
Local replacement requires that each process select from only its own set of allocated frames.

EXAMPLE: consider an allocation scheme where in we allow high-priority processes to select frames from low-priority processes for replacement.

A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes.

In local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process.

Non-Uniform Memory Access

In some systems, a given CPU can access some sections of main memory faster than it can access others. These performance differences are caused by how CPUs and memory are interconnected in the system.
A system is made up of several system boards, each containing multiple CPUs and some memory.
The CPUs on a particular board can access the memory on that board with less delay than they can access memory on other boards in the system.
The systems in which the memory access time are uniform is called as Uniform memory access.
Systems in which memory access times vary significantly are known collectively as **non-uniform memory access (NUMA)** systems, and they are slower than systems in which memory and CPUs are located on the same motherboard.

Managing which page frames are stored at which locations can significantly affect performance in NUMA systems.
If we treat memory as uniform in such a system, CPUs may wait significantly longer for memory access.

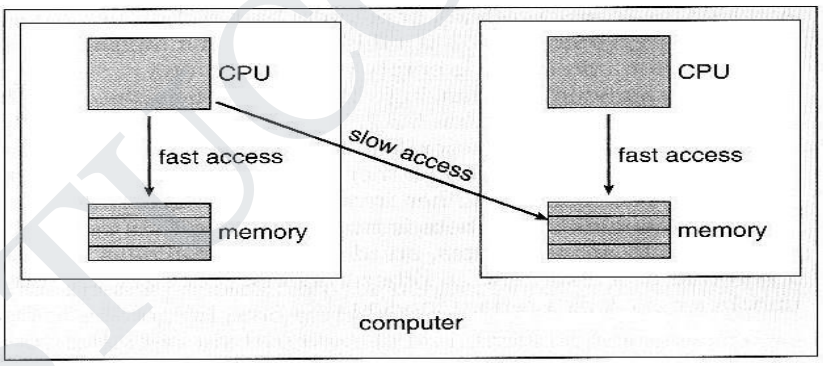


Figure 5.9 NUMA and CPU scheduling.

The goal is to have memory frames allocated ―as close as possible‖ to the CPU on which the process is running so that the memory access can be faster.

In NUMA systems the scheduler tracks the last CPU on which each process ran. If the scheduler tries to schedule each process onto its previous CPU, and the memory-management system tries to allocate frames for the process close to the CPU on which it is being scheduled, then improved cache hits and decreased memory access times will result.

THRASHING:

If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. If all its pages are in active use, it must replace a page that will be needed again right away. So it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called thrashing.

A process is thrashing if it is spending more time paging than executing.

Causes of Thrashing:

The operating system monitors CPU utilization If CPU utilization is too low; we increase the degree of multiprogramming by introducing a new Process to the system.

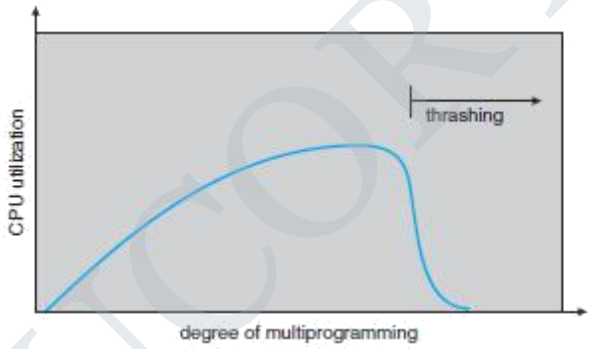
Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes.

A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong.

These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device.

As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more. **Thrashing has occurred,** and system throughput plunges.



At this point, to increase CPU utilization and stop thrashing, we must ***decrease*** the degree of Multi programming.

We can limit the effects of thrashing by using a **local replacement algorithm.** With local replacement, if one process starts thrashing, it cannot steal frames from another process, so the page fault of one process does not affect the other process.

To prevent thrashing, we must provide a process with as many frames as it needs. The Os need to know how many frames are required by the process.

The working-set strategy starts by looking at how many frames a process is actually using. This approach defines the **locality model** of process execution.

A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap.

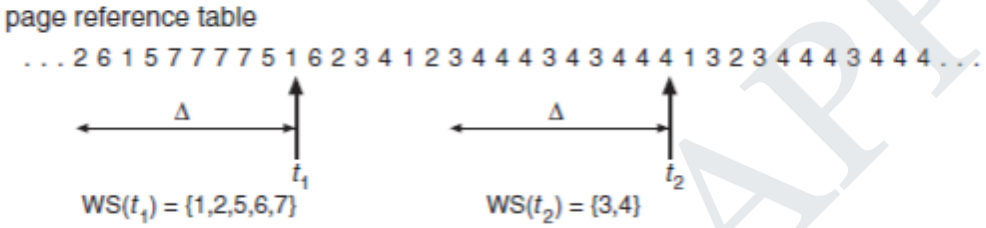
Suppose we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities.
If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

Working-Set Model

The **working-set model** is based on the assumption of locality.

- This model uses a parameter to define the **working-set window**.
- The idea is to examine the most recent page references.
- The set of pages in the most recent page references is the **working set.**
- If a page is in active use, it will be in the working set.
- If it is no longer being used, it will drop from the working set time units after its last reference.

EXAMPLE: Consider the sequence of memory references shown. If = 10 memory references, then the working set at time *t*1 is *{*1, 2, 5, 6, 7*}*. By time *t*2, the working set has changed to *{*3, 4*}*.



- If is too small, it will not encompass the entire locality;
- If is too large, it may overlap several localities.
- If is infinite, the working set is the set of pages touched during the process execution.

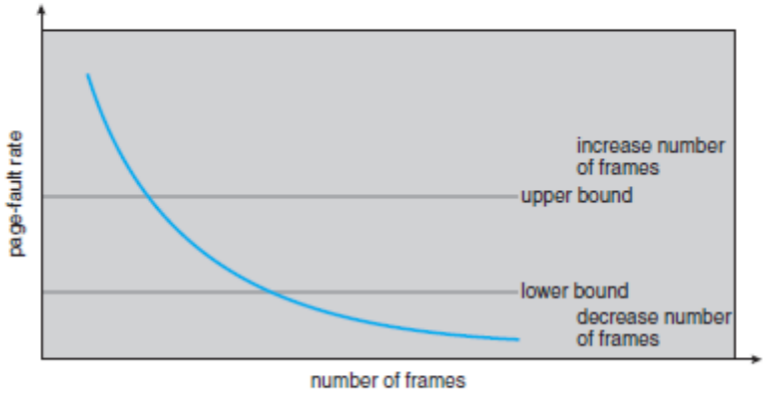
The most important property of the working set, then, is its size. If we compute the working-set size, *WSSi* , for each process in the system, we can then consider that
Here *D* is the total demand for frames.
Thus, process *i* needs *WSSi* frames. If the total demand is greater than the total number of available frames (*D*> *m*), thrashing will occur, because some processes will not have enough frames.
If there are enough extra frames, another process can be initiated.
If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend.
This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible.

Page-Fault Frequency

The page fault frequency is calculated by the total number of faults to the total number of references.

Page fault frequency = No. of page Faults / No. of References

Thrashing has a high page-fault rate. Thus, we want to control the page-fault rate.
When it is too high, we know that the process needs more frames.
Conversely, if the page-fault rate is too low, then the process may have too many frames.
Establish an upper and lower bounds on the desired page-fault rate.
If the actual page-fault rate exceeds the upper limit, we allocate the process another frame.



If the page-fault rate falls below the lower limit, we remove a frame from the process.
Thus, we can directly measure and control the page-fault rate to prevent thrashing.
If the page-fault rate increases and no free frames are available, we must select some process and swap it out to backing store.
The freed frames are then distributed to processes with high page-fault rates

ALLOCATING KERNEL MEMORY

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel.
If a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.
Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:
The kernel requests memory for data structures of varying sizes, some of which are less than a page in size
Certain hardware devices interact directly with physical memory and consequently may require memory residing in physically contiguous pages.

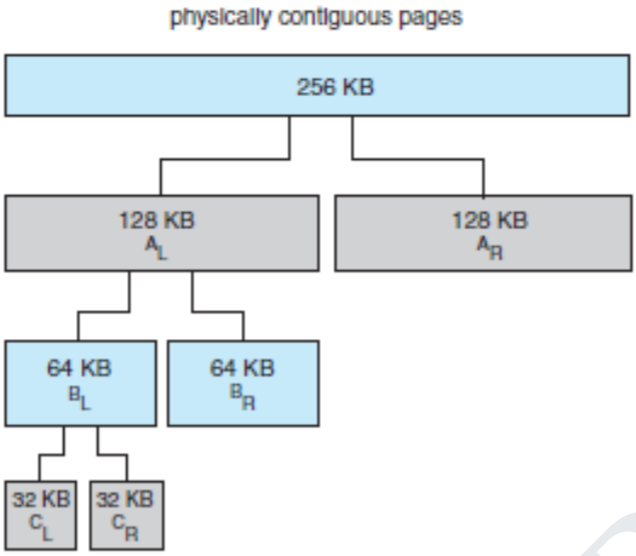
We examine two strategies for managing free memory that is assigned to kernel processes:

Buddy system
Slab allocation

Buddy Systems:

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages.
Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth).

EXAMPLE: Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two **buddies**—which we will call *AL* and *AR*—each 128 KB in size.



One of these buddies is further divided into two 64-KB buddies— *BL* and *BR*. However, the next-highest power of 2 from 21 KB is 32 KB so either *BL* or *BR* is again divided into two 32-KB buddies, *CL* and *CR*. One of these Buddies is used to satisfy the request.

Advantage: COALESCING: An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**. **Disadvantage:** It Leads to Internal Fragmentation.

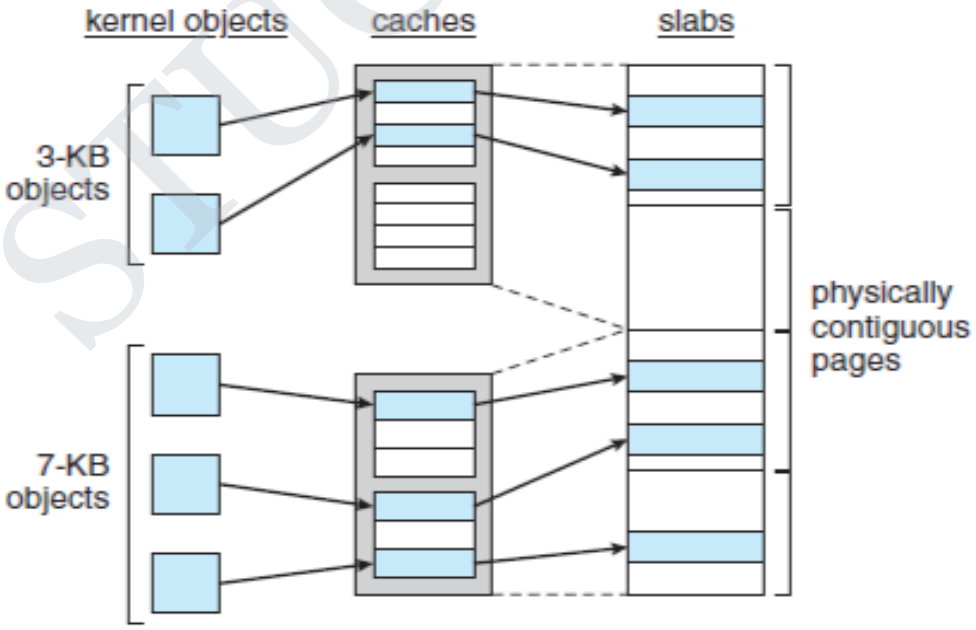
The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments.

Slab Allocation

A **slab** is made up of one or more physically contiguous pages.
There is a single cache for each unique kernel data structure.

Example: A separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores.

A **cache** consists of one or more slabs.



The slab-allocation algorithm uses caches to store kernel objects

When a cache is created, a number of objects—which are initially marked as free—are allocated to the cache.

Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used..

In Linux, a slab may be in one of three possible states:

Full. All objects in the slab are marked as used.

Empty. All objects in the slab are marked as free.

Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache.

Advantages: The slab allocator provides two main benefits:

No memory is wasted due to fragmentation.

Memory requests can be satisfied quickly.

OS EXAMPLES:

CASE STUDY: How Windows and Solaris implement virtual memory.

Windows:

Windows implements virtual memory using demand paging with **clustering**.

Clustering handles page faults by bringing in not only the faulting page but also several pages following the faulting page.

When a process is first created, it is assigned a working-set minimum and maximum.

The **working-set minimum** is the minimum number of pages the process is guaranteed to have in memory.

If sufficient memory is available, a process may be assigned as many pages as its **working-set maximum.**

The virtual memory manager maintains a list of free page frames.

Associated with this list is a threshold value that is used to indicate whether sufficient free memory is available.

If a page fault occurs for a process that is below its working-set maximum, the virtual memory manager allocates a page from this list of free pages.

If a process that is at its working-set maximum incurs a page fault, it must select a page for replacement using a local LRU page-replacement policy.

When the amount of free memory falls below the threshold, the virtual memory manager uses a tactic known as **automatic working-set trimming** to restore the value above the threshold.

If a process has been allocated more pages than its working-set minimum, the virtual memory Manager removes pages until the process reaches its working-set minimum.

2.Solaris:

In Solaris, when a thread incurs a page fault, the kernel assigns a page to the faulting thread from the list of free pages it maintains.

Associated with this list of free pages is a parameter—lotsfree—that represents a threshold to begin paging.

The lotsfree parameter is typically set to 1/64 the size of the physical memory. Four times per second, the kernel checks whether the amount of free memory is less than lotsfree.

If the number of free pages falls below lotsfree, a process known as a **pageout** starts up.

The front hand of the clock scans all pages in memory, setting the reference bit to 0.

Later, the back hand of the clock examines the reference bit for the pages in memory, appending each page whose reference bit is still set to 0 to the free list and writing to disk its contents if modified.

Solaris maintains a cache list of pages that have been ―freed‖ but have not yet been overwritten.

The free list contains frames that have invalid contents. Pages can be reclaimed from the cache list if they are accessed before being moved to the free list.

UNIT - IV FILE SYSTEMS AND I/O SYSTEMS

Mass Storage system – Overview of Mass Storage Structure, Disk Structure, Disk Scheduling and Management, swap space management; File-System Interface - File concept, Access methods, Directory Structure, Directory organization, File system mounting, File Sharing and Protection; File System Implementation- File System Structure, Directory implementation, Allocation Methods, Free Space Management, Efficiency and Performance, Recovery; I/O Systems – I/O Hardware, Application I/O interface, Kernel I/O subsystem, Streams, Performance.

MASS STORAGE STRUCTURE- OVERVIEW

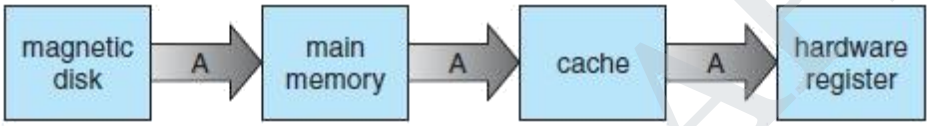
Main memory is usually too small to store all needed programs and data permanently.

Main memory is a **volatile** storage device that loses its contents when power is turned off or otherwise lost.

Thus, 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.

Most of the secondary storage devices are internal to the computer such as the hard disk drive, the tape disk drive and even the compact disk drive and floppy disk drive.



Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems.

Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches.

The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

A read–write head ―flies‖ just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit.

The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**.

CYLINDER: The set of tracks that are at one arm position makes up a **cylinder**.

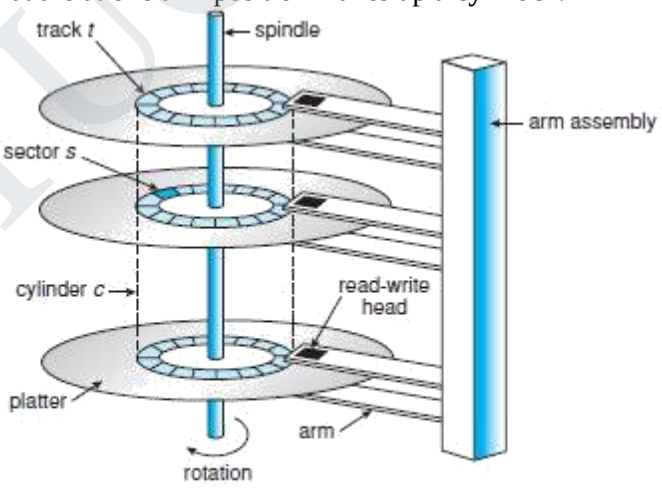


Figure 10.1 Moving-head disk mechanism.

The storage capacity of common disk drives is measured in gigabytes. When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute.

Disk speed has two parts.

The **transfer rate** is the rate at which data flow between the drive and the computer.

The **positioning time**, or **random-access time**

SEEK TIME: The time necessary to move the disk arm to the desired cylinder, is called the **seek time**.

ROTATIONAL LATENCY: The time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.

HEAD CRASH:

The disk read write head has a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash**.

A disk drive is attached to a computer by a set of wires called an **I/O bus**. Several kinds of buses are available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA universal serial bus (USB)**, and **fibre channel (FC)**.

The data transfers on a bus are carried out by special electronic processors called **controllers**. The **host controller** is the controller at the computer end of the bus.

A **disk controller** is built into each disk drive. To perform a disk I/O operation, the computer places a command into the host controller, typically using memory-mapped I/O ports

Magnetic Tapes:

Magnetic tape was used as an early secondary-storage medium.

It is relatively permanent and can hold large quantities of data.

Its access time is slow compared with that of main memory and magnetic disk.

In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

Disk Structure

Magnetic disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer.

The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to have a different logical block size, such as 1,024 bytes.

The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder.

The number of sectors per track is not constant on some drives.

For the disks that use **constant linear velocity (CLV)**, the density of bits per track is uniform.

In **constant angular velocity (CAV)** the density of bits decreases from inner tracks to outer tracks to keep the data rate constant.

Disk Attachment

Computers access disk storage in the following ways

- Host attached Storage
- Network Attached Storage
- Storage Area Network.

HOST ATTACHED STORAGE: Host-attached storage is storage accessed through local I/O ports. The typical desktop PC uses an I/O bus architecture called IDE or ATA.

NETWORK ATTACHED STORAGE: A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network .Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX systems or CIFS for Windows machines.

STORAGE AREA NETWORK: A storage-area network (SAN) is a private network connecting servers and storage units.

DISK SCHEDULING:

Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

Whether this operation is input or output
What the disk address for the transfer is
What the memory address for the transfer is
What the number of sectors to be transferred is

Disk Scheduling: If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. When one request is completed, the operating system chooses which pending request to service next. This is called as Disk Scheduling.

Disk Components:
The two major components of the hard disk are Seek time and Rotational Latency.

Seek time: The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector.

Rotational latency: The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head.

Disk bandwidth: The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Scheduling Algorithms:

First Come First Serve
Shortest Seek Time First
Scan Algorithm
Circular Scan Algorithm
Look Algorithm
Circular Look Algorithm

FCFS Scheduling:

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm.
This algorithm is easy to implement but it generally does not provide the fastest service.

Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.

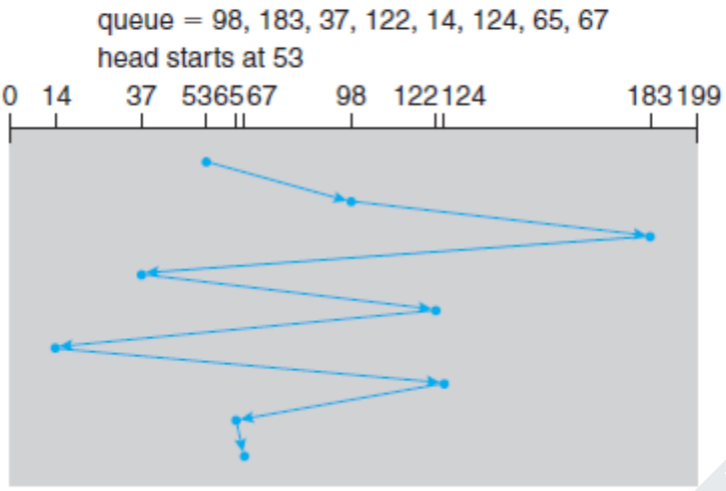


Figure 10.4 FCFS disk scheduling.

If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 24, 65, and finally to 67.

The total head movements is Head Movements = (53-98)+(98-183)+((183-37)+(122-14)+(14-124)+(124-65)+(65-67) = 640 Head Movements.

Disadvantage:

The request from 122 to 14 and then back to 124 increases the total head movements.

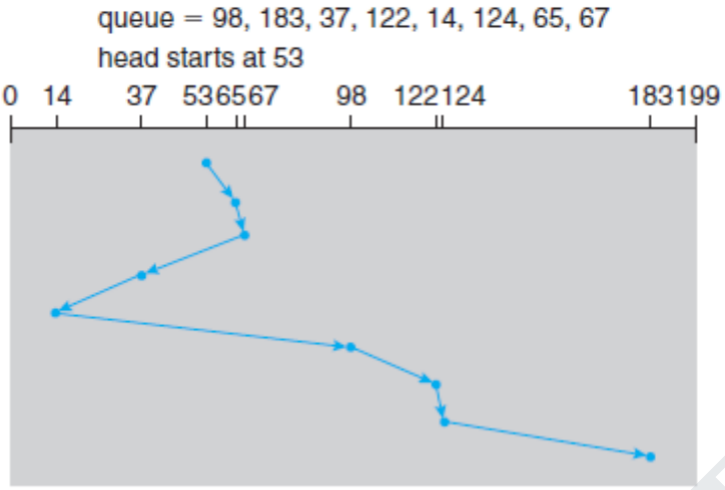
If the requests for cylinders 37 and 14 could be serviced together, before or after the requests for 122 and 124, the total head movement could be decreased and performance could be thereby improved.

SSTF Scheduling:

The **shortest-seek-time-first (SSTF) algorithm** selects the request with the least seek time from the current head position. It chooses the pending request closest to the current head position.

Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.

The closest request to the initial head position (53) is at cylinder 65.
Once we are at cylinder 65, the next closest request is at cylinder 67.
From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next.
Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183.
This scheduling method results in a total head movement of only 236 cylinders
SSTF algorithm gives a substantial improvement in performance.

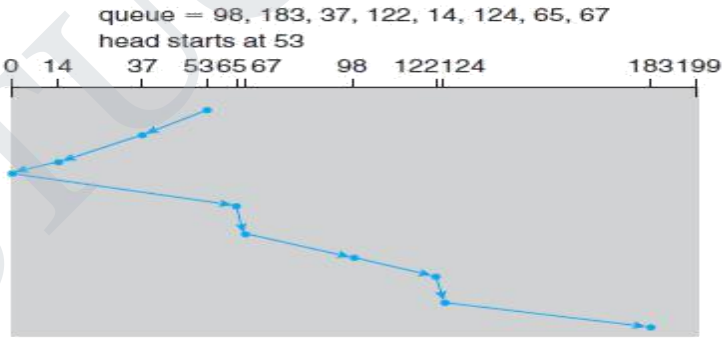


Disadvantage:
SSTF may cause starvation of some requests.

STARVATION: Suppose that we have two requests in the queue, for cylinders 14 and 186, and while the request from 14 is being serviced, a new request near 14 arrives. This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.

3) SCAN Scheduling:
In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk
At the other end, the direction of head movement is reversed, and servicing continues.
The head continuously scans back and forth across the disk.
The SCAN algorithm is sometimes called the **elevator algorithm.**

Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.
Assuming that the disk arm is moving toward 0 the head will next service 37 and then 14.



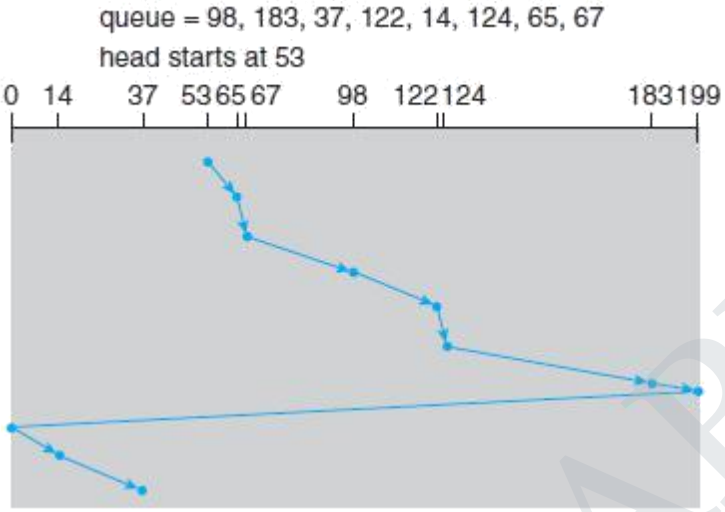
At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183.
If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

Circular SCAN Algorithm:
Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.

C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.

The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.



LOOK scheduling:

The **LOOK** algorithm is the same as the **SCAN** algorithm in that it also services the requests on both directions of the disk head, but it —Looks" ahead to see if there are any requests pending in the direction of head movement.

If no requests are pending in the direction of head movement, then the disk head traversal will be reversed to the opposite direction and requests on the other direction can be served.

In LOOK scheduling, the arm goes only as far as final requests in each direction and then reverses direction without going all the way to the end.

Consider an example, given a disk with 200 cylinders (0-199), suppose we have 8 pending requests: 98, 183, 37, 122, 14, 124, 65, 67 and that the read/write head is currently at cylinder 53. In order to complete these requests, the arm will move in the increasing order first and then will move in decreasing order after reaching the end. So, the order in which it will execute is 65, 67, 98, 122, 124, 183, 37, and 14.

Note :(Draw the Diagram and calculate the head movements for the previous example)

C-LOOK Scheduling:

This is just an enhanced version of C-SCAN.

Arm only goes as far as the last request in each direction, then reverses direction immediately, without servicing all the way to the end of the disk and then turns the next direction to provide the service.

Note :(Draw the Diagram and calculate the head movements for the previous example)

DISK MANAGEMENT:

The operating system is responsible for disk management.

The Major Responsibility includes

- Disk Formatting,
- Booting from disk
- Bad-block recovery.

1. Disk Formatting:

The Disk can be formatted in two ways,
Physical or Low level Formatting,
Logical Or High Level Formatting

Physical or Low level Formatting:

Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called **low-level formatting**, or **physical formatting**.
Low-level formatting fills the disk with a special data structure for each sector.
The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer.
The Header contains the Sector Number and the Trailer contains the Error correction code.
When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area.
When the sector is read, the ECC is recalculated and compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad. It then reports a recoverable **soft error**.

Logical Formatting Or High Level Formatting:

The operating record its own data structures on the disk during Logical formatting.
It does so in two steps.
The first step is to **partition** the disk into one or more groups of cylinders.
The operating system can treat each partition as though it were a separate disk.
For instance, one partition can hold a copy of the operating system's executable code, while another holds user files.
The second step is **logical formatting**, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the disk.
These data structures may include maps of free and allocated space and an initial empty directory.

CLUSTERS: To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters**.

RAW DISK: Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the **raw disk,** and I/O to this array is termed **raw I/O.**

Boot Block

The Process of Starting a computer System by loading the Operating system in the main memory is called as System Booting.
This is done by a initial program called as Bootstrap program initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system.
The bootstrap is stored in **read-only memory (ROM)**.The Problem here is that changing this bootstrap code requires changing the ROM hardware chips.
To overcome this most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk.
The full bootstrap program can be changed easily: a new version is simply written onto the disk.
BOOT DISK: The full bootstrap program is stored in the ―boot blocks‖ at a fixed location on the disk. A disk that has a boot partition is called a **boot disk** or **system disk**.
The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code which in turn loads the entire Operating System.

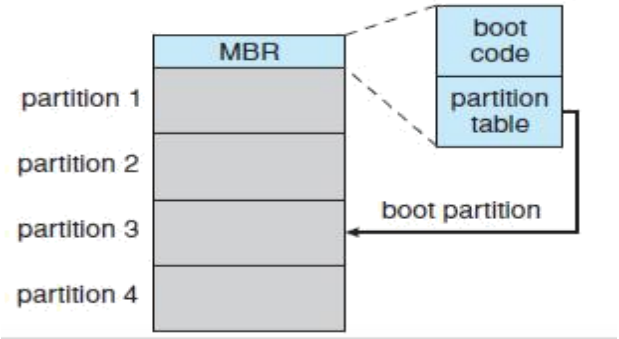
EXAMPLE: Boot Process in Windows.

BOOT PARTITION: Windows allows a hard disk to be divided into partitions, and one partition called as the **boot partition** contains the operating system and device drivers.

The Windows system places its boot code in the first sector on the hard disk, which it terms the **master boot record**, or **MBR**.

Booting begins by running code that is resident in the system's ROM memory. This code directs the system to read the boot code from the MBR.

Once the system identifies the boot partition, it reads the first sector from that partition and continues with the remainder of the boot process, which includes loading the various subsystems and system services.



Bad Blocks

A bad block is a damaged area of magnetic storage media that cannot reliably be used to store and retrieve data. Depending on the disk and controller in use, these blocks are handled in a variety of ways.

One strategy is to scan the disk to find bad blocks while the disk is being formatted. Any bad blocks that are discovered are flagged as unusable so that the file system does not allocate them.

In Some systems the controller maintains a list of bad blocks on the disk. This can be handled in two ways

Sector Sparing
Sector Slipping

SECTOR SPARING: Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**.

Example:

The operating system tries to read logical block 87.

The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.

The next time the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.

After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

SECTOR SLIPPING: The Process of moving all the sectors down one position from the bad sector is called as sector slipping.

Example: If the logical block 17 becomes defective and the first available spare follows sector 202. Sector slipping then remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18 so that sector 17 can be mapped to it.

FILE SYSTEM STORAGE:

The File System provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system.

The file system consists of two distinct parts:

a collection of files, each storing related data,
a directory structure, which organizes and provides information about all the files in the system.

FILE CONCEPTS:

A file is defined as a named collection of related information that is stored on secondary storage device.
Many different types of information may be stored in a file such as source or executable programs, numeric or text data, photos, music, video, and so on.
A file has a certain defined structure, which depends on its type.
Types of Files: A **text file** is a sequence of characters organized into lines.
A **source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements.
An **executable file** is a series of code sections that the loader can bring into memory and execute.

FILE ATTRIBUTES:

A file's attributes vary from one operating system to another but typically consist of these:

Name. The file name is the information kept in human readable form.
Identifier. This unique tag, usually a number, identifies the file within the file system
Type. This information is needed for systems that support different types of files.
Location. This information is a pointer to a device and to the location of the file on that device.
Size. The current size of the file (in bytes, words, or blocks)
Protection. Access-control information determines who can do reading, writing, executing
Time, date, and user identification. This information may be kept for creation, last modification, and last use.

FILE OPERATIONS:

A file is an abstract data type.
The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. The Basic Operations on a file includes

Creating a File
Writing a File
Reading a File
Repositioning within a File
Deleting a file
Truncating a file

Creating a file: OS first finds space in the file system for the file. Second, an entry for the new file must be made in the directory.

Writing a file. To write a file, we make a system specifying both the name of the file and the information to be written to the file. The system must keep a **write pointer** to the location in the file where the next write is to take place.

Reading a file. To read from a file, we use a system call that specifies the name of the file. The system needs to keep a **read pointer** to the location in the file where the next read is to take place.

Current File Position Pointer: Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current file- position pointer.**

Repositioning within a file. The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. This file operation is also known as files seek.

Deleting a file. To delete a file, we search the directory for the named file.

Truncating a file. The user may want to erase the contents of a file but keep its attributes

Open File Table:

Most of the file operations involve searching the directory for the entry associated with the named file.
The operating system keeps a table, called the **open-file table**, containing information about all open files.
When a file operation is requested, the file is specified via an index into this table, so no searching is required.
When the file is no longer being actively used, it is closed by the process, and the operating system removes its entry from the open-file table.

The open-file table also has an **open count** associated with each file to indicate how many processes have the file open Each close() decreases this open count, and when the open count reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.

When several processes may open the file , the operating system uses two levels of internal tables:

A per-process table

A system-wide table.

The per process table tracks all files that a process has open. It information regarding the process's use of the file such as the current file pointer, access rights.

The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size.

FILE LOCKS:

File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes.

Shared Lock

Exclusive Lock

A **shared lock** is similar to a reader lock in that several processes can acquire the lock concurrently.

An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock.

Operating systems may provide either **mandatory** or **advisory** file-locking mechanisms.

If a lock is mandatory, then once a process acquires an exclusive lock, the operating system will prevent any other process from accessing the locked file.

If a lock is advisory then the Operating System will allow the process to access the locked file.

A common technique for implementing file types is to include the type as part of the file name.

The name is split into two parts—a name and an extension, usually separated by a period or a dot.

The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

Example: Java compilers expect source files to have a .java extension, and the Microsoft Word word processor expects its files to end with a .doc or .docx extension.

file type	usual extension	Function
Executable	exe, com, bin or none	ready-to-run machine-language program
Object	obj, o	Compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
Batch	bat, sh	Commands to the command interpreter
Markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
Library	lib, a , so, dll	Libraries of routines for programmers
point or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
Archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
Multimedia	mpeg, mov, mp3, mp4,avi	binary file containing audio or A/V information

FILE ACCESS METHODS:

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways.

Sequential access Method

Direct Access method

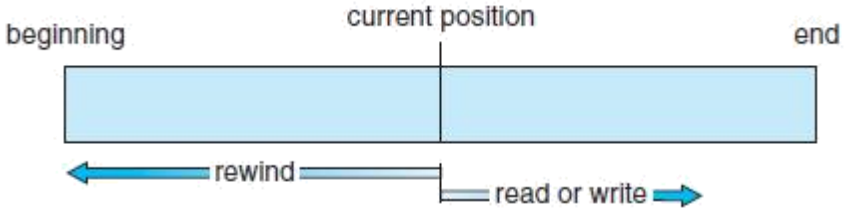
Indexed access method

Sequential access Method: The simplest access method is **sequential access**. Information in the file is processed in order, one record after the other.

Example: Editors and compilers usually access files in Sequential manner.

A read operation—read next ()—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.

Similarly, the write operation—write next ()—appends to the end of the file and advances to the end of the newly written material (the new end of file).



Direct Access Method:

A file is made up of fixed-length **logical records** that allow programs to read and write records rapidly in no particular order.

The file is viewed as a numbered sequence of blocks or records.

Direct-access files are of great use for immediate access to large amounts of information.

EXAMPLE: Databases are often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.

For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have read (n), where n is the block number, write (n) rather than write next ().

RELATIVE BLOCK NUMBER: The block number provided by the user to the operating system is normally a **relative block number**

sequential access	implementation for direct access
reset	cp
read_next	read cp; cp = cp + 1;
write_next	write cp; cp = cp + 1;

Indexed Access Methods:

These methods generally involve the construction of an index for the file.

The **index**, like an index in the back of a book, contains pointers to the various blocks.

To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.

With large files, the index file itself may become too large to be kept in memory.

One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items

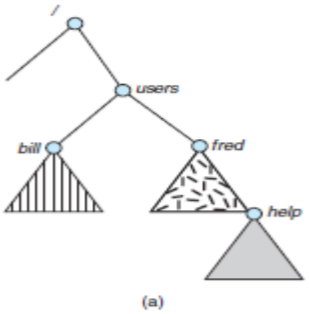
FILE SYSTEM MOUNTING:

File System Mounting is defined as the process of attaching an additional file system to the currently accessible file system of a computer. **A file system** is a hierarchy of directories that is used to organize files on a computer or storage media.

The operating system is given the name of the device and the **mount point**

Mount Point: It is the location within the file structure where the file system is to be attached. A mount point is an empty directory.

Example: A file system containing a user's home directories might be mounted as /home. To access the directory structure within that file system, we could precede the directory names with /home, as in /home/Jane. Mounting that file system under /users would result in the path name /users/Jane, which we could use to reach the same directory.



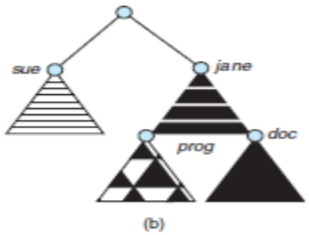


Figure 11.14 File system. (a) Existing system. (b) Unmounted volume.

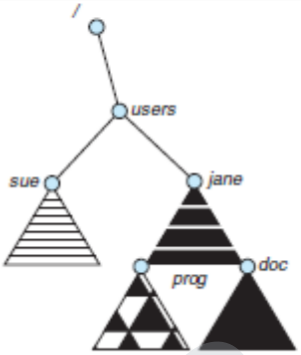
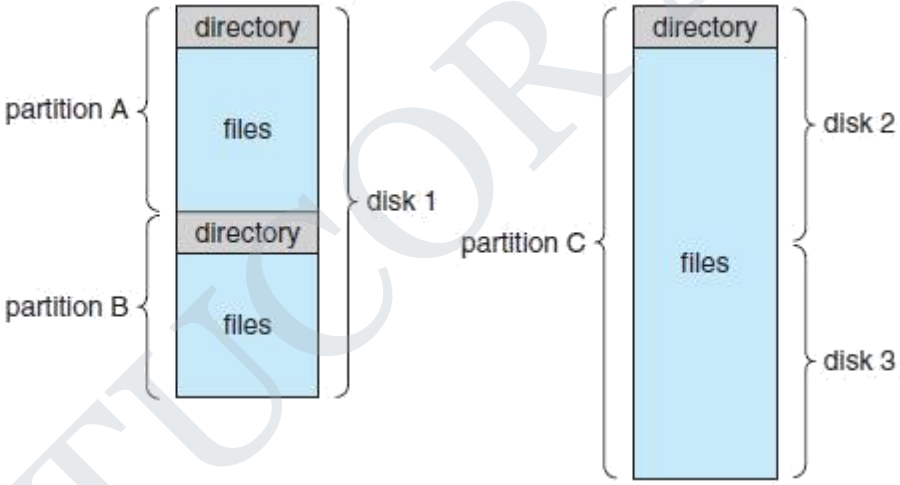


Figure 11.15 Mount point.

DIRECTORY AND DISK STRUCTURE

The Operating System divides the hard disk into various partitions and stores its File System in each Partition.
Any entity containing a file system is generally known as a **volume**.
Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**.
The device directory records information—such as name, location, size, and type—for all files on that volume.



The directory translates file names into their directory entries. Operations that are to be performed on a directory includes

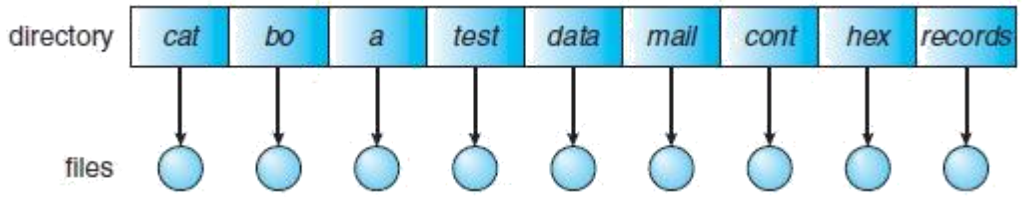
Search for a file
Create a File
Delete a file
List a Directory
Rename a File
List a directory
Traverse the file system.

Single level Directory
Two Level Directory
Tree Structured Directory

Acyclic Graph directory
General Graph directory.

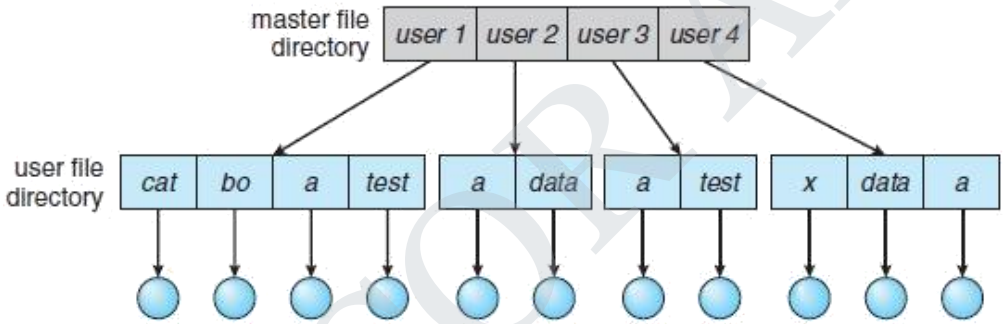
Single level Directory:

The simplest directory structure is the single-level directory. All files are contained in the same directory.
When the system has more than one user, the files created by different users should have unique names because all the files will be stored under a single directory
When the number of files increases the user may find it difficult to remember the names of all the files



Two Level directory:

In the two-level directory structure, each user has his own directory called **user file directory.**
The UFDs have similar structures, but each lists only the files of a single user.
When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched.
The MFD is indexed by user name or account number, and each entry points to the UFD for that user.
When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.
To create a file for a user, the operating system searches only that user's UFD to check whether another file of that name exists.
To delete a file, the operating system searches the local UFD; thus, it cannot accidentally delete another user's file that has the same name.



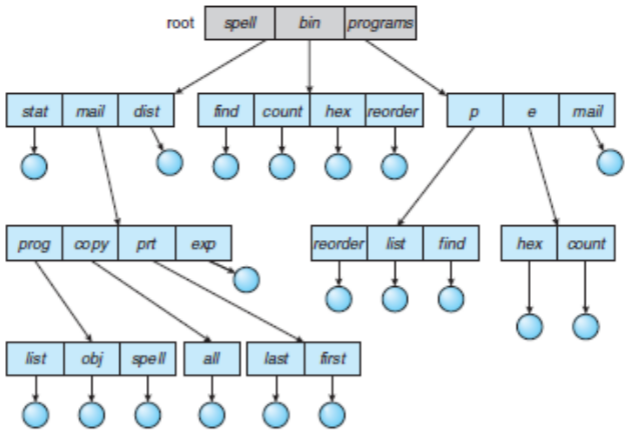
Advantage: The two-level directory structure solves the name-collision problem.

Disadvantage: It isolates one user from another user. It's a major disadvantage when the users want to cooperate on some task and to access one another's files.

Some systems simply do not allow local user files to be accessed by other users.
If access is to be permitted, one user must have the ability to name a file in another user's directory.
PATH NAME: A user name and a file name define a **path name**.
A two-level directory can be similar to that of a tree structure of height 2. The root of the tree is the MFD. Its direct Childs are the UFDs. The Childs of the UFDs are the files themselves. The files are the leaves of the tree.

Tree structured directory:

Tree structured directory is an expansion of two level directory extended to a tree of arbitrary height.
It allows users to create their own subdirectories and to organize their files.
The tree has a root directory, and every file in the system has a unique path name.
The **current directory** should contain most of the files that are of current need to the process.



Path names can be of two types:
- Absolute path name
- Relative path name

An **absolute path name** begins at the root and follows a path down to the specified file.
A **relative path name** defines a path from the current directory.
Users can be allowed to access the files of other users by specifying the path name.
A directory (or subdirectory) contains a set of files or subdirectories.
A bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).

FILE DELETION: If a directory is empty, its entry in the directory that contains it can simply be deleted.

If the directory to be deleted is not empty and contains several files or subdirectories. One of two approaches can be taken.
Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory.

If when a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted.

Acyclic Graph Directories:

An **acyclic graph is** a graph with no cycles.
It allows directories to share subdirectories and files
The same file or subdirectory may be in two different directories.

With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other.

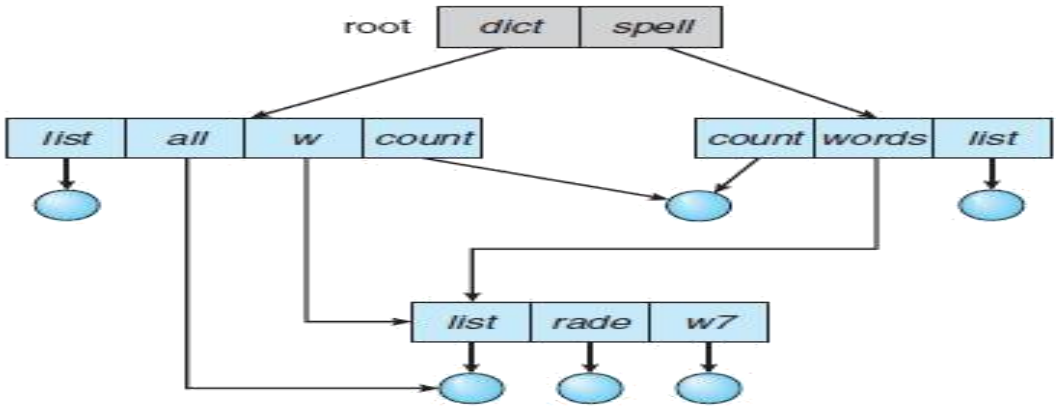
All the files the user wants to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory.

SHARED FILE IMPLEMENTATION:

Shared files and subdirectories can be implemented in several ways.

The first way is to create a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory.
When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information.
We **resolve** the link by using that path name to locate the real file.
Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories.

A major problem with duplicate directory entries is maintaining consistency when a file is mod



FILE DELETION:

One method is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file.

The second method is the deletion of a link. It does not affect the original file; only the link is removed.

Another approach to deletion is to preserve the file until all references to it are deleted.

We could keep a list of all references to a file.

When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

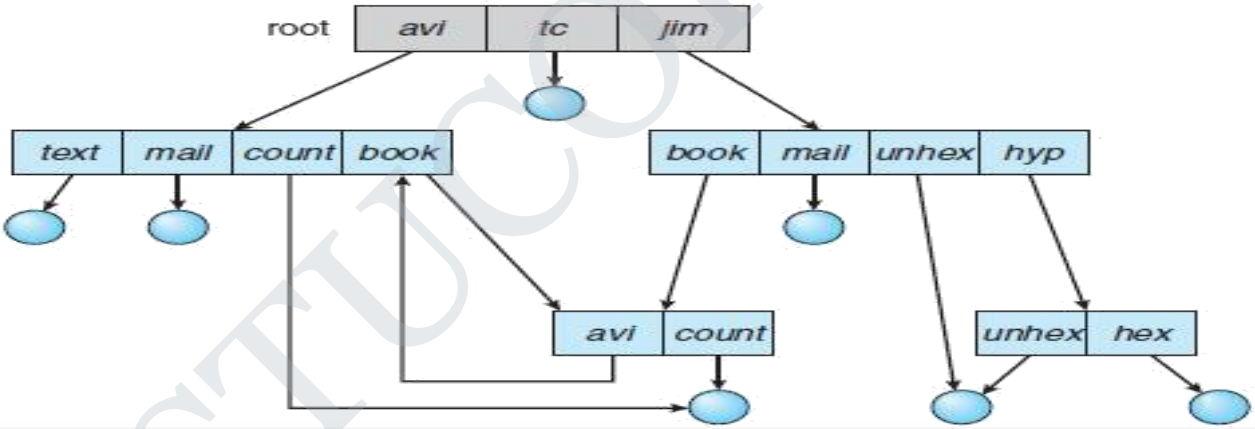
General Graph Directory:

If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results.

Adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature.

When we add links, the tree structure is destroyed, resulting in a simple graph structure.

If cycles are allowed to exist in the directory, we want to avoid searching any component twice, to improve the performance.



With acyclic-graph directory structures, a value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted.

However, when cycles exist, the reference count may not be 0 even when it is no longer possible to refer to a directory or file. In this case we use Garbage collection.

Garbage collection:

This scheme is used to determine when the last reference has been deleted and the disk space can be reallocated.

Garbage collection involves traversing the entire file system, marking everything that can be accessed.
Then, a second pass collects everything that is not marked onto a list of free space.

FILE SHARING AND PROTECTION:

File Sharing:

File sharing is very important for users who want to cooperate their files with each other and to reduce the effort required to achieve a computing goal.

Multiple users share files. When multiple users are allowed to share files, then there is a need to extend sharing to multiple file systems, including remote file systems.

File sharing includes

- Multiple users
- Remote File Systems
  - Client server model
  - Distributed Information systems
  - Failure Modes
- Consistency semantics
  - Unix Semantics
  - Session Semantics
  - Immutable Shared File Semantics

Multiple Users:

The system with multiple users can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.

To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system.

The systems uses the concepts of file **owner** (or **user**) and **group for File sharing.**

The owner is the user who can change attributes and grant access and who has the most control over the file.

The group attribute defines a subset of users who can share access to the file.

The owner and group IDs of a given file are stored with the other file attributes.

When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file.

If he is not the owner of the file, the group IDs can be compared. The result indicates which permissions are applicable.

The system then applies those permissions to the requested operation and allows or denies it.

Remote File Systems:

Networking allows the sharing of resources across a campus or even around the world.

The first implemented method for remote file systems involves manually transferring files between machines via programs like FTP.

The second major method uses a **distributed file system (DFS)** in which remote directories is visible from a local machine.

The third method is the **World Wide Web** where the browser is needed to gain access to the remote files.

Client Server Model:

Remote file systems allow a computer to mount one or more file systems from one or more remote machines.

The machine containing the files is the **server**, and the machine seeking access to the files is the **client.**

The server declares that a resource is available to clients and specifies exactly which resource is shared by which clients.

A server can serve multiple clients, and a client can use multiple servers.

A client can be specified by a network name or other identifier, such as an IP address but these can be **spoofed**, or imitated.

As a result of spoofing, an unauthorized client could be allowed access to the server.

In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information.

The user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files.

Distributed Information Systems

Distributed information systems, also known as **distributed naming services**, provide unified access to the information needed for remote computing.

The **domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet.

Distributed information systems provide **user name/password/user ID/group ID** space for a distributed facility.

In the case of Microsoft's **common Internet file system (CIFS)**, network information is used in conjunction with user authentication to create a network login that the server uses to decide whether to allow or deny access to a requested file system.

Microsoft uses **active directory** as a distributed naming structure to provide a single name space for users. Once established, the distributed naming facility is used by all clients and servers to authenticate users.

LDAP: **lightweight directory-access protocol (LDAP)** is a secure distributed naming mechanism.

Failure Modes:

Local file systems can fail for a variety of reasons that includes

- Failure of the disk containing the file system,
- Corruption of the directory structure or other disk-management information
- Disk-controller failure,
- Cable failure,
- Host-adapter failure
- User or system-administrator failure

Remote file systems have more failure modes because of the complexity of network systems and the required interactions between remote machines.

The failure semantics are defined and implemented as part of the remote-file-system protocol.

Termination of all operations can result in users' losing data.

To implement the recovery from failure, some kind of **state information** may be maintained on both the client and the server.

If both server and client maintain knowledge of their current activities and open files, then they can recover from a failure.

Consistency Semantics:

Consistency semantics represent a criterion for evaluating any file system that supports file sharing.

The semantics specify how multiple users of a system are to access a shared file simultaneously.

They specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

Consistency semantics are directly related to the process synchronization algorithms.

A series of file accesses attempted by a user to the same file is always enclosed between the open() and close() operations.

The series of accesses between the open() and close() operations makes up a **file session**

The Examples Of Consistency semantics includes

- Unix Semantics
- Session Semantics
- Immutable shared File Semantics

i) Unix Semantics:

The UNIX file system uses the following consistency semantics.

- Writes to an open file by a user are visible immediately to other users who have this file open.
- One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users

ii) Session Semantics:

The Andrew file system uses the following consistency semantics:

Writes to an open file by a user are not visible immediately to other users that have the same file open.

Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

Immutable Shared File Semantics:

The Immutable Shared file system uses the following consistency semantics

Once a file is declared as shared by its creator, it cannot be modified.

An immutable file has two key properties: its name may not be reused, and its contents may not be altered

An immutable file signifies that the contents of the file are fixed.

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested.

File Protection is also defined as the process of protecting the file of a user from unauthorized access or any other physical damage.

Goals of Protection:

To prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.

To ensure that each shared resource is used only in accordance with system policies, which may be set either by system designers or by system administrators.

To ensure that errant programs cause the minimal amount of damage possible.

Types of Access:

The need to protect files is a direct result of the ability to access files.

Systems that do not permit access to the files of other users do not need protection Several different types of operations may be controlled:

Read. Read from the file.

Write. Write or rewrite the file.

Execute. Load the file into memory and execute it.

Append. Write new information at the end of the file.

Delete. Delete the file and free its space for possible reuse.

List. List the name and attributes of the file.

These higher-level functions may be implemented by a system program that makes lower-level system calls.

Access control:

The most common approach to the protection problem is to make access dependent on the identity of the user.

Different users may need different types of access to a file or directory.

The general scheme to implement identity dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user.

When a user requests access to a particular file, the operating system checks the access list associated with that file.

If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

The main problem with access lists is their length.

This technique has two undesirable consequences:

Constructing such a list may be a tedious and unrewarding task, especially if we do not know in advance the list of users in the system.

The directory entry, previously of fixed size, now must be of variable size, resulting in more complicated space management

To reduce the length of the access-control list, many systems recognize three classifications of users in connection with each file:

Owner. The user who created the file is the owner.

Group. A set of users who are sharing the file and need similar access

Universe. All other users in the system constitute the universe

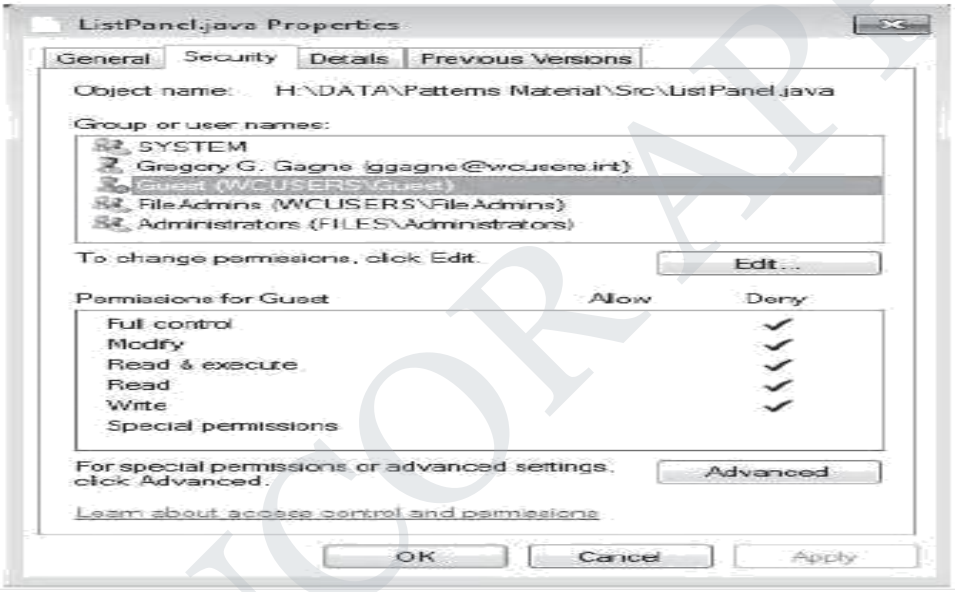
EXAMPLE: consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named book.tex

The protection associated with this file is as follows:

Sara should be able to invoke all operations on the file.
Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
All other users should be able to read, but not write, the file.

In the UNIX system, groups can be created and modified only by the manager of the facility
Three fields are needed to define protection. Each field is a collection of bits, and each bit either allows or prevents the access associated with it.
The UNIX system defines three fields of 3 bits each—rwx where r controls read access, w controls write access, and x controls execution.
A separate field is kept for the file owner, for the file's group, and for all other users.
In this scheme, 9 bits per file are needed to record protection information.
Thus, for our example, the protection fields for the file book.tex are as follows

For the owner Sara, all bits are set;
For the group text, the r and w bits are set;
For the universe, only the r bit is set.



Windows users typically manage access-control lists via the Graphical User Interface .

Password Protection:

Another approach to the protection problem is to associate a password with each file.
Access to each file can be controlled with the help of passwords.

If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file.

Disadvantages: The number of passwords that a user needs to remember may become large.

If only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis.
In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories;

We need to provide a mechanism for directory protection. The directory operations that must be protected are somewhat different from the file operations. We want to control the creation and deletion of files in a directory.

FILE SYSTEM STRUCTURE:

The file system provides the mechanism for on-line storage and access to file contents, including data and programs.

The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently.

Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.

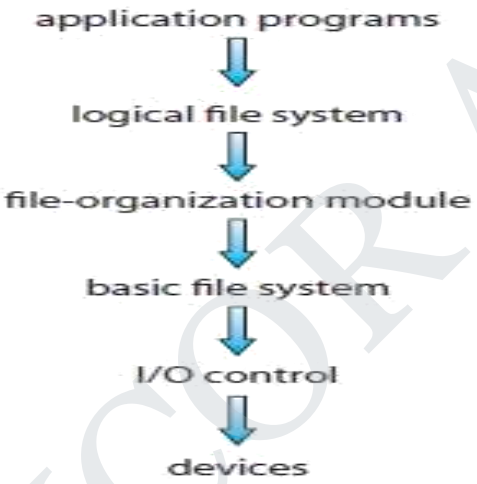
A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read–write heads and waiting for the disk to rotate.

A file system poses two quite different design problems.

The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure files.

The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels .



I/O Control:

The **I/O control** level consists of device drivers and interrupts handlers to transfer information between the main memory and the disk system.

The device driver usually writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take.

Basic File System:

The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

Each physical block is identified by its numeric disk address.

This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks

A block in the buffer is allocated before the transfer of a disk block can occur

Caches are used to hold frequently used file-system metadata to improve performance

File Organization Module:

The **file-organization module** knows about files and their logical blocks, as well as physical blocks.
The file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Logical File System:

The **logical file system** manages metadata information
The logical file system manages the directory structure to provide the file-organization module with the information it needs.
It maintains file structure via file-control blocks
A **file control block (FCB)** (an **inode** in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents.

Advantages of Layered File system :

When a layered structure is used for file-system implementation, duplication of code is minimized
Each file system can then have its own logical file-system and file-organization modules

Disadvantages: The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

EXAMPLE FILE SYSTEMS:

UNIX uses the **UNIX file system (UFS)**, which is based on the Berkeley Fast File System (FFS).
Windows supports disk file-system formats of FAT, FAT32, and NTFS (or Windows NT File System), as well as CD-ROM and DVD file-system formats.
Although Linux supports over forty different file systems, the standard Linux file system is known as the **extended file system**, with the most common versions being ext3 and ext4.

On-disk and in-memory structures are used to implement a file system.
These structures vary depending on the operating system and the file system
The On Disk Structure of File system Provides the details such as

- Boot Control Block
- Volume Control Block
- Directory Structure
- File Control Block

A **boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume.
If the disk does not contain an operating system, this block can be empty.
In UFS (Unix File System), it is called the **boot block**. In NTFS (New Technology File System), it is the **partition boot sector**.

Volume Control Block:

volume control block (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers.
In UFS, this is called a **superblock**. In NTFS, it is stored in the **master file table**.

Directory Structure:

A directory structure (per file system) is used to organize the files.
In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.

File Control Block:

A per-file FCB contains many details about the file.
It has a unique identifier number to allow association with a directory entry.
In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

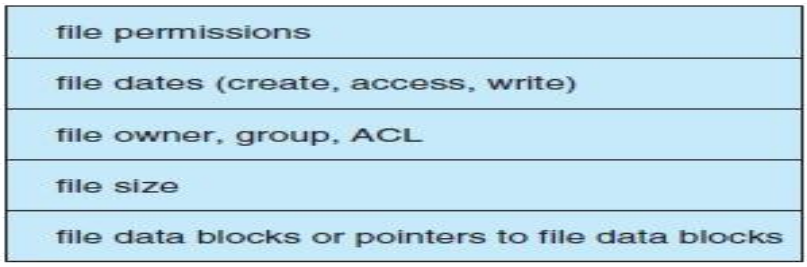


Figure 12.2 A typical file-control block.

The in memory Structure of File system provides the details such as

- An in-memory mount table
- An in-memory directory-structure cache
- The system-wide open-file table
- The per-process open-file table
- Buffers hold file-system blocks

An in-memory mount table contains information about each mounted volume.
An in-memory directory-structure cache holds the directory information of recently accessed directories.
The system-wide open-file table contains a copy of the FCB of each open file, as well as other information
The per-process open-file table contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
Buffers hold file-system blocks when they are being read from disk or written to disk.

To create a new file, an application program calls the logical file system. To create a new file, it allocates a new FCB.

The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk.

Example: A file has been created, and it can be used for I/O.It must be opened to perform I/O.

The open() call passes a file name to the logical file system. The open () system call first searches the system-wide open-file table to see if the file is already in use by another process.
If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
If the file is not already open, the directory structure is searched for the given file name.
Once the file is found, the FCB is copied into a system-wide open-file table in memory.
Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.

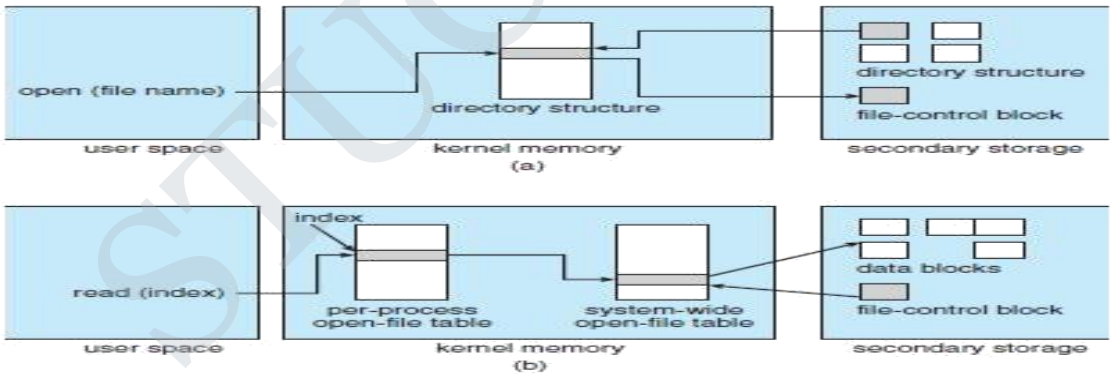
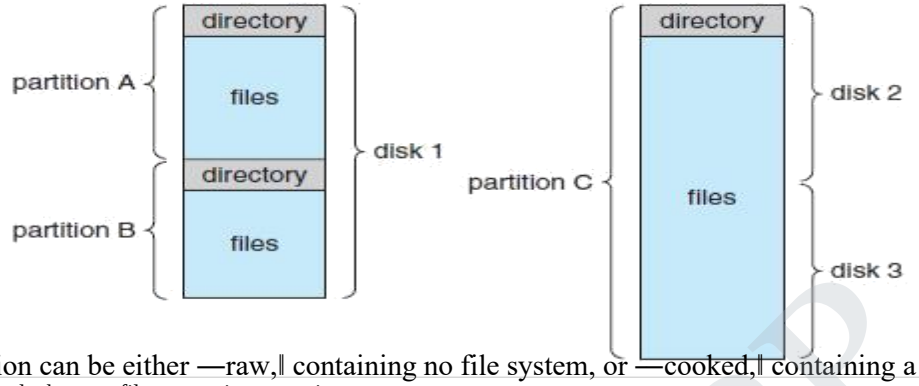


Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented.

PARTITIONING AND MOUNTING:

A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks



Each partition can be either —raw,‖ containing no file system, or —cooked,‖ containing a file system.

Raw disk is used where no file system is appropriate.

Boot information can be stored in a separate partition,

The **boot loader** in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing.

Many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system.

The **root partition**, which contains the operating-system kernel and sometimes other system files, is mounted at boot time.

Microsoft Windows–based systems mount each volume in a separate name space, denoted by a letter and a colon.

VIRTUAL FILE SYSTEMS:

Modern operating systems must concurrently support multiple types of file systems. Virtual File Systems allow multiple types of file systems to be integrated into a directory structure.

Implementing multiple types of file systems requires writing directory and file routines for each type.

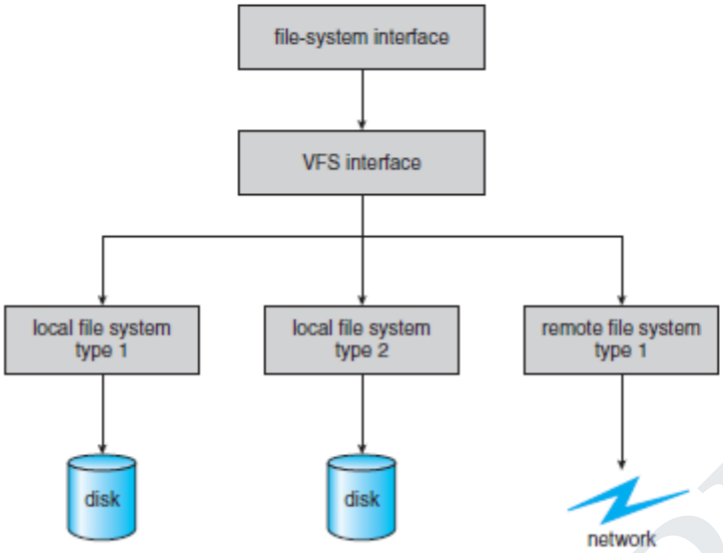
Users can access files contained within multiple file systems on the local disk or even on file systems available across the network.

Thus, the file-system implementation consists of three major layers

File System Interface

Virtual File System

Remote File System Protocol



The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors

The second layer is called the **virtual file system (VFS)** layer. The VFS layer serves two important functions:

It separates file-system-generic operations from their implementation by defining a clean VFS interface.

It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a **vnode** that contains a numerical designator for a network-wide unique file.

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.

The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.

EXAMPLE: VFS architecture in Linux

The four main object types defined by the Linux VFS are:

The **inode object**, which represents an individual file

The **file object**, which represents an open file

The **superblock object**, which represents an entire file system

The **dentry object**, which represents an individual directory entry

For each of these four object types, the VFS defines a set of operations that may be implemented.

Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that particular object.

int open(. . .)—Open a file.

int close(...)—Close an already-open file.

ssize t read(. . .)—Read from a file.

ssize t write(. . .)—Write to a file.

int mmap(. . .)—Memory-map a file.

Thus, the VFS software layer can perform an operation on one of these objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with.

FILE / DISK ALLOCATION TECHNIQUES:

Many files are stored on the same disk. The File System allocates space to these files so that disk space is utilized effectively and files can be accessed quickly.

Three major methods of allocating disk space are,

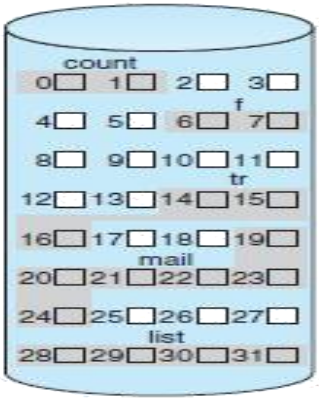
Contiguous Allocation

Linked Allocation

Indexed Allocation

Contiguous Allocation:

Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk.
Disk addresses define a linear ordering on the disk.
It supports both direct and sequential access. For direct access to block i of a file that starts at block b, we can immediately access block b + i.
If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2... b + n − 1.
The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Figure 12.5 Contiguous allocation of disk space.

Advantages:

The number of disk seeks required for accessing contiguously allocated files is minimal.
Contiguous allocation of a file is defined by the disk address and length of the first block.
Accessing a file that has been allocated contiguously is easy.
One difficulty is finding space for a new file.
Contiguous memory allocation suffers from the problem of external fragmentation.

EXTERNAL FRAGMENTATION: As files are allocated and deleted, the free disk space is broken into little pieces. The total available space may not be enough to satisfy a request. Storage is fragmented into a number of holes, none of which is large enough to store the data.

SOLUTION: One strategy for preventing external fragmentation is to copy an entire file system onto another disk. The original disk is then freed completely, creating one large contiguous free space .We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This is called as **Compaction.**

Another Problem with contiguous allocation is determining how much space is needed for a file.
Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient.
To minimize these drawbacks, some operating systems use a contiguous chunk of space that is allocated initially.
If that amount proves not to be large enough, another chunk of contiguous space, known as an **extent**, is added.

Linked allocation:

Linked allocation solves all problems of contiguous allocation
With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
The directory contains a pointer to the first and last blocks of the file. If each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.
Each directory entry has a pointer to the first disk block of the file.

A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.
To read a file, we simply read blocks by following the pointers from block to block.

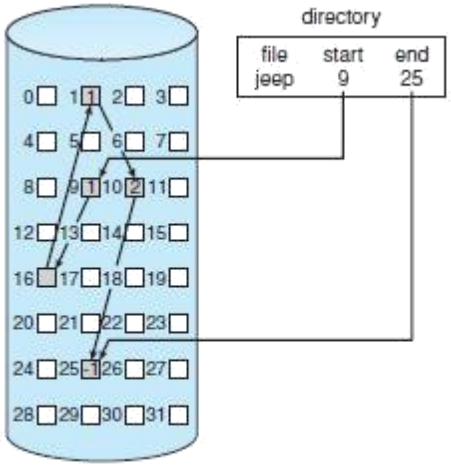


Figure 12.6 Linked allocation of disk space.

Advantages:

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.
The size of a file need not be declared when the file is created.
A file can continue to grow as long as free blocks are available

It is inefficient to support a direct-access capability for linked-allocation files.
It requires more disk space for storing the pointers.
Solution: The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks.
An important variation on linked allocation is the use of a **file-allocation table (FAT)**.
A section of disk at the beginning of each volume contains the table.
The table has one entry for each disk block and is indexed by block number.
The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file.
The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block.

Indexed Allocation

In Indexed allocation each file has its own index block, which is an array of disk-block addresses. The directory contains the address of the index block
The i_{th} entry in the index block points to the i_{th} block of the file.
To find and read the i_{th} block, we use the pointer in the i_{th} index-block entry.
When the file is created, all pointers in the index block are set to null.
When the i_{th} block is first written, a block is obtained from the free-space manager, and its address is put in

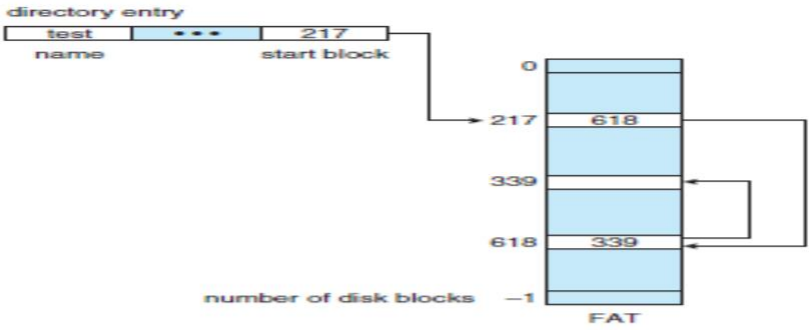


Figure 12.7 File-allocation table.

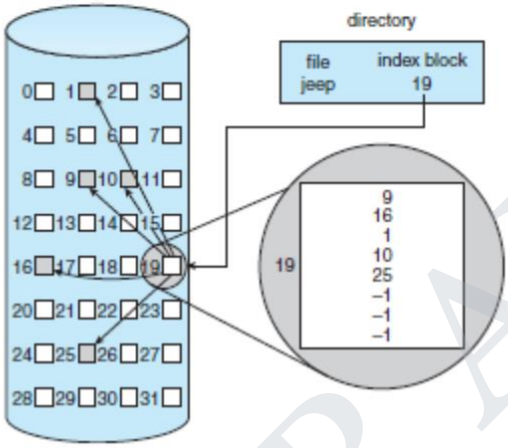


Figure 12.8 Indexed allocation of disk space.

Advantages:

Indexed allocation supports direct access

It does not suffer from External fragmentation. because any free block on the disk can satisfy a request for more Space

Indexed allocation does suffer from wasted space

The Pointer overhead of the index block is generally greater than the pointer overhead of linked allocation

Every file must have an index block, so we want the index block to be as small as possible.

Mechanisms for implementing the index block includes:

Linked scheme.

Multilevel index

Combined scheme.

An index block is normally one disk block. Thus, it can be read and written directly by itself.

To allow for large files, we can link together several index blocks.

Example: An index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses.

The next address is a pointer to another index block.

Multilevel index representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks.

To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block.

This approach could be continued to a third or fourth level, depending on the desired maximum file size.

Combined scheme:

Combined scheme is a combination of both linked and multilevel implementation.

EXAMPLE: Consider a UNIX-based file systems, which keeps 15 pointers of the index block in the file's inode(FAT).

The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file.

The next three pointers point to **indirect blocks**.

The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data.

The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.

The last pointer contains the address of a **triple indirect block**.

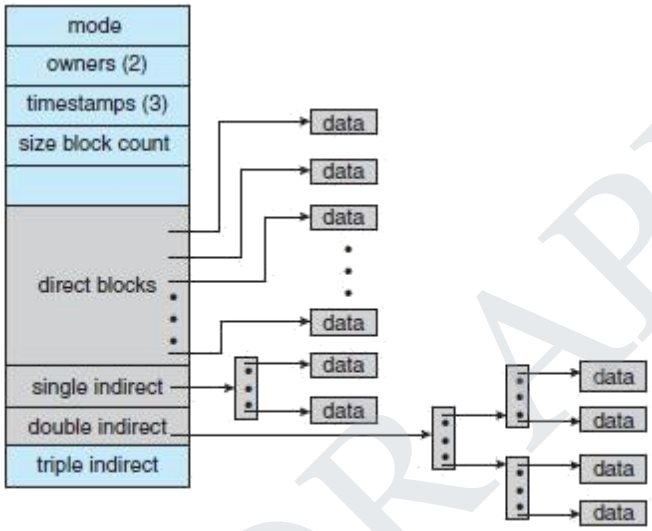


Figure 12.9 The UNIX inode.

FREE SPACE MANAGEMENT:

A number of files can be created and deleted by the user in a secondary storage device.

Since disk space is limited, we need to reuse the space from deleted files for new files.

FREE SPACE LIST:

To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all free disk blocks—those not allocated to some file or directory.

To create a file, we search the free-space list for the required amount of space and allocate that space to the new file.

This space is then removed from the free-space list.

When a file is deleted, its disk space is added to the free-space list.

The Free space list can be implemented in the following ways

- Bit Vector or Bit Map
- Linked list
- Groping
- Counting
- Space maps

Bit Vector:

The free-space list is implemented as a **bit map** or **bit vector**.

Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

Example: Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17,

18, 25, 26, and 27 are free and the rest of the blocks are allocated.

Free-space bit map:
01111001111110001100000011100000...

One technique for finding the first free block on a system that uses a bit-vector is to sequentially check each word in the bit map to see whether that value is not 0.

The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is **(number of bits per word) × (number of 0-value words) + offset of first 1 bit.**

Advantages:

The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.

Bit vectors are inefficient unless the entire vector is kept in main memory.

If the disk size constantly increases, the problem with bit vectors will continue to increase.

Linked list:

Linked list implementation link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.

This first block contains a pointer to the next free disk block, and so on

Example: Blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 in the disk were free and the rest of the blocks were allocated.

We would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on

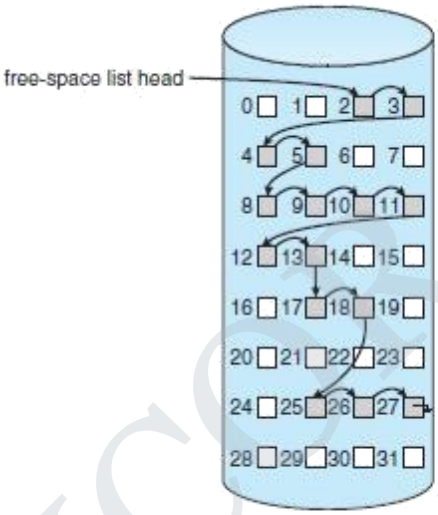


Figure 12.10 Linked free-space list on disk.

Disadvantages:

This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

Grouping:

A modification of the free-list approach stores the addresses of n free blocks in the first free block.

The first n−1 of these blocks are actually free. The last block contains the addresses of other n free blocks, and so on.

The addresses of a large number of free blocks can now be found quickly.

Counting:

Free space list keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block.

Each entry in the free-space list then consists of a disk address and a count.

This method of tracking free space is similar to the extent method of allocating blocks.

These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

Space maps:

Oracle's **ZFS** file system creates **Meta slabs** to divide the space on the device into chunks of manageable size.
A given volume may contain hundreds of Meta slabs. Each Meta slab has an associated space map.
ZFS uses the counting algorithm to store information about free blocks.
A space map uses log-structured file-system techniques to record the information about the free blocks.
The space map is a log of all block activity (allocating and freeing), in time order, in counting format.
When ZFS decides to allocate or free space from a Meta slab, it loads the associated space map into memory in a balanced-tree structure, indexed by offset, and replays the log into that structure.
The in-memory space map is then an accurate representation of the allocated and free space in the Meta slab.

I/O SYSTEMS:

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. This is met by a combination of hardware device controllers and software device driver techniques**.**

Computers operate a great many kinds of devices. These devices are grouped under various categories that includes

Storage devices – Disks and Tapes

Transmission devices – Networks cards, modems, Bluetooth

Human Interface devices – Monitor, Keyboard, Mouse, Printer.

A device communicates with a computer system by sending signals over a cable.
PORT: The device communicates with the machine via a connection point called a port.
BUS: If devices share a common set of wires, the connection is called a bus. A bus is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires.
DAISY CHAIN: When device A has a cable that plugs into device B, and device B has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a daisy chain. A daisy chain usually operates as a bus.

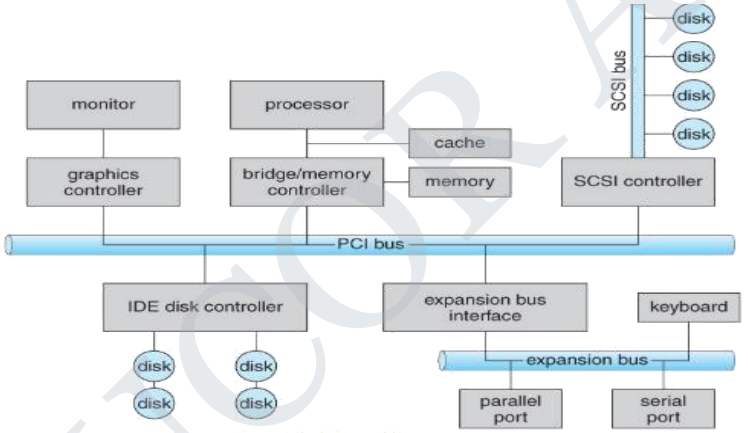


Figure 13.1 - A typical PC bus structure.

PCI – Peripheral Component Interconnect

SCSI – Small Computer Systems Interface

IDE – Integrated Drive Electronics

A PCI bus (the common PC system bus) connects the processor–memory subsystem to fast devices
An expansion bus connects relatively slow devices, such as the keyboard and serial and USB ports.
A Device controller is a collection of electronics that can operate a port, a bus, or a device.
The processor gives commands and data to a controller to accomplish an I/O transfer.
The controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers.

MEMORY MAPPED I/O: The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers at their mapped locations in physical memory.

An I/O port typically consists of four registers, called the status, control, data-in, and data-out registers.

The data-in register is read by the host to get input.

The data-out register is written by the host to send output.

The status register contains that indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.

The control register can be written by the host to start a command or to change the mode of a device.

POLLING:

HANDSHAKING: It is a protocol for interaction between the host and a controller.

The controller indicates its state through the busy bit in the status register.

The controller sets the busy bit when it is busy working and clears the busy bit when it is ready to accept the next command.

The host signals its wishes via the command-ready bit in the command register.

The host sets the command-ready bit when a command is available for the controller to execute.

POLLING: The host repeatedly reads the busy bit until that bit becomes clear. This process is called as polling or busy waiting.

The host sets the write bit in the command register and writes a byte into the data-out register.

The host sets the command-ready bit.

When the controller notices that the command-ready bit is set, it sets the busy bit.

The controller reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device.

The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

INTERUPPTS:

An interrupt is a signal to the kernel (i.e., the core of the operating system) that an event has occurred, and this result in changes in the sequence of instructions that is executed by the CPU.

The CPU hardware has a wire called the **interrupt-request line** that the CPU senses after executing every instruction.

When the CPU detects that a controller has asserted a signal on the interrupt-request line, the CPU performs a state save and jumps to the **interrupt-handler routine** at a fixed address in memory.

The device controller **raises** an interrupt by asserting a signal on the interrupt request line, the CPU **catches** the interrupt and **dispatches** it to the interrupt handler, and the handler **clears** the interrupt by servicing the device.

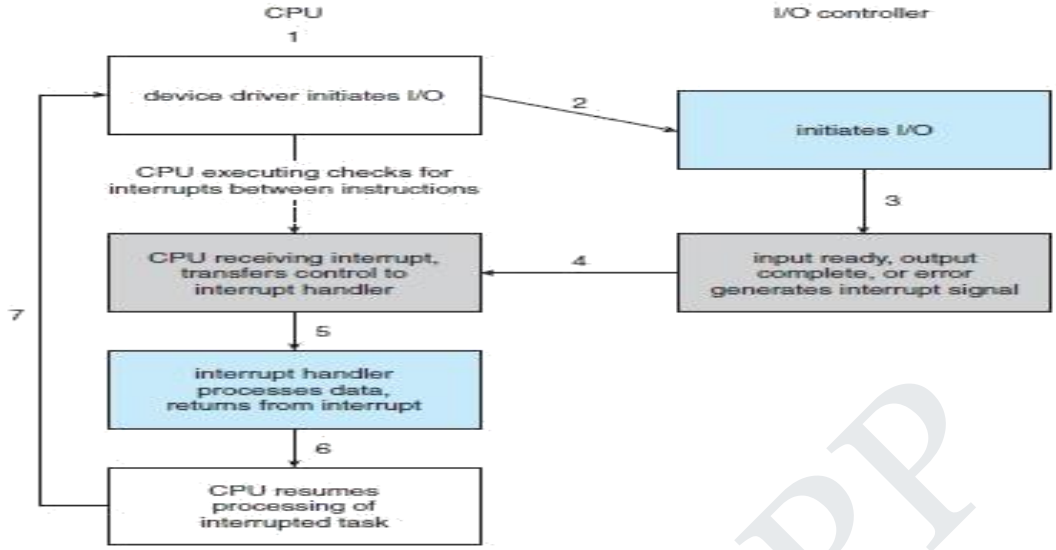


Figure 13.3 Interrupt-driven I/O cycle.

In a modern operating system, we need more sophisticated interrupt-handling features

We need the ability to suspend interrupt handling during critical processing.

We need an efficient way to dispatch to the proper interrupt handler for a device without first polling all the devices to see which one raised the interrupt.

We need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts and can respond with the appropriate degree of urgency.

Most CPUs have two interrupt request lines.

Non Maskable Interrupt

Maskable Interrupt

The Non maskable interrupts are those that cannot be ignored by the processor. It is reserved for events such as unrecoverable memory errors.

The Maskable interrupts are those that can be ignored by the processor. The maskable interrupt is used by device controllers to request service.

Interrupt vector: The interrupt mechanism accepts an **address**—a number that selects a specific interrupt-handling routine from a small set. This address is an offset in a table called the **interrupt vector**.

The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.

Interrupt Chaining: A common way to solve this problem is to use **interrupt chaining**, in which each element in the interrupt vector points to the head of a list of interrupt handlers.

The interrupt mechanism also implements a system of **interrupt priority levels**.

These levels enable the CPU to suspend the handling of low-priority interrupts without masking all interrupts and makes it possible for a high priority interrupt to preempt the execution of a low-priority interrupt.

The interrupt mechanism is also used to handle a wide variety of **exceptions**, such as dividing by 0, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode.

EXAMPLE: A page fault is an exception that raises an interrupt. The interrupt suspends the current process and jumps to the page-fault handler in the kernel.

This handler saves the state of the process, moves the process to the wait queue, performs page-cache management, schedules an I/O operation to fetch the page, schedules another process to resume execution, and then returns from the interrupt.

Interrupts can also be used to manage the flow of control within the kernel.

Thus interrupts are used throughout modern operating systems to handle asynchronous events and to trap to supervisor-mode routines in the kernel.

DIRECT MEMORY ACCESS:

In Programmed I/O, when the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. The I/O module performs the requested action and takes no action to alert the processor and it does not interrupt the processor. The processor periodically checks the status of the I/O module until it finds that the operation is complete. This burdens the CPU.

Many computers avoid burdening the main CPU with Programmed I/O by offloading some of this work to a special-purpose processor called a **direct-memory-access (DMA)** controller.

To initiate a DMA transfer, the host writes a DMA command block into memory.

This block contains a pointer to

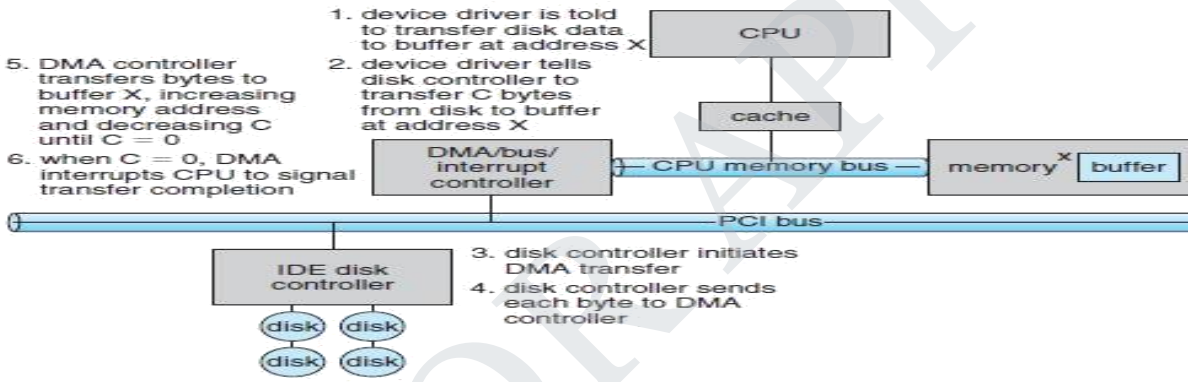
The source of a transfer,

A pointer to the destination of the transfer,

A count of the number of bytes to be transferred.

The CPU writes the address of this command block to the DMA controller, then goes on with other work.

The DMA controller proceeds to operate the memory bus directly, placing addresses on the bus to perform transfers without the help of the main CPU.



Handshaking between the DMA controller and the device controller:

Handshaking between the DMA controller and the device controller is performed via a pair of wires called

DMA-request and **DMA-acknowledge**.

The device controller places a signal on the DMA-request wire when a word of data is available for transfer.

When the device controller receives the DMA-acknowledge signal, it transfers the word of data to memory and removes the DMA-request signal.

When the entire transfer is finished, the DMA controller interrupts the CPU.

CYCLE STEALING: When the DMA controller seizes the memory bus, the CPU is prevented from accessing main memory, although it can still access data items in its primary and secondary caches.This is called as Cycle stealing.

This **cycle stealing** can slow down the CPU computation, offloading the data-transfer work to a DMA controller generally improves the total system performance.

DIRECT VIRTUAL MEMORY ADDRESS: Some computer architectures use physical memory addresses for DMA, but others perform **direct virtual memory access (DVMA)**, using virtual addresses that undergo translation to physical addresses.

DVMA can perform a transfer between two memory-mapped devices without the intervention of the CPU or the use of main memory.

On protected-mode kernels, the operating system generally prevents processes from issuing device commands directly.

This direct access can be used to achieve high performance, since it can avoid kernel communication, context switches, and layers of kernel software.

APPLICATION I/O INTERFACE:

The operating system acts as an interface between the application programs and the I/O Devices.
The interface involves abstraction, encapsulation, and software layering.
The kernel modules called device drivers are internally custom-tailored to specific devices.
The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel
The kernels allows the existing I/O Subsystem for new devices to be compatible with an existing host controller interface or they write device drivers to interface the new hardware to popular operating systems.

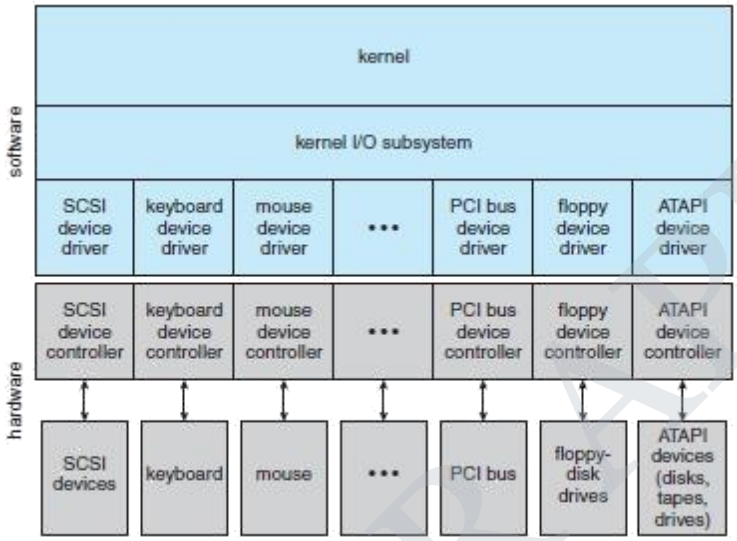


Figure 13.6 A kernel I/O structure.

Devices vary on many dimensions such as

- Character-stream or block
- Sequential or random access
- Synchronous or asynchronous
- Sharable or dedicated
- Speed of operation
- Read–write, read only, or write only

Character-stream or block

Character-stream or block: A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.

Sequential or random access: A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

Synchronous or asynchronous. A synchronous device performs data transfers with predictable response times and an asynchronous device exhibits irregular or unpredictable response times.

Sharable or dedicated. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

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

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

For the application programs, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types.

Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer.
Some operating systems provide a set of system calls for graphical display, video, and audio devices.
ESCAPE: Most operating systems also have an **escape** (or **back door**) that transparently passes arbitrary commands from an application to a device driver.

Block and Character Devices:

The **block-device interface** captures all the aspects necessary for accessing disk drives and other block-oriented devices.

The device is expected to understand commands such as read () and write (). If it is a random-access device, it is also expected to have a seek () command to specify which block to transfer next.
The read (), write (), and seek () system calls are the essential behaviors of block-storage devices, so that applications are insulated from the low-level differences among those devices.
RAW I/O: The operating system itself, as well as special applications such as database management

Systems may prefer to access a block device as a simple linear array of blocks. This mode of access is called **raw I/O**.

DIRECT I/O: operating system allows a mode of operation on a file that disables buffering and locking. In the UNIX, this is called **direct I/O**.
A keyboard is an example of a device that is accessed through a **character stream interface**. The basic system calls in this interface enable an application to get () or put () one character.

Network devices:

Most operating systems provide a network I/O interface that is different from the read()–write()–seek() interface used for disks, Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O.

One interface available in many operating systems, including UNIX and Windows, is the network **socket** interface.

Many other approaches to interprocess communication and network communication have also been implemented
EXAMPLE: Windows provides one interface to the network interface card and a second interface to the network protocols.

Clocks and Timers:

Most computers have hardware clocks and timers that provide three basic functions:

Give the current time.
Give the elapsed time.
Set a timer to trigger operation X at time T.

PROGRAMMABLE INTERVAL TIMER: The hardware to measure elapsed time and to trigger operations is called a programmable interval timer.
It can be set to wait a certain amount of time and then generate an interrupt, and it can be set to do this once or to repeat the process to generate periodic interrupts.

EXAMPLE: The scheduler uses this mechanism to generate an interrupt that will preempt a process at the end of its time slice.

Nonblocking and Asynchronous I/O:

When an application issues a **blocking** system call, the execution of the application is suspended.
The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution.
The physical actions performed by I/O devices are generally asynchronous.
Some user-level processes need nonblocking I/O. One example is a user interface that receives keyboard and mouse input while processing and displaying data on the screen.

An alternative to a nonblocking system call is an asynchronous system call. An asynchronous call returns immediately, without waiting for the I/O to complete.

The difference between nonblocking and asynchronous system calls is that a nonblocking read() returns immediately with whatever data are available and an asynchronous read() call requests a transfer that will be performed in its entirety but will complete at some future time.

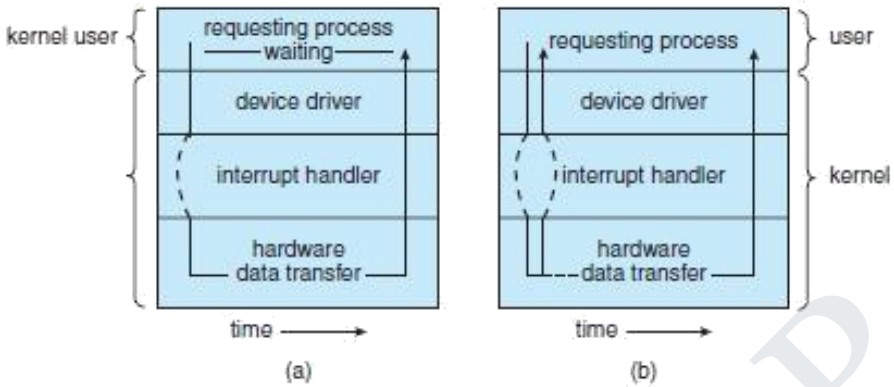


Figure 13.8 Two I/O methods: (a) synchronous and (b) asynchronous.

Vectored I/O:

Vectored I/O allows one system call to perform multiple I/O operations involving multiple locations.

EXAMPLE:

SCATTER – GATHER METHOD: A system call in UNIX accepts a vector of multiple buffers and either reads from a source to that vector or writes from that vector to a destination**.** The same transfer could be caused by several individual invocations of system calls.

KERNEL I/O SUBSYSTEM:

Kernels provide many services related to I/O. The I/O subsystem is also responsible for protecting itself from errant processes and malicious users. The Services include

I/O Scheduling
Buffering
Caching
Spooling and device reservation
Error Handling
I/O Protection
Kernel Data Structures

I/O Scheduling:

Scheduling can improve overall system performance, can share device access among processes, and can reduce the average waiting time for I/O to complete.

EXAMPLE: A disk arm is near the beginning of a disk and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in the order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling.

The OS must be able to keep track of many I/O requests at the same time. For this purpose, the operating system might attach the wait queue to a device-status table.

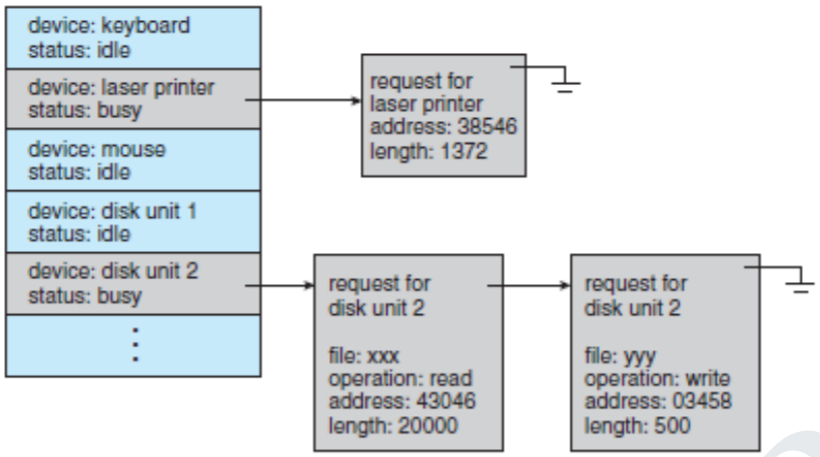


Figure 13.9 Device-status table.

Buffering:

A buffer, of course, is a memory area that stores data being transferred between two devices or between a device and an application

Buffering is done for three reasons.

One reason is to cope with a speed mismatch between the producer and consumer of a data stream

A second use of buffering is to provide adaptations for devices that have different data-transfer sizes

COPY SEMANTICS: A third use of buffering is to support copy semantics for application I/O. With copy semantics, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer.

EXAMPLE: Consider a file that is being received via modem for storage on the hard disk. A buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation.

DOUBLE BUFFERING: The modem still needs a place to store additional incoming data, two buffers are used. After the modem fills the first buffer, the disk write is requested. The modem then starts to fill the second buffer while the first buffer is written to disk. By the time the modem has filled the second buffer, the disk write from the first one should have completed, so the modem can switch back to the first buffer while the disk writes the second one. This is called as double buffering.

Caching:

A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original

The Instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches.

The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, holds a copy on faster storage of an item that resides.

A **spool** is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Spooling is a way of operating systems to coordinate concurrent output

A printer can serve only one job at a time

Several applications may wish to print their output concurrently, without having their output mixed together.

The operating system solves this problem by intercepting all output to the printer.

Each application's output is spooled to a separate disk file.

When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer.
The spooling system copies the queued spool files to the printer one at a time.
The spool can be managed in two ways

System daemon process
In-kernel thread.

Some operating systems (including VMS) provide support for exclusive device access by enabling a process to allocate an idle device and to deallocate that device when it is no longer needed.

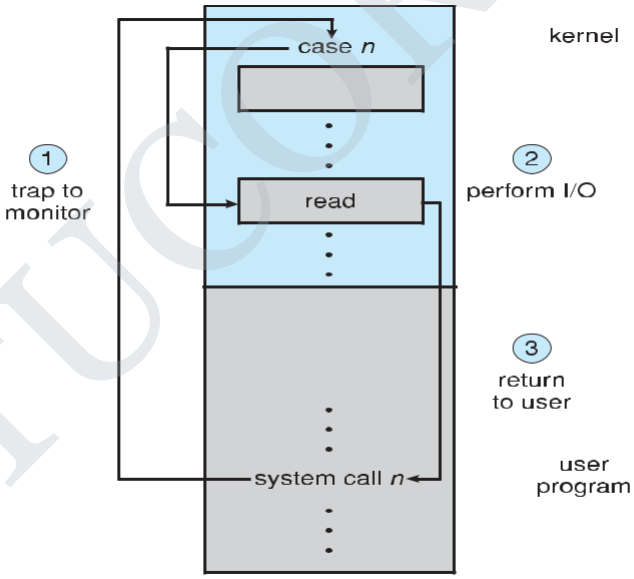
An operating system that uses protected memory can guard against many kinds of hardware and application errors.
Devices and I/O transfers can fail when a network becomes overloaded, or when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures.
A disk read() failure results in a read() retry, and a network send() error results in a resend(), if the protocol so specifies.

An I/O system call will return one bit of information about the status of the call, signifying either success or failure.
In the UNIX operating system, an additional integer variable named errno is used to return an error code indicating the general nature of the failure.

EXAMPLE: A failure of a SCSI device is reported by the SCSI protocol in three levels of detail:

Sense Key - Identifies the general nature of the failure
Additional sense code - states the category of failure
Additional sense-code qualifier -Specifies which command parameter was in error or which hardware subsystem failed its self-test.

A user process may attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions.
To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly.
To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf. The operating system, executing in monitor mode, checks that the request is valid and, if it is, does the I/O requested. The operating system then returns to the user.
Any memory-mapped and I/O port memory locations must be protected from user access by the memory-protection system.



Kernel Data Structures:

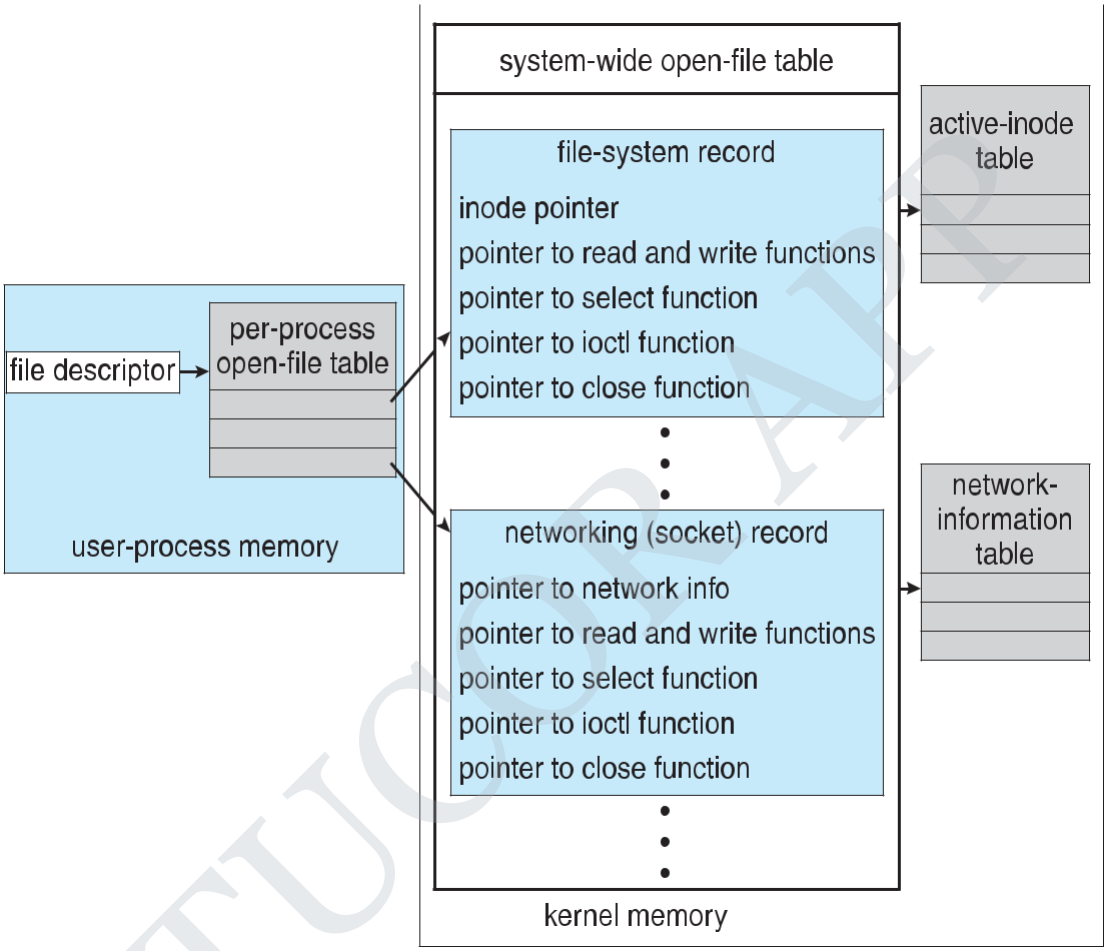
The kernel needs to keep state information about the use of I/O components which is done through a variety of in-kernel data structures such as open file table, per process open file table, etc..

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

To read a user file, the kernel needs to probe the buffer cache before deciding whether to perform a disk I/O.

To read a raw disk, the kernel needs to ensure that the request size is a multiple of the disk sector size and is aligned on a sector boundary.

An I/O request is converted into a message that is sent through the kernel to the I/O manager and then to the device driver, each of which may change the message contents. For output, the message contains the data to be written. For input, the message contains a buffer to receive the data



UNIT V CASE STUDY

Linux System - Design Principles, Kernel Modules, Process Management, Scheduling, Memory Management, Input-Output Management, File System, Inter-process Communication; Mobile OS - iOS and Android - Architecture and SDK Framework, Media Layer, Services Layer, Core OS Layer, File System.

Linux System

Linux kernel forms the core of the Linux project, but other components make up a complete Linux operating system. Whereas the Linux kernel is composed entirely of code written from scratch specifically for the Linux project, much of the supporting software that makes up the Linux system is not exclusive to Linux but is common to a number of UNIX-like operating systems.

In particular, Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.

This sharing of tools has worked in both directions.

The main system libraries of Linux were originated by the GNU project, but the Linux community greatly improved the libraries by addressing omissions, inefficiencies, and bugs. Other components, such as the **GNU C compiler (gcc)**, were already of sufficiently high quality to be used directly in Linux.

The network administration tools under Linux were derived from code first developed for 4.3 BSD, but more recent BSD derivatives, such as FreeBSD, have borrowed code from Linux in return. Examples of this sharing include the Intel floating-point-emulation math library and the PC sound-hardware device drivers.

The Linux system as a whole is maintained by a loose network of developers collaborating over the Internet, with small groups or individuals having responsibility for maintaining the integrity of specific components.

A small number of public Internet file-transfer-protocol (FTP) archive sites act as de facto standard repositories for these components.

The **File System Hierarchy Standard** document is also maintained by the Linux community as a means of ensuring compatibility across the various system components. This standard specifies the overall layout of a standard Linux file system; it determines under which directory names configuration files, libraries, system binaries, and run-time data files should be stored.

Linux distributions, include much more than just the basic Linux system. They typically include extra system-installation and management utilities, as well as precompiled and ready-to-install packages of many of the common UNIX tools, such as news servers, web browsers, text-processing and editing tools, and even games.

The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places.

One of the important contributions of modern distributions, however, is advanced package management.

Today's Linux distributions include a package-tracking database that allows packages to be installed, upgraded, or removed painlessly.

The SLS distribution, dating back to the early days of Linux, was the first collection of Linux packages that was recognizable as a complete distribution. Although it could be installed as a single entity, SLS lacked the package-management tools now expected of Linux distributions.

The **Slackware** distribution represented a great improvement in overall quality, even though it also had poor package management. In fact, it is still one of the most widely installed distributions in the Linux community.

Since Slackware's release, many commercial and noncommercial Linux distributions have become available.

Red Hat and **Debian** are particularly pop-ular distributions; the first comes from a commercial Linux support company and the second from the free-software Linux community.

Other commercially supported versions of Linux include distributions from **Canonical** and **SuSE**, and others too numerous to list here.

There are too many Linux distributions in circulation for us to list all of them here. The variety of distributions does not prevent Linux distributions from being compatible, however.

The RPM package file format is used, or at least understood, by the majority of distributions, and commercial applications distributed in this format can be installed and run on any distribution that can accept RPM files.

Linux Licensing

The Linux kernel is distributed under version 2.0 of the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation.

Linux is not public-domain software. **Public domain** implies that the authors have waived copyright rights in the software, but copyright rights in Linux code are still held by the code's various authors. Linux is ***free*** software, however, in the sense that people can copy it, modify it, use it in any manner they want, and give away (or sell) their own copies.

The main implication of Linux's licensing terms is that nobody using Linux, or creating a derivative of Linux (a legitimate exercise), can distribute the derivative without including the source code.

Software released under the GPL cannot be redistributed as a binary-only product. If you release software that includes any components covered by the GPL, then, under the GPL, you must

make source code available alongside any binary distributions. (This restriction does not prohibit making or even selling binary software distributions, as long as anybody who receives binaries is also given the opportunity to get the originating source code for a reasonable distribution charge.)

Design Principles

In its overall design, Linux resembles other traditional, nonmicrokernel UNIX implementations. It is a multiuser, preemptively multitasking system with a full set of UNIX-compatible tools.

Linux's file system adheres to traditional UNIX semantics, and the standard UNIX networking model is fully implemented.

The internal details of Linux's design have been influenced heavily by the history of this operating system's development.

Although Linux runs on a wide variety of platforms, it was originally developed exclusively on PC architecture. A great deal of that early devel-opment was carried out by individual enthusiasts rather than by well-funded development or research facilities, so from the start Linux attempted to squeeze as much functionality as possible from limited resources.

Today, Linux can run happily on a multiprocessor machine with many gigabytes of main memory and many terabytes of disk space, but it is still capable of operating usefully in under 16 MB of RAM.

As PCs became more powerful and as memory and hard disks became cheaper, the original, minimalist Linux kernels grew to implement more UNIX functionality.

Speed and efficiency are still important design goals, but much recent and current work on Linux has concentrated on a third major design goal: standardization. One of the prices paid for the diversity of UNIX implementations currently available is that source code written for one may not necessarily compile or run correctly on another.

Even when the same system calls are present on two different UNIX systems, they do not necessarily behave in exactly the same way.

The POSIX standards comprise a set of specifications for different aspects of operating-system behavior. There are POSIX documents for common operating-system functionality and for extensions such as process threads and real-time operations.

Linux is designed to comply with the relevant POSIX documents, and at least two Linux distributions have achieved official POSIX certification.

Because it gives standard interfaces to both the programmer and the user, Linux presents few surprises to anybody familiar with UNIX. We do not detail these interfaces here. The sections on the programmer interface and user interface of BSD apply equally well to Linux.

By default, however, the Linux programming interface adheres to SVR4 UNIX semantics, rather than to BSD behavior. A separate set of libraries is available to implement BSD semantics in places where the two behaviors differ significantly.

Many other standards exist in the UNIX world, but full certification of Linux with respect to these standards is sometimes slowed because certification is often available only for a fee, and the expense involved in certifying an operating system's compliance with most standards is substantial.

However, supporting a wide base of applications is important for any operating system, so implementation of standards is a major goal for Linux development, even if the implementation is not formally certified.

In addition to the basic POSIX standard, Linux currently supports the POSIX threading extensions Pthreads and a subset of the POSIX extensions for real-time process control.

Components of a Linux System

The Linux system is composed of three main bodies of code, in line with most traditional UNIX implementations:

Kernel. The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.

System libraries. The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code. The most important system library is the **C library**, known as libc. In addition to providing the standard C library, libc implements the user mode side of the Linux system call interface, as well as other critical system-level interfaces.

System utilities. The system utilities are programs that perform indi-vidual, specialized management tasks. Some system utilities are invoked just once to initialize and configure some aspect of the system. Others — known as **daemons** in UNIX terminology — run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

Figure 5.1 illustrates the various components that make up a full Linux system.

The most important distinction here is between the kernel and everything else. All the kernel code executes in the processor's privileged mode with full access to all the physical resources of the computer.

Linux refers to this privileged mode as **kernel mode**. Under Linux, no user code is built into the kernel. Any operating-system-support code that does not need to run in kernel mode is placed into the system libraries and runs in **user mode**.

Unlike kernel mode, user mode has access only to a controlled subset of the system's resources.

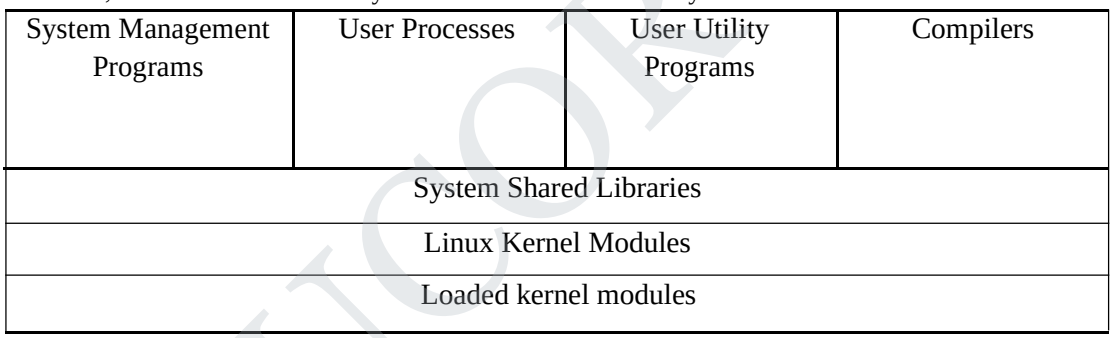


Fig:5.1 Components of the Linux system

Although various modern operating systems have adopted a message-passing architecture for their kernel internals, Linux retains UNIX's historical model: the kernel is created as a single, monolithic binary. The main reason is performance.

Because all kernel code and data structures are kept in a single address space, no context switches are necessary when a process calls an operating-system function or when a hardware interrupt is delivered. More-over, the kernel can pass data and make requests between various subsystems using relatively cheap C function invocation and not more complicated inter-process communication (IPC).

This single address space contains not only the core scheduling and virtual memory code but all kernel code, including all device drivers, file systems, and networking code.

Even though all the kernel components share this same melting pot, there is still room for modularity.

In the same way that user applications can load shared libraries at run time to pull in a needed piece of code, so the Linux kernel can load (and unload) modules dynamically at run time.

The kernel does not need to know in advance which modules may be loaded they are truly independent loadable components.
The Linux kernel forms the core of the Linux operating system. It provides all the functionality necessary to run processes, and it provides system services to give arbitrated and protected access to hardware resources.
The kernel implements all the features required to qualify as an operating system. On its own, however, the operating system provided by the Linux kernel is not a complete UNIX system. It lacks much of the functionality and behavior of UNIX, and the features that it does provide are not necessarily in the format in which a UNIX application expects them to appear.
The operating-system interface visible to running applications is not maintained directly by the kernel. Rather, applications make calls to the system libraries, which in turn call the operating-system services as necessary.
The system libraries provide many types of functionality. At the simplest level, they allow applications to make system calls to the Linux kernel. Making a system call involves transferring control from unprivileged user mode to privileged kernel mode; the details of this transfer vary from architecture to architecture.
The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.
The libraries may also provide more complex versions of the basic system calls. For example, the C language's buffered file-handling functions are all implemented in the system libraries, providing more advanced control of file I/O than the basic kernel system calls.
The libraries also provide routines that do not correspond to system calls at all, such as sorting algorithms, mathematical functions, and string-manipulation routines. All the functions necessary to support the running of UNIX or POSIX applications are implemented in the system libraries.

The Linux system includes a wide variety of user-mode programs both system utilities and user utilities. The system utilities include all the programs necessary to initialize and then administer the system, such as those to set up networking interfaces and to add and remove users from the system.
User utilities are also necessary to the basic operation of the system but do not require elevated privileges to run. They include simple file-management utilities such as those to copy files, create directories, and edit text files.
One of the most important user utilities is the **shell**, the standard command-line interface on UNIX systems. Linux supports many shells; the most common is the **bourne-Again shell (bash)**.

Kernel Modules

The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand. These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run.
In theory, there is no restriction on what a kernel module is allowed to do. Among other things, a kernel module can implement a device driver, a file system, or a networking protocol.
Kernel modules are convenient for several reasons. Linux's source code is free, so anybody wanting to write kernel code is able to compile a modified kernel and to reboot into that new functionality. However, recompiling, relinking, and reloading the entire kernel is a cumbersome cycle to undertake when you are developing a new driver.
If you use kernel modules, you do not have to make a new kernel to test a new driver the driver can be compiled on its own and loaded into the already running kernel. Of course, once a new driver is written, it can be distributed as a module so that other users can benefit from it without having to rebuild their kernels.
This latter point has another implication. Because it is covered by the GPL license, the Linux kernel cannot be released with proprietary components added to it unless those new components are also released under the GPL and the source code for them is made available on demand.
The kernel's module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.
Kernel modules allow a Linux system to be set up with a standard minimal kernel, without any extra device drivers built in.
Any device drivers that the user needs can be either loaded explicitly by the system at startup or loaded automatically by the system on demand and unloaded when not in use.

For example, a mouse driver can be loaded when a USB mouse is plugged into the system and unloaded when the mouse is unplugged.

The module support under Linux has four components:

The **module-management system** allows modules to be loaded into memory and to communicate with the rest of the kernel.

The **module loader and unloader**, which are user-mode utilities, work with the module-management system to load a module into memory.

The **driver-registration system** allows modules to tell the rest of the kernel that a new driver has become available.

A**conflict-resolution mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

Loading a module requires more than just loading its binary contents into kernel memory. The system must also make sure that any references the module makes to kernel symbols or entry points are updated to point to the correct locations in the kernel's address space

Linux deals with this reference updating by splitting the job of module loading into two separate sections: the management of sections of module code in kernel memory and the handling of symbols that modules are allowed to reference.

Linux maintains an internal symbol table in the kernel. This symbol table does not contain the full set of symbols defined in the kernel during the latter's compilation; rather, a symbol must be explicitly exported.

The set of exported symbols constitutes a well-defined interface by which a module can interact with the kernel.Although exporting symbols from a kernel function requires an explicit request by the programmer, no special effort is needed to import those symbols into a module. A module writer just uses the standard external linking of the C language.

Any external symbols referenced by the module but not declared by it are simply marked as unresolved in the final module binary produced by the compiler. When a module is to be loaded into the kernel, a system utility first scans the module for these unresolved references.

All symbols that still need to be resolved are looked up in the kernel's symbol table, and the correct addresses of those symbols in the currently running kernel are substituted into the module's code.

Only then is the module passed to the kernel for loading. If the system utility cannot resolve all references in the module by looking them up in the kernel's symbol table, then the module is rejected.

The loading of the module is performed in two stages.

First, the module-loader utility asks the kernel to reserve a continuous area of virtual kernel memory for the module. The kernel returns the address of the memory allocated, and the loader utility can use this address to relocate the module's machine code to the correct loading address.

A second system call then passes the module, plus any symbol table that the new module wants to export, to the kernel. The module itself is now copied verbatim into the previously allocated space, and the kernel's symbol table is updated with the new symbols for possible use by other modules not yet loaded.

The final module-management component is the module requester. The kernel defines a communication interface to which a module-management program can connect.

With this connection established, the kernel will inform the management process whenever a process requests a device driver, file system, or network service that is not currently loaded and will give the manager the opportunity to load that service.

The original service request will complete once the module is loaded. The manager process regularly queries the kernel to see whether a dynamically loaded module is still in use and unloads that module when it is no longer actively needed.

Driver Registration

Once a module is loaded, it remains no more than an isolated region of memory until it lets the rest of the kernel know what new functionality it provides.

The kernel maintains dynamic tables of all known drivers and provides a set of routines to allow drivers to be added to or removed from these tables at any time.

The kernel makes sure that it calls a module's startup routine when that module is loaded and calls the module's cleanup routine before that module is unloaded. These routines are responsible for registering the module's functionality.

A module may register many types of functionality; it is not limited to only one type.

For example, a device driver might want to register two separate mechanisms for accessing the device. Registration tables include, among others, the following items:

Device drivers. These drivers include character devices (such as printers, terminals, and mice), block devices (including all disk drives), and network interface devices.

File systems. The file system may be anything that implements Linux's virtual file system calling routines. It might implement a format for storing files on a disk, but it might equally well be a network file system, such as NFS, or a virtual file system whose contents are generated on demand, such as Linux's /proc file system.

Network protocols. A module may implement an entire networking protocol, such as TCP or simply a new set of packet-filtering rules for a network firewall.

Binary format. This format specifies a way of recognizing, loading, and executing a new type of executable file.

Conflict Resolution

Commercial UNIX implementations are usually sold to run on a vendor's own hardware. One advantage of a single-supplier solution is that the software vendor has a good idea about what hardware configurations are possible.

PC hardware, however, comes in a vast number of configurations, with large numbers of possible drivers for devices such as network cards and video display adapters.

The problem of managing the hardware configuration becomes more severe when modular device drivers are supported, since the currently active set of devices becomes dynamically variable.

Linux provides a central conflict-resolution mechanism to help arbitrate access to certain hardware resources. Its aims are as follows:

To prevent modules from clashing over access to hardware resources

To prevent **autoprobes** — device-driver probes that auto-detect device configuration — from interfering with existing device drivers

To resolve conflicts among multiple drivers trying to access the same hardware — as, for example, when both the parallel printer driver and the parallel line IP (PLIP) network driver try to talk to the parallel port.

To these ends, the kernel maintains lists of allocated hardware resources. The PC has a limited number of possible I/O ports (addresses in its hardware I/O address space), interrupt lines, and DMA channels. When any device driver wants to access such a resource, it is expected to reserve the resource with.

Process Management

A process is the basic context in which all user-requested activity is serviced within the operating system. To be compatible with other UNIX systems, Linux must use a process model similar to those of other versions of UNIX.

Linux operates differently from UNIX in a few key places, however. In this section, we review the traditional UNIX process model and introduce Linux's threading model.

The fork() and exec() Process Model

The basic principle of UNIX process management is to separate into two steps two operations that are usually combined into one: the creation of a new process and the running of a new program.

A new process is created by the fork() system call, and a new program is run after a call to exec(). These are two distinctly separate functions. We can create a new process with fork() without running a new program the new subprocess simply continues to execute exactly the same program, at exactly the same point, that the first (parent) process was running.

In the same way, running a new program does not require that a new process be created first. Any process may call exec() at any time. A new binary object is loaded into the process's address space and the new executable starts executing in the context of the existing process.

This model has the advantage of great simplicity. It is not necessary to specify every detail of the environment of a new program in the system call that runs that program.

The new program simply runs in its existing environment. If a parent process wishes to modify the environment in which a new program is to be run, it can fork and then, still running the original executable in a child process, make any system calls it requires to modify that child process before finally executing the new program.

Under UNIX, then, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program.

Under Linux, we can break down this context into a number of specific sections. Broadly, process properties fall into three groups: the process identity, environment, and context.

Process Identity

A process identity consists mainly of the following items:

Process ID (PID). Each process has a unique identifier. The PID is used to specify the process to the operating system when an application makes a system call to signal, modify, or wait for the process. Additional identifiers associate the process with a process group (typically, a tree of processes forked by a single user command) and login session.

Credentials. Each process must have an associated user ID and one or more group IDs (user groups are discussed in Section 11.6.2) that determine the rights of a process to access system resources and files.

Personality. Process personalities are not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain varieties of UNIX.

Namespace. Each process is associated with a specific view of the file-system hierarchy, called its **namespace**. Most processes share a common namespace and thus operate on a shared file-system hierarchy. Processes and their children can, however, have different namespaces, each with a unique file-system hierarchy — their own root directory and set of mounted file systems.

Most of these identifiers are under the limited control of the process itself. The process group and session identifiers can be changed if the process wants to start a new group or session. Its credentials can be changed, subject to appropriate security checks. However, the primary PID of a process is unchangeable and uniquely identifies that process until termination.

Process Environment

A process's environment is inherited from its parent and is composed of two null-terminated vectors: the argument vector and the environment vector. The **argument vector** simply lists the command-line arguments used to invoke the running program; it conventionally starts with the name of the program itself. The **environment vector** is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values. The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack.

The argument and environment vectors are not altered when a new process is created. The new child process will inherit the environment of its parent. However, a completely new environment is set up when a new program is invoked. On calling exec(), a process must supply the environment for the new program. The kernel passes these environment variables to the next program, replacing the process's current environment. The kernel otherwise leaves the environment and command-line vectors alone — their interpretation is left entirely to the user-mode libraries and applications.

The passing of environment variables from one process to the next and the inheriting of these variables by the children of a process provide flexible ways to pass information to components of the user-mode system software. Various important environment variables have conventional meanings to related parts of the system software. For example, the TERM variable is set up to name the type of terminal connected to a user's login session. Many programs use this variable to determine how to perform operations on the user's display, such as moving the cursor and scrolling a region of text. Programs with multilingual support use the LANG variable to determine the language in which to display system messages for programs that include multilingual support.

The environment-variable mechanism custom-tailors the operating system on a per-process basis. Users can choose their own languages or select their own editors independently of one another.

Process Context

The process identity and environment properties are usually set up when a process is created and not changed until that process exits. A process may choose to change some aspects of its identity if it needs to do so, or it may alter its environment. In contrast, process context is the state of the running program at any one time; it changes constantly. Process context includes the following parts:

Scheduling context. The most important part of the process context is its scheduling context — the information that the scheduler needs to suspend and restart the process. This information includes saved copies of all the process's registers. Floating-point registers are stored separately and are restored only when needed. Thus, processes that do not use floating-point arithmetic do not incur the overhead of saving that state. The scheduling context also includes information about scheduling priority and about any outstanding signals waiting to be delivered to the process. A key part of the scheduling context is the process's kernel stack, a separate area of kernel memory reserved for use by kernel-mode code. Both system calls and interrupts that occur while the process is executing will use this stack.

Accounting. The kernel maintains accounting information about the resources currently being consumed by each process and the total resources consumed by the process in its entire lifetime so far.

File table. The file table is an array of pointers to kernel file structures representing open files. When making file-I/O system calls, processes refer to files by an integer, known as a **file descriptor (fd)**, that the kernel uses to index into this table.

File-system context. Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The file-system context includes the process's root directory, current working directory, and namespace.

Signal-handler table. UNIX systems can deliver asynchronous signals to a process in response to various external events. The signal-handler table defines the action to take in response to a specific signal. Valid actions include ignoring the signal, terminating the process, and invoking a routine in the process's address space.

Virtual memory context. The virtual memory context describes the full contents of a process's private address space; Linux provides the fork() system call, which duplicates a process without loading a new executable image. Linux also provides the ability to create threads via the clone() system call. Linux does not distinguish between processes and threads, however. In fact, Linux generally uses the term ***task*** — rather than ***process*** or ***thread*** — when referring to a flow of control within a program. The clone() system call behaves identically to fork(), except that it accepts as arguments a set of flags that dictate what resources are shared between the parent and child (whereas a process created with fork() shares no resources with its parent). The flags include:

flag	Meaning
CLONE_FS	File-system information is shared
CLONE_VM	The same memory space is shared
CLONE_SIGHAND	Signal Handlers are shared
CLONE_FILES	The set of open files is shared

Thus, if clone() is passed the flags CLONE FS, CLONE VM, CLONE SIGHAND, and CLONE FILES, the parent and child tasks will share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files. Using clone() in this fashion is equivalent to creating a thread in other systems, since the parent task shares most of its resources with its child task. If none of these flags is set when clone() is invoked, however, the associated resources are not shared, resulting in functionality similar to that of the fork() system call.

The lack of distinction between processes and threads is possible because Linux does not hold a process's entire context within the main process data structure. Rather, it holds the context within independent subcontexts. Thus, process's file-system context, file-descriptor table, signal-handler table, and virtual memory context are held in

separate data structures. The process data structure simply contains pointers to these other structures, so any number of processes can easily share a subcontext by pointing to the same subcontext and incrementing a reference count.

The arguments to the clone() system call tell it which subcontexts to copy and which to share. The new process is always given a new identity and a new scheduling context — these are the essentials of a Linux process. According to the arguments passed, however, the kernel may either create new subcontext data structures initialized so as to be copies of the parent's or set up the new process to use the same subcontext data structures being used by the parent. The fork() system call is nothing more than a special case of clone() that copies all subcontexts, sharing none.

Scheduling

Scheduling is the job of allocating CPU time to different tasks within an operating system. Linux, like all UNIX systems, supports **preemptive multitasking**. In such a system, the process scheduler decides which process runs and when.

Making these decisions in a way that balances fairness and performance across many different workloads is one of the more complicated challenges in modern operating systems.

Normally, we think of scheduling as the running and interrupting of user processes, but another aspect of scheduling is also important to Linux: the running of the various kernel tasks. Kernel tasks encompass both tasks that are requested by a running process and tasks that execute internally on behalf of the kernel itself, such as tasks spawned by Linux's I/O subsystem.

Process Scheduling

Linux has two separate process-scheduling algorithms. One is a time-sharing algorithm for fair, preemptive scheduling among multiple processes. The other is designed for real-time tasks, where absolute priorities are more important than fairness.

The scheduling algorithm used for routine time-sharing tasks received a major overhaul with version 2.6 of the kernel. Earlier versions ran a variation of the traditional UNIX scheduling algorithm. This algorithm does not provide adequate support for SMP systems, does not scale well as the number of tasks on the system grows, and does not maintain fairness among interactive tasks, particularly on systems such as desktops and mobile devices. The process scheduler was first overhauled with version 2.5 of the kernel. Version 2.5 implemented a scheduling algorithm that selects which task to run in constant time — known as *O*(1) regardless of the number of tasks or processors in the system. The new scheduler also provided increased support for SMP, including processor affinity and load balancing. These changes, while improving scalability, did not improve interactive performance or fairness and, in fact, made these problems worse under certain workloads. Consequently, the process scheduler was overhauled a second time, with Linux kernel version 2.6. This version ushered in the

Completely Fair Scheduler (CFS).

The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a **real-time** range from 0 to 99 and a nice **value** ranging from 20 to 19. Smaller nice values indicate higher priorities. Thus, by increasing the nice value, you are decreasing your priority and being "nice" to the rest of the system.

CFS is a significant departure from the traditional UNIX process scheduler. In the latter, the core variables in the scheduling algorithm are priority and time slice. The **time slice** is the length of time — the ***slice*** of the processor that a process is afforded. Traditional UNIX systems give processes a fixed time slice, perhaps with a boost or penalty for high- or low-priority processes, respectively. A process may run for the length of its time slice, and higher-priority processes run before lower-priority processes. It is a simple algorithm that many non-UNIX systems employ. Such simplicity worked well for early time-sharing systems but has proved incapable of delivering good interactive performance and fairness on today's modern desktops and mobile devices.

CFS introduced a new scheduling algorithm called **fair scheduling** that eliminates time slices in the traditional sense. Instead of time slices, all processes are allotted a proportion of the processor's time. CFS calculates how long a process should run as a function of the total number of runnable processes.

To start, CFS says that if there are *N* runnable processes, then each should be afforded 1/*N* of the processor's time. CFS then adjusts this allotment by weighting each process's allotment by its nice value. Processes with the default nice value have a weight of 1 their priority is unchanged. Processes with a smaller nice value (higher priority) receive a higher weight, while processes with a larger nice value (lower priority) receive a lower weight. CFS then runs

each process for a "time slice" proportional to the process's weight divided by the total weight of all runnable processes.

To calculate the actual length of time a process runs, CFS relies on a configurable variable called **target latency**, which is the interval of time during which every runnable task should run at least once. For example, assume that the target latency is 10 milliseconds. Further assume that we have two runnable processes of the same priority. Each of these processes has the same weight and therefore receives the same proportion of the processor's time. In this case, with a target latency of 10 milliseconds, the first process runs for 5 milliseconds, then the other process runs for 5 milliseconds, then the first process runs for 5 milliseconds again, and so forth. If we have 10 runnable processes, then CFS will run each for a millisecond before repeating.

But what if we had, say, 1, 000 processes? Each process would run for 1 microsecond if we followed the procedure just described. Due to switching costs, scheduling processes for such short lengths of time is inefficient. CFS consequently relies on a second configurable variable, the **minimum granularity**, which is a minimum length of time any process is allotted the processor. All processes, regardless of the target latency, will run for at least the minimum granularity. In this manner, CFS ensures that switching costs do not grow unacceptably large when the number of runnable processes grows too large. In doing so, it violates its attempts at fairness. In the usual case, however, the number of runnable processes remains reasonable, and both fairness and switching costs are maximized.

With the switch to fair scheduling, CFS behaves differently from traditional UNIX process schedulers in several ways. Most notably, as we have seen, CFS eliminates the concept of a static time slice. Instead, each process receives a proportion of the processor's time. How long that allotment is depends on how many other processes are runnable. This approach solves several problems in mapping priorities to time slices inherent in preemptive, priority-based scheduling algorithms. It is possible, of course, to solve these problems in other ways without abandoning the classic UNIX scheduler. CFS, however, solves the problems with a simple algorithm that performs well on interactive workloads such as mobile devices without compromising throughput performance on the largest of servers.

Real-Time Scheduling

Linux's real-time scheduling algorithm is significantly simpler than the fair scheduling employed for standard time-sharing processes. Linux implements the two real-time scheduling classes required by POSIX.1b: first-come, first-served (FCFS) and round-robin. In both cases, each process has a priority in addition to its scheduling class. The scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest. The only difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves.

Linux's real-time scheduling is soft rather than hard real time. The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees about how quickly a real-time process will be scheduled once that process becomes runnable. In contrast, a hard real-time system can guarantee a minimum latency between when a process becomes runnable and when it actually runs.

Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules processes. A request for kernel-mode execution can occur in two ways. A running program may request an operating-system service, either explicitly via a system call or implicitly for example, when a page fault occurs. Alternatively, a device controller may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem for the kernel is that all these tasks may try to access the same internal data structures. If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption. This fact relates to the idea of critical sections portions of code that access shared data and thus must not be allowed to execute concurrently. As a result, kernel synchronization involves much more than just process scheduling. A framework is required that allows kernel tasks to run without violating the integrity of shared data.

Prior to version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted even if a higher-priority process became available to run. With version 2.6, the Linux kernel became fully preemptive. Now, a task can be preempted when it is running in the kernel.

The Linux kernel provides spinlocks and semaphores (as well as reader – writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock, and the kernel is designed so that spinlocks are held for only short durations. On single-processor machines, spinlocks are not appropriate for use and are replaced by enabling and disabling kernel preemption. That is, rather than holding a spinlock, the task disables kernel preemption. When the task would otherwise release the spinlock, it enables kernel preemption. This pattern is summarized below:

Single processor	Multiple processors
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

Linux uses an interesting approach to disable and enable kernel pre-emption. It provides two simple kernel interfaces preempt disable() and preempt enable(). In addition, the kernel is not preemptible if a kernel-mode task is holding a spinlock. To enforce this rule, each task in the system has a thread-info structure that includes the field preempt count, which is a counter indicating the number of locks being held by the task. The counter is incremented when a lock is acquired and decremented when a lock is released. If the value of preempt count for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted, assuming there are no outstanding calls to preempt disable().

Spinlocks along with the enabling and disabling of kernel preemption are used in the kernel only when the lock is held for short durations. When a lock must be held for longer periods, semaphores are used.

The second protection technique used by Linux applies to critical sections that occur in interrupt service routines. The basic tool is the processor's interrupt-control hardware. By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access to shared data structures.

However, there is a penalty for disabling interrupts. On most hardware architectures, interrupt enable and disable instructions are not cheap. More importantly, as long as interrupts remain disabled, all I/O is suspended, and any device waiting for servicing will have to wait until interrupts are reenabled; thus, performance degrades. To address this problem, the Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled. This ability is especially useful in the networking code. An interrupt in a network device driver can signal the arrival of an entire network packet, which may result in a great deal of code being executed to disassemble, route, and forward that packet within the interrupt service routine.

Linux implements this architecture by separating interrupt service routines into two sections: the top half and the bottom half. The ***top half*** is the standard interrupt service routine that runs with recursive interrupts disabled. Interrupts of the same number (or line) are disabled, but other interrupts may run. The ***bottom half*** of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves. The bottom-half scheduler is invoked automatically whenever an interrupt service routine exits.

This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself. If another interrupt occurs while a bottom half is executing, then that interrupt can request that the same bottom half execute, but the execution will be deferred until the one currently running completes. Each execution of the bottom half can be interrupted by a top half but can never be interrupted by a similar bottom half.

The top-half/bottom-half architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code. The kernel can code critical sections easily using this system. Interrupt handlers can code their critical sections as bottom halves; and when the foreground kernel wants to enter a critical section, it can disable any relevant bottom halves to prevent any other critical sections from interrupting it. At the end of the critical section, the kernel can reenable the bottom halves and run any bottom-half tasks that have been queued by top-half interrupt service routines during the critical section.

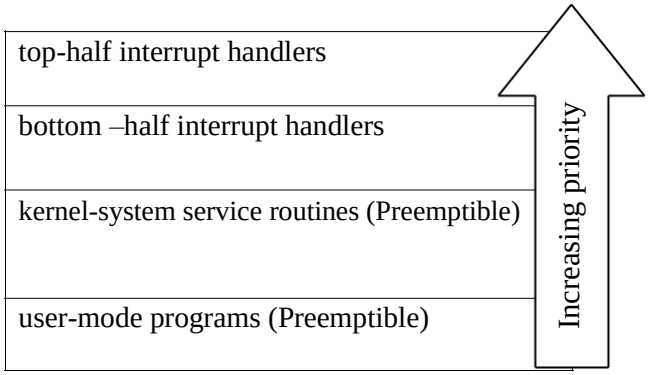


Fig 5.2: Interrupt protection levels

Figure 5.2 summarizes the various levels of interrupt protection within the kernel. Each level may be interrupted by code running at a higher level but will never be interrupted by code running at the same or a lower level. Except for user-mode code, user processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.

Symmetric Multiprocessing

The Linux 2.0 kernel was the first stable Linux kernel to support **symmetric multiprocessor (SMP)** hardware, allowing separate processes to execute in parallel on separate processors. The original implementation of SMP imposed the restriction that only one processor at a time could be executing kernel code.

In version 2.2 of the kernel, a single kernel spinlock (sometimes termed **BKL** for "big kernel lock") was created to allow multiple processes (running on different processors) to be active in the kernel concurrently. However, the BKL provided a very coarse level of locking granularity, resulting in poor scalability to machines with many processors and processes. Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel's data structures. Such spinlocks are described in Section 18.5.3. The 3.0 kernel provides additional SMP enhancements, including ever-finer locking, processor affinity, and load-balancing algorithms.

Memory Management

Memory management under Linux has two components. The first deals with allocating and freeing physical memory — pages, groups of pages, and small blocks of RAM. The second handles virtual memory, which is memory-mapped into the address space of running processes. In this section, we describe these two components and then examine the mechanisms by which the loadable components of a new program are brought into a process's virtual memory in response to an exec() system call.

Management of Physical Memory

Due to specific hardware constraints, Linux separates physical memory into four different **zones**, or regions:

ZONE DMA
ZONE DMA32
ZONE NORMAL
ZONE HIGHMEM

These zones are architecture specific. For example, on the Intel x86-32 architec-ture, certain ISA (industry standard architecture) devices can only access the lower 16 MB of physical memory using DMA. On these systems, the first 16 MB of physical memory comprise ZONE DMA. On other systems, certain devices can only access the first 4 GB of physical memory, despite supporting 64-bit addresses. On such systems, the first 4 GB of physical memory comprise ZONE DMA32. ZONE HIGHMEM (for "high memory") refers to physical memory that is not mapped into the kernel address space. For example, on the 32-bit Intel architecture (where 2^{32} provides a 4-GB address space), the kernel is mapped into the first 896 MB of the address space; the remaining memory is referred to as **high memory** and is allocated from ZONE HIGHMEM. Finally, ZONE NORMAL comprises everything else — the normal, regularly

mapped pages. Whether an architecture has a given zone depends on its constraints. A modern, 64-bit architecture such as Intel x86-64 has a small 16 MB ZONE DMA (for legacy devices) and all the rest of its memory in ZONE NORMAL, with no "high memory".

The relationship of zones and physicāl addresses on the Intel x86-32 architecture is shown in Figure 5.3. The kernel maintains a list of free pages for each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.

The primary physical-memory manager in the Linux kernel is the **page allocator**. Each zone has its own allocator, which is responsible for allocating and freeing all physical pages for the zone and is capable of allocating ranges of physically contiguous pages on request. The allocator uses a buddy system (Section 9.8.1) to keep track of available physical pages. In this scheme, adjacent units of allocatable memory are paired together (hence its name). Each allocatable memory region has an adjacent partner (or buddy). Whenever two allocated partner regions are freed up, they are combined to form a larger region — a **buddy heap**. That larger region also has a partner, with which it can combine to form a still larger free region. Conversely, if a small memory request cannot be satisfied by allocation of an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request. Separate linked lists are used to record the free memory regions of each allowable size. Under Linux, the smallest size allocatable under this mechanism is a single physical page. Figure 5.3 shows an example of buddy-heap allocation. A 4-KB region is being allocated, but the smallest available region is 16 KB. The region is broken up recursively until a piece of the desired size is available.

zone	physical memory
ZONE_DMA	<16MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

Figure 5.3 Relationship of zones and physical addresses in Intel x86-32.

Ultimately, all memory allocations in the Linux kernel are made either statically, by drivers that reserve a contiguous area of memory during system boot time, or dynamically, by the page allocator. However, kernel functions do not have to use the basic allocator to reserve memory. Several specialized memory-management subsystems use the underlying page allocator to man-age their own pools of memory. The most important are the virtual memory system, the kmalloc() variable-length allocator; the slab allocator, used for allocating memory for kernel data structures; and the page cache, used for caching pages belonging to files.

Many components of the Linux operating system need to allocate entire pages on request, but often smaller blocks of memory are required. The kernel provides an additional allocator for arbitrary-sized requests, where the size of a request is not known in advance and may be only a few bytes. Analogous to the C language's malloc() function, this kmalloc() service allocates entire physical pages on demand but then splits them into smaller pieces. The kernel maintains lists of pages in use by the kmalloc() service. Allocating memory involves determining the appropriate list and either taking the first free piece available on the list or allocating a new page and splitting it up. Memory regions claimed by the kmalloc() system are allocated permanently until they are freed explicitly with a corresponding call to kfree(); the kmalloc() system cannot reallocate or reclaim these regions in response to memory shortages.

Another strategy adopted by Linux for allocating kernel memory is known as slab allocation. A **slab** is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for each unique kernel data structure — for example, a cache for the data structure representing process descriptors, a cache for file objects, a cache for inodes, and so forth.

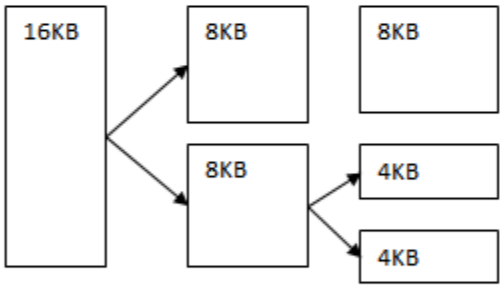


Fig: 5.4 Splitting of memory in the buddy system

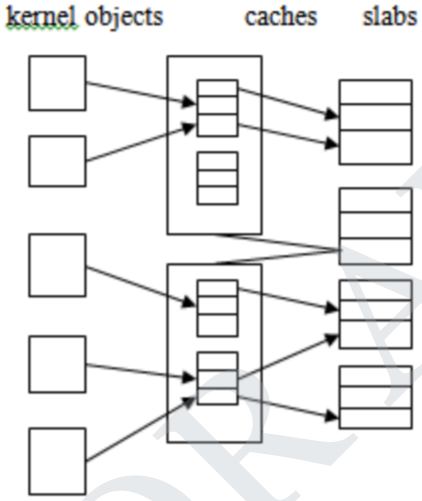


Fig 5.5 Slab allocation in Linux

Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing inodes stores instances of inode structures, and the cache representing process descriptors stores instances of process descriptor structures. The relationship among slabs, caches, and objects is shown in Figure 5.5. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size. These objects are stored in the respective caches for 3-KB and 7-KB objects.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all the objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type struct task struct, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the struct task struct object from its cache. The cache will fulfill the request using a struct task struct object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

Full. All objects in the slab are marked as used.

Empty. All objects in the slab are marked as free.

Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exist, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

Two other main subsystems in Linux do their own management of physical pages: the page cache and the virtual memory system. These systems are closely related to each other. The **page cache** is the kernel's main cache for files and is the main mechanism through which I/O to block devices is performed. File systems of all types, including the native Linux disk-based file systems and the NFS networked file system, perform their I/O through the page cache. The page cache stores entire pages of file contents and is not limited to block devices. It can also cache networked data. The virtual memory system manages the contents of each process's virtual address space. These two systems interact closely with each other because reading a page of data into the page cache requires mapping pages in the page cache using the virtual memory system.

Virtual Memory

The Linux virtual memory system is responsible for maintaining the address space accessible to each process. It creates pages of virtual memory on demand and manages loading those pages from disk and swapping them back out to disk as required. Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of separate regions and as a set of pages.

The first view of an address space is the logical view, describing instructions that the virtual memory system has received concerning the layout of the address space. In this view, the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space. Each region is described internally by a single vm area struct structure that defines the properties of the region, including the process's read, write, and execute permissions in the region as well as information about any files associated with the region. The regions for each address space are linked into a balanced binary tree to allow fast lookup of the region corresponding to any virtual address.

The kernel also maintains a second, physical view of each address space. This view is stored in the hardware page tables for the process. The page-table entries identify the exact current location of each page of virtual memory, whether it is on disk or in physical memory. The physical view is managed by a set of routines, which are invoked from the kernel's software-interrupt handlers whenever a process tries to access a page that is not currently present in the page tables. Each vm area struct in the address-space description contains a field pointing to a table of functions that implement the key page-management functionality for any given virtual memory region. All requests to read or write an unavailable page are eventually dispatched to the appropriate handler in the function table for the vm area struct, so that the central memory-management routines do not have to know the details of managing each possible type of memory region.

Virtual Memory Regions

Linux implements several types of virtual memory regions. One property that characterizes virtual memory is the backing store for the region, which describes where the pages for the region come from. Most memory regions are backed either by a file or by nothing. A region backed by nothing is the simplest type of virtual memory region. Such a region represents **demand-zero memory**: when a process tries to read a page in such a region, it is simply given back a page of memory filled with zeros.

A region backed by a file acts as a viewport onto a section of that file. Whenever the process tries to access a page within that region, the page table is filled with the address of a page within the kernel's page cache corresponding to the appropriate offset in the file. The same page of physical memory is used by both the page cache and the process's page tables, so any changes made to the file by the file system are immediately visible to any processes that have mapped that file into their address space. Any number of processes can map the same region of the same file, and they will all end up using the same page of physical memory for the purpose.

A virtual memory region is also defined by its reaction to writes. The mapping of a region into the process's address space can be either ***private*** or ***shared***. If a process writes to a privately mapped region, then the pager detects that a copy-on-write is necessary to keep the changes local to the process. In contrast, writes to a shared region result in updating of the object mapped into that region, so that the change will be visible immediately to any other process that is mapping that object.

Lifetime of a Virtual Address Space

The kernel creates a new virtual address space in two situations: when a process runs a new program with the exec() system call and when a new process is created by the fork() system call. The first case is easy. When a new

program is executed, the process is given a new, completely empty virtual address space. It is up to the routines for loading the program to populate the address space with virtual memory regions.

The second case, creating a new process with fork(), involves creating a complete copy of the existing process's virtual address space. The kernel copies the parent process's vm area struct descriptors, then creates a new set of page tables for the child. The parent's page tables are copied directly into the child's, and the reference count of each page covered is incremented. Thus, after the fork, the parent and child share the same physical pages of memory in their address spaces.

A special case occurs when the copying operation reaches a virtual memory region that is mapped privately. Any pages to which the parent process has written within such a region are private, and subsequent changes to these pages by either the parent or the child must not update the page in the other process's address space. When the page-table entries for such regions are copied, they are set to be read only and are marked for copy-on-write. As long as neither process modifies these pages, the two processes share the same page of physical memory. However, if either process tries to modify a copy-on-write page, the reference count on the page is checked. If the page is still shared, then the process copies the page's contents to a brand-new page of physical memory and uses its copy instead. This mechanism ensures that private data pages are shared between processes whenever possible and copies are made only when absolutely necessary.

Swapping and Paging

An important task for a virtual memory system is to relocate pages of memory from physical memory out to disk when that memory is needed. Early UNIX systems performed this relocation by swapping out the contents of entire processes at once, but modern versions of UNIX rely more on paging — the movement of individual pages of virtual memory between physical memory and disk. Linux does not implement whole-process swapping; it uses the newer paging mechanism exclusively.

The paging system can be divided into two sections. First, the **policy algorithm** decides which pages to write out to disk and when to write them. Second, the **paging mechanism** carries out the transfer and pages data back into physical memory when they are needed again.

Linux's **pageout policy** uses a modified version of the standard clock (or second-chance) algorithm. Under Linux, a multiple-pass clock is used, and every page has an ***age*** that is adjusted on each pass of the clock. The age is more precisely a measure of the page's youthfulness, or how much activity the page has seen recently. Frequently accessed pages will attain a higher age value, but the age of infrequently accessed pages will drop toward zero with each pass. This age valuing allows the pager to select pages to page out based on a least frequently used (LFU) policy.

The paging mechanism supports paging both to dedicated swap devices and partitions and to normal files, although swapping to a file is significantly slower due to the extra overhead incurred by the file system. Blocks are allocated from the swap devices according to a bitmap of used blocks, which is maintained in physical memory at all times. The allocator uses a next-fit algorithm to try to write out pages to continuous runs of disk blocks for improved performance. The allocator records the fact that a page has been paged out to disk by using a feature of the page tables on modern processors: the page-table entry's page-not-present bit is set, allowing the rest of the page-table entry to be filled with an index identifying where the page has been written.

Kernel Virtual Memory

Linux reserves for its own internal use a constant, architecture-dependent region of the virtual address space of every process. The page-table entries that map to these kernel pages are marked as protected, so that the pages are not visible or modifiable when the processor is running in user mode. This kernel virtual memory area contains two regions. The first is a static area that contains page-table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run. The core of the kernel, along with all pages allocated by the normal page allocator, resides in this region.

The remainder of the kernel's reserved section of address space is not reserved for any specific purpose. Page-table entries in this address range can be modified by the kernel to point to any other areas of memory. The kernel provides a pair of facilities that allow kernel code to use this virtual memory. The vmalloc() function allocates an arbitrary number of physical pages of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory. The vremap() function maps a sequence of virtual addresses to point to an area of memory used by a device driver for memory-mapped I/O.

Execution and Loading of User Programs

The Linux kernel's execution of user programs is triggered by a call to the exec() system call. This exec() call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program. The first job of this system service is to verify that the calling process has permission rights to the file being executed. Once that matter has been checked, the kernel invokes a loader routine to start running the program. The loader does not necessarily load the contents of the program file into physical memory, but it does at least set up the mapping of the program into virtual memory.

There is no single routine in Linux for loading a new program. Instead, Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an exec() system call is made. The initial reason for this loader table was that, between the releases of the 1.0 and 1.2 kernels, the standard format for Linux's binary files was changed. Older Linux kernels understood the a.out format for binary files — a relatively simple format common on older UNIX systems. Newer Linux systems use the more modern **ELF** format, now supported by most current UNIX implementations. ELF has a number of advantages over a.out, including flexibility and extendability. New sections can be added to an ELF binary (for example, to add extra debugging information) without causing the loader routines to become confused. By allowing registration of multiple loader routines, Linux can easily support the ELF and a.out binary formats in a single running system.

Mapping of Programs into Memory

Under Linux, the binary loader does not load a binary file into physical memory. Rather, the pages of the binary file are mapped into regions of virtual memory. Only when the program tries to access a given page will a page fault result in the loading of that page into physical memory using demand paging.

It is the responsibility of the kernel's binary loader to set up the initial memory mapping. An ELF-format binary file consists of a header followed by several page-aligned sections. The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

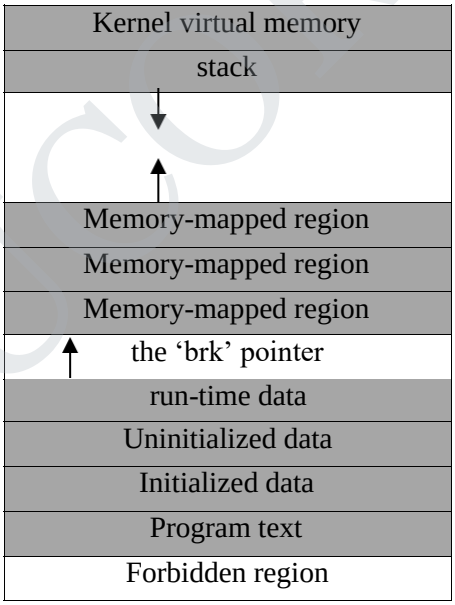


Fig 5.6 Memory layout for ELF programs

Figure 5.6 shows the typical layout of memory regions set up by the ELF loader. In a reserved region at one end of the address space sits the kernel, in its own privileged region of virtual memory inaccessible to normal user-mode programs. The rest of virtual memory is available to applications, which can use the kernel's memory-mapping functions to create regions that map a portion of a file or that are available for application data.

The loader's job is to set up the initial memory mapping to allow the execution of the program to start. The regions that need to be initialized include the stack and the program's text and data regions.

The stack is created at the top of the user-mode virtual memory; it grows downward toward lower-numbered addresses. It includes copies of the arguments and environment variables given to the program in the exec() system

call. The other regions are created near the bottom end of virtual memory. The sections of the binary file that contain program text or read-only data are mapped into memory as a write-protected region. Writable initialized data are mapped next; then any uninitialized data are mapped in as a private demand-zero region.

Directly beyond these fixed-sized regions is a variable-sized region that programs can expand as needed to hold data allocated at run time. Each process has a pointer, brk, that points to the current extent of this data region, and processes can extend or contract their brk region with a single system call— sbrk().

Once these mappings have been set up, the loader initializes the process's program-counter register with the starting point recorded in the ELF header, and the process can be scheduled.

Static and Dynamic Linking

Once the program has been loaded and has started running, all the necessary contents of the binary file have been loaded into the process's virtual address space. However, most programs also need to run functions from the system libraries, and these library functions must also be loaded. In the simplest case, the necessary library functions are embedded directly in the program's executable binary file. Such a program is statically linked to its libraries, and statically linked executables can commence running as soon as they are loaded.

The main disadvantage of static linking is that every program generated must contain copies of exactly the same common system library functions. It is much more efficient, in terms of both physical memory and disk-space usage, to load the system libraries into memory only once. Dynamic linking allows that to happen.

Linux implements dynamic linking in user mode through a special linker library. Every dynamically linked program contains a small, statically linked function that is called when the program starts. This static function just maps the link library into memory and runs the code that the function contains. The link library determines the dynamic libraries required by the program and the names of the variables and functions needed from those libraries by reading the information contained in sections of the ELF binary. It then maps the libraries into the middle of virtual memory and resolves the references to the symbols contained in those libraries. It does not matter exactly where in memory these shared libraries are mapped: they are compiled into **position-independent code (PIC)**, which can run at any address in memory.

File Systems

Linux retains UNIX's standard file-system model. In UNIX, a file does not have to be an object stored on disk or fetched over a network from a remote file server. Rather, UNIX files can be anything capable of handling the input or output of a stream of data. Device drivers can appear as files, and interprocess-communication channels or network connections also look like files to the user.

The Linux kernel handles all these types of files by hiding the implemen-tation details of any single file type behind a layer of software, the virtual file system (VFS). Here, we first cover the virtual file system and then discuss the standard Linux file system — ext3.

The Virtual File System

The Linux VFS is designed around object-oriented principles. It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects. The VFS defines four main object types:

An **inode object** represents an individual file.

A **file object** represents an open file.

A **superblock object** represents an entire file system.

A **dentry object** represents an individual directory entry.

For each of these four object types, the VFS defines a set of operations. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that object. For example, an abbreviated API for some of the file object's operations includes:

int open(. . .) — Open a file.

ssize t read(. . .) — Read from a file.

ssize t write(. . .) — Write to a file.

int mmap(. . .) — Memory-map a file.

The complete definition of the file object is specified in the struct file operations, which is located in the file /usr/include/linux/fs.h. An implementation of the file object (for a specific file type) is required to implement each function specified in the definition of the file object.

The VFS software layer can perform an operation on one of the file-system objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a networked file, a disk file, a network socket, or a directory file. The appropriate function for that file's read() operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

The inode and file objects are the mechanisms used to access files. An inode object is a data structure containing pointers to the disk blocks that contain the actual file contents, and a file object represents a point of access to the data in an open file. A process cannot access an inode's contents without first obtaining a file object pointing to the inode. The file object keeps track of where in the file the process is currently reading or writing, to keep track of sequential file I/O. It also remembers the permissions (for example, read or write) requested when the file was opened and tracks the process's activity if necessary to perform adaptive read-ahead, fetching file data into memory before the process requests the data, to improve performance.

File objects typically belong to a single process, but inode objects do not. There is one file object for every instance of an open file, but always only a single inode object. Even when a file is no longer in use by any process, its inode object may still be cached by the VFS to improve performance if the file is used again in the near future. All cached file data are linked onto a list in the file's inode object. The inode also maintains standard information about each file, such as the owner, size, and time most recently modified.

Directory files are dealt with slightly differently from other files. The UNIX programming interface defines a number of operations on directories, such as creating, deleting, and renaming a file in a directory. The system calls for these directory operations do not require that the user open the files concerned, unlike the case for reading or writing data. The VFS therefore defines these directory operations in the inode object, rather than in the file object.

The superblock object represents a connected set of files that form a self-contained file system. The operating-system kernel maintains a single superblock object for each disk device mounted as a file system and for each networked file system currently connected. The main responsibility of the superblock object is to provide access to inodes. The VFS identifies every inode by a unique file-system/inode number pair, and it finds the inode corresponding to a particular inode number by asking the superblock object to return the inode with that number.

Finally, a dentry object represents a directory entry, which may include the name of a directory in the path name of a file (such as /usr) or the actual file (such as stdio.h). For example, the file /usr/include/stdio.h contains the directory entries (1) /, (2) usr, (3) include, and (4) stdio.h. Each of these values is represented by a separate dentry object.

As an example of how dentry objects are used, consider the situ-ation in which a process wishes to open the file with the pathname /usr/include/stdio.h using an editor. Because Linux treats directory names as files, translating this path requires first obtaining the inode for the root — /. The operating system must then read through this file to obtain the inode for the file include. It must continue this process until it obtains the inode for the file stdio.h. Because path-name translation can be a time-consuming task, Linux maintains a cache of dentry objects, which is consulted during path-name translation. Obtaining the inode from the dentry cache is considerably faster than having to read the on-disk file.

The Linux ext3 File System

The standard on-disk file system used by Linux is called **ext3**, for historical reasons. Linux was originally programmed with a Minix-compatible file system, to ease exchanging data with the Minix development system, but that file system was severely restricted by 14-character file-name limits and a maximum file-system size of 64 MB. The Minix file system was superseded by a new file system, which was christened the **extended file system (extfs)**. A later redesign to improve performance and scalability and to add a few missing features led to the **second extended file system (ext2)**. Further development added journaling capabilities, and the system was renamed the **third extended file system (ext3)**. Linux kernel developers are working on augmenting ext3 with modern file-system features such as extents. This new file system is called the **fourth extended file system (ext4)**. The rest of this section discusses ext3, however, since it remains the most-deployed Linux file system. Most of the discussion applies equally to ext4.

Linux's ext3 has much in common with the BSD Fast File System (FFS). It uses a similar mechanism for locating the data blocks belonging to a specific file, storing data-block pointers in indirect blocks throughout the file

system with up to three levels of indirection. As in FFS, directory files are stored on disk just like normal files, although their contents are interpreted differently. Each block in a directory file consists of a linked list of entries. In turn, each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers.

The main differences between ext3 and FFS lie in their disk-allocation policies. In FFS, the disk is allocated to files in blocks of 8 KB. These blocks are subdivided into fragments of 1 KB for storage of small files or partially filled blocks at the ends of files. In contrast, ext3 does not use fragments at all but performs all its allocations in smaller units. The default block size on ext3 varies as a function of the total size of the file system. Supported block sizes are 1, 2, 4, and 8 KB.

To maintain high performance, the operating system must try to perform I/O operations in large chunks whenever possible by clustering physically adjacent I/O requests. Clustering reduces the per-request overhead incurred by device drivers, disks, and disk-controller hardware. A block-sized I/O request size is too small to maintain good performance, so ext3 uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

The ext3 allocation policy works as follows: As in FFS, an ext3 file system is partitioned into multiple segments. In ext3, these are called **block groups**. FFS uses the similar concept of **cylinder groups**, where each group corresponds to a single cylinder of a physical disk. (Note that modern disk-drive technology packs sectors onto the disk at different densities, and thus with different cylinder sizes, depending on how far the disk head is from the center of the disk. Therefore, fixed-sized cylinder groups do not necessarily correspond to the disk's geometry.)

When allocating a file, ext3 must first select the block group for that file. For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated. For inode allocations, it selects the block group in which the file's parent directory resides for nondirectory files. Directory files are not kept together but rather are dispersed throughout the available block groups. These policies are designed not only to keep related information within the same block group but also to spread out the disk load among the disk's block groups to reduce the fragmentation of any one area of the disk.

Within a block group, ext3 tries to keep allocations physically contiguous if possible, reducing fragmentation if it can. It maintains a bitmap of all free blocks in a block group. When allocating the first blocks for a new file, it starts searching for a free block from the beginning of the block group. When extending a file, it continues the search from the block most recently allocated to the file. The search is performed in two stages. First, ext3 searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit. The search for free bytes aims to allocate disk space in chunks of at least eight blocks where possible.

Once a free block has been identified, the search is extended backward until an allocated block is encountered. When a free byte is found in the bitmap, this backward extension prevents ext3 from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found. Once the next block to be allocated has been found by either bit or byte search, ext3 extends the allocation forward for up to eight blocks and preallocates these extra blocks to the file. This preallocation helps to reduce fragmentation during interleaved writes to separate files and also reduces the CPU cost of disk allocation by allocating multiple blocks simultaneously. The preallocated blocks are returned to the free-space bitmap when the file is closed.

Figure 5.7 illustrates the allocation policies. Each row represents a sequence of set and unset bits in an allocation bitmap, indicating used and free blocks on disk. In the first case, if we can find any free blocks sufficiently near the start of the search, then we allocate them no matter how fragmented they may be. The fragmentation is partially compensated for by the fact that the blocks are close together and can probably all be read without any disk seeks. Furthermore, allocating them all to one file is better in the long run than allocating isolated blocks to separate files once large free areas become scarce on disk. In the second case, we have not immediately found a free block close by, so we search forward for an entire free byte in the bitmap. If we allocated that byte as a whole, we would end up creating a fragmented area of free space between it and the allocation preceding it. Thus, before allocating, we back up to make this allocation flush with the allocation preceding it, and then we allocate forward to satisfy the default allocation of eight blocks.

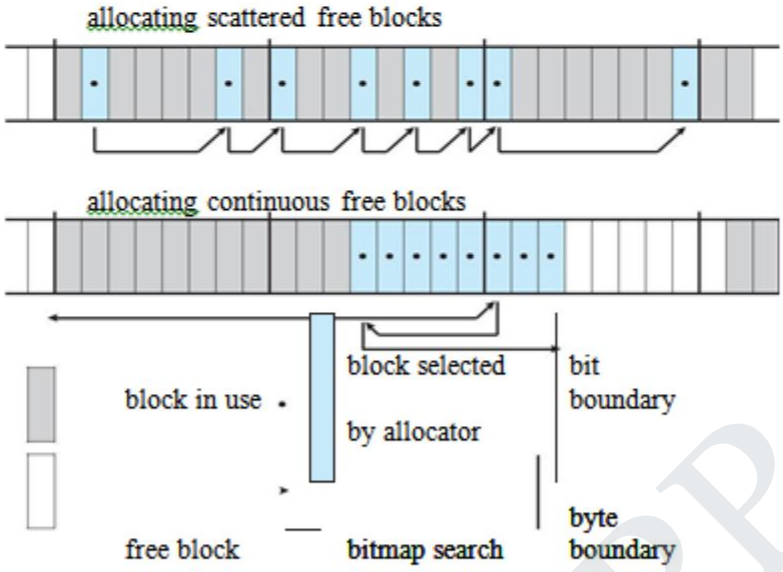


Figure 5.7 ext3 block-allocation policies.

Journaling

The ext3 file system supports a popular feature called **journaling**, whereby modifications to the file system are written sequentially to a journal. A set of operations that performs a specific task is a **transaction**. Once a transaction is written to the journal, it is considered to be committed. Meanwhile, the journal entries relating to the transaction are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the journal. The journal, which is actually a circular buffer, may be in a separate section of the file system, or it may even be on a separate disk spindle. It is more efficient, but more complex, to have it under separate read – write heads, thereby decreasing head contention and seek times.

If the system crashes, some transactions may remain in the journal. Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed once the system recovers. The transactions can be executed from the pointer until the work is complete, and the file-system structures remain consistent. The only problem occurs when a transaction has been aborted — that is, it was not committed before the system crashed. Any changes from those transactions that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating all problems with consistency checking.

Journaling file systems may perform some operations faster than non-journaling systems, as updates proceed much faster when they are applied to the in-memory journal rather than directly to the on-disk data structures. The reason for this improvement is found in the performance advantage of sequential I/O over random I/O. Costly synchronous random writes to the file system are turned into much less costly synchronous sequential writes to the file system's journal. Those changes, in turn, are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of file-system metadata-oriented operations, such as file creation and deletion. Due to this performance improvement, ext3 can be configured to journal only metadata and not file data.

The Linux Process File System

The flexibility of the Linux VFS enables us to implement a file system that does not store data persistently at all but rather provides an interface to some other functionality. The Linux **process file system**, known as the /proc file system, is an example of a file system whose contents are not actually stored anywhere but are computed on demand according to user file I/O requests.

A /proc file system is not unique to Linux. SVR4 UNIX introduced a /proc file system as an efficient interface to the kernel's process debugging support. Each subdirectory of the file system corresponded not to a directory on any disk but rather to an active process on the current system. A listing of the file system reveals one directory per process, with the directory name being the ASCII decimal representation of the process's unique process identifier (PID).

Linux implements such a /proc file system but extends it greatly by adding a number of extra directories and text files under the file system's root directory. These new entries correspond to various statistics about the kernel and the associated loaded drivers. The /proc file system provides a way for programs to access this information as plain text files; the standard UNIX user environment provides powerful tools to process such files. For example, in the past, the traditional UNIX ps command for listing the states of all running processes has been implemented as a privileged process that reads the process state directly from the kernel's virtual memory. Under Linux, this command is implemented as an entirely unprivileged program that simply parses and formats the information from /proc.

The /proc file system must implement two things: a directory structure and the file contents within. Because a UNIX file system is defined as a set of file and directory inodes identified by their inode numbers, the /proc file system must define a unique and persistent inode number for each directory and the associated files. Once such a mapping exists, the file system can use this inode number to identify just what operation is required when a user tries to read from a particular file inode or to perform a lookup in a particular directory inode. When data are read from one of these files, the /proc file system will collect the appropriate information, format it into textual form, and place it into the requesting process's read buffer.

The mapping from inode number to information type splits the inode number into two fields. In Linux, a PID is 16 bits in size, but an inode number is 32 bits. The top 16 bits of the inode number are interpreted as a PID, and the remaining bits define what type of information is being requested about that process.

A PID of zero is not valid, so a zero PID field in the inode number is taken to mean that this inode contains global rather than process-specific information. Separate global files exist in /proc to report information such as the kernel version, free memory, performance statistics, and drivers currently running.

Input and Output

To the user, the I/O system in Linux looks much like that in any UNIX system. That is, to the extent possible, all device drivers appear as normal files. Users can open an access channel to a device in the same way they open any other file — devices can appear as objects within the file system. The system administrator can create special files within a file system that contain references to a specific device driver, and a user opening such a file will be able to read from and write to the device referenced. By using the normal file-protection system, which determines who can access which file, the administrator can set access permissions for each device.

Linux splits all devices into three classes: block devices, character devices, and network devices. Figure 5.8 illustrates the overall structure of the device-driver system.

Block devices include all devices that allow random access to completely independent, fixed-sized blocks of data, including hard disks and floppy disks, CD-ROMs and Blu-ray discs, and flash memory. Block devices are typically used to store file systems, but direct access to a block device is also allowed so that programs can create and repair the file system that the device contains. Applications can also access these block devices directly if they wish. For example, a database application may prefer to perform its own fine-tuned layout of data onto a disk rather than using the general-purpose file system.

Character devices include most other devices, such as mice and keyboards. The fundamental difference between block and character devices is random access block devices are accessed randomly, while character devices are accessed serially. For example, seeking to a certain position in a file might be supported for a DVD but makes no sense for a pointing device such as a mouse.

Network devices are dealt with differently from block and character devices. Users cannot directly transfer data to network devices. Instead, they must communicate indirectly by opening a connection to the kernel's networking subsystem.

Block Devices

Block devices provide the main interface to all disk devices in a system. Performance is particularly important for disks, and the block-device system must provide functionality to ensure that disk access is as fast as possible. This functionality is achieved through the scheduling of I/O operations.

In the context of block devices, a block represents the unit with which the kernel performs I/O. When a block is read into memory, it is stored in a buffer. The **request manager** is the layer of software that manages the reading and writing of buffer contents to and from a block-device driver.

A separate list of requests is kept for each block-device driver. Traditionally, these requests have been scheduled according to a unidirectional-elevator (C-SCAN) algorithm that exploits the order in which requests are inserted in and removed from the lists. The request lists are maintained in sorted order of increasing starting-sector number. When a request is accepted for processing by a block-device driver, it is not removed from the list. It is removed only after the I/O is complete, at which point the driver continues with the next request in the list, even if new requests have been inserted in the list before the active request. As new I/O requests are made, the request manager attempts to merge requests in the lists.

Linux kernel version 2.6 introduced a new I/O scheduling algorithm. Although a simple elevator algorithm remains available, the default I/O scheduler is now the **Completely Fair Queueing (CFQ)** scheduler. The CFQ I/O scheduler is fundamentally different from elevator-based algorithms. Instead of sorting requests into a list, CFQ maintains a set of lists by default, one for each process. Requests originating from a process go in that process's list. For example, if two processes are issuing I/O requests, CFQ will maintain two separate lists of requests, one for each process. The lists are maintained according to the C-SCAN algorithm.

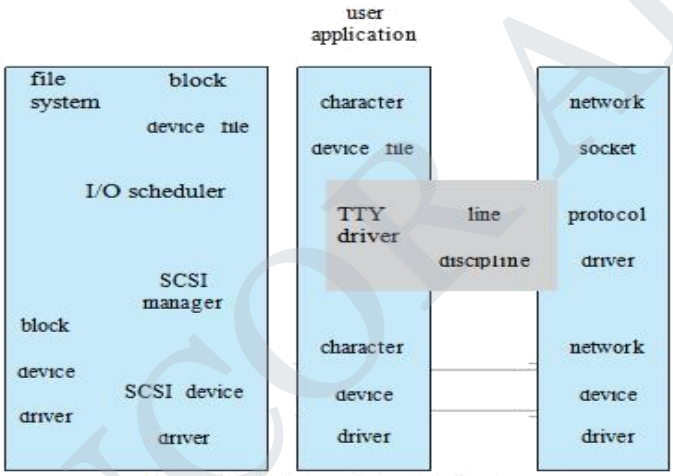


Figure 5.8 Device-driver block structure.

Character Devices

A character-device driver can be almost any device driver that does not offer random access to fixed blocks of data. Any character-device drivers registered to the Linux kernel must also register a set of functions that implement the file I/O operations that the driver can handle. The kernel performs almost no preprocessing of a file read or write request to a character device. It simply passes the request to the device in question and lets the device deal with the request.

The main exception to this rule is the special subset of character-device drivers that implement terminal devices. The kernel maintains a standard interface to these drivers by means of a set of tty struct structures. Each of these structures provides buffering and flow control on the data stream from the terminal device and feeds those data to a line discipline.

A **line discipline** is an interpreter for the information from the terminal device. The most common line discipline is the tty discipline, which glues the terminal's data stream onto the standard input and output streams of a user's running processes, allowing those processes to communicate directly with the user's terminal. This job is complicated by the fact that several such processes may be running simultaneously, and the tty line discipline is responsible for attaching and detaching the terminal's input and output from the various processes connected to it as those processes are suspended or awakened by the user.

Other line disciplines also are implemented that have nothing to do with I/O to a user process. The PPP and SLIP networking protocols are ways of encoding a networking connection over a terminal device such as a serial line. These protocols are implemented under Linux as drivers that at one end appear to the terminal system as line disciplines and at the other end appear to the networking system as network-device drivers. After one of these line

disciplines has been enabled on a terminal device, any data appearing on that terminal will be routed directly to the appropriate network-device driver.

Interprocess Communication

Linux provides a rich environment for processes to communicate with each other. Communication may be just a matter of letting another process know that some event has occurred, or it may involve transferring data from one process to another.

Synchronization and Signals

The standard Linux mechanism for informing a process that an event has occurred is the **signal**. Signals can be sent from any process to any other process, with restrictions on signals sent to processes owned by another user. However, a limited number of signals are available, and they cannot carry information. Only the fact that a signal has occurred is available to a process. Signals are not generated only by processes. The kernel also generates signals internally. For example, it can send a signal to a server process when data arrive on a network channel, to a parent process when a child terminates, or to a waiting process when a timer expires.

Internally, the Linux kernel does not use signals to communicate with processes running in kernel mode. If a kernel-mode process is expecting an event to occur, it will not use signals to receive notification of that event. Rather, communication about incoming asynchronous events within the kernel takes place through the use of scheduling states and wait queue structures. These mechanisms allow kernel-mode processes to inform one another about relevant events, and they also allow events to be generated by device drivers or by the networking system. Whenever a process wants to wait for some event to complete, it places itself on a wait queue associated with that event and tells the scheduler that it is no longer eligible for execution. Once the event has completed, every process on the wait queue will be awoken. This procedure allows multiple processes to wait for a single event. For example, if several processes are trying to read a file from a disk, then they will all be awakened once the data have been read into memory successfully.

Passing of Data among Processes

Linux offers several mechanisms for passing data among processes. The stan-dard UNIX **pipe** mechanism allows a child process to inherit a communication channel from its parent; data written to one end of the pipe can be read at the other. Under Linux, pipes appear as just another type of inode to virtual file system software, and each pipe has a pair of wait queues to synchronize the reader and writer. UNIX also defines a set of networking facilities that can send streams of data to both local and remote processes. Another process communications method, shared memory, offers an extremely fast way to communicate large or small amounts of data. Any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space. The main disadvantage of shared memory is that, on its own, it offers no synchronization. A process can neither ask the operating system whether a piece of shared memory has been written to nor suspend execution until such a write occurs. Shared memory becomes particularly powerful when used in conjunction with another interprocess-communication mechanism that provides the missing synchronization.

A shared-memory region in Linux is a persistent object that can be created or deleted by processes. Such an object is treated as though it were a small, independent address space. The Linux paging algorithms can elect to page shared-memory pages out to disk, just as they can page out a process's data pages. The shared-memory object acts as a backing store for shared-memory regions, just as a file can act as a backing store for a memory-mapped memory region. When a file is mapped into a virtual address space region, then any page faults that occur because the appropriate page of the file to be mapped into virtual memory. Similarly, shared-memory mappings direct page faults to map in pages from a persistent shared-memory object. Also just as for files, shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

Mobile OS - iOS and Android

In The Anatomy of an iPhone 4 we looked at the hardware that is contained within an iPhone 4 device. When we develop apps for the iPhone Apple does not allow us direct access to any of this hardware. In fact, all hardware interaction takes place exclusively through a number of different layers of software that act as intermediaries between the application code and device hardware. These layers make up what is known as an operating system. In the case of the iPhone, this operating system is known as iOS.

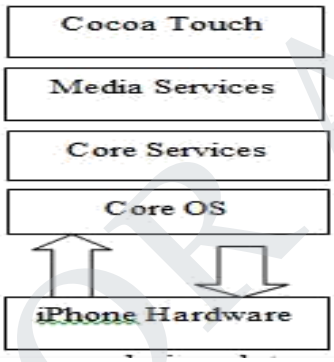
In order to gain a better understanding of the iPhone development environment, this chapter will look in detail at the different layers that comprise the iOS operating system and the frameworks that allow us, as developers, to write iPhone applications.

iPhone OS becomes iOS

Prior to the release of the first iPad in 2010, the operating system running on the iPhone was referred to as iPhone OS. Given that the operating system used for the iPad is essentially the same as that on the iPhone it didn't make much sense to name it iPad OS. Instead, Apple decided to adopt a more generic and non-device specific name for the operating system. Given Apple's predilection for names prefixed with the letter 'i' (iTunes, iBookstore, iMac etc) the logical choice was, of course, iOS. Unfortunately, iOS is also the name used by Cisco for the operating system on its routers (Apple, it seems, also has a predilection for ignoring trademarks). When performing an internet search for iOS, therefore, be prepared to see large numbers of results for Cisco's iOS which have absolutely nothing to do with Apple's iOS.

An Overview of the iOS 4 Architecture

As previously mentioned, iOS consists of a number of different software layers, each of which provides programming frameworks for the development of applications that run on top of the underlying hardware. These operating system layers can be presented diagrammatically as illustrated in the following figure:



some diagrams designed to graphically depict the iOS software stack show an additional box positioned above the Cocoa Touch layer to indicate the applications running on the device. In the above diagram we have not done so since this would suggest that the only interface available to the app is Cocoa Touch. In practice, an app can directly call down any of the layers of the stack to perform tasks on the physical device.

That said, however, each operating system layer provides an increasing level of abstraction away from the complexity of working with the hardware. As an iOS developer you should, therefore, always look for solutions to your programming goals in the frameworks located in the higher level iOS layers before resorting to writing code that reaches down to the lower level layers. In general, the higher level of layer you program to, the less effort and fewer lines of code you will have to write to achieve your objective. And as any veteran programmer will tell you, the less code you have to write the less opportunity you have to introduce bugs.

Now that we have identified the various layers that comprise iOS 4 we can now look in more detail at the services provided by each layer and the corresponding frameworks that make those services available to us as application developers.

The Cocoa Touch Layer

The Cocoa Touch layer sits at the top of the iOS stack and contains the frameworks that are most commonly used by iPhone application developers. Cocoa Touch is primarily written in Objective-C, is based on the standard Mac OS X Cocoa API (as found on Apple desktop and laptop computers) and has been extended and modified to meet the needs of the iPhone. The Cocoa Touch layer provides the following frameworks for iPhone app development:

UIKit Framework (UIKit.framework)

The UIKit framework is a vast and feature rich Objective-C based programming interface. It is, without question, the framework with which you will spend most of your time working. Entire books could, and probably will, be written about the UIKit framework alone. Some of the key features of UIKit are as follows:

User interface creation and management (text fields, buttons, labels, colors, fonts etc)

Application lifecycle management

Application event handling (e.g. touch screen user interaction)
Multitasking
Wireless Printing
Data protection via encryption
Cut, copy, and paste functionality
Web and text content presentation and management
Data handling
Inter-application integration
Push notification in conjunction with Push Notification Service
Local notifications (a mechanism whereby an application running in the background can gain the user's attention)
Accessibility
Accelerometer, battery, proximity sensor, camera and photo library interaction.
Touch screen gesture recognition
File sharing (the ability to make application files stored on the device available via iTunes)
Blue tooth based peer to peer connectivity between devices
Connection to external displays

Map Kit Framework (MapKit.framework)

If you have spent any appreciable time with an iPhone then the chances are you have needed to use the Maps application more than once, either to get a map of a specific area or to generate driving directions to get you to your intended destination. The Map Kit framework provides a programming interface that enables you to build map based capabilities into your own applications. This allows you to, amongst other things, display scrollable maps for any location, display the map corresponding to the current geographical location of the device and annotate the map in a variety of ways.

Push Notification Service

The Push Notification Service allows applications to notify users of an event even when the application is not currently running on the device. Since the introduction of this service it has most commonly been used by news based applications. Typically when there is breaking news the service will generate a message on the device with the news headline and provide the user the option to load the corresponding news app to read more details. This alert is typically accompanied by an audio alert and vibration of the device. This feature should be used sparingly to avoid annoying the user with frequent interruptions.

Message UI Framework (MessageUI.framework)

The Message UI framework provides everything you need to allow users to compose and send email messages from within your application. In fact, the framework even provides the user interface elements through which the user enters the email addressing information and message content. Alternatively, this information can be pre-defined within your application and then displayed for the user to edit and approve prior to sending.

Address Book UI Framework (AddressUI.framework)

Given that a key function of the iPhone is as a communications device and digital assistant it should not come as too much of a surprise that an entire framework is dedicated to the integration of the address book data into your own applications. The primary purpose of the framework is to enable you to access, display, edit and enter contact information from the iPhone address book from within your own application.

Game Kit Framework (GameKit.framework)

The Game Kit framework provides peer-to-peer connectivity and voice communication between multiple devices and users allowing those running the same app to interact. When this feature was first introduced it was anticipated by Apple that it would primarily be used in multi-player games (hence the choice of name) but the possible applications for this feature clearly extend far beyond games development.

iAd Framework (iAd.framework)

The purpose of the iAd Framework is to allow developers to include banner advertising within their applications. All advertisements are served by Apple's own ad service.

Event Kit UI Framework

The Event Kit UI framework was introduced in iOS 4 and is provided to allow the calendar events to be accessed and edited from within an application.

The iOS Media Layer

The role of the Media layer is to provide iOS with audio, video, animation and graphics capabilities. As with the other layers comprising the iOS stack, the Media layer comprises a number of frameworks that may be utilized when developing iPhone apps. In this section we will look at each one in turn.

Core Video Framework (CoreVideo.framework)

A new framework introduced with iOS 4 to provide buffering support for the Core Media framework. Whilst this may be utilized by application developers it is typically not necessary to use this framework.

Core Text Framework (CoreText.framework)

The iOS Core Text framework is a C-based API designed to ease the handling of advanced text layout and font rendering requirements.

Image I/O Framework (ImageIO.framework)

The Image IO framework, the purpose of which is to facilitate the importing and exporting of image data and image metadata, was introduced in iOS 4. The framework supports a wide range of image formats including PNG, JPEG, TIFF and GIF.

Assets Library Framework (AssetsLibrary.framework)

The Assets Library provides a mechanism for locating and retrieving video and photo files located on the iPhone device. In addition to accessing existing images and videos, this framework also allows new photos and videos to be saved to the standard device photo album.

Core Graphics Framework (CoreGraphics.framework)

The iOS Core Graphics Framework (otherwise known as the Quartz 2D API) provides a lightweight two dimensional rendering engine. Features of this framework include PDF document creation and presentation, vector based drawing, transparent layers, path based drawing, anti-aliased rendering, color manipulation and management, image rendering and gradients. Those familiar with the Quartz 2D API running on MacOS X will be pleased to learn that the implementation of this API is the same on iOS.

Quartz Core Framework (QuartzCore.framework)

The purpose of the Quartz Core framework is to provide animation capabilities on the iPhone. It provides the foundation for the majority of the visual effects and animation used by the UIKit framework and provides an Objective-C based programming interface for creation of specialized animation within iPhone apps.

OpenGL ES framework (OpenGLES.framework)

For many years the industry standard for high performance 2D and 3D graphics drawing has been OpenGL. Originally developed by the now defunct Silicon Graphics, Inc (SGI) during the 1990s in the form of GL, the open version of this technology (OpenGL) is now under the care of a non-profit consortium comprising a number of major companies including Apple, Inc., Intel, Motorola and ARM Holdings.

OpenGL for Embedded Systems (ES) is a lightweight version of the full OpenGL specification designed specifically for smaller devices such as the iPhone.

iOS 3 or later supports both OpenGL ES 1.1 and 2.0 on certain iPhone models (such as the iPhone 3GS and iPhone 4). Earlier versions of iOS and older device models support only OpenGL ES version 1.1.

iOS Audio Support

iOS is capable of supporting audio in AAC, Apple Lossless (ALAC), A-law, IMA/ADPCM, Linear PCM, µ-law, DVI/Intel IMA ADPCM, Microsoft GSM 6.10 and AES3-2003 formats through the support provided by the following frameworks.

AV Foundation framework (AVFoundation.framework)

An Objective-C based framework designed to allow the playback, recording and management of audio content.

Core Audio Frameworks (CoreAudio.framework, AudioToolbox.framework and AudioUnit.framework)

The frameworks that comprise Core Audio for iOS define supported audio types, playback and recording of audio files and streams and also provide access to the device's built-in audio processing units.

Open Audio Library (OpenAL)

OpenAL is a cross platform technology used to provide high-quality, 3D audio effects (also referred to as positional audio). Positional audio can be used in a variety of applications though is typically using to provide sound effects in games.

Media Player framework (MediaPlayer.framework)

The iOS Media Player framework is able to play video in .mov, .mp4, .m4v, and .3gp formats at a variety of compression standards, resolutions and frame rates.

Core Midi Framework (CoreMIDI.framework)

Introduced in iOS 4, the Core MIDI framework provides an API for applications to interact with MIDI compliant devices such as synthesizers and keyboards via the iPhone's dock connector.

The iOS Core Services Layer

<google>IOSBOX</google> The iOS Core Services layer provides much of the foundation on which the previously referenced layers are built and consists of the following frameworks.

Address Book framework (AddressBook.framework)

The Address Book framework provides programmatic access to the iPhone Address Book contact database allowing applications to retrieve and modify contact entries.

CFNetwork Framework (CFNetwork.framework)

The CFNetwork framework provides a C-based interface to the TCP/IP networking protocol stack and low level access to BSD sockets. This enables application code to be written that works with HTTP, FTP and Domain Name servers and to establish secure and encrypted connections using Secure Sockets Layer (SSL) or Transport Layer Security (TLS).

Core Data Framework (CoreData.framework)

This framework is provided to ease the creation of data modeling and storage in Model-View-Controller (MVC) based applications. Use of the Core Data framework significantly reduces the amount of code that needs to be written to perform common tasks when working with structured data in an application.

Core Foundation Framework (CoreFoundation.framework)

The Core Foundation is a C-based Framework that provides basic functionality such as data types, string manipulation, raw block data management, URL manipulation, threads and run loops, date and times, basic XML manipulation and port and socket communication. Additional XML capabilities beyond those included with this framework are provided via the libXML2 library. Though this is a C-based interface, most of the capabilities of the Core Foundation framework are also available with Objective-C wrappers via the Foundation Framework.

Core Media Framework (CoreMedia.framework)

The Core Media framework is the lower level foundation upon which the AV Foundation layer is built. Whilst most audio and video tasks can, and indeed should, be performed using the higher level AV Foundation framework, access is also provided for situations where lower level control is required by the iOS application developer.

Core Telephony Framework (CoreTelephony.framework)

The iOS Core Telephony framework is provided to allow applications to interrogate the device for information about the current cell phone service provider and to receive notification of telephony related events.

EventKit Framework (EventKit.framework)

An API designed to provide applications with access to the calendar and alarms on the device.

Foundation Framework (Foundation.framework)

The Foundation framework is the standard Objective-C framework that will be familiar to those that have programmed in Objective-C on other platforms (most likely Mac OS X). Essentially, this consists of Objective-C wrappers around much of the C-based Core Foundation Framework.

Core Location Framework (CoreLocation.framework)

The Core Location framework allows you to obtain the current geographical location of the device (latitude and longitude) and compass readings from with your own applications. The method used by the device to provide coordinates will depend on the data available at the time the information is requested and the hardware support provided by the particular iPhone model on which the app is running (GPS and compass are only featured on recent models). This will either be based on GPS readings, Wi-Fi network data or cell tower triangulation (or some combination of the three).

Mobile Core Services Framework (MobileCoreServices.framework)

The iOS Mobile Core Services framework provides the foundation for Apple's Uniform Type Identifiers (UTI) mechanism, a system for specifying and identifying data types. A vast range of predefined identifiers have been defined by Apple including such diverse data types as text, RTF, HTML, JavaScript, PowerPoint .ppt files, PhotoShop images and MP3 files.

Store Kit Framework (StoreKit.framework)

The purpose of the Store Kit framework is to facilitate commerce transactions between your application and the Apple App Store. Prior to version 3.0 of iOS, it was only possible to charge a customer for an app at the point that they purchased it from the App Store. iOS 3.0 introduced the concept of the "in app purchase" whereby the user can be given the option make additional payments from within the application. This might, for example, involve implementing a subscription model for an application, purchasing additional functionality or even buying a faster car for you to drive in a racing game.

SQLite library

Allows for a lightweight, SQL based database to be created and manipulated from within your iPhone application.

System Configuration Framework (SystemConfiguration.framework)

The System Configuration framework allows applications to access the network configuration settings of the device to establish information about the "reachability" of the device (for example whether Wi-Fi or cell connectivity is active and whether and how traffic can be routed to a server).

Quick Look Framework (QuickLook.framework)

One of the many new additions included in iOS 4, the Quick Look framework provides a useful mechanism for displaying previews of the contents of files types loaded onto the device (typically via an internet or network connection) for which the application does not already provide support. File format types supported by this framework include iWork, Microsoft Office document, Rich Text Format, Adobe PDF, Image files, public.text files and comma separated (CSV).

The iOS Core OS Layer

The Core OS Layer occupies the bottom position of the iOS stack and, as such, sits directly on top of the device hardware. The layer provides a variety of services including low level networking, access to external accessories and the usual fundamental operating system services such as memory management, file system handling and threads.

Accelerate Framework (Accelerate.framework)

Introduced with iOS 4, the Accelerate Framework provides a hardware optimized C-based API for performing complex and large number math, vector, digital signal processing (DSP) and image processing tasks and calculations.

External Accessory framework (ExternalAccessory.framework)

Provides the ability to interrogate and communicate with external accessories connected physically to the iPhone via the 30-pin dock connector or wirelessly via Bluetooth.

Security Framework (Security.framework)

The iOS Security framework provides all the security interfaces you would expect to find on a device that can connect to external networks including certificates, public and private keys, trust policies, keychains, encryption, digests and Hash-based Message Authentication Code (HMAC).

System (LibSystem)

As we have previously mentioned, the iOS is built upon a UNIX-like foundation. The System component of the Core OS Layer provides much the same functionality as any other UNIX like operating system. This layer includes the operating system kernel (based on the Mach kernel developed by Carnegie Mellon University) and device drivers. The kernel is the foundation on which the entire iOS is built and provides the low level interface to the underlying hardware. Amongst other things the kernel is responsible for memory allocation, process lifecycle management, input/output, inter-process communication, thread management, low level networking, file system access and thread management.

As an app developer your access to the System interfaces is restricted for security and stability reasons. Those interfaces that are available to you are contained in a C-based library called LibSystem. As with all other layers of the iOS stack, these interfaces should be used only when you are absolutely certain there is no way to achieve the same objective using a framework located in a higher iOS layer.