CS8493- Operating System

UNIT-I

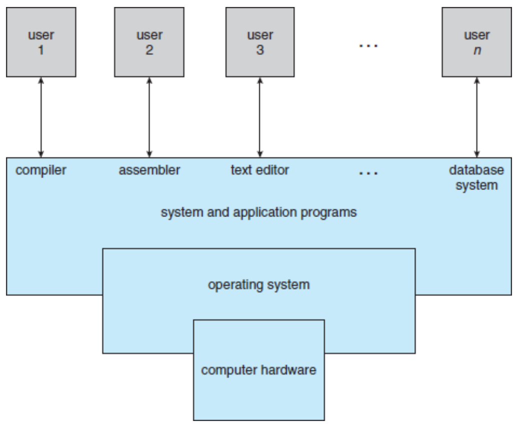
1.1 introduction

An *operating system* acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.

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. An amazing aspect of operating systems is how they vary in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be ***convenient,*** others to be ***efficient,*** and others to be some combination of the two.

1.2 Computer system overview

A computer system can be divided roughly into four components: the ***hardware,*** the ***operating system,*** the ***application programs,*** and the ***users*** (Figure 1.1). The **hardware**—the **central processing unit (CPU)**, the **memory**, and the **input/output (I/O) devices**—provides the basic computing resources for the system. The **application programs**—such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.



1.1.1 User View

The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**—how various hardware and software resources are shared.

Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.

1.1.2 System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer.

1.3 Basic elements

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 (Figure 1.2). Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, or video displays). The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller synchronizes access to the memory.

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial program, or **bootstrap program**, tends to be simple. Typically, 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**. It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory. Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel, by system programs that are loaded into memory at boot time to become **system processes**, or **system daemons** that run the entire time the kernel is running.

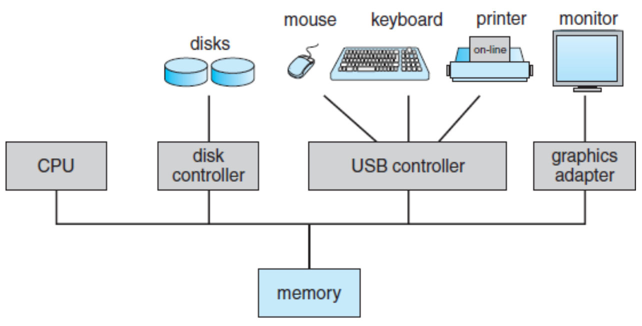


Figure 1.2 A modern computer system.

1.4 Instruction Execution and Interrupts

The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus. Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call**).

When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed location usually contains the starting address where the service routine for the interrupt is located. The interrupt service routine executes; on completion, the CPU resumes the
Interrupted computation. A timeline of this operation is shown in Figure 1.3.

Interrupts are an important part of computer architecture. Each computer design has its own interrupt mechanism, but several functions are common. The interrupt must transfer control to the appropriate interrupt service routine. The straightforward method for handling this transfer would be to invoke a generic routine to examine the interrupt information. The routine, in turn, would call the interrupt-specific handler. However, interrupts must be handled quickly. Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, the table of pointers is stored in low memory (the first hundred or so locations). These locations hold the addresses of the interrupt service routines for the various devices. This array, or **interrupt vector**, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.

Chapter 1 Introduction

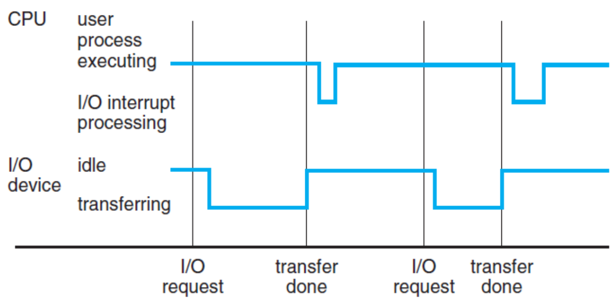


Figure 1.3 Interrupt timeline for a single process doing output.

1.5 Memory Hierarchy

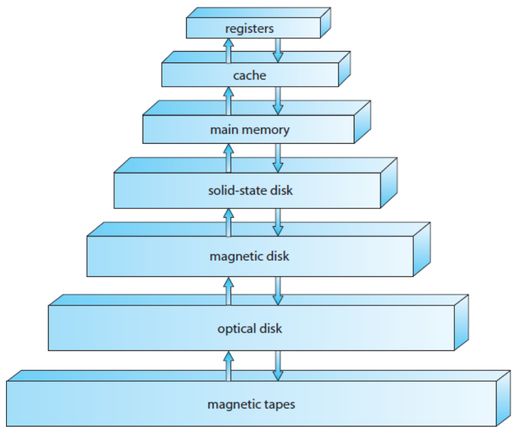


Figure 1.4 Storage-device hierarchy.

The wide variety of storage systems can be organized in a hierarchy (Figure 1.4) 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. This trade-off is reasonable; if a given storage system were both faster and less expensive than another—other properties being the same—then there would be no reason to use the slower, more expensive memory. In fact, many early storage devices, including paper tape and core memories, are relegated to museums now that magnetic tape and **semiconductor memory** have become faster and cheaper. The top four levels of memory in Figure 1.4 may be constructed using semiconductor memory.

In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile. As mentioned earlier, **volatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. In the hierarchy shown in Figure 1.4, the storage systems above the solid-state disk are volatile, whereas those including the solid-state disk and below are nonvolatile.

Solid-state disks have several variants but in general are faster than magnetic disks and are nonvolatile. One type of solid-state disk stores data in a large DRAM array during normal operation but also contains a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, this solid-state disk's controller copies the data from RAM to the magnetic disk. When external power is restored, the controller copies the data back into RAM. Another form of solid-state disk is flash memory, which is popular in cameras and **personal digital assistants (PDAs)**, in robots, and increasingly for storage on general-purpose computers. Flash memory is slower than DRAM but needs no power to retain its contents. Another form of nonvolatile storage is **NVRAM**, which is DRAM with battery backup power. This memory can be as fast as DRAM and (as long as the battery lasts) is nonvolatile.

1.6 Cache Memory

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

1.7 Direct Memory Access

Interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for bulk data movement such as disk I/O. To solve this problem, **direct memory access (DMA)** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed, rather than the one interrupt per byte generated for low-speed devices. While the device controller is performing these operations, the CPU is available to accomplish other work.

Some high-end systems use switch rather than bus architecture. On these systems, multiple components can talk to other components concurrently, rather than competing for cycles on a shared bus. In this case, DMA is even more effective. Figure 1.5 shows the interplay of all components of a computer system.

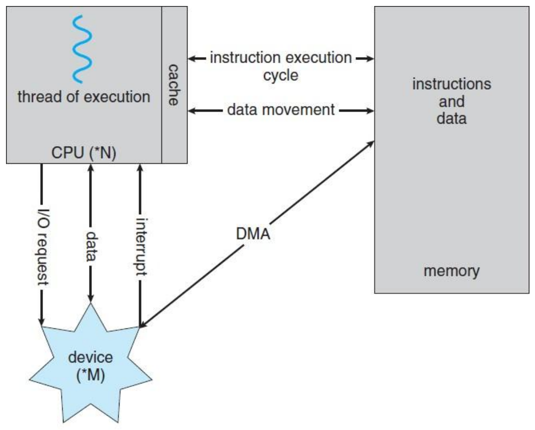


Figure 1.5 How a modern computer system works.

1.7 Multiprocessor and Multicore Organization

1.7.1 Multiprocessor Organization

Within the past several years, **multiprocessor systems** (also known as **parallel systems** or **multicore systems**) have begun to dominate the landscape of computing. Such systems have two or more processors in close communication, sharing the computer bus and sometimes the clock, memory, and peripheral devices. Multiprocessor systems first appeared prominently appeared in servers and have since migrated to desktop and laptop systems. Recently, multiple processors have appeared on mobile devices such as smart phones and tablet computers. Multiprocessor systems have three main advantages:

1. Increased throughput. By increasing the number of processors, we expect to get more work done in less time. The speed-up ratio with *N* processors is not *N*, however; rather, it is less than *N*. When multiple processors cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly. This overhead, plus contention for shared resources, lowers the expected gain from additional processors. Similarly, *N* programmers working closely together do not produce *N* times the amount of work a single programmer would produce.

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 runs only 10 percent slower, rather than failing altogether.

The multiple-processor systems in use today are of two types. Some systems use **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. The most common systems use **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. Figure 1.6 illustrates a typical SMP architecture.

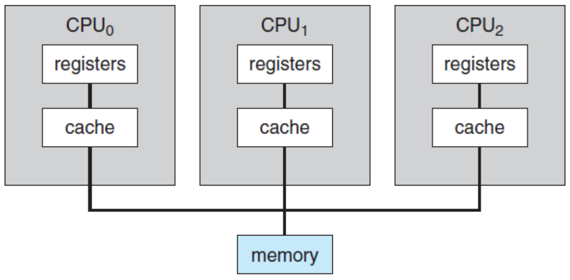


Figure 1.6 Symmetric multiprocessing architecture.

Multiprocessing adds CPUs to increase computing power. If the CPU has an integrated memory controller, then adding CPUs can also increase the amount of memory addressable in the system. Either way, multiprocessing can cause a system to change its memory access model from uniform memory access (**UMA**) to non-uniform memory access (**NUMA**). UMA is defined as the situation in which access to any RAM from any CPU takes the same amount of time. With NUMA, some parts of memory may take longer to access than other parts, creating a performance penalty. Operating systems can minimize the NUMA penalty through resource management

1.7.2 Multicore Organization

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating system (Section 1.3.2). Whether the cores appear across CPU chips or within CPU chips, we call these systems **multicore** or **multiprocessor** systems. Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), 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 (Figure 4.4).

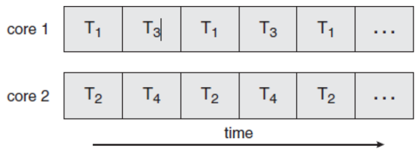


Figure 4.4 Parallel execution on a multicore system.

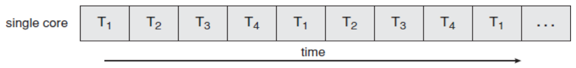
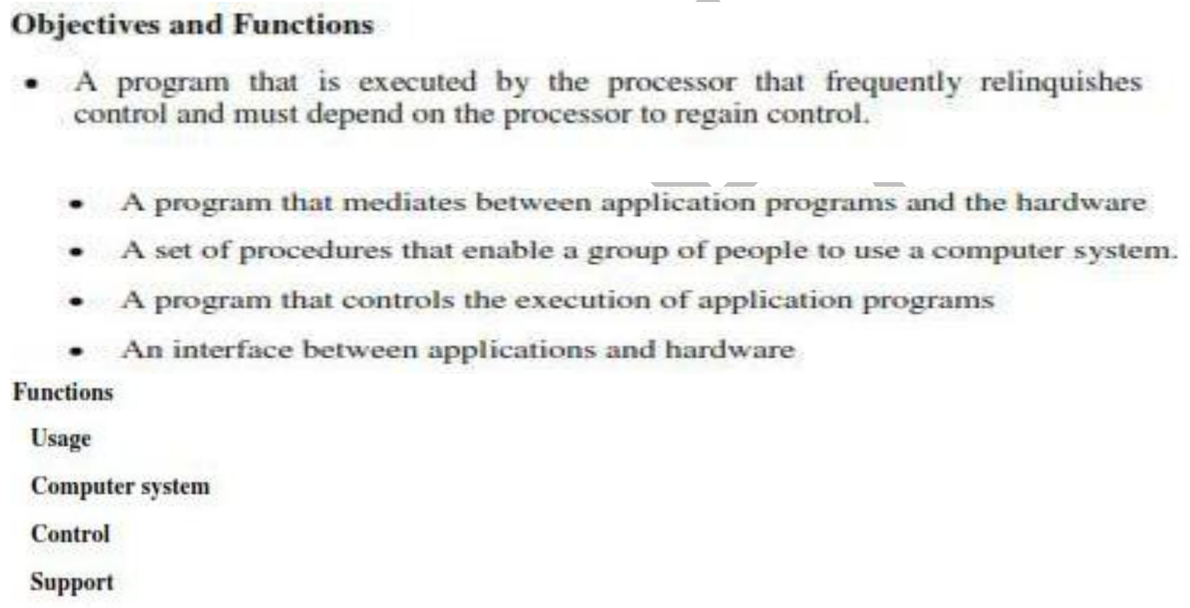


Figure 4.3 Concurrent execution on a single-core system.

Notice the distinction between ***parallelism*** and ***concurrency*** in this discussion. A system is parallel if it can perform more than one task simultaneously. In contrast, 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. Before the advent of SMP and multicore architectures, most computer systems had only a single processor. CPU schedulers were designed to provide the illusion of parallelism by rapidly switching between processes in the system, thereby allowing each process to make progress. Such processes were running concurrently, but not in parallel.

As systems have grown from tens of threads to thousands of threads, CPU designers have improved system performance by adding hardware to improve thread performance. Modern Intel CPUs frequently support two threads per core, while the Oracle T4 CPU supports eight threads per core. This support means that multiple threads can be loaded into the core for fast switching. Multicore computers will no doubt continue to increase in core counts and hardware thread support.

1.8 Operating system overview-objectives and functions.



Objectives and Functions

- A program that is executed by the processor that frequently relinquishes control and must depend on the processor to regain control.

- A program that mediates between application programs and the hardware
- A set of procedures that enable a group of people to use a computer system.
- A program that controls the execution of application programs
- An interface between applications and hardware

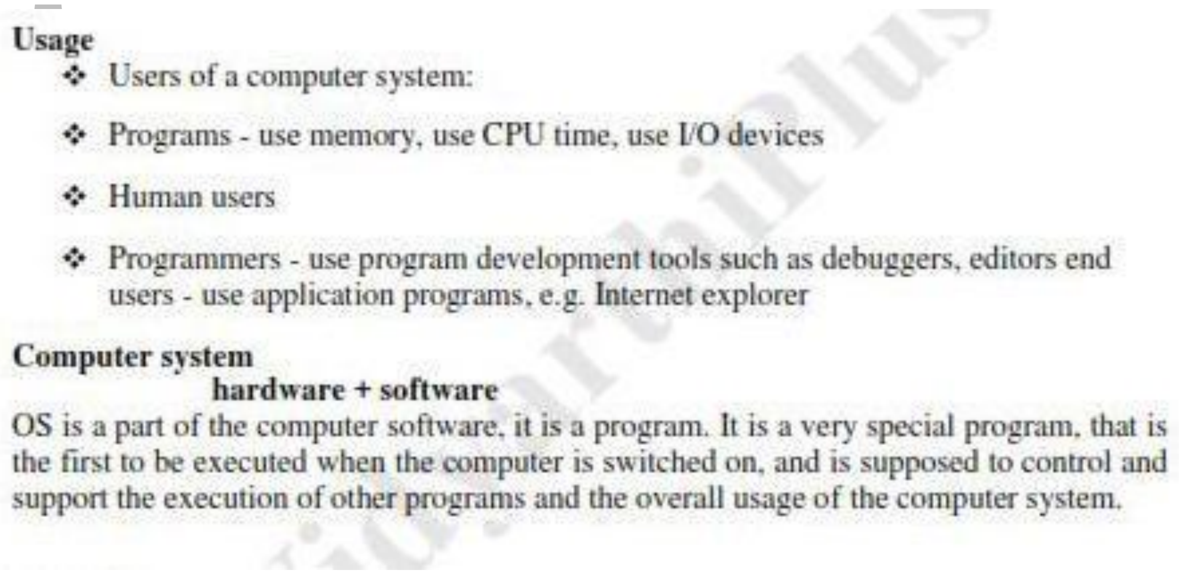
Functions

Usage

Computer system

Control

Support



Usage

- ❖ Users of a computer system:
- ❖ Programs - use memory, use CPU time, use I/O devices
- ❖ Human users
- ❖ Programmers - use program development tools such as debuggers, editors end users - use application programs, e.g. Internet explorer

Computer system

hardware + software

OS is a part of the computer software, it is a program. It is a very special program, that is the first to be executed when the computer is switched on, and is supposed to control and support the execution of other programs and the overall usage of the computer system.

Control
The operating system controls the usage of the computer resources - hardware devices and software utilities. We can think of an operating system as a ***Resource Manager.*** Here are some of the resources managed by the OS:

- Processors,
- Main memory,
- Secondary Memory,
- Peripheral devices,
- Information.

Support

- ✓ The operating system provides a number of services to assist the users of the computer system:

For the programmers:

Utilities - debuggers, editors, file management, etc.

For the end users - provides the interface to the application programs

For programs - loads instructions and data into memory, prepares I/O devises for usage, handles interrupts and error conditions.

1.9 Evolution of Operating System

1.9. **Serial Processing** - 1940's – 1950's programmer interacted directly with hardware. No operating system.

Problems

Scheduling - users sign up for machine time. Wasted computing time

Setup Time- Setup included loading the compiler, source program, saving compiled program, and loading and linking. If an error occurred - start over.

1.9.2 Simple Batch Systems

Improve the utilization of computers.

Jobs were submitted on cards or tape to an operator who batches jobs together sequentially. The program that controls the execution of the jobs was called **monitor** - a simple version of an operating system. The interface to the monitor was accomplished through Job Control Language (JCL). For example, a JCL request could be to run the compiler for a particular programming language, then to link and load the program, then to run the user program.

Hardware features:

Memory protection: do not allow the memory area containing the monitor to be altered

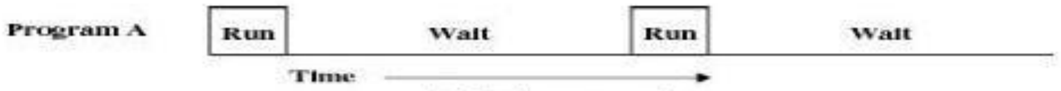
Timer: prevents a job from monopolizing the system

Problems:

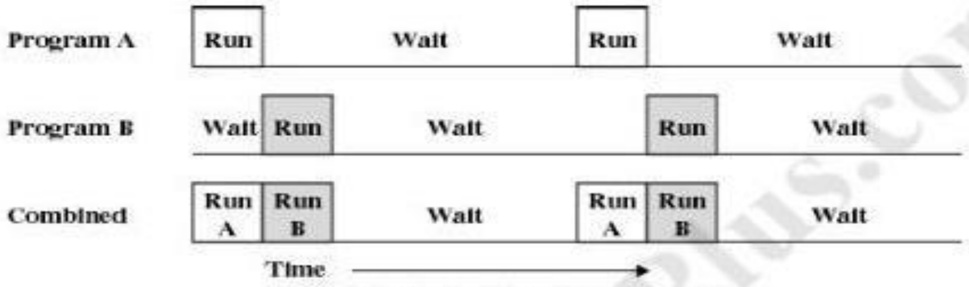
Bad utilization of CPU time - the processor stays idle while I/O devices are in use.

1.9.3 Multiprogrammed Batch Systems

More than one program resides in the main memory. While a program A uses an I/O device the processor does not stay idle, instead it runs another program B.



(a) Uniprogramming



(b) Multiprogramming with two programs

New features:

Memory management - to have several jobs ready to run, they must be kept in main memory

Job scheduling - the processor must decide which program to run.

1.9.4 Time-Sharing Systems

Multiprogramming systems: Several programs use the computer system.

Time-sharing systems: Several (human) users use the computer system interactively.

Characteristics:

- Using multiprogramming to handle multiple interactive jobs
- Processor's time is shared among multiple users
- Multiple users simultaneously access the system through terminals

1.9.5 Operating-System Services

The OS provides certain services to programs and to the users of those programs.

1. **Program execution:**

   The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

2. **I/O operations:**

   A running program may require I/O. This I/O may involve a file or an I/O device.

3. **File-system manipulation:**

   The program needs to read, write, create and delete files.

4. **Communications :**

   In many circumstances, one process needs to exchange information with another process. Such communication can occur in two major ways. The first takes place between processes that are executing on the same computer; the second takes place between processes that are executing on different computer systems that are tied together by a computer network.

5. **Error detection:**

   The operating system constantly needs to be aware of possible errors. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on tape, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing.

6. **Resource allocation:**

Different types of resources are managed by the Os.

When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them.

7. **Accounting:**

We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.

8. **Protection:**

The owners of information stored in a multiuser computer system may want to control use of that information. Security of the system is also important.

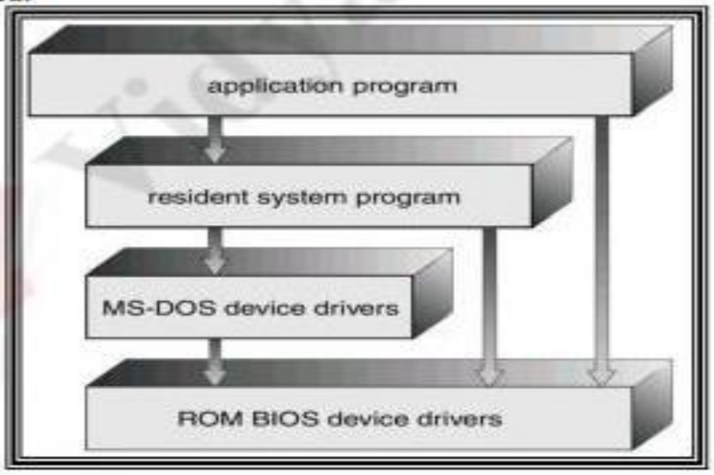
1.10 Computer System Organization

1.10.1 Operating System Structure and Operations

1.10. **Operating System Structure**

1.10.1.1 **MS-DOS System Structure**

- ✓ MS-DOS – written to provide the most functionality in the least space.
- ✓ Not divided into modules.
- ✓ Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated.



1.10.1.2 **Unix System Structure**

- ➢ **UNIX** – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts.

- **Systems programs** – use kernel supported system calls to provide useful functions such as compilation and file manipulation.
- **The kernel** - Consists of everything below the system-call interface and above the physical hardware

1.10.1.3 Layered Approach

- ✓ The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- ✓ An OS layer is an implementation of an abstract object that is the encapsulation of data and operations that can manipulate those data. These operations (routines) can be invoked by higher-level layers. The layer itself can invoke operations on lower-level layers.
- ✓ Layered approach provides modularity. With modularity, layers are selected such that each layer uses functions (operations) and services of only lower-level layers.
- ✓ Each layer is implemented by using only those operations that are provided lower level layers.
- ✓ The major difficulty is appropriate definition of various layers.

1.10.1.4 Microkernel System Structure

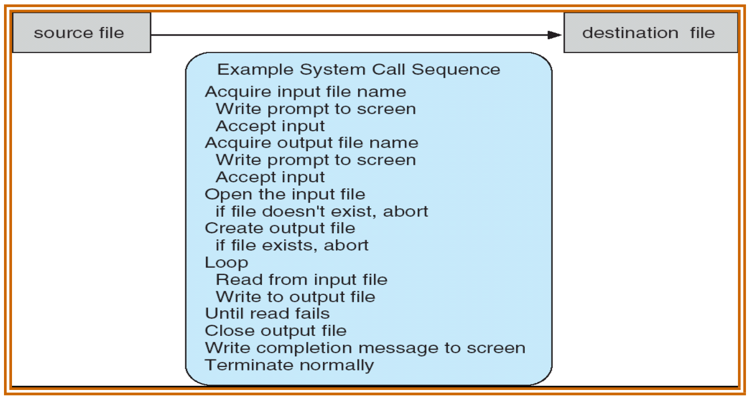
- ✓ Moves as much from the kernel into "user" space.
- ✓ Communication takes place between user modules using message passing.
  - ❖ Benefits:
    - Easier to extend a microkernel
    - Easier to port the operating system to new architectures

1.10. Operating-System Operations

- ✓ If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signaled by the occurrence of an interrupt or a trap.
- ✓ A trap (or an exception) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure.
- ✓ Without protection against these sorts of errors, either the computer must execute only one process at a time or all output must be suspect. A properly designed operating system must ensure that an incorrect (or malicious) program cannot cause other programs to execute incorrectly.

✓ **System Calls**

- **Programming interface to the services provided by the OS**
- **Typically written in a high-level language (C or C++)**
- **Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use**
- **Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)**
- **System call sequence to copy the contents of one file to another file**



1.11.1 System Call Implementation

- **Typically, a number associated with each system call**

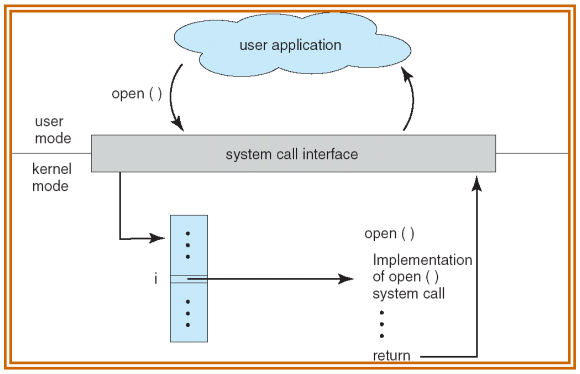
✓ **System-call interface maintains a table indexed according to these numbers**

- **The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values**
- **The caller need know nothing about how the system call is implemented**

✓ **Just needs to obey API and understand what OS will do as a result call**

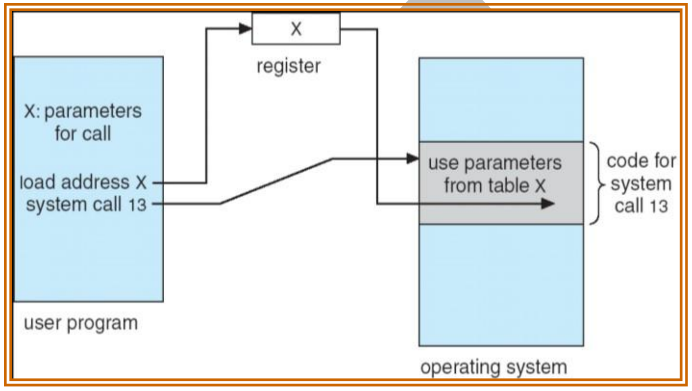
✓ **Most details of OS interface hidden from programmer by API**

✓ **Managed by run-time support library (set of functions built into libraries included with compiler)**



1.11.2 System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed



Types of System Calls

- **Process control**
- **File management**
- **Device management**
- **Information maintenance**
- **Communications**

1.12 System Programs

- System programs provide a convenient environment for program development and execution. The can be divided into:

  - File manipulation
  - Status information
  - File modification

    - Programming language support
    - Program loading and execution
    - Communications
    - Application programs

- Most users' view of the operation system is defined by system programs, not the actual system calls
- Provide a convenient environment for program development and execution
    - Some of them are simply user interfaces to system calls; others are considerably more complex
- File management - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- Status information
    - Some ask the system for info - date, time, amount of available memory, disk space, number of users
    - Others provide detailed performance, logging, and debugging information
    - Typically, these programs format and print the output to the terminal or other output devices
    - Some systems implement a registry - used to store and retrieve configuration information
- File modification
    - Text editors to create and modify files
    - Special commands to search contents of files or perform transformations of the text
- Programming-language support - Compilers, assemblers, debuggers and interpreters sometimes provided
- Program loading and execution- Absolute loaders, relocatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language
- Communications - Provide the mechanism for creating virtual connections among processes, users, and computer systems
    - Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another

1.12 OS Generation and System Boot.

1.12.1 OS Generation

Historically operating systems have been tightly related to the computer architecture, it is good idea to study the history of operating systems from the architecture of the computers on which they run.

Operating systems have evolved through a number of distinct phases or generations which corresponds roughly to the decades.

The 1940's - First Generations

The earliest electronic digital computers had no operating systems. Machines of the time were so primitive that programs were often entered one bit at time on rows of mechanical switches (plug boards). Programming languages were unknown (not even assembly languages). Operating systems were unheard of .

The 1950's - Second Generation

By the early 1950's, the routine had improved somewhat with the introduction of punch cards. The General Motors Research Laboratories implemented the first operating systems in early 1950's for their IBM 701. The system of the 50's generally ran one job at a time. These were called single-stream batch processing systems because programs and data were submitted in groups or batches.

The 1960's - Third Generation

The systems of the 1960's were also batch processing systems, but they were able to take better advantage of the computer's resources by running several jobs at once. So operating systems designers developed the concept of multiprogramming in which several jobs are in main memory at once; a processor is switched from job to job as needed to keep several jobs advancing while keeping the peripheral devices in use.

For example, on the system with no multiprogramming, when the current job paused to wait for other I/O operation to complete, the CPU simply sat idle until the I/O finished. The solution for this problem that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU.

Another major feature in third-generation operating system was the technique called spooling (simultaneous peripheral operations on line). In spooling, a high-speed device like a disk interposed between a running program and a low-speed device involved with the program in input/output. Instead of writing directly to a printer, for example, outputs are written to the disk. Programs can run to completion faster, and other programs can be initiated sooner when the printer becomes available, the outputs may be printed.

Note that spooling technique is much like thread being spun to a spool so that it may be later be unwound as needed.

Another feature present in this generation was time-sharing technique, a variant of multiprogramming technique, in which each user has an on-line (i.e., directly connected) terminal. Because the user is present and interacting with the computer, the computer system must respond quickly to user requests, otherwise user productivity could suffer. Timesharing systems were developed to multiprogram large number of simultaneous interactive users.

Fourth Generation

With the development of LSI (Large Scale Integration) circuits, chips, operating system entered in the system entered in the personal computer and the workstation age. Microprocessor technology evolved to the point that it become possible to build desktop computers as powerful as the mainframes of the 1970s. Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the large personal computers using the Motorola 6899 CPU family.

12.2 System Boot.

- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
  - When power initialized on system, execution starts at a fixed memory location
    - Firmware used to hold initial boot code

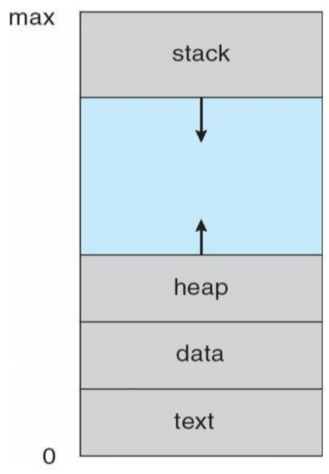
CS8493- Operating System

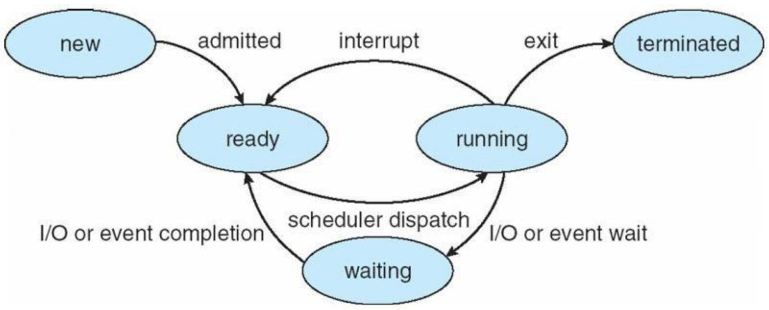
UNIT-II

PROCESS MANAGEMENT

Processes-Process Concept:

- An operating system executes a variety of programs:
    - Batch system – **—jobs”**
    - Time-shared systems – **—user programs”** or **—tasks”**
- We will use the terms ***job*** and ***process*** almost interchangeably
- **Process** – is a program in execution (informal definition)
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
    - Program becomes process when executable file loaded into memory
- Execution of program started via GUI, command line entry of its name, etc
- One program can be several processes
    - Consider multiple users executing the same program
- In memory, a process consists of **multiple parts:**
    - **Program code**, also called **text section**
    - **Current activity** including
        - **program counter**
        - processor registers
    - **Stack** containing temporary data
        - Function parameters return addresses, local variables
    - **Data section** containing global variables
    - **Heap** containing memory dynamically allocated during run time





- As a process executes, it changes **state**
    - **new**: The process is being created
    - **ready**: The process is waiting to be assigned to a processor
    - **running**: Instructions are being executed
    - **waiting**: The process is waiting for some event to occur
    - **terminated**: The process has finished execution

PROCESS CONTROL BLOCK (PCB)

Each process is represented in the operating system by a **process control block (PCB)**—also called a **task control block**. A PCBis shown in 3.3. It contains many pieces of information associated with a specific process, including these:

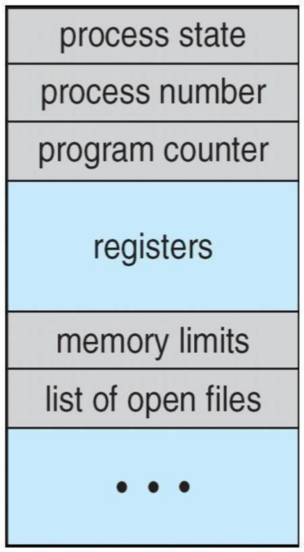


Figure 3.3 Process control block (PCB)

• **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, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure 3.4).
• **CPU-scheduling information**:This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

• **Memory-management information**: This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system

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

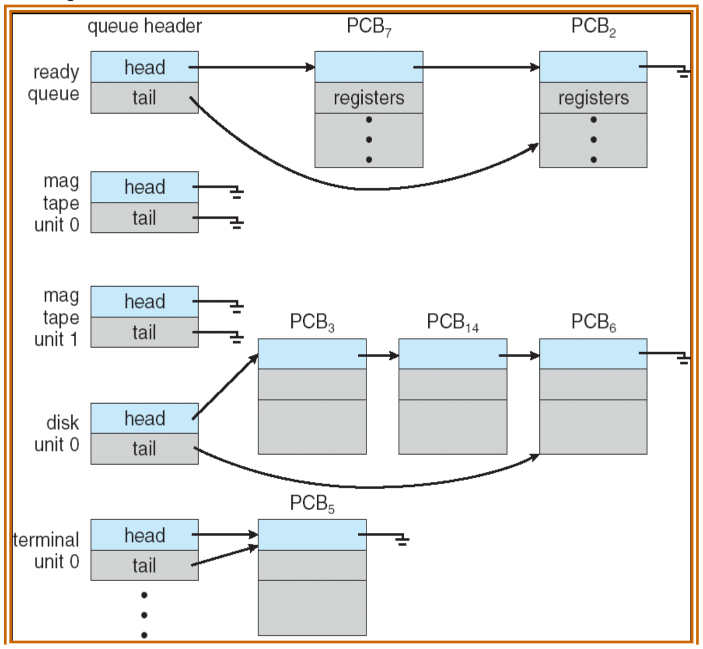
• **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

<u>Process Scheduling:</u>

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. the **process scheduler** selects an available process (possibly from a set of several available processes) 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.

Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device Processes migrate among the various queues.



A common representation of process scheduling is a **queueing diagram**. Two types of queues are present: the 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. 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:

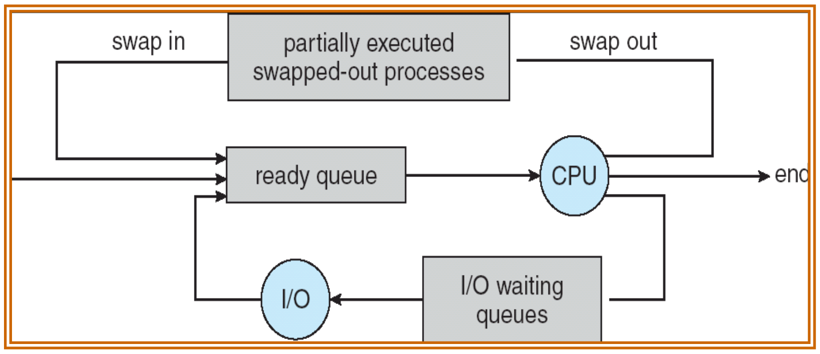
• The process could issue an I/O request and then be placed in an I/O queue.

• 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

- **Long-term scheduler (or job scheduler)** – selects which processes should be brought into the ready queue
- **Short-term scheduler (or CPU scheduler)** – selects which process should be executed next and allocates CPU



- Short-term scheduler is invoked very frequently (milliseconds) → (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) → (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming.

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

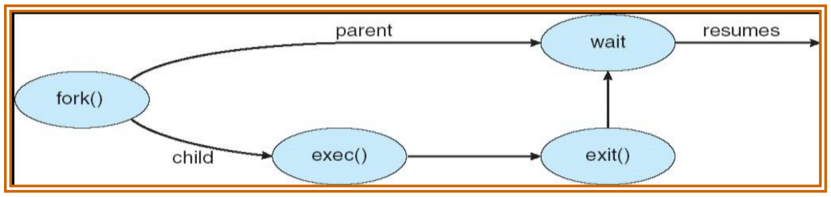
Operations on Processes

Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space

- Child duplicate of parent
- Child has a program loaded into it

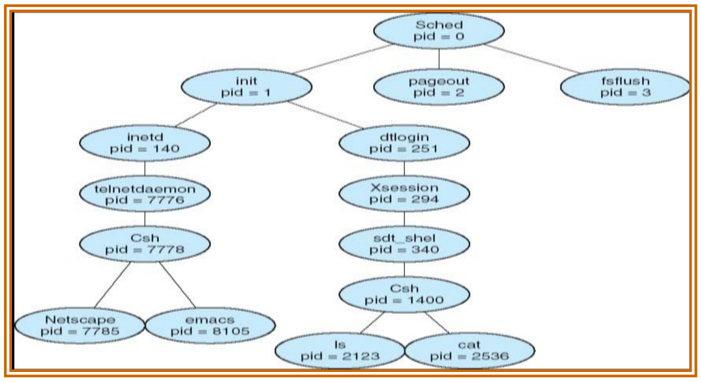
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



C Program Forking Separate Process

```
int main()
{
pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-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");
            exit(0);
        }
}
```

A tree of processes on a typical Solaris



Process Termination

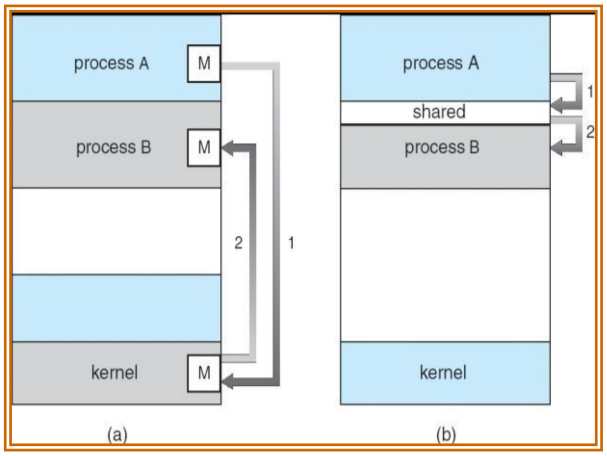
- Process executes last statement and asks the operating system to delete it (**exit**)
    - Output data from child to parent (via **wait**)
    - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - If parent is exiting
        - Some operating system do not allow child to continue if its parent terminates
            - All children terminated - *cascading termination*

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
    - Information sharing
    - Computation speed-up
    - Modularity
    - Convenience

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
    - **send**(*message*) – message size fixed or variable
    - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
    - establish a *communication link* between them
    - exchange messages via send/receive
- Implementation of communication link
    - physical (e.g., shared memory, hardware bus)
    - logical (e.g., logical properties)



Direct Communication

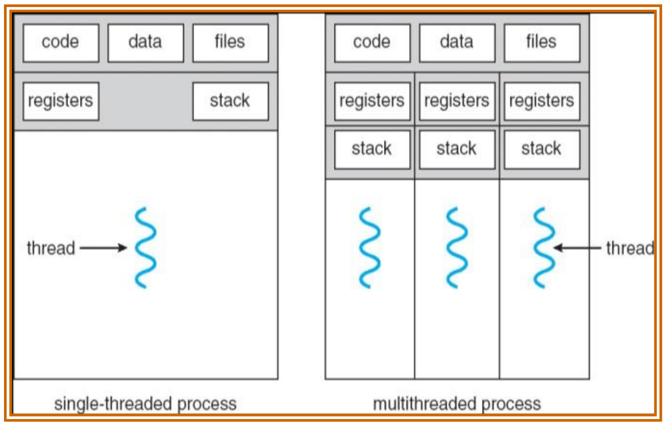
- Processes must name each other explicitly:
    - **send** (*P*, *message*) – send a message to process P
    - **receive**(*Q*, *message*) – receive a message from process Q
- Properties of communication link
    - Links are established automatically
    - A link is associated with exactly one pair of communicating processes
    - Between each pair there exists exactly one link
    - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
    - Each mailbox has a unique id
    - Processes can communicate only if they share a mailbox
- Properties of communication link
    - Link established only if processes share a common mailbox
    - A link may be associated with many processes
    - Each pair of processes may share several communication links
    - Link may be unidirectional or bi-directional

Threads- Overview

A thread is a basic unit of CPU utilization; 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 and signals. A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.



Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

2. **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Multicore Programming

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A more recent, similar trend in system design is to place multiple computing cores on a single chip. Each core appears as a separate processor to the operating Whether the cores appear across CPU chips or within CPU chips, we call these systems **multicore** or **multiprocessor** systems.

Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. 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,

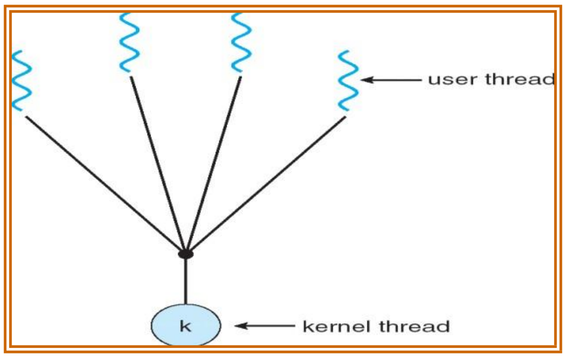
Concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core .Notice the distinction between ***parallelism*** and ***concurrency*** in this discussion. A system is parallel if it can perform more than one task simultaneously. In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress.

Multithreading Models

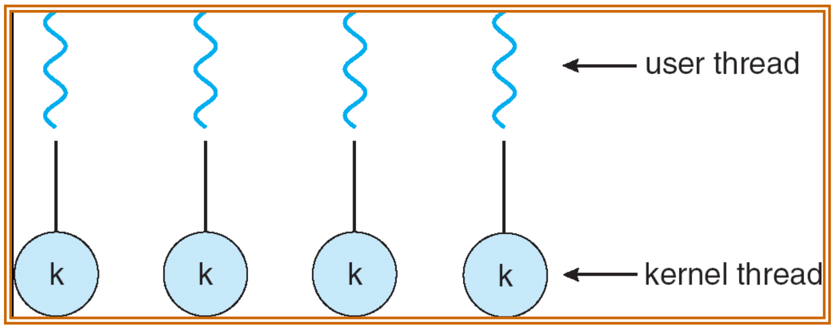
- Many-to-One
- One-to-One
- Many-to-Many

1. Many-to-One

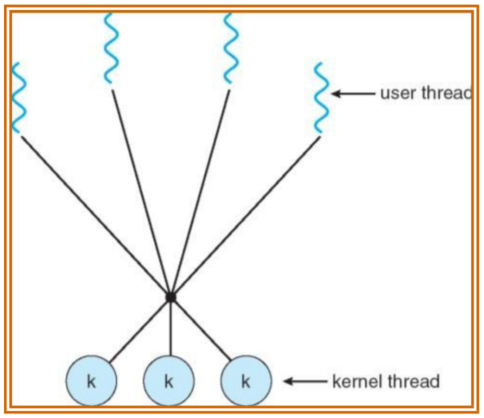
- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



2. **One-to-One**
   - Each user-level thread maps to kernel thread
   - Examples
     - Windows NT/XP/2000
     - Linux
     - Solaris 9 and later



3. **Many-to-Many Model**
   - Allows many user level threads to be mapped to many kernel threads
   - Allows the operating system to create a sufficient number of kernel threads
   - Solaris prior to version 9
   - Windows NT/2000 with the *ThreadFiber* package



Windows 7

Windows implements the Windows API, which is the primary API for the family of Microsoft operating systems (Windows 98, NT, 2000, and XP, as well as Windows 7). Indeed, much of what is mentioned in this section applies to this entire family of operating systems. A Windows application runs as a separate process, and each process may contain one or more threads.

The general components of a thread include:
• A thread ID uniquely identifying the thread
• A register set representing the status of the processor
• A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode

• A private storage area used by various run-time libraries and dynamic link libraries (DLLs).

The register set, stacks, and private storage area are known as the **context** of the thread. The primary data structures of a thread include:

• ETHREAD—executive thread block
• KTHREAD—kernel thread block
• TEB—thread environment block

Process Synchronization

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-butter problem allows at most *n* – 1 items in buffer at the same time. A solution, where all *N* buffers are used is not simple.
- Suppose that we modify the producer-consumer code by adding a variable *counter*, initialized to 0 and increment it each time a new item is added to the buffer
- **Race condition:** The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, **concurrent processes** must be **synchronized.**

The Critical-Section Problem:

- There are n processes that are competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Requirements to be satisfied for a Solution to the Critical-Section Problem:

1. **Mutual Exclusion -** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress -** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting -** A bound must exist 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.

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

- Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**.
- A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

Mutex Locks

- □ A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- □ Only one process may be active within the monitor at atime monitor monitor-name

  {

  // shared variable declarations

  procedure body P1 (…) { …. }

  …

  procedure body Pn (…) {……}

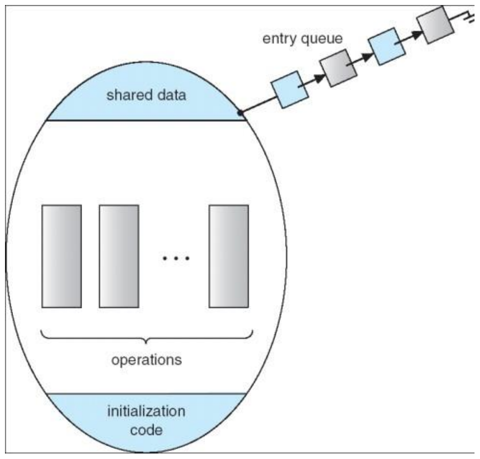
  {

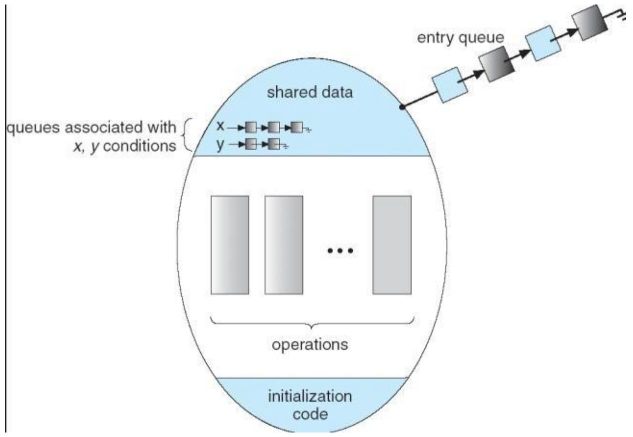
  initialization code

  }

  }

- □ To allow a process to wait within the monitor, a condition variable must be declared as o condition x, y;
- □ Two operations on a condition variable:
- □ x.wait ()–a process that invokes the operation is suspended.
- □ x.signal ()–resumes one of the suspended processes(if any)





Solution to Dining Philosophers Problem

```
Monitor DP
{
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 left and right neighbors
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;
}
}
```

Semophores

- It is a synchronization tool that is used to generalize the solution to the critical section problem in complex situations.
- A Semaphore s is an integer variable that can only be accessed via two indivisible (atomic) operations namely

```
    wait (s)
    {
   1. wait or P operation ( to test )
   2. signal or V operation ( to increment )
   while(s□ 0);
     s--;
   }
    signal (s)
    {
     s++;
    }
```

Mutual Exclusion Implementation using semaphore

```
do
{
wait(mutex);
        critical section

    remainder section
  } while (1);
signal(mutex);
```

Semaphore Implementation

- The semaphore discussed so far requires a busy waiting. That is if a process is in critical-section, the other process that tries to enter its critical-section must loop continuously in the entry code.
- To overcome the busy waiting problem, the definition of the semaphore operations wait and signal should be modified.
    - When a process executes the wait operation and finds that the semaphore value is not positive, the process can block itself. The block operation places the process into a waiting queue associated with the semaphore.
    - A process that is blocked waiting on a semaphore should be restarted when some other process executes a signal operation. The blocked process should be restarted by a wakeup operation which put that process into ready queue.
- To implemented the semaphore, we define a semaphore as a record as:

```
typedef struct {

int value;

struct process *L;

} semaphore;
```

Deadlock & starvation:

Example: Consider a system of two processes , P0 & P1 each accessing two semaphores ,S & Q, set to the value 1.

```
  P0     P1
Wait (S) Wait (Q)
Wait (Q) Wait (S)
   .     .
   .     .
   .     .
Signal(S) Signal(Q)
Signal(Q) Signal(S)
```

- Suppose that P0 executes wait(S), then 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). Since these signal operations cannot be executed, P0 & P1 are deadlocked.
- Another problem related to deadlock is indefinite blocking or starvation, a situation where a process wait indefinitely within the semaphore. Indefinite blocking may occur if we add or remove processes from the list associated with a semaphore in LIFO order.

Types of Semaphores

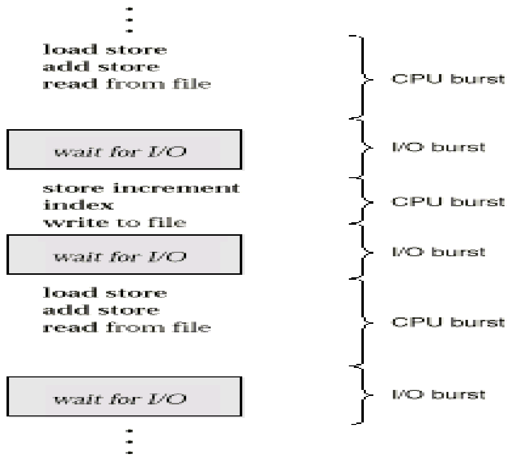
- *Counting* semaphore – any positive integer value
- *Binary* semaphore – integer value can range only between 0 and 1

CPU Scheduling

- CPU scheduling is the basis of multi programmed operating systems.
- The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization.
- Scheduling is a fundamental operating-system function.
- Almost all computer resources are scheduled before use.

CPU-I/O Burst Cycle

- 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,** then another CPU burst, then another I/O burst, and so on.
- Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.



CPU Scheduler
Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed.
The selection process is carried out by the **short-term scheduler** (or CPU scheduler).
The ready queue is not necessarily a first-in, first-out (FIFO) queue. It may be a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

Preemptive Scheduling

- CPU scheduling decisions may take place under the following four circumstances:
  1. When a process switches from the running state to the waiting state
  2. When a process switches from the running state to the ready state
  3. When a process switches from the waiting state to the ready state
  4. When a process terminates
- Under 1 & 4 scheduling scheme is non preemptive.
- Otherwise the scheduling scheme is preemptive.

Non-preemptive Scheduling
□ In non preemptive scheduling, once the CPU has been allocated a process, the process keeps the CPU until it releases the CPU either by termination or by switching to the waiting state.
□ This scheduling method is used by the Microsoft windows environment.

Dispatcher
The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
This function involves:
1. Switching context
2. Switching to user mode
3. Jumping to the proper location in the user program to restart that program

Scheduling Criteria

1. CPU utilization: The CPU should be kept as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
2. Throughput: Itis the number of processes completed per time unit. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.

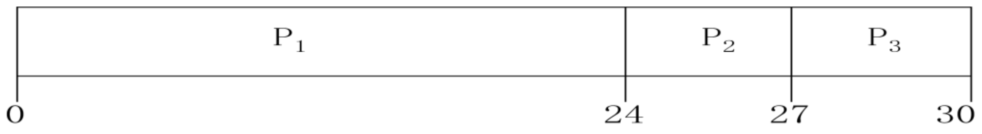
3. Turnaround time: The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
4. Waiting time: Waiting time is the sum of the periods spent waiting in the ready queue.
5. Response time: It is the amount of time it takes to start responding, but not the time that it takes to output that response.

CPU Scheduling Algorithms
1. First-Come, First-Served Scheduling
2. Shortest Job First Scheduling
3. Priority Scheduling
4. Round Robin Scheduling

First-Come, First-Served (FCFS) Scheduling

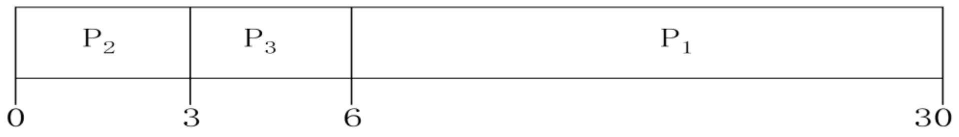
Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order



The Gantt chart for the schedule is:

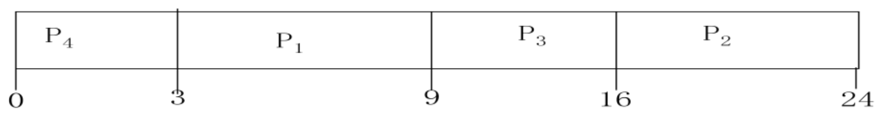
- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- *Convoy effect* short process behind long process

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

- SJF scheduling chart

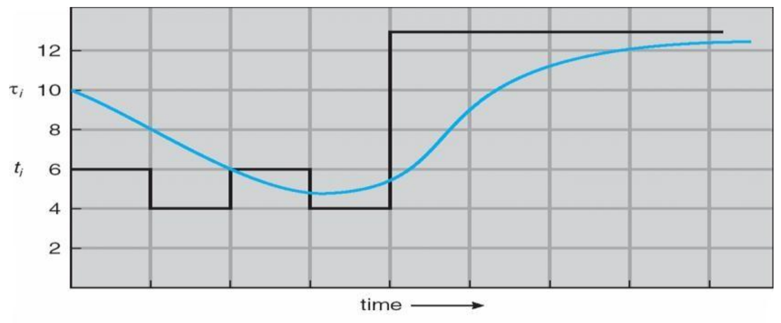


- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \Sigma \alpha \Sigma 1$
4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_n - 1 + \ldots$$

$$+ (1 - \alpha)^j\, \alpha\, t_{n-j} + \ldots$$

$$+ (1 - \alpha)^{n+1}\, \tau_0$$

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ÷ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ÷ **Starvation** – low priority processes may never execute
- Solution ÷ **Aging** – as time progresses increase the priority of the process

Round Robin (RR)

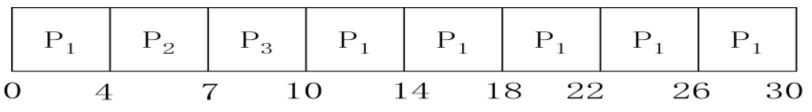
- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large $\rightarrow$ FIFO

– *q* small → *q* must be large with respect to context switch, otherwise overhead is too high

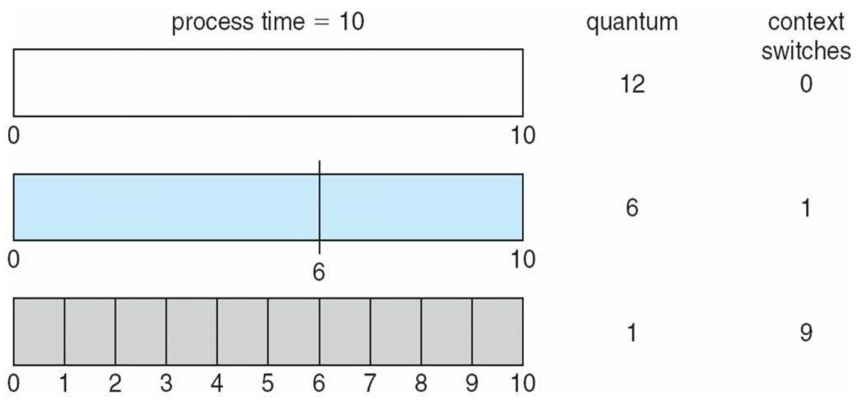
Example of RR with Time Quantum = 4

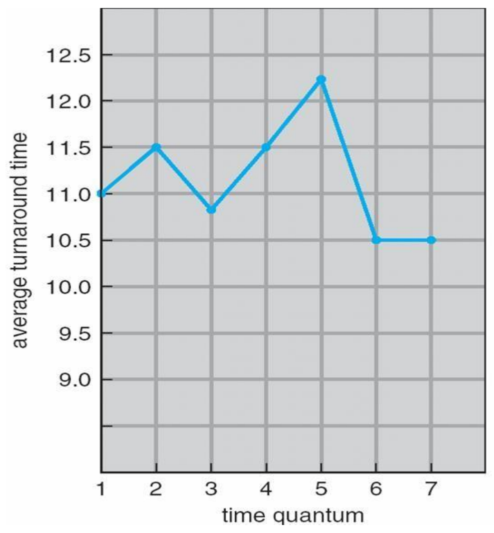
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

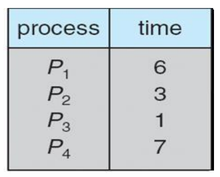
- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*



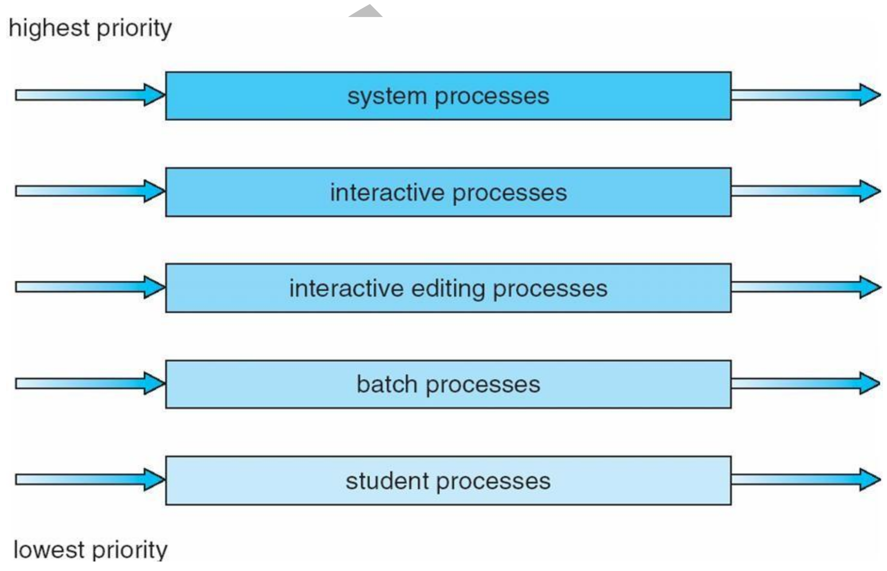




process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

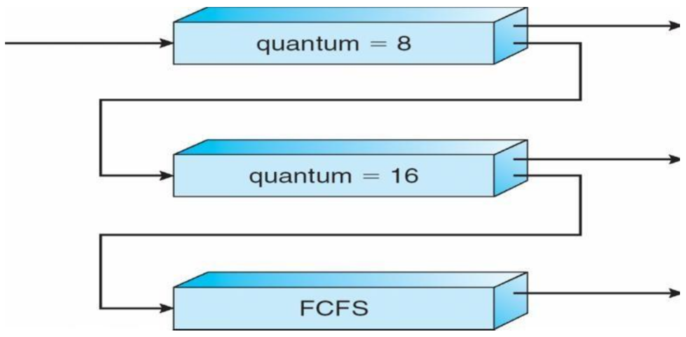
- Ready queue is partitioned into separate queues:
  foreground (interactive)
  background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process

– method used to determine which queue a process will enter when that process needs service



Deadlocks

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

System Model

- Resource types $R_1, R_2, \ldots, R_m$

CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

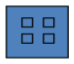
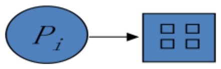
Deadlock can arise if four conditions hold simultaneously.

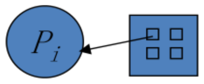
- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.

- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

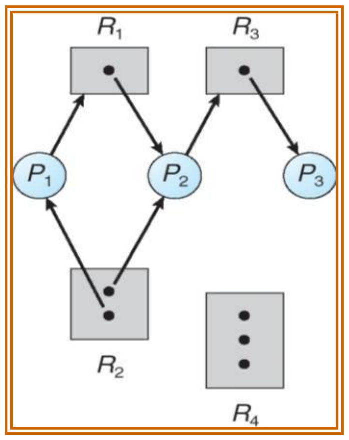
Resource-Allocation Graph

A set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$
- Process
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

Example of a Resource Allocation Graph



Basic Facts

- If graph contains no cycles $\rightarrow$ no deadlock.
- If graph contains a cycle $\rightarrow$
  - if only one instance per resource type, then deadlock.

    - if several instances per resource type, possibility of deadlock.

Deadlock Prevention

- Mutual Exclusion – not required for sharable resources; must hold for non-sharable resources.
- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
    - Low resource utilization; starvation possible.
- **No Preemption** –
    - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
    - Preempted resources are added to the list of resources for which the process is waiting.
    - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

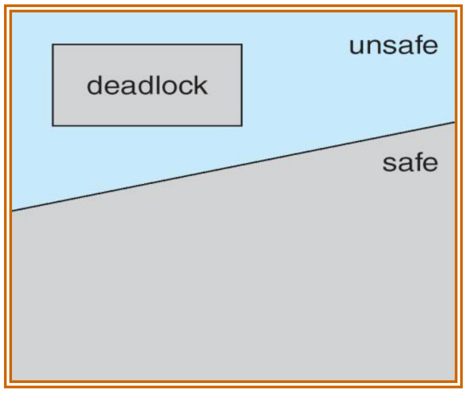
Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
- That is:
    - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.
    - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
    - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

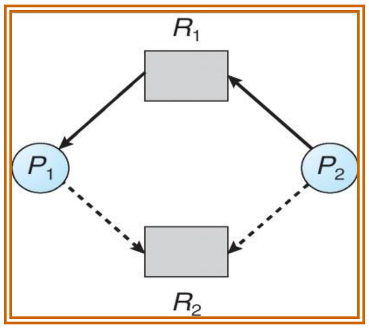


Avoidance algorithms

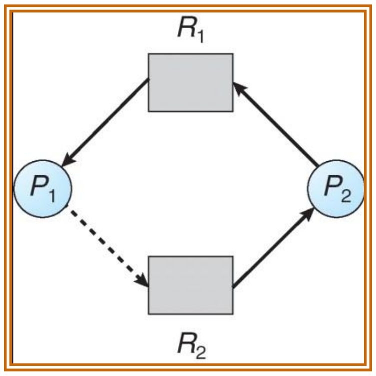
- Single instance of a resource type. Use a resource-allocation graph
- Multiple instances of a resource type. Use the banker's algorithm

Resource-Allocation Graph Scheme

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.
- Let n = number of processes, and m = number of resources types.

    - ***Available:*** Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
    - ***Max:*** n x m matrix. If *Max* $[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
    - ***Allocation:*** n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j.
    - ***Need:*** n x m matrix. If *Need*$[i,j] = k$, then P_i may need k more instances of R_j to complete its task.
      Need $[i,j]$ = *Max*$[i,j]$ – *Allocation* $[i,j]$.

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

3 resource types:

A (10 instances), *B* (5instances), and *C* (7 instances).

- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- The content of the matrix *Need* is defined to be *Max* – *Allocation*.

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria.

Deadlock Detection

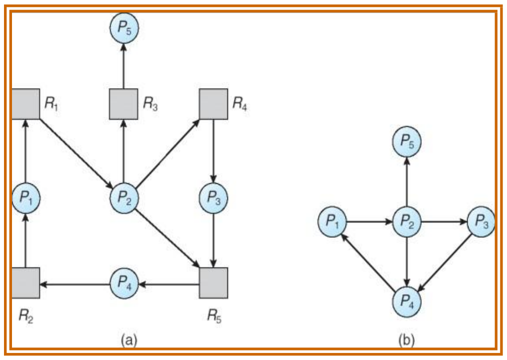
- **Allow system to enter deadlock state**
- **Detection algorithm**
- **Recovery scheme**

Single Instance of Each Resource Type

- Maintain *wait-for* graph

- Nodes are processes.
- $P_i \rightarrow P_j$ if P_i is waiting for P_j.

- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.



Several Instances of a Resource Type

- *Available:* A vector of length *m* indicates the number of available resources of each type.
- *Allocation:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- *Request:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:

(a) Work = Available

(b) For $i = 1,2, \ldots, n$, if $Allocation_i$ σ 0, then
Finish[i] = false;otherwise, *Finish*[i] = *true*.

2. Find an index *i* such that both:

(a) *Finish*[*i*] == *false*

(b) $Request_i$ Σ *Work*

If no such *i* exists, go to step 4.

3. Work = Work + $Allocation_i$
Finish[*i*] = *true*
go to step 2.

4. If *Finish*[*i*] == false, for some *i*, 1 Σ *i* Σ *n*, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types
  A (7 instances), *B* (2 instances), and *C* (6 instances).
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*.
- P_2 requests an additional instance of type *C*.

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?
- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

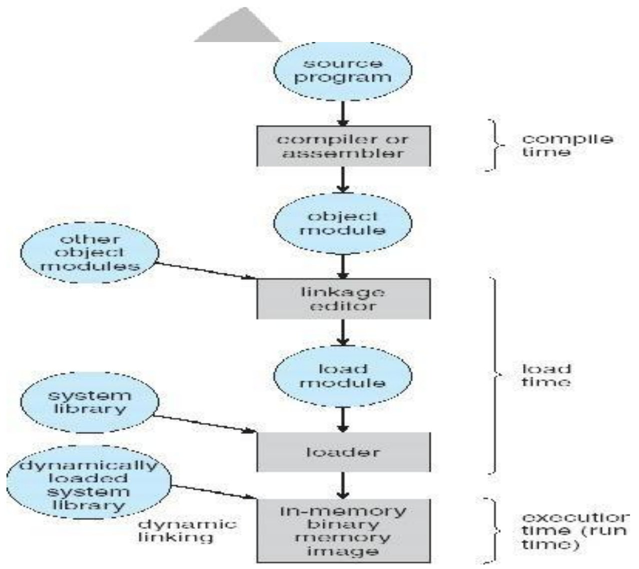
UNIT-III
STORAGE MANAGEMENT

Memory Management: Background

- In general, to rum a program, it must be brought into memory.
- Input queue – collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run
- Address binding: Mapping of instructions and data from one address to another address in memory.

Three different stages of binding:

1. Compile time: Must generate absolute code if memory location is known in prior.
2. Load time: Must generate relocatable code if memory location is not known at compile time
3. Execution time: Need hardware support for address maps (e.g., base and limit registers).



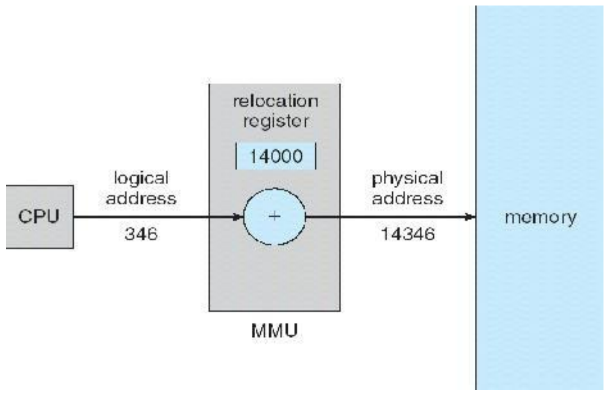
Logical vs. Physical Address Space

- **Logical address** – generated by the CPU; also referred to as **"virtual address"**
- **Physical address** – address seen by the memory unit.
- Logical and physical addresses are the **same** in compile-time and load-time address-binding schemes"
- Logical (virtual) and physical addresses **differ** in execution-time address- binding scheme"

Memory-Management Unit (MMU)

- It is a hardware device that maps virtual / Logical address to physical address
- In this scheme, the relocation register's value is added to Logical address generated by a user process.

- The user program deals with *logical* addresses; it never sees the *real* physical addresses
- Logical address range: 0 to max
- Physical address range: R+0 to R+max, where R—value in relocation register.



Dynamic Loading

- Through this, the routine is not loaded until it is called.
  - o Better memory-space utilization; unused routine is never loaded
  - o Useful when large amounts of code are needed to handle infrequently occurring cases
  - o No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time & is particularly useful for libraries
- Small piece of code called stub, used to locate the appropriate memory- resident library routine or function.
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Shared libraries: Programs linked before the new library was installed will continue using the older library.

Swapping

- A process can be swapped temporarily out of memory to a backing store (SWAP OUT)and then brought back into memory for continued execution (SWAP IN).
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users & it must provide direct access to these memory images
- **Roll out, roll in –** swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- **Transfer time:** Major part of swap time is transfer time. Total transfer time is directly proportional to the amount of memory swapped.

□ **Example**: Let us assume the user process is of size 1MB & the backing store is a standard hard disk with a transfer rate of 5MBPS.

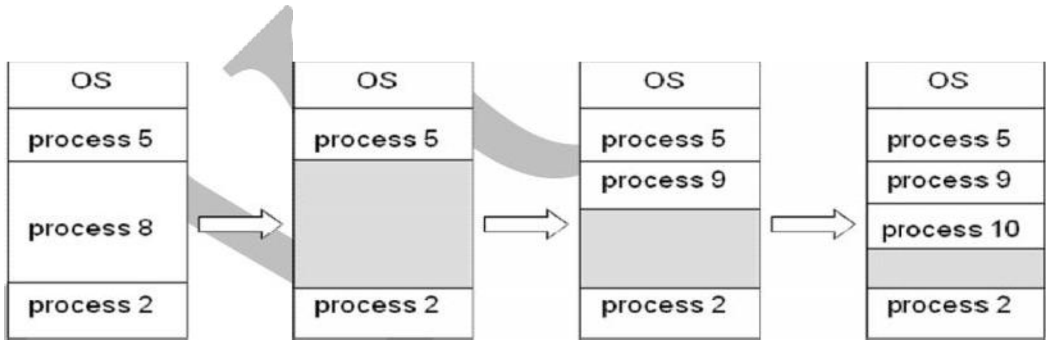
Transfer time = 1000KB/5000KB per second
= 1/5 sec = 200ms

Contiguous Allocation

- Each process is contained in a single contiguous section of memory.
- There are two methods namely:
  - □ Fixed – Partition Method
  - □ Variable – Partition Method

- **Fixed – Partition Method** :
  - o Divide memory into fixed size partitions, where each partition has exactly one process.
  - o The drawback is memory space unused within a partition is wasted.(eg.when process size < partition size)

- **Variable-partition method:**
  - o Divide memory into variable size partitions, depending upon the size of the incoming process.
  - o When a process terminates, the partition becomes available for another process.
  - o As processes complete and leave they create holes in the main memory.
  - o ***Hole*** – block of available memory; holes of various size are scattered throughout memory.



Dynamic Storage- Allocation Problem:

How to satisfy a request of size n‘ from a list of free holes?

Solution:

o First-fit: Allocate the first hole that is big enough.
o Best-fit: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
o Worst-fit: Allocate the largest hole; must also search entire list. Produces the largest leftover hole.

NOTE: First-fit and best-fit are better than worst-fit in terms of speed and storage utilization

- **Fragmentation:**

o **External Fragmentation** – This takes place when enough total memory space exists to satisfy a request, but it is not contiguous i.e, storage is fragmented into a large number of small holes scattered throughout the main memory.

o **Internal Fragmentation** – Allocated memory may be slightly larger than requested memory.

Example: hole = 184 bytes
Process size = 182 bytes.
We are left with a hole of 2 bytes.

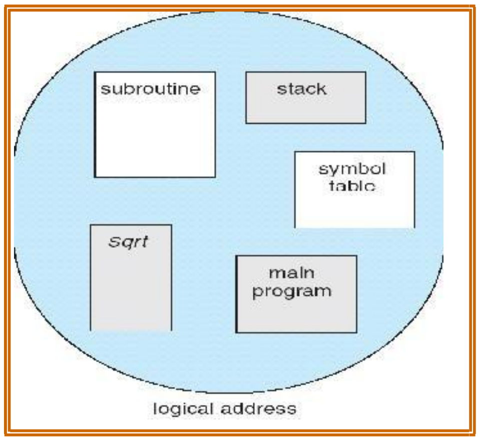
o **Solutions**

1. **Coalescing:** Merge the adjacent holes together.
2. **Compaction:** Move all processes towards one end of memory, hole towards other end of memory, producing one large hole of available memory. This scheme is expensive as it can be done if relocation is dynamic and done at execution time.
3. Permit the logical address space of a process to be **non-contiguous**. This is achieved through two memory management schemes namely **paging** and **segmentation**.

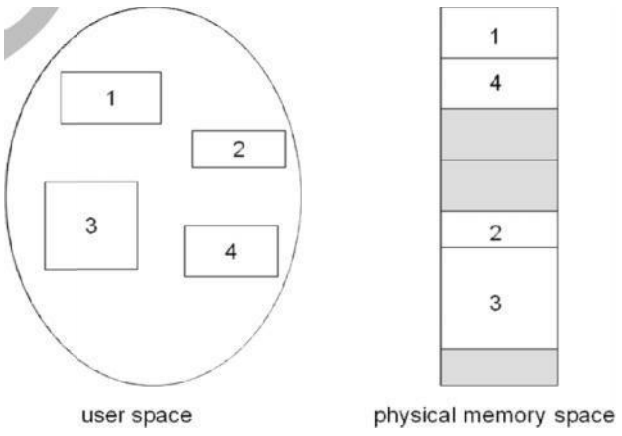
Segmentation

o Memory-management scheme that supports user view of memory

o A program is a collection of segments. A segment is a logical unit such as:Main program, Procedure, Function, Method, Object, Local variables, global variables, Common block, Stack, Symbol table, arrays



Logical View of Segmentation



Segmentation Hardware

o Logical address consists of a two tuple :

<Segment-number, offset>

o **Segment table** – maps two-dimensional physical addresses; each table entry has:

□ **Base** – contains the starting physical address where the segments reside in memory

□ **Limit** – specifies the length of the segment

o ***Segment-table base register (STBR)*** points to the segment table's location in memory

o ***Segment-table length register (STLR)*** indicates number of segments used by a program;

Segment number=s' is legal, if *s* < STLR

o **Relocation**.

□ dynamic

□ by segment table

o **Sharing**.

□ shared segments

□ same segment number

o **Allocation**.

□ first fit/best fit

□ external fragmentation

o **Protection:** With each entry in segment table associate:

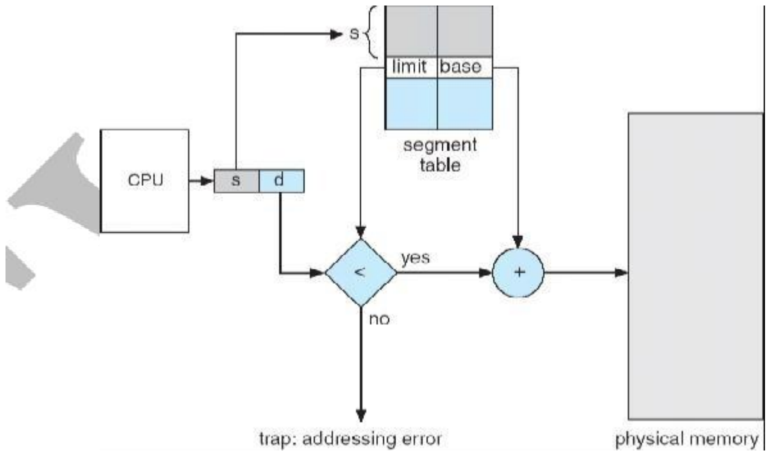
□ validation bit = 0 □ illegal segment

□ read/write/execute privileges

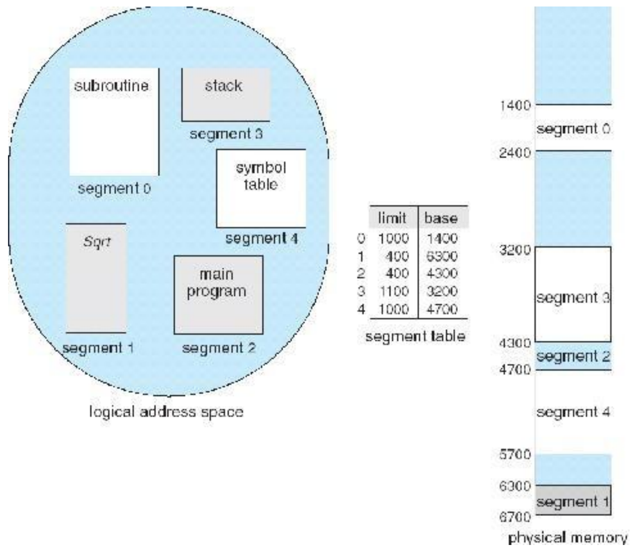
o Protection bits associated with segments; code sharing occurs at segment level

o Since segments vary in length, memory allocation is a dynamic storage- allocation problem

o A segmentation example is shown in the following diagram



EXAMPLE:



o Another advantage of segmentation involves the sharing of code or data.

o Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU.

o Segments are shared when entries in the segment tables of two different processes point to the same physical location.

Segmentation with paging

o The IBM OS/ 2.32 bit version is an operating system running on top of the Intel 386 architecture. The 386 uses segmentation with paging for memory management. The maximum number of segments per process is 16 KB, and each segment can be as large as 4 gigabytes.

o The local-address space of a process is divided into two partitions.

□ The first partition consists of up to 8 KB segments that are private to that process.

□ The second partition consists of up to 8KB segments that are shared among all the processes.

o Information about the first partition is kept in the **local descriptor table (LDT)**, information about the second partition is kept in the **global descriptor table (GDT)**.

o Each entry in the LDT and GDT consist of 8 bytes, with detailed information about a particular segment including the base location and length of the segment.

The logical address is a pair (selector, offset) where the selector is a16-bit number:

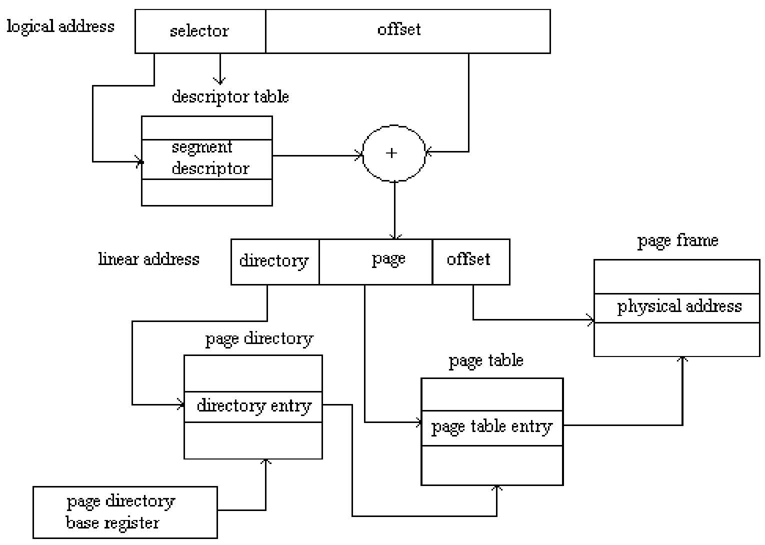
s	g	p
13	1	2

Where 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 within the segment in question.

- The base and limit information about the segment in question are used to generate a linear-address.
- First, the limit is used to check for address validity. If the address is not valid, a memory fault is generated, resulting in a trap to the operating system. If it is valid, then the value of the offset is added to the value of the base, resulting in a 32-bit linear address. This address is then translated into a physical address.
- The linear address is divided into a page number consisting of 20 bits, and a page offset consisting of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page directory pointer and a 10-bit

  page table pointer. The logical address is as follows.

p_1	p_2	d
10	10	12



o To improve the efficiency of physical memory use. Intel 386 page tables can be swapped to disk. In this case, an invalid bit is used in the page directory entry to indicate whether the table to which the entry is pointing is in memory or on disk.

o If the table is on disk, the operating system can use the other 31 bits to specify the disk location of the table; the table then can be brought into memory on demand.

Paging

- It is a memory management scheme that permits the physical address space of a process to be noncontiguous.
- It avoids the considerable problem of fitting the varying size memory chunks on to the backing store.

(i) Basic Method:

o Divide logical memory into blocks of same size called **"pages"**. o Divide physical memory into fixed-sized blocks called **"frames"** o Page size is a power of 2, between 512 bytes and 16MB.

<u>Address Translation Scheme</u>

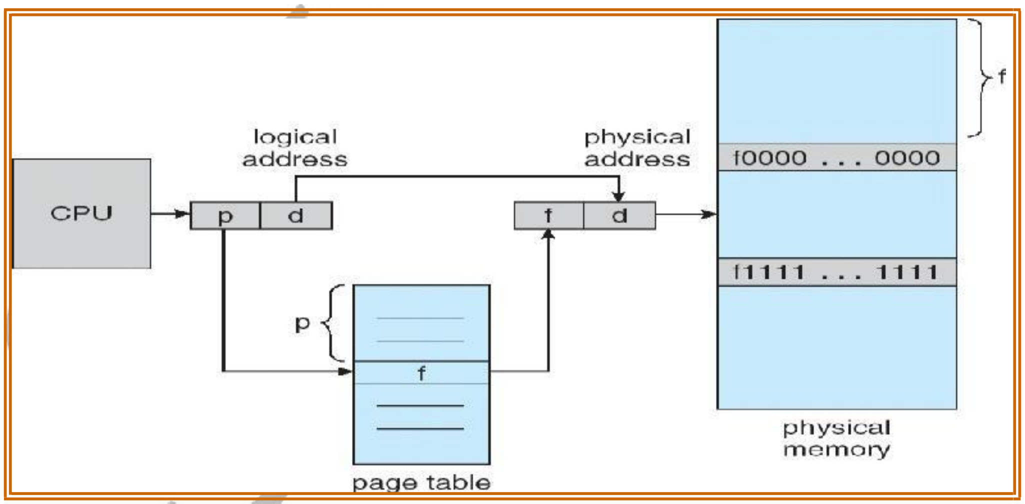
○ Address generated by CPU(logical address) is divided into:

□ **Page number *(p)*** – used as an index into a page table which contains base address of each page in physical memory

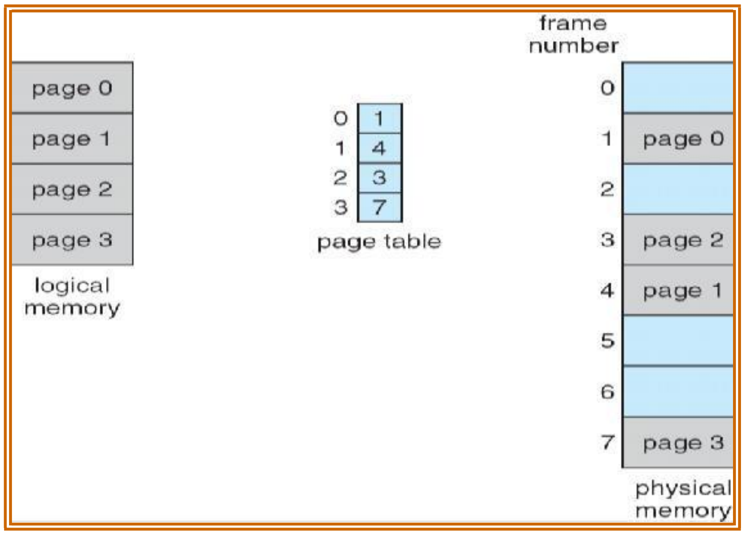
□ **Page offset *(d)*** –combined with base address to define the physical address i.e.,

Physical address = base address + offset

Paging Hardware



Paging model of logical and physical memory



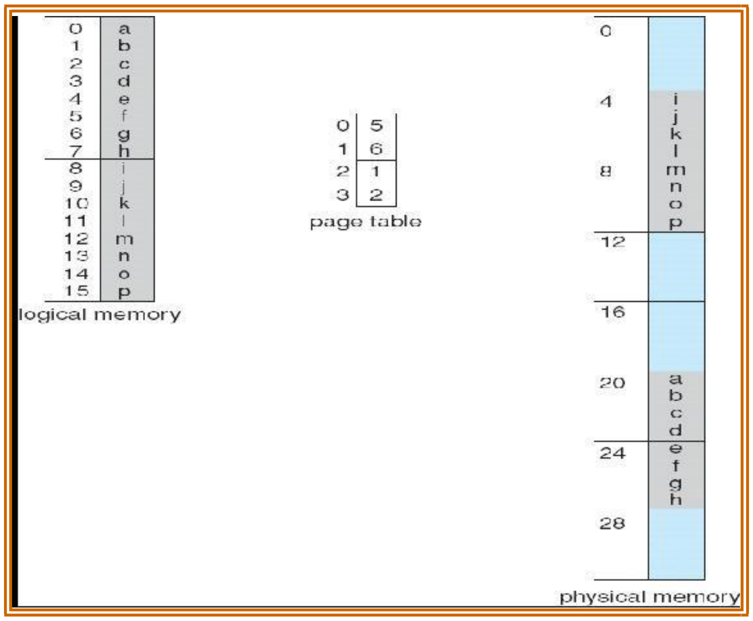
Paging example for a 32-byte memory with 4-byte pages

Page size = 4 bytes

Physical memory size = 32 bytes i.e (4 X 8 = 32 so, 8 pages)

Logical address 0‘ maps to physical address 20 i.e ((5 X 4) +0)

Where Frame no = 5, Page size = 4, Offset = 0



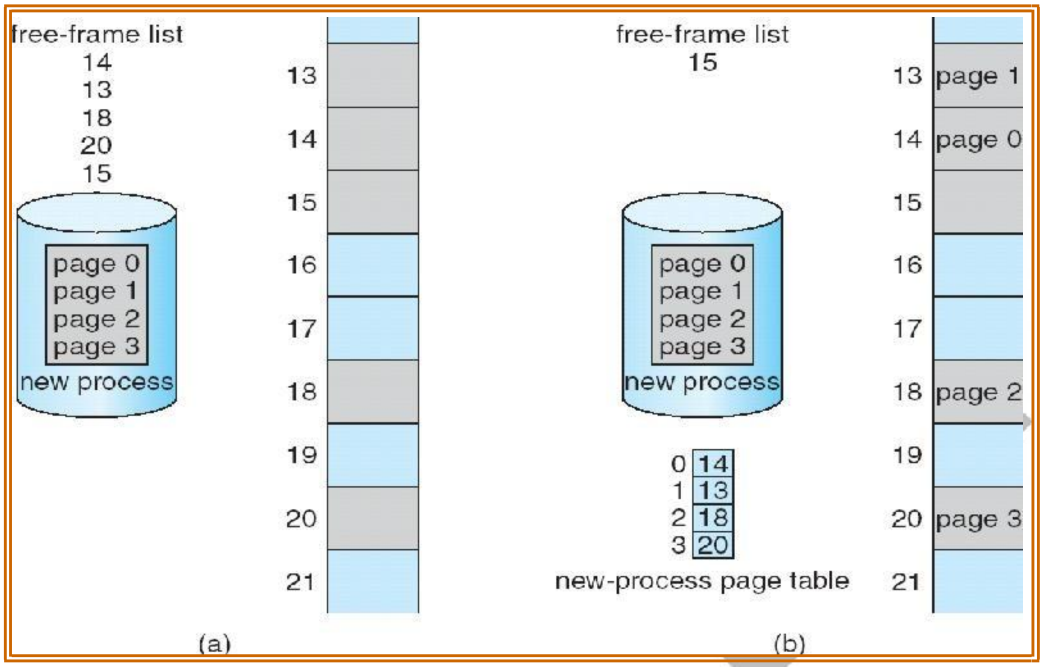
Allocation

o When a process arrives into the system, its size (expressed in pages) is examined.
o Each page of process needs one frame. Thus if the process requires n‘ pages, at least n‘ frames must be available in memory.

o If n‘ frames are available, they are allocated to this arriving process.

o The 1st page of the process is loaded into one of the allocated frames & the frame number is put into the page table.

o Repeat the above step for the next pages & so on.



(a) Before Allocation (b) After Allocation

Frame table: It is used to determine which frames are allocated, which frames are available, how many total frames are there, and so on.(ie) It contains all the information about the frames in the physical memory.

(ii) Hardware implementation of Page Table

o This can be done in several ways :

1. Using PTBR
2. TLB

o The simplest case is **Page-table base register (PTBR**), is an index to point the page table.

o **TLB (Translation Look-aside Buffer)**

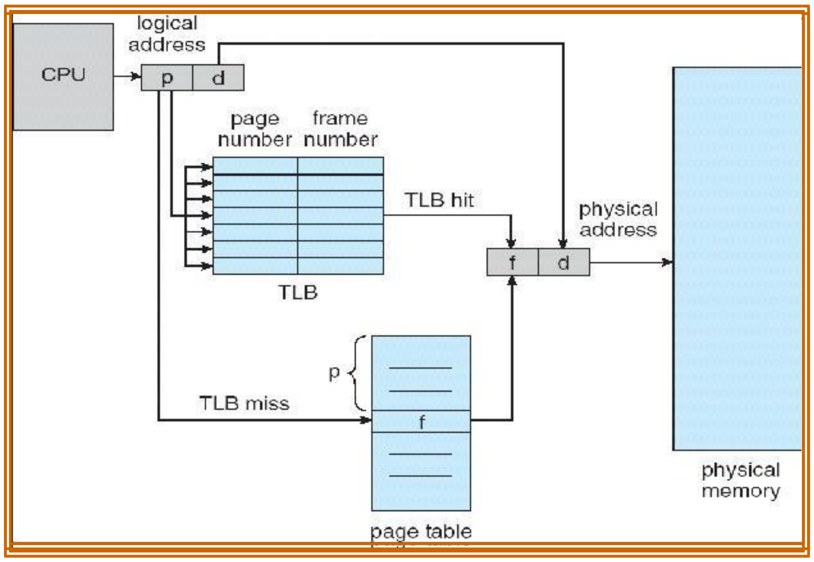
□ It is a fast lookup hardware cache.

□ It contains the recently or frequently used page table entries.

□ It has two parts: Key (tag) & Value.

□ More expensive.

<u>**Paging Hardware with TLB**</u>



○ When a logical address is generated by CPU, its page number is presented to TLB.

o **TLB hit**: If the page number is found, its frame number is immediately available & is used to access memory

o **TLB miss**: If the page number is not in the TLB, a memory reference to the page table must be made.

o **Hit ratio:** Percentage of times that a particular page is found in the TLB.

□ For example hit ratio is 80% means that the desired page number in the TLB is 80% of the time.

o **Effective Access Time:**

□ Assume hit ratio is 80%.

□ If it takes 20ns to search TLB & 100ns to access memory, then the memory access takes 120ns(TLB hit)

□ If we fail to find page no. in TLB (20ns), then we must 1st access memory for page table (100ns) & then access the desired byte in memory (100ns).

Therefore Total = 20 + 100 + 100

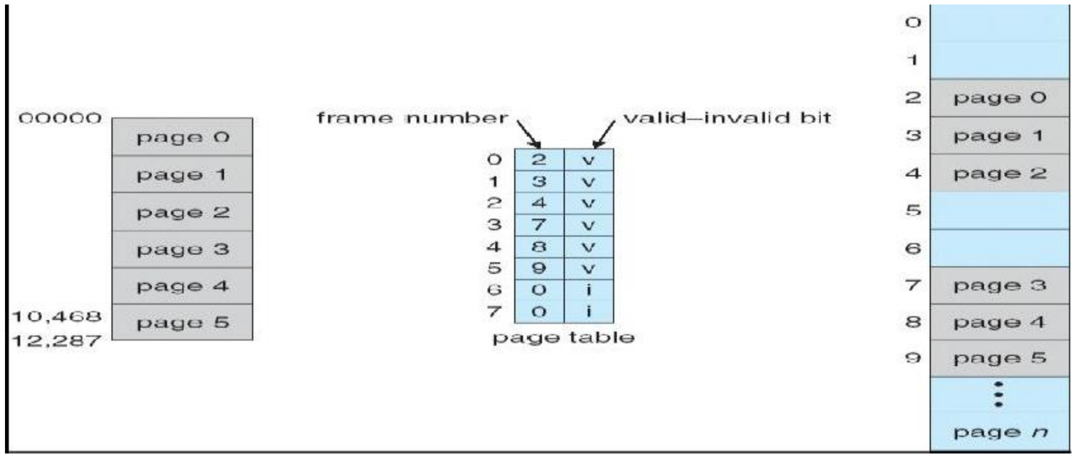
= 220 ns(TLB miss).

Then Effective Access Time (EAT) = 0.80 X (120 + 0.20) X 220.

= 140 ns.

(iii) Memory Protection

o Memory protection implemented by associating protection bit with each frame
o Valid-invalid bit attached to each entry in the page table:
□ ***"valid (v)" indicates that the associated page is in the process' logical address space, and is thus a legal page***
□ ***"invalid (i)" indicates that the page is not in the process' logical address spaces***



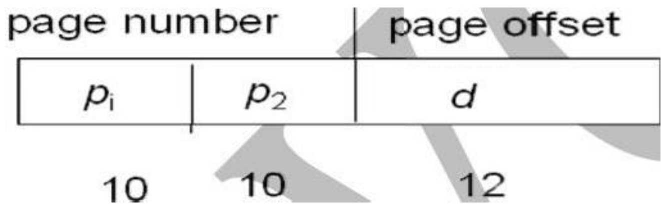
(iv) Structures of the Page Table

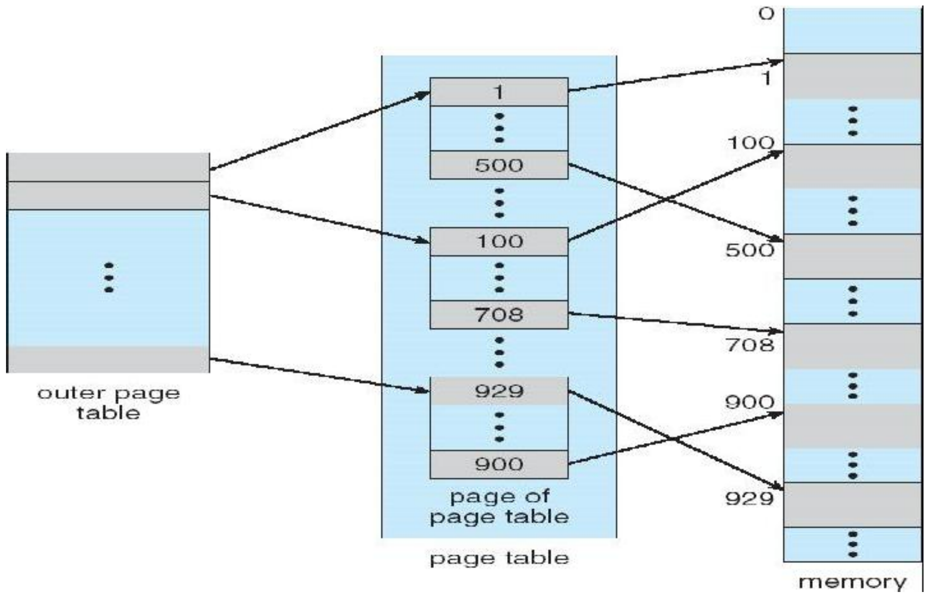
a) Hierarchical Paging b)
Hashed Page Tables c) Inverte
Page Tables

a) <u>Hierarchical Paging</u>

o Break up the Page table into smaller pieces. Because if the page table is too large then it is qui difficult to search the page number.

Example: "Two-Level Paging "





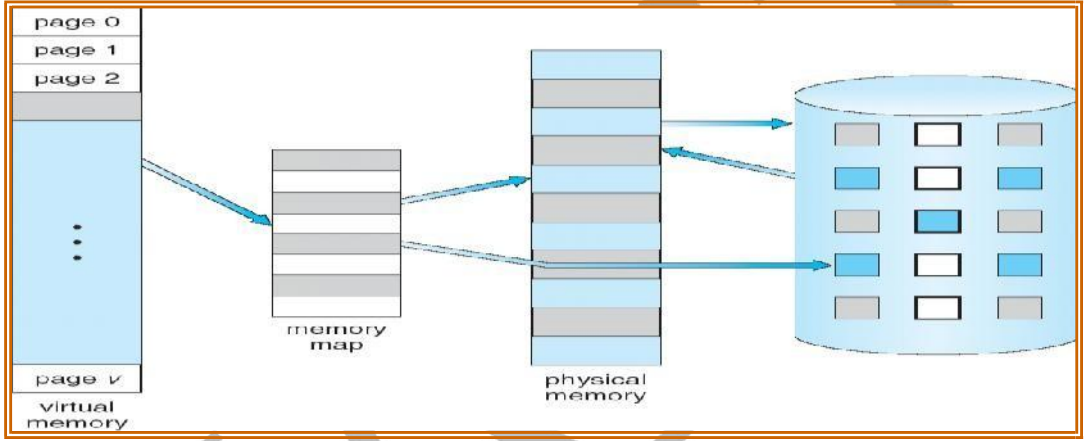
Virtual Memory

- It is a technique that allows the execution of processes that may not be completely in main memory.
- **Advantages:**
  - Allows the program that can be larger than the physical memory.
  - Separation of user logical memory from physical memory
  - Allows processes to easily share files & address space.
  - Allows for more efficient process creation.

Virtual memory can be implemented using

- Demand paging
- Demand segmentation

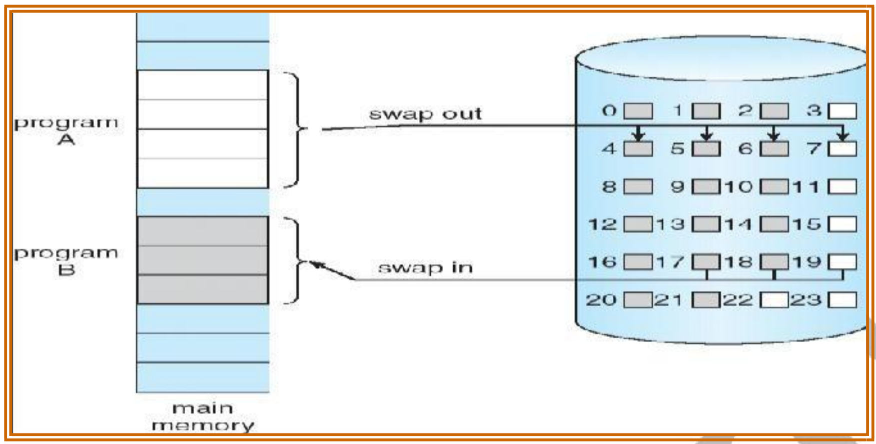
Virtual Memory That is Larger than Physical Memory



Demand Paging

- It is similar to a paging system with swapping.
- Demand Paging - Bring a page into memory only when it is needed
- To execute a process, swap that entire process into memory. Rather than swapping the entire process into memory however, we use Lazy Swapper‖
- **Lazy Swapper -** Never swaps a page into memory unless that page will be needed.
- **Advantages**
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

Transfer of a paged memory to contiguous disk space



Basic Concepts:

o Instead of swapping in the whole processes, the pager brings only those necessary pages into memory. Thus,

1. It avoids reading into memory pages that will not be used anyway.
2. Reduce the swap time.
3. Reduce the amount of physical memory needed.

o To differentiate between those pages that are in memory & those that are on the disk we use the **Valid-Invalid bit**

o A valid – invalid bit is associated with each page table entry.

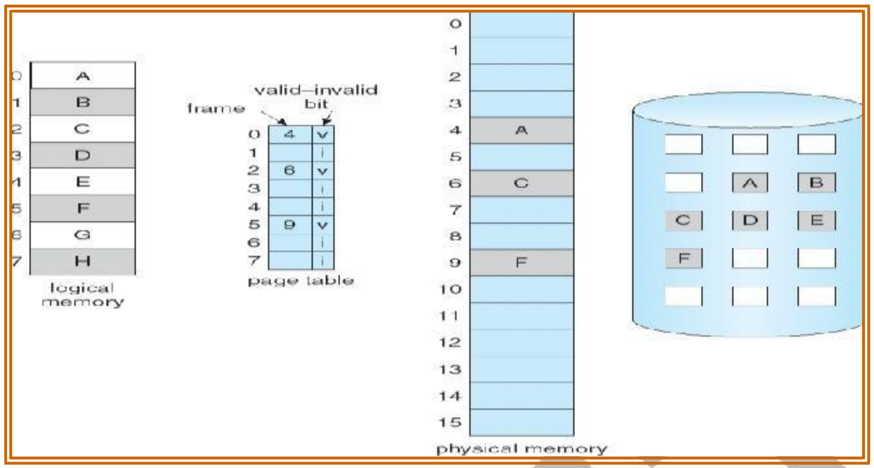
o Valid □ associated page is in memory.

In-Valid □

□ invalid page

□ valid page but is currently on the disk

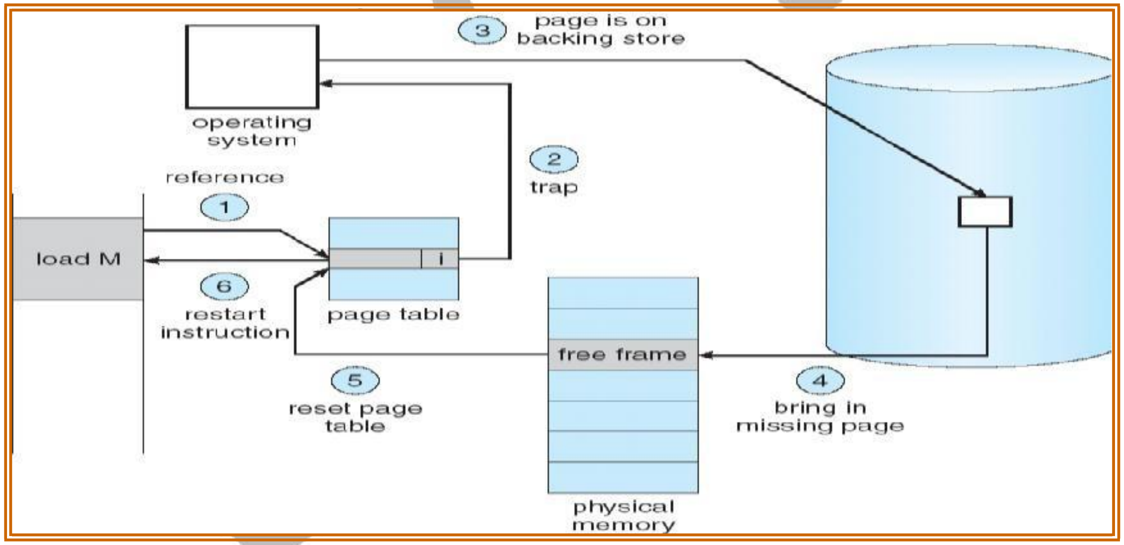
Page table when some pages are not in main memory



Page Fault

o Access to a page marked invalid causes a page fault trap**.**

Steps in Handling a Page Fault



1. Determine whether the reference is a valid or invalid memory access
2. a) If the reference is invalid then terminate the process.
   b) If the reference is valid then the page has not been yet brought into main memory.
3. Find a free frame.
4. Read the desired page into the newly allocated frame.
5. Reset the page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted .

Pure demand paging

- o Never bring a page into memory until it is required.
- o We could start a process with no pages in memory**.**
- o When the OS sets the instruction pointer to the 1st instruction of the process, which is on the non-memory resident page, then the process immediately faults for the page.
- o After this page is bought into the memory, the process continue to execute, faulting as necessary until every page that it needs is in memory.

Performance of demand paging

- o Let p be the probability of a page fault 0 □ *p* □ 1
- o Effective Access Time (EAT)

  EAT = (1 – *p*) x ma + *p* x page fault time.

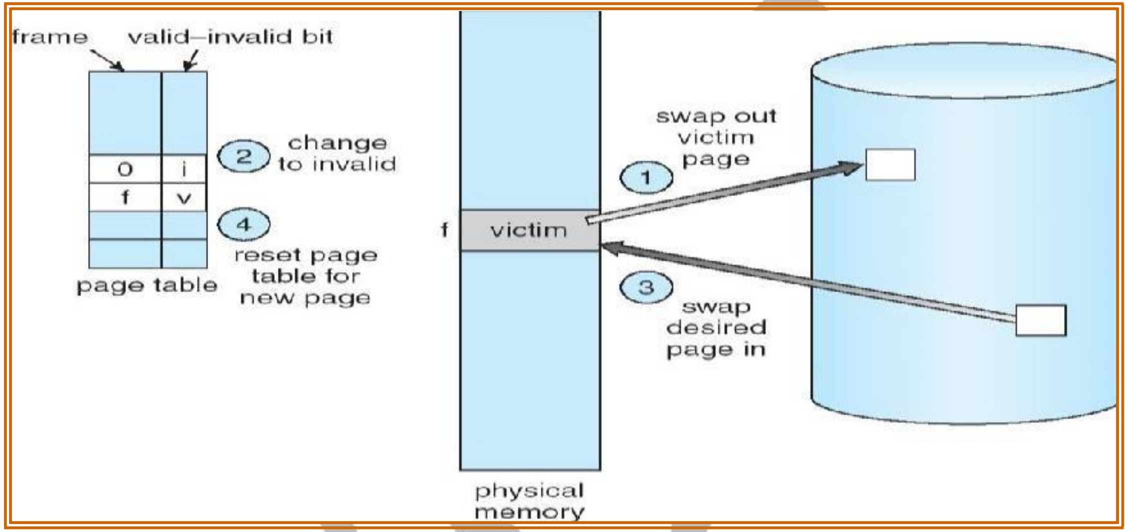
  Where ma □ memory access, p □ Probability of page fault $(0 \leq p \leq 1)$
- o The memory access time denoted ma is in the range 10 to 200 ns.
- o If there are no page faults then EAT = ma.
- o To compute effective access time, we must know how much time is needed to service a page fault.
- o A page fault causes the following sequence to occur:
  1. Trap to the OS
  2. Save the user registers and process state.
  3. Determine that the interrupt was a page fault.
  4. Check whether the reference was legal and find the location of page on disk.
  5. Read the page from disk to free frame.
     a. Wait in a queue until read request is serviced. b. Wait for seek time and latency time.
     c. Transfer the page from disk to free frame.
  6. While waiting ,allocate CPU to some other user.
  7. Interrupt from disk.
  8. Save registers and process state for other users.
  9. Determine that the interrupt was from disk.
  7. Reset the page table to indicate that the page is now in memory.
  8. Wait for CPU to be allocated to this process again.
  9. Restart the instruction that was interrupted .

Page Replacement

- o If no frames are free, we could find one that is not currently being used & free it.
- o We can free a frame by writing its contents to swap space & changing the page table to indicate that the page is no longer in memory.
- o Then we can use that freed frame to hold the page for which the process faulted.

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
   - If there is a free frame , then use it.
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame
   - Write the victim page to the disk, change the page & frame tables accordingly.
3. Read the desired page into the (new) free frame. Update the page and frame tables.
4. Restart the process



Page Replacement Algorithms

1. FIFO Page Replacement
2. Optimal Page Replacement
3. LRU Page Replacement
4. LRU Approximation Page Replacement
5. Counting-Based Page Replacement

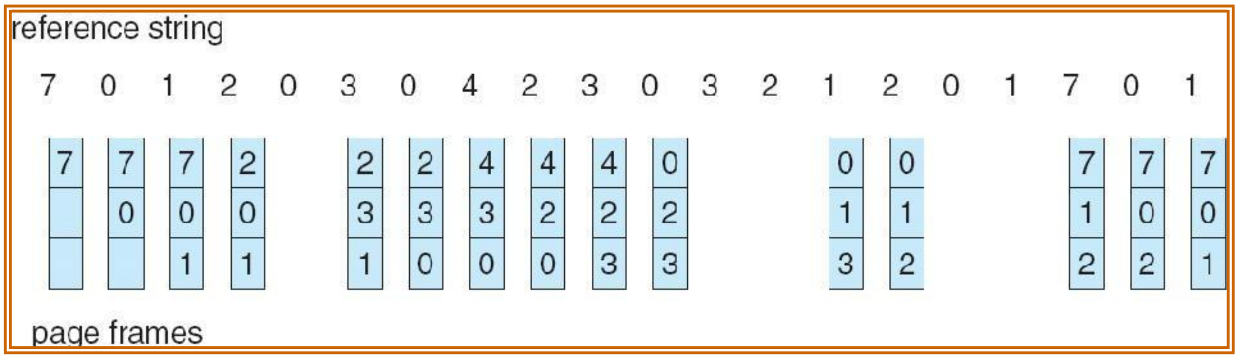
(a) FIFO page replacement algorithm

- **Replace the oldest page.**
- This algorithm associates with each page ,the time when that page was brought in.

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3 (3 pages can be in memory at a time per process)



No. of page faults = 15

Drawback:

- FIFO page replacement algorithm s performance is not always good.
- To illustrate this, consider the following example:

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- If No.of available frames -= 3 then the no.of page faults=9
- If No.of available frames =4 then the no.of page faults=10
- Here the no. of page faults increases when the no.of frames increases .This is called as **Belady's Anomaly.**

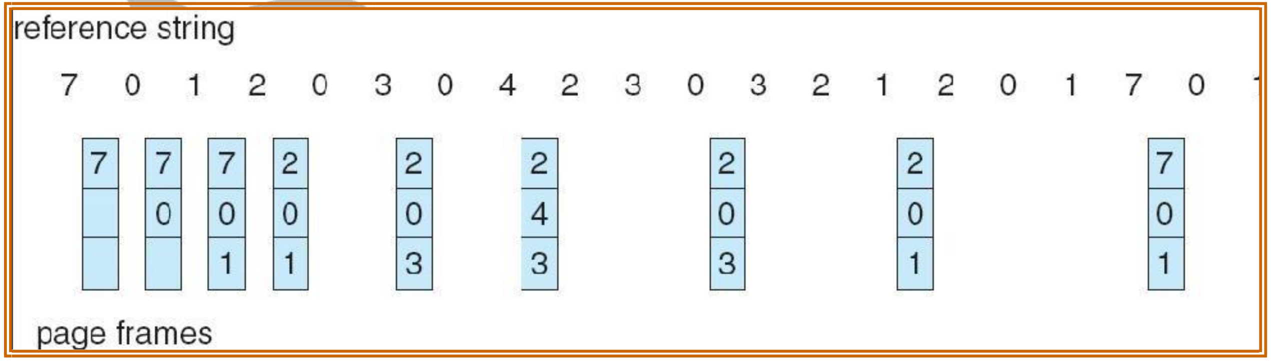
(b) Optimal page replacement algorithm

- **Replace the page that will not be used for the longest period of time.**

Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3



No. of page faults = 9

Drawback:

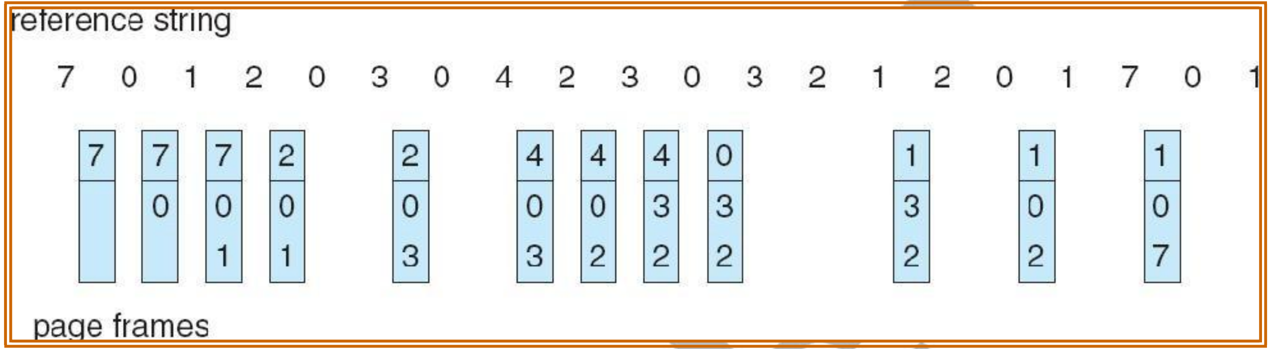
o It is difficult to implement as it requires future knowledge of the reference string.

(c) LRU(Least Recently Used) page replacement algorithm

o **Replace the page that has not been used for the longest period of time.**
Example:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

No.of available frames = 3

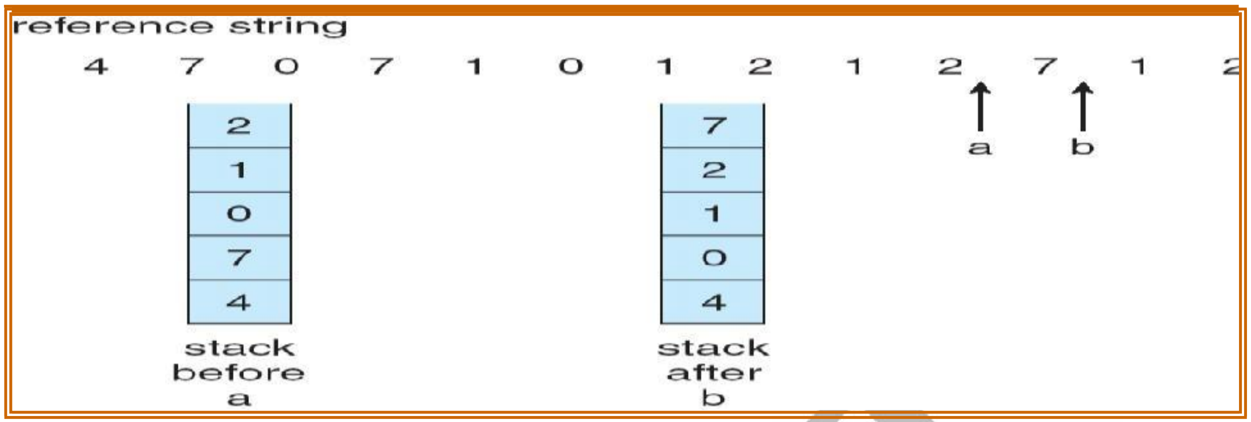


No. of page faults = 12

o LRU page replacement can be implemented using

1. **Counters**
   - □ Every page table entry has a time-of-use field and a clock or counter is associated with the CPU.
   - □ The counter or clock is incremented for every memory reference.
   - □ Each time a page is referenced , copy the counter into the time- of-use field.
   - □ When a page needs to be replaced, replace the page with the smallest counter value.
2. **Stack**
   - □ Keep a stack of page numbers
   - □ Whenever a page is referenced, remove the page from the stack and put it on top of the stack.
   - □ When a page needs to be replaced, replace the page that is at the bottom of the stack.(LRU page)

Use of A Stack to Record The Most Recent Page References



(d) LRU Approximation Page Replacement

- o Reference bit
  - □ With each page associate a reference bit, initially set to 0
  - □ When page is referenced, the bit is set to 1
- o When a page needs to be replaced, replace the page whose reference bit is 0
- o The order of use is not known , but we know which pages were used and which were not used.

(i) Additional Reference Bits Algorithm

- o Keep an 8-bit byte for each page in a table in memory.
- o At regular intervals , a timer interrupt transfers control to OS.
- o The OS shifts reference bit for each page into higher- order bit shifting the other bits right 1 bit and discarding the lower-order bit.

Example:

oIf reference bit is 00000000 then the page has not been used for 8 time periods.

oIf reference bit is 11111111 then the page has been used atleast once each time period.

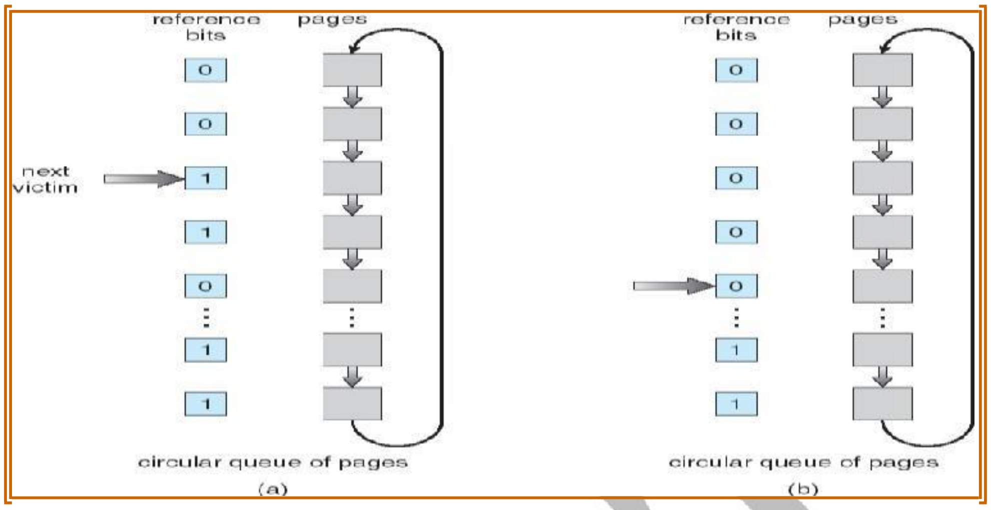
oIf the reference bit of page 1 is 11000100 and page 2 is 01110111 then page 2 is the LRU page.

(ii) Second Chance Algorithm

oBasic algorithm is FIFO

oWhen a page has been selected , check its reference bit.

- □ If 0 proceed to replace the page
- □ If 1 give the page a second chance and move on to the next FIFO page.
- □ When a page gets a second chance, its reference bit is cleared and arrival time is reset to current time.
- □ Hence a second chance page will not be replaced until all other pages are replaced.



(iii) Enhanced Second Chance Algorithm o Consider both reference bit and modify bit o There are four possible classes

1. (0,0) – neither recently used nor modified □ Best page to replace
2. (0,1) – not recently used but modified □ page has to be written out before replacement.
3. (1,0) - recently used but not modified □ page may be used again
4. (1,1) – recently used and modified □ page may be used again and page has to be written to disk

(e) Counting-Based Page Replacement

o Keep a counter of the number of references that have been made to each page

1. **Least Frequently Used (LFU)Algorithm**: replaces page with smallest count
2. **Most Frequently Used (MFU)Algorithm**: replaces page with largest count

□ It 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

o These are used along with page replacement algorithms to improve their performance

Technique 1:

o A pool of free frames is kept.
o When a page fault occurs, choose a victim frame as before.
o Read the desired page into a free frame from the pool
o The victim frame is written onto the disk and then returned to the pool of free frames.

Technique 2:

- Maintain a list of modified pages.
- Whenever the paging device is idles, a modified is selected and written to disk and its modify bit is reset.

Technique 3:

- A pool of free frames is kept.
- Remember which page was in each frame.
- If frame contents are not modified then the old page can be reused directly from the free frame pool when needed

Allocation of Frames

- There are two major allocation schemes
  - Equal Allocation
  - Proportional Allocation
- **Equal allocation**
  - If there are n processes and m frames then allocate m/n frames to each process.
  - **Example:** If there are 5 processes and 100 frames, give each process 20 frames.
- **Proportional allocation**
  - Allocate according to the size of process

Let s_i be the size of process i.

Let m be the total no. of frames

Then $S = \sum s_i$

$a_i = s_i / S * m$

where a_i is the no.of frames allocated to process i.

Global vs. Local Replacement

- **Global replacement** – each process selects a replacement frame from the set of all frames; one process can take a frame from another.
- **Local replacement** – each process selects from only its own set of allocated frames.

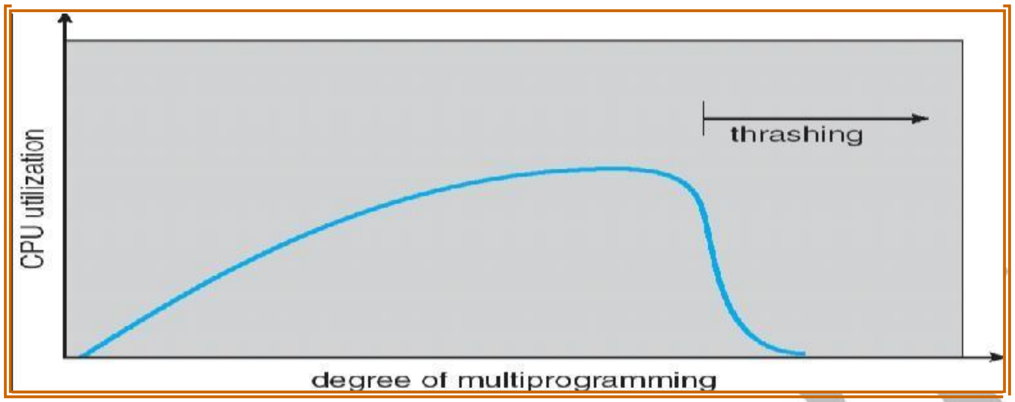
Thrashing

- High paging activity is called **thrashing**.
- If a process does not have enough‖ pages, the page-fault rate is very high.

  This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process is added to the system
- When the CPU utilization is low, the OS increases the degree of multiprogramming.
- If global replacement is used then as processes enter the main memory they tend to steal frames belonging to other processes.
- Eventually all processes will not have enough frames and hence the page fault rate becomes very

high.

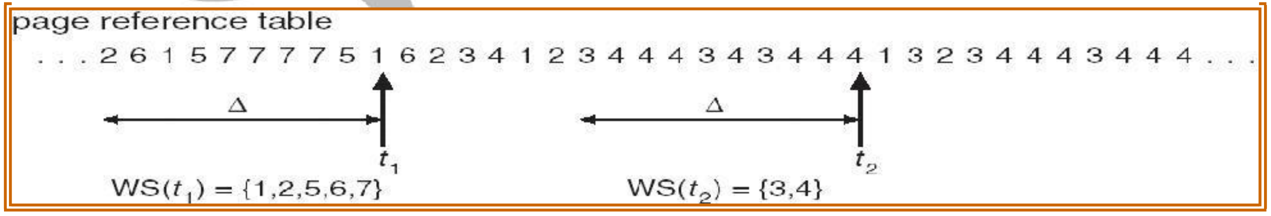
- o Thus swapping in and swapping out of pages only takes place.
- o This is the cause of thrashing.



- o To **limit thrashing**, we can use a **local replacement** algorithm.
- o To prevent thrashing, there are two methods namely ,
  - □ Working Set Strategy
  - □ Page Fault Frequency

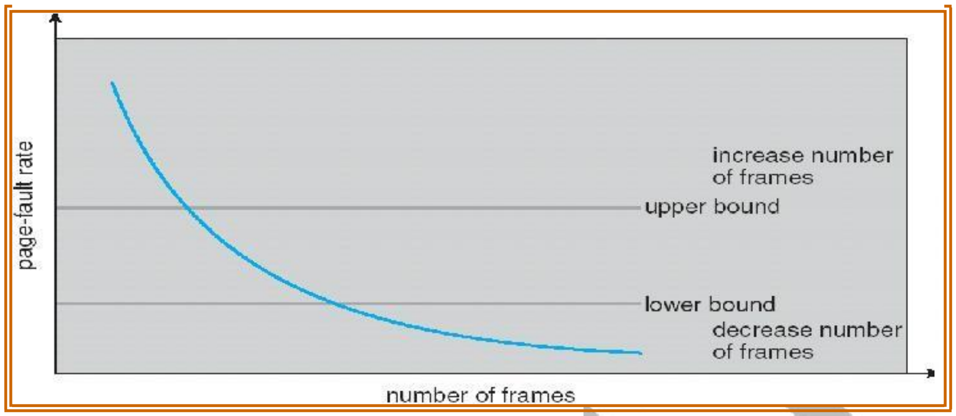
1. Working-Set Strategy

- o It is based on the assumption of the model of locality.
- o Locality is defined as the set of pages actively used together.
- o Working set is the set of pages in the most recent □ page references
- o □ is the working set window.
  - □ if □ too small , it will not encompass entire locality
  - □ if □ too large ,it will encompass several localities
  - □ if □ = □□ it will encompass entire program
- o D = □ WSS_i
  - □ Where WSS_i is the working set size for process i.
  - □ D is the total demand of frames
  - o if $D > m$ then Thrashing will occur.



2. Page-Fault Frequency Scheme

- If actual rate too low, process loses frame
- If actual rate too high, process gains frame



Other Issues

- **Prepaging**
  - ☐ To reduce the large number of page faults that occurs at process startup
  - ☐ Prepage all or some of the pages a process will need, before they are referenced
  - ☐ But if prepaged pages are unused, I/O and memory are wasted

- **Page Size**

  Page size selection must take into consideration:

  - fragmentation
  - table size
  - I/O overhead
  - locality
- **TLB Reach**
  - ☐ TLB Reach - The amount of memory accessible from the TLB
  - ☐ TLB Reach = (TLB Size) X (Page Size)
  - ☐ Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.
  - ☐ Increase the Page Size. This may lead to an increase in fragmentation as not all applications require a large page size
  - ☐ Provide Multiple Page Sizes. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.
- **I/O interlock**
  - ☐ Pages must sometimes be locked into memory
  - ☐ Consider I/O. Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

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. This list is typically populated using a page-replacement algorithm such as those discussed in Section 9.4 and most likely contains free pages scattered throughout physical memory, as explained earlier. Remember, too, that 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:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.

2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

Buddy System

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 (4KB,8KB,16KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16K segment

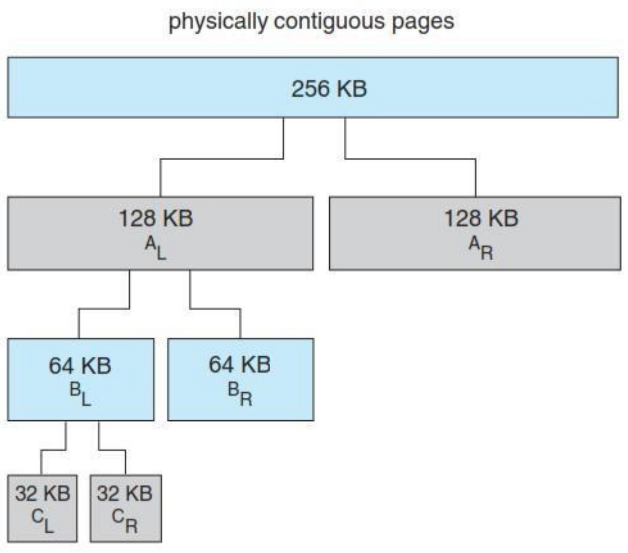


Figure 9.26 Buddy system allocation.

OS Examples

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**. (In some circumstances, a process may be allowed to exceed 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.

Solaris

In Solaris, when a thread incurs a page fault, the kernel assigns a page to the faulting thread fromthe list of free pages it maintains. Therefore, it is imperative that the kernel keep a sufficient amount of free memory available. 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 pageout process is similar to the second

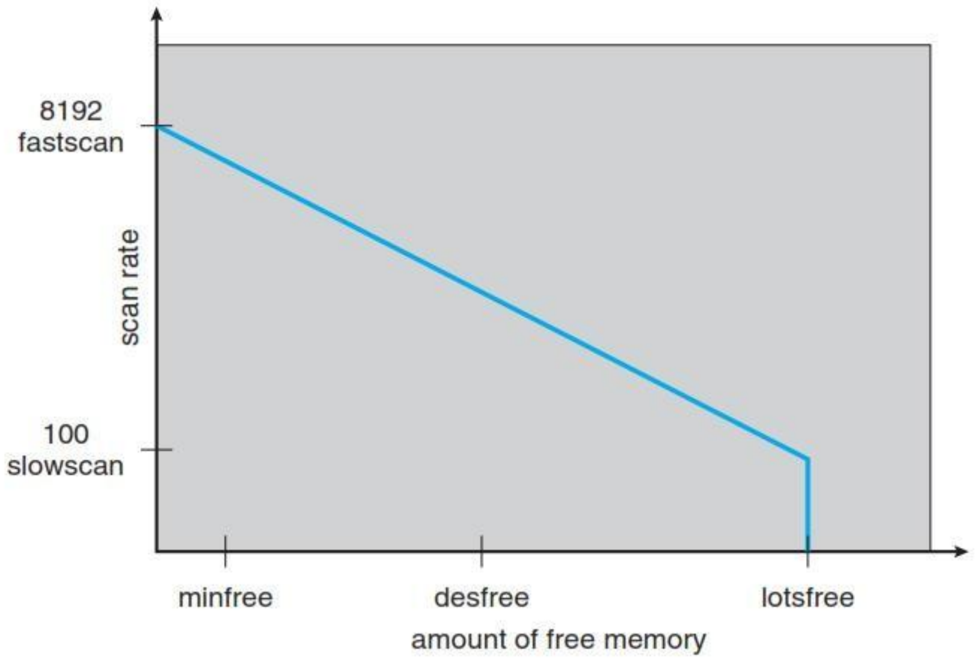


Figure 9.29 Solaris page scanner.

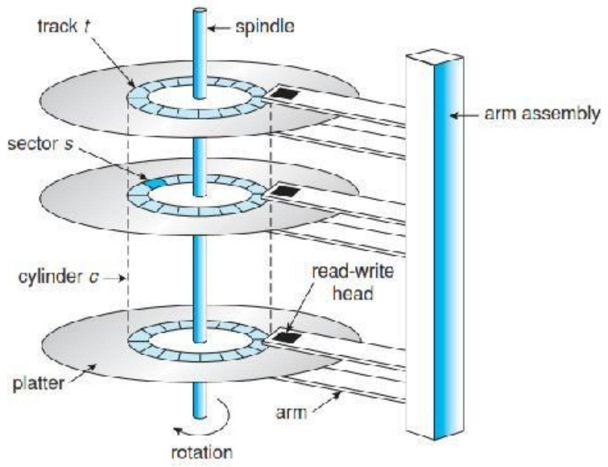
CS6401- Operating System

UNIT-IV

I/O SYSTEMS

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**. The set of tracks that are at one arm position makes up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

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.

The host controller then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command. Disk controllers usually have a built-in cache. Data transfer at the disk drive happens between the cache and the disk surface, and data transfer to the host, at fast electronic speeds, occurs between the cache and the host controller.

Solid-State Disks

Sometimes old technologies are used in new ways as economics change or the technologies evolve. An example is the growing importance of **solid-state disks**,or **SSDs**. Simply described, an SSD is nonvolatile memory that is used like a hard drive. There are many variations of this technology, from DRAM with a battery to allow it to maintain its state in a power failure through flash-memory technologies like single-level cell (SLC) and multilevel cell (MLC) chips.

SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency. In addition, they consume less power. However, they are more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks, so their uses are somewhat limited. One use for SSDs is in storage arrays, where they hold file-system metadata that require high performance. SSDs are also used in some laptop computers to make them smaller, faster, and more energy-efficient. Because SSDs can be much faster than magnetic disk drives, standard bus interfaces can cause a major limit on throughput.

Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. Although 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.

A tape is kept in a spool and is wound or rewound past a read–write head. Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimetres and 1/4 and 1/2 inch. Some are named according to technology, such as LTO-5 and SDLT.

Disk Scheduling and Management

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives,

1. A fast access time and
2. High disk bandwidth.

- The **access time** has two major components;

□ The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector.

□ The **rotational latency** is the additional time waiting for the disk to rotate the desired sector to the disk head.

- 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.

One of the responsibilities of the operating system is to use the hardware efficiently.
For the disk drives,
1. A fast access time and
2. High disk bandwidth.

- The **access time** has two major components;
  - ✓ The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector.
  - ✓ The **rotational latency** is the additional time waiting for the disk to rotate the desired sector to the disk head.
- 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.

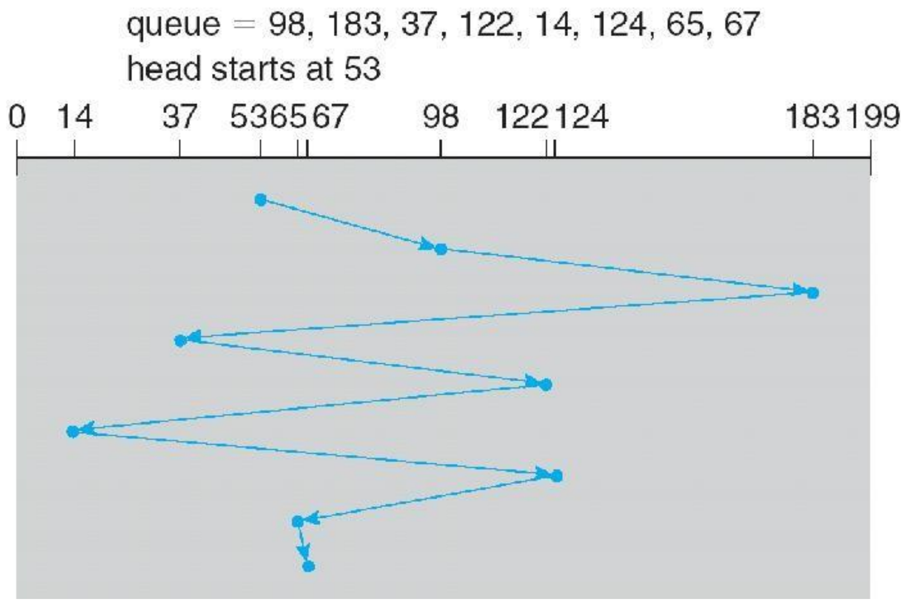
We can improve both the access time and the bandwidth by disk scheduling.
Disk scheduling: Servicing of disk I/O requests in a good order.

1. FCFS Scheduling:

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

I/O to blocks on cylinders

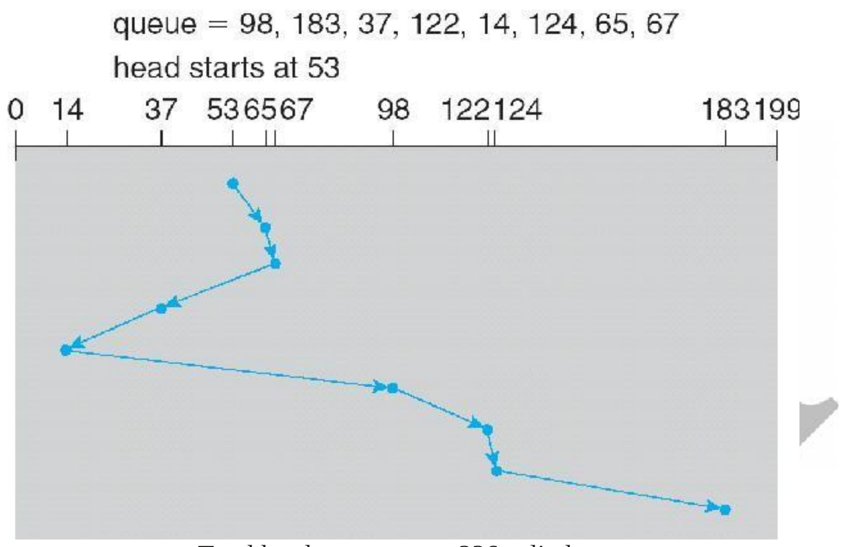
98, 183, 37, 122, 14, 124, 65, 67,



If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for **a total head movement of 640 cylinders**. The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule. 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 substantially, and performance could be thereby improved.

2. **SSTF (shortest-seek-time-first)Scheduling**

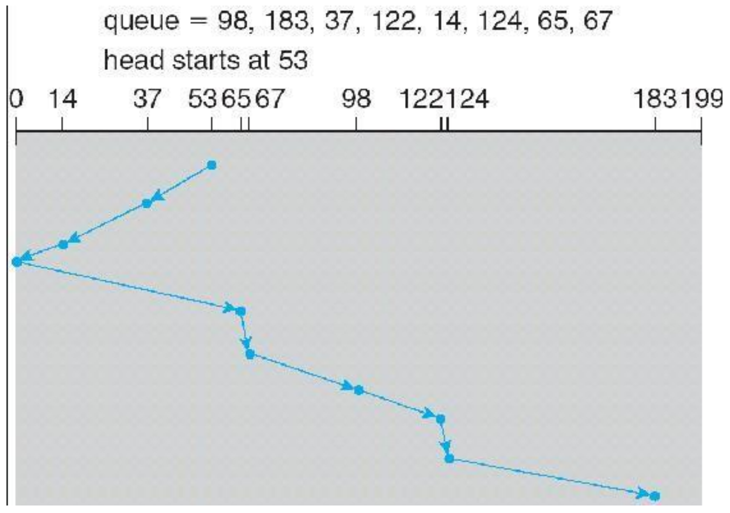
Service all the requests close to the current head position, before moving the head far away to service other requests. That is selects the request with the minimum seek time from the current head position.



Total head movement = 236 cylinders

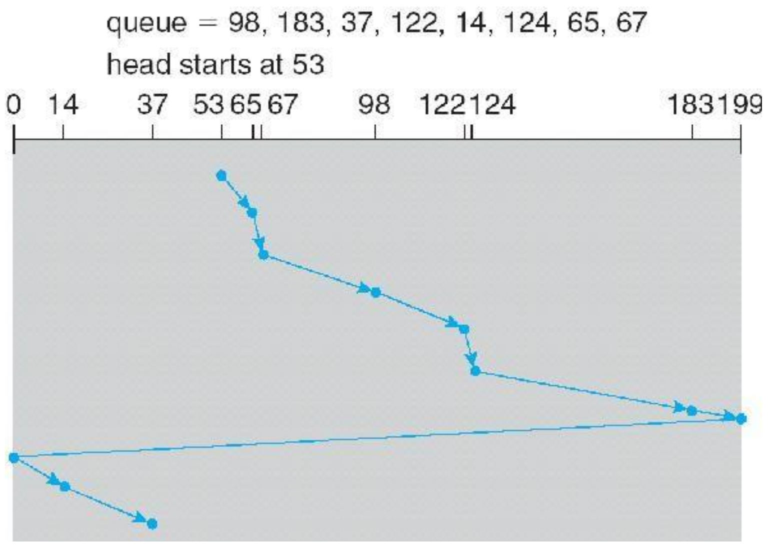
3. **SCAN Scheduling**

The disk head 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.

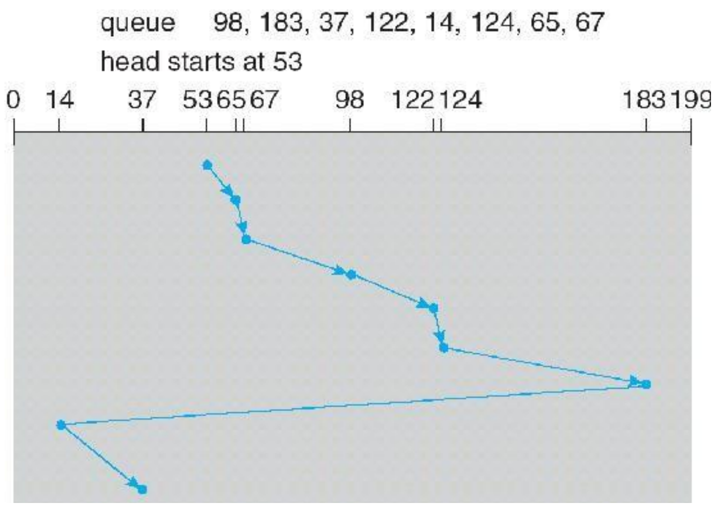


4. **C-SCAN Scheduling**

Variant of SCAN designed to provide a more uniform wait time. It 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.



5. **LOOK Scheduling**
   Both SCAN and C-SCAN move the disk arm across the full width of the disk. In this, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.



Disk Management

1. Disk Formatting:

Before a disk can store data, the sector is divided into various partitions. This process is called low-level formatting or physical formatting. It fills the disk with a special data structure for each sector. The data structure for a sector consists of

✓ Header,

✓ Data area (usually 512 bytes in size), and

✓ Trailer.

The header and trailer contain information used by the disk controller, such as a sector number and an **error-correcting code (ECC).**

This formatting enables the manufacturer to

1. Test the disk and
2. To initialize the mapping from logical block numbers

To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps.

(a) The first step is **Partition** the disk into one or more groups of cylinders. Among the partitions, one partition can hold a copy of the OS's executable code, while another holds user files.

(b) The second step is **logical formatting** .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.

2. Boot Block:

For a computer to start running-for instance, when it is powered up or rebooted-it needs to have an initial program to run. This initial program is called bootstrap program & it should be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system.

To do its job, the bootstrap program

1. Finds the operating system kernel on disk,
2. Loads that kernel into memory, and
3. Jumps to an initial address to begin the operating-system execution. The

bootstrap is stored in read-only memory **(ROM).**

Advantages:

1. ROM needs no initialization.
2. It is at a fixed location that the processor can start executing when powered up or reset.
3. It cannot be infected by a computer virus. Since, ROM is read only.

The full bootstrap program is stored in a partition called 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.

Bootstrap loader: load the entire operating system from a non-fixed location on disk, and to start the operating system running.

3. Bad Blocks:

The disk with defected sector is called as bad block.

Depending on the disk and controller in use, these blocks are handled in a variety of ways;

Method 1: "Handled manually‖

If blocks go bad during normal operation, a **special program** must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

Method 2: "sector sparing or forwarding"

The controller maintains a list of bad blocks on the disk. Then 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.

A typical bad-sector transaction might be as follows:

1. The operating system tries to read logical block 87.
2. The controller calculates the ECC and finds that the sector is bad.
3. It reports this finding to the operating system.
4. The next time that the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.
5. After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

Method 3: "sector slipping"

For an example, suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and 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 sector 17 can be mapped to it.

File System Storage-File Concepts

File Concept

A file is a named collection of related information that is recorded on secondary storage.

• From a user's perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.

Examples of files:

• A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker.

An executable file is a series of code sections that the loader can bring into memory and execute.

File Attributes

• **Name:** The symbolic file name is the only information kept in human readable form.
• **Identifier:** This unique tag, usually a number identifies the file within the file system. It is the non-human readable name for the file.
• **Type:** This information is needed for those systems that support different types.

• **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)and possibly the maximum allowed size are included in this attribute.
• **Protection:** Access-control information determines who can do reading, writing, executing and so on.
• **Time, date and user identification:** This information may be kept for creation, last modification and last use. These data can be useful for protection, security and usage monitoring.

File Operations
• Creating a file
• Writing a file
• Reading a file
• Repositioning within a file
• Deleting a file
• Truncating a file

Access Methods

1. **Sequential Access**
    a. The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.

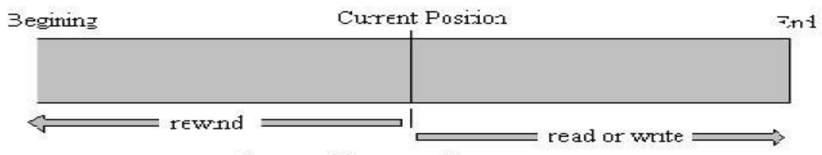


Fig 4.10 Sequential-access file

The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning and, on some systems, a program may be able to skip forward or back ward n records, for some integer n-perhaps only for n=1. Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random – access ones.

2. Direct Access

Another method is direct access (or relative access). A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct- access methods is based on a disk model of a file, since disks allow random access to any file block.

For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. Thus, we may read block 14, then read block 53, and then write block7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct – access files are of great use for immediate access to large amounts of information. Database is 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.

Directory and Disk Structure

There are five directory structures. They are
1. Single-level directory
2. Two-level directory
3. Tree-Structured directory
4. Acyclic Graph directory
5. General Graph directory

1. Single – Level Directory

- The simplest directory structure is the single- level directory.
- All files are contained in the same directory.

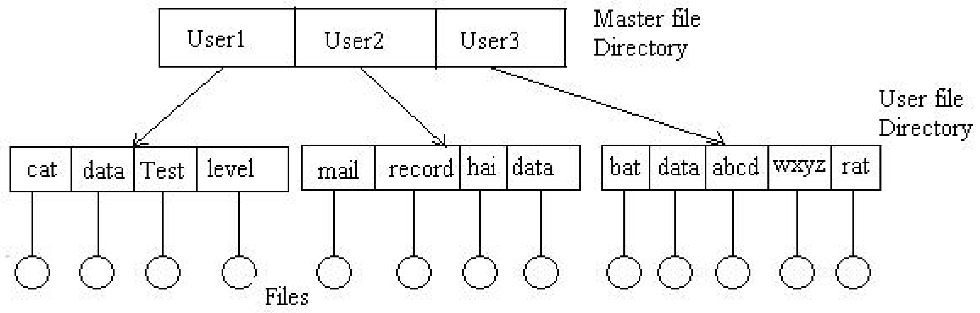
- **Disadvantage:**
  - ➢ When the number of files increases or when the system has more than one user, since all files are in the same directory, they must have unique names.

2. Two – Level Directory

- In the two level directory structures, each user has her own user file directory (UFD).
- 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.
- Although the two – level directory structure solves the name-collision problem

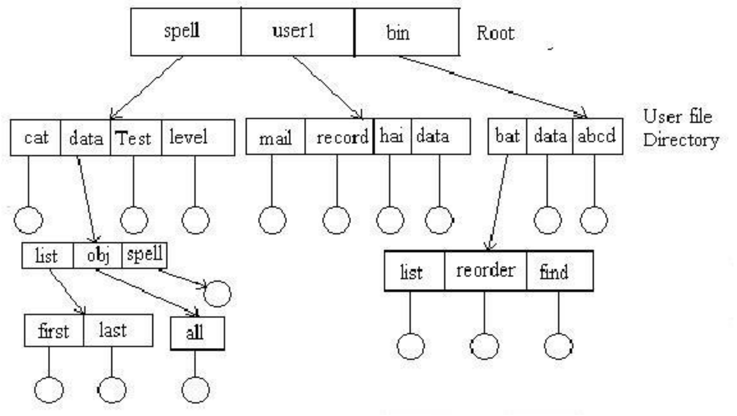
Disadvantage:

➢ Users cannot create their own sub-directories.



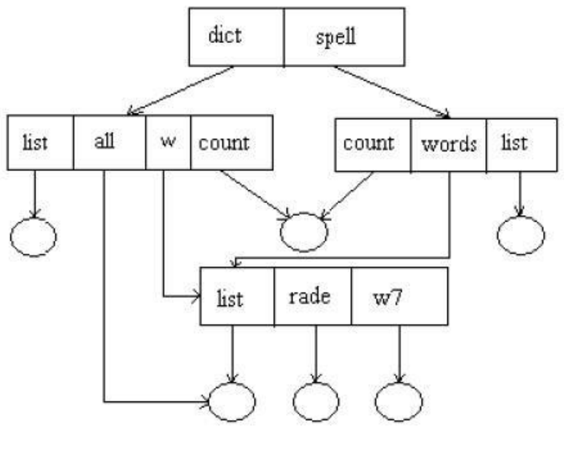
3. Tree – Structured Directory
- A tree is the most common directory structure.
- The tree has a root directory. Every file in the system has a unique path name.
- A **path name** is the path from the root, through all the subdirectories to a specified file.
- A directory (or sub directory) contains a set of files or sub directories.
- A directory is simply another file. But it is treated in a special way.
- All directories have the same internal format.

- One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).
- Special system calls are used to create and delete directories.
- Path names can be of two types: absolute path names or relative path names.
- An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.
- A relative path name defines a path from the current directory.



4. Acyclic Graph Directory.

- An acyclic graph is a graph with no cycles.
- To implement shared files and subdirectories this directory structure is used.
- An acyclic – graph directory structure is more flexible than is a simple tree structure, but it is also more complex. In a system where sharing is implemented by symbolic link, this situation is somewhat easier to handle. The deletion of a link does not need to 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. To implement this approach, we must have some mechanism for determining that the last reference to the file has been deleted.



Sharing and Protection

File Sharing

1. Multiple Users:

- When an operating system accommodates multiple users, the issues of file sharing, file naming and file protection become preeminent.
- The system either can allow user to access the file of other users by default, or it may require that a user specifically grant access to the files.
- These are the issues of access control and protection.
- To implementing sharing and protection, the system must maintain more file and directory attributes than a on a single-user system.
- The owner is the user who may change attributes, grand access, and has the most control over the file or directory.
- The group attribute of a file is used to define a subset of users who may share access to the file.
- Most systems implement owner attributes by managing a list of user names and associated user identifiers (user Ids).
- When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all of user's processes and threads. When they need to be user readable, they are translated, back to the user name via the user name list. Likewise, group functionality can be implemented as a system wide list of group names and group identifiers.
- Every user can be in one or more groups, depending upon operating system design decisions. The user's group Ids is also included in every associated process and thread.

2. Remote File System:

- Networks allowed communications between remote computers.
- Networking allows the sharing or resource spread within a campus or even around the world.
- User manually transfer files between machines via programs like **ftp**.
- A **distributed file system** (DFS) in which remote directories is visible from the local machine.
- The **World Wide Web**: A browser is needed to gain access to the remote file and separate operations (essentially a wrapper for ftp) are used to transfer files.

a) The client-server Model:

- Remote file systems allow a computer to a mount one or more file systems from one or more remote machines.
- A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client –server facility.
- Client identification is more difficult. Clients can be specified by their network name or other identifier, such as IP address, but these can be spoofed (or imitate). An unauthorized client can spoof the server into deciding that it is authorized, and the unauthorized client could be allowed access.

b) Distributed Information systems:

- Distributed information systems, also known as distributed naming service, have been devised to provide a unified access to the information needed for remote computing.
- Domain name system (DNS) provides host-name-to-network address translations for their entire Internet (including the World Wide Web).
- Before DNS was invented and became widespread, files containing the same information were sent via e-mail of ftp between all networked hosts.

c) Failure Modes:

- **Redundant arrays of inexpensive disks** (**RAID**) can prevent the loss of a disk from resulting in the loss of data.

• Remote file system has more failure modes. By nature of the complexity of networking system and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.

d) Consistency Semantics:

• It is characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously.

• These semantics should specify when modifications of data by one user are observable by other users.

• The semantics are typically implemented as code with the file system.

• A series of file accesses (that is reads and writes) attempted by a user to the same file is always enclosed between the open and close operations.

• The series of access between the open and close operations is a **file session**.

(i) UNIX Semantics:

The UNIX file system uses the following consistency semantics:

1. Writes to an open file by a user are visible immediately to other users that have this file open at the same time.

2. 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 (AFS) uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously.

2. 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 this change.

(iii) Immutable –shared File 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.

File Protection

(i) Need for file protection.

• When information is kept in a computer system, we want to keep it safe from **physical damage** (reliability) and **improper access** (protection).

• Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or though computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.

• File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally. Bugs in the file-system software can also cause file contents to be lost.

• Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multi-user system, however, other mechanisms are needed.

(ii) Types of Access

• Complete protection is provided by prohibiting access.

• Free access is provided with no protection.

• Both approaches are too extreme for general use.

• What is needed is **controlled access**.

• 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. Several different types of operations may be controlled:

1. **Read:** Read from the file.
2. **Write:** Write or rewrite the file.
3. **Execute:** Load the file into memory and execute it.
4. **Append:** Write new information at the end of the file.
5. **Delete:** Delete the file and free its space for possible reuse.
6. **List:** List the name and attributes of the file.

(iii) Access Control

• Associate with each file and directory an access-control list (ACL) specifying the user name 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.

• 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 needs to be of variable size, resulting in more complicated space management.

• To condense 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 is a group, or work group.
- ➢ **Universe:** All other users in the system constitute the universe.

File System Implementation- File System Structure

• **Disk** provide the bulk of secondary storage on which a file system is maintained.

• **Characteristics of a disk:**

1. They can be rewritten in place, it is possible to read a block from the disk, to modify the block and to write it back into the same place.
2. They can access directly any given block of information to the disk.

• To produce an efficient and convenient access to the disk, the operating system imposes one or more file system to allow the data to be stored, located and retrieved easily.

• The file system itself is generally composed of many different levels. Each level in the design uses the features of lower level to create new features for use by higher levels.

Layered File System

• The **I/O control** consists of device drivers and interrupt handlers to transfer information between the main memory and the disk 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 (for example, drive –1, cylinder 73, track 2, sector 10)

Directory Implementation

1. Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointer to the data blocks.
- A linear list of directory entries requires a linear search to find a particular entry.
- This method is simple to program but time- consuming to execute. To create a new file, we must first search the but time – consuming to execute.
- The real disadvantage of a linear list of directory entries is the linear search to find a file.

2. Hash Table

- In this method, a linear list stores the directory entries, but a hash data structure is also used.
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
- Therefore, it can greatly decrease the directory search time.
- Insertion and deleting are also fairly straight forward, although some provision must be made for collisions – situation where two file names hash to the same location.
- The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

Allocation Methods

- The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.
- There are three major methods of allocating disk space:

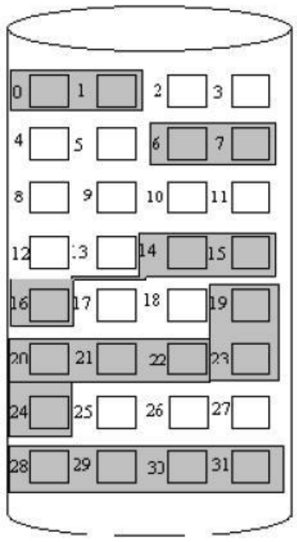
    1. Contiguous Allocation

    2. Linked Allocation

    3. Indexed Allocation

1. Contiguous Allocation

- The contiguous – allocation method requires each file to occupy a set of contiguous blocks on the disk.



Directory

file	start	length
Count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. 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.

Disadvantages:

1. Finding space for a new file.

• The contiguous disk space-allocation problem suffer from the problem of external fragmentation. As file are allocated and deleted, the free disk space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data.

2. Determining how much space is needed for a file.

• When the file is created, the total amount of space it will need must be found an allocated how does the creator know the size of the file to be created?

• If we allocate too little space to a file, we may find that file cannot be extended. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions may be repeated as long as space exists, although it can be time – consuming. However, in this case, the user never needs to be informed explicitly about what is happening ; the system continues despite the problem, although more and more slowly.

• Even if the total amount of space needed for a file is known in advance pre-allocation may be inefficient.

• A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time the file, therefore has a large amount of internal fragmentation.

To overcome these disadvantages:

• Use a modified contiguous allocation scheme, in which a contiguous chunk of space called as an **extent** is allocated initially and then, when that amount is not large enough another chunk of contiguous space an extent is added to the initial allocation.

• Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can be a problem as extents of varying sizes are allocated and deallocated.

2. Linked Allocation

• Linked allocation solves all problems of contiguous allocation.

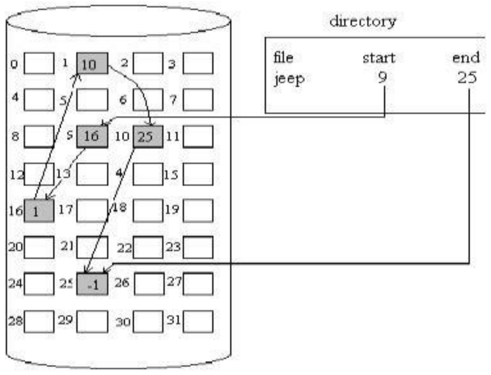
• With linked allocation, each file is a linked list of disk blocks, the disk blocks may be scattered any where on the disk.

• The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally bock 25.

• Each block contains a pointer to the next block. These pointers are not made available to the user.

• 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 does not need to the declared when that file is created. A file can continue to grow as long as free blocks are available consequently, it is never necessary to compacts disk space.



Disadvantages:

1. Used effectively only for sequential access files.

• To find the ith block of a file, we must start at the beginning of that file, and follow the pointers until we get to the ith block. Each aces to a pointer requires a disk read, and sometimes a disk seek consequently, it is inefficient to support a direct- access capability for linked allocation files.

2. Space required for the pointers

• If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

• Solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate the clusters rather than blocks. For instance, the file system may define a clusters as 4 blocks, and operate on the disk in only cluster units.

3. Reliability

• Since the files are linked together by pointers scattered all over the disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free- space list or into another file. Partial solution are to use doubly linked lists or to store the file names in a relative block number in each block; however, these schemes require even more over head for each file.

File Allocation Table(FAT)

• An important variation on the linked allocation method is the use of a file allocation table(FAT).

• This simple but efficient method of disk- space allocation is used by the MS-DOS and OS/2 operating systems.

• A section of disk at beginning of each partition is set aside to contain the table.

• The table has entry for each disk block, and is indexed by block number.

• The FAT is much as is a linked list.

• The directory entry contains the block number 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 the last block which has a special end – of – file value as the table entry.

• Unused blocks are indicated by a 0 table value.

• Allocating a new block file is a simple matter of finding the first 0 – valued table entry, and replacing the previous end of file value with the address of the new block.

• The 0 is replaced with the end – of – file value, an illustrative example is the FAT structure for a file consisting of disk blocks 217,618, and 339.

3. Indexed Allocation

• Linked allocation solves the external – fragmentation and size- declaration problems of contiguous allocation.

• Linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order.

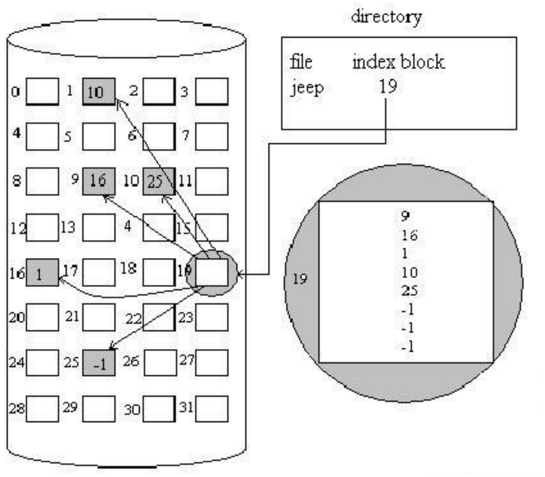
• Indexed allocation solves this problem by bringing all the pointers together into one location: the **index block**.

• Each file has its own index block, which is an array of disk – block addresses.

• The ith entry in the index block points to the ith block of the file.

• The directory contains the address of the index block .

• To read the ith block, we use the pointer in the ith index – block entry to find and read the desired block this scheme is similar to the paging scheme .



• When the file is created, all pointers in the pointers in the index block are set to nil. when the ith block is first written, a block is obtained from the free space manager, and its address is put in the ith index – block entry.
• Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.

Disadvantages

1.Pointer Overhead

• Indexed allocation does suffer from wasted space. The pointer over head of the index block is generally greater than the pointer over head of linked allocation.

2. Size of Index block

If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue:

• **Linked 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 may link together several index blocks.

• **Multilevel index:** A variant of the linked representation is to use a first level index block to point to a set of second – level index blocks.

• **Combined scheme:**

o Another alternative, used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's inode.

o The first 12 of these pointers point to direct blocks; that is for small (no more than 12 blocks) files do not need a separate index block

o The next pointer is the address of a single indirect block.

 The single indirect block is an index block, containing not data, but rather the addresses of blocks that do contain data.

o Then there is a double indirect block pointer, which contains the address of a block that contain pointers to the actual data blocks. The last pointer would contain pointers to the actual data blocks.

o The last pointer would contain the address of a triple indirect block.

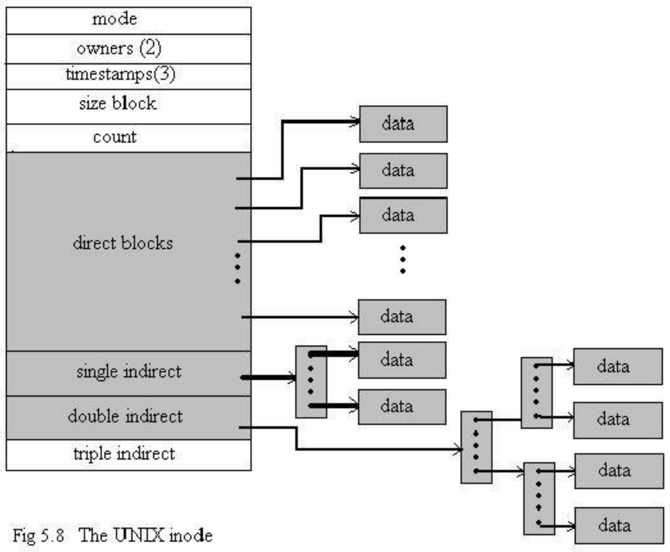


Fig 5.8 The UNIX inode

Free-space Management

- Since disk space is limited, we need to reuse the space from deleted files for new files, if possible.
- 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.

1. Bit Vector

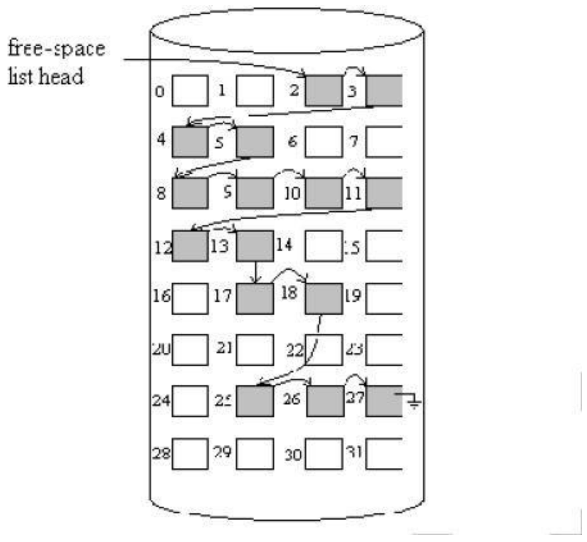
- The free-space list is implemented as a bit map or bit vector.
- Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where block 2,3,4,5,8,9,10,11,12,13,17,18,25,26 and 27 are free, and the rest of the block are allocated. The free space bit map would be

001111001111110001100000011100000 …

- The main **advantage** of this approach is its relatively simplicity and efficiency in finding the first free block, or n consecutive free blocks on the disk.

2. Linked List

- Another approach to free-space management is to 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.
- In our example, 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.
- However, this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.
- The FAT method incorporates free-block accounting data structure. No separate method is needed.



3. Grouping

• A modification of the free-list approach is to store 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 another n free blocks, and so on.

• The importance of this implementation is that the addresses of a large number of free blocks can be found quickly.

4. Counting

• We can 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.

• Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

I/O Systems

I/O Hardware

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. A device communicates with a computer system by sending signals over a cable or even through the air.

Port: The device communicates with the machine via a connection point (or port), for example, a serial port.

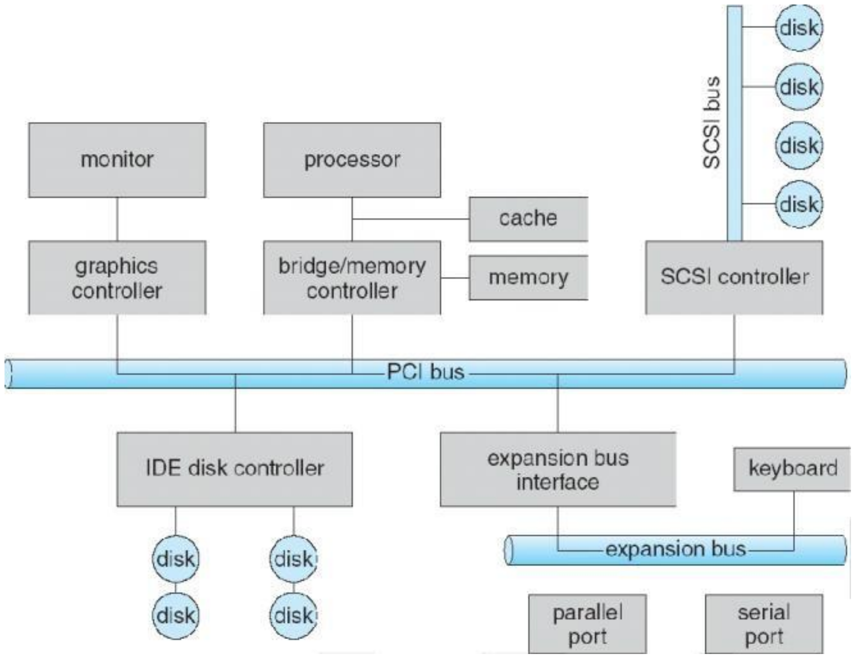
Bus: If one or more devices use a common set of wires, the connection is called a bus.

Daisy chain: 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.

PC bus structure:

A PCI bus that connects the processor-memory subsystem to the fast devices, and an expansion bus that connects relatively slow devices such as the keyboard and serial and parallel ports. In the upper-right portion of the figure, four disks are connected together on a SCSI bus plugged into a SCSI controller.

A **controller or host adapter** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. It is a single chip in the computer that controls the signals on the wires of a serial port. By contrast, a SCSI bus controller is not simple. Because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board. It typically contains a processor, microcode, and some private memory. Some devices have their own built-in controllers.



- How can the processor give commands and data to a controller to accomplish an I/O transfer?
  - o Direct I/O instructions

  - o Memory-mapped I/O

Direct I/O instructions
Use special I/O instructions that specify the transfer of a byte or word to an I/O port address. The I/O instruction triggers bus lines to select the proper device and to move bits into or out of a device register
Memory-mapped I/O
The device-control registers are mapped into the address space of the processor. The CPU executes I/O requests using the standard data-transfer instructions to read and write the device-control registers.
An I/O port typically consists of four registers: status, control, data-in, and data-out registers.

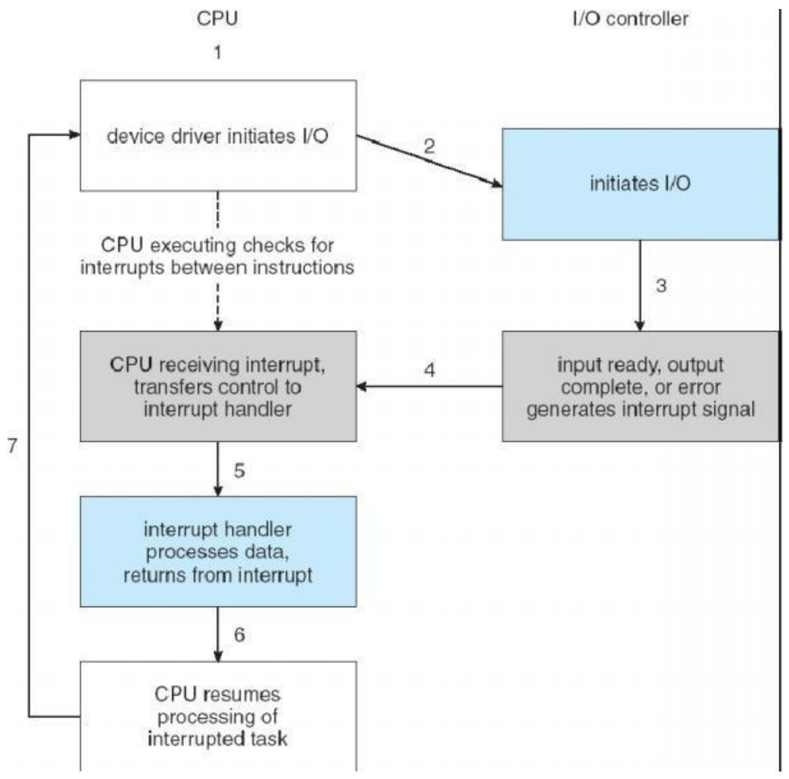
Status register	Read by the host to indicate states such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether there has been a device error.
Control register	Written by the host to start a command or to change the mode of a device.
data-in register	Read by the host to get input
data-out register	Written by the host to send output

Polling
Interaction between the host and a controller
- 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 sets the command ready bit when a command is available for the controller to execute.

Coordination between the host & the controller is done by handshaking as follows:
1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the busy bit.
5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte, and does the I/O to the device.
6. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.
In step 1, the host is **busy-waiting or polling** : It is in a loop, reading the status register over and over until the busy bit becomes clear.

Interrupts
The CPU hardware has a wire called the interrupt-request line .
The basic interrupt mechanism works as follows;
1. Device controller raises an interrupt by asserting a signal on the interrupt request line.
2. The CPU catches the interrupt and dispatches to the interrupt handler and
3. The handler clears the interrupt by servicing the device.
Two interrupt request lines:
1. **Nonmaskable interrupt**: which is reserved for events such as unrecoverable memory errors?
2. **Maskable interrupt**: Used by device controllers to request service


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

Application I/O Interface
- I/O system calls encapsulate device behaviors in generic classes
- Device-driver layer hides differences among I/O controllers from kernel

- Devices vary in many dimensions

1. Character-stream or block
2. Sequential or random-access
3. Sharable or dedicated
4. Speed of operation
5. read-write, read only, or write only

Types	**Description**	**Example**
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.	Terminal, Disk
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.	Modem, CD-ROM
Sharable or dedicated	A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.	Tape, Keyboard
Speed of operation	Latency, seek time, transfer rate, delay between operations	
read-write, read only, or write only	Some devices perform both input and output, but others support only one data direction.	CD-ROM, Graphics controller, Disk

Block and Character Devices

Block-device: The block-device interface captures all the aspects necessary for accessing disk drives and other block-oriented devices. The device should understand the commands such as read () & write (), and if it is a random access device, it has a seek() command to specify which block to transfer next.

Applications normally access such a device through a file-system interface. The OS itself, and 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 sometimes called **raw I/O.**

Memory-mapped file access can be layered on top of block-device drivers. Rather than offering read and write operations, a memory-mapped interface provides access to disk storage via an array of bytes in main memory.

Character Devices: A keyboard is an example of a device that is accessed through a character stream interface. The basic system calls in this interface enable an application to get() or put() one character.

On top of this interface, libraries can be built that offer line-at-a-time access, with buffering and editing services.

(+) This style of access is convenient for input devices where it produce input "spontaneously".

(+) This access style is also good for output devices such as printers or audio boards, which naturally fit the concept of a linear stream of bytes.

Network Devices

Because the performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the read0 -write() - seek() interface used for disks.

• Windows NT provides one interface to the network interface card, and a second interface to the network protocols.

• In UNIX, we find half-duplex pipes, full-duplex FIFOs, full-duplex STREAMS, message queues and sockets.

Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

1. Give the current time
2. Give the elapsed time
3. Set a timer to trigger operation X at time T

These functions are used by the operating system & also by time sensitive applications. 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 to generate an interrupt. To generate periodic interrupts, it can be set to do this operation once or to repeat.

Blocking and Non-blocking I/O (or) synchronous & asynchronous:

Blocking 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, at which time it will receive the values returned by the system call.

Non-blocking I/O: Some user-level processes need non-blocking I/O.

Examples: 1. User interface that receives keyboard and mouse input while processing and displaying data on the screen.

2. Video application that reads frames from a file on disk while simultaneously decompressing and displaying the output on the display.

Kernel I/O Subsystem

Kernels provide many services related to I/O.

- One way that the I/O subsystem improves the efficiency of the computer is by scheduling I/O operations.
- Another way is by using storage space in main memory or on disk, via techniques called buffering, caching, and spooling.

I/O Scheduling:

To determine a good order in which to execute the set of I/O requests.

Uses:

a) It can improve overall system performance,

b) It can share device access fairly among processes, and

c) It can reduce the average waiting time for 1/0 to complete.

Implementation: OS developers implement scheduling by maintaining a queue of requests‖ for each device.

1. When an application issues a blocking I/O system call,
2. The request is placed on the queue for that device.

3. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications.

Buffering:

Buffer: A memory area that stores data while they are transferred between two devices or between a device and an application.

Reasons for buffering:

a) To cope with a speed mismatch between the producer and consumer of a data stream.

b) To adapt between devices that have different data-transfer sizes.

c) To support copy semantics for application I/O.

Copy semantics: Suppose that an application has a buffer of data that it wishes to write to disk. It calls the write () system call, providing a pointer to the buffer and an integer specifying the number of bytes to write.

After the system call returns, what happens if the application changes the contents of the buffer? With copy semantics, the version of the data written to disk is guaranteed to be the version at the time of the application system call, independent of any subsequent changes in the application's buffer.

A simple way that the operating system can guarantee copy semantics is for the write() system call to copy the application data into a kernel buffer before returning control to the application. The disk write is performed from the kernel buffer, so that subsequent changes to the application buffer have no effect.

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

Cache vs buffer: A buffer may hold the only existing copy of a data item, whereas a cache just holds a copy on faster storage of an item that resides elsewhere.

When the kernel receives a file I/O request,

1. The kernel first accesses the buffer cache to see whether that region of the file is already available in main memory.

2. If so, a physical disk I/O can be avoided or deferred. Also, disk writes are accumulated in the buffer cache for several seconds, so that large transfers are gathered to allow efficient write schedules.

Spooling and Device Reservation:

Spool: A buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. 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 os provides a control interface that enables users and system administrators ;

a) To display the queue,

b) To remove unwanted jobs before those jobs print,

c) To suspend printing while the printer is serviced, and so on.

Device reservation - provides exclusive access to a device

- System calls for allocation and de-allocation
- Watch out for deadlock

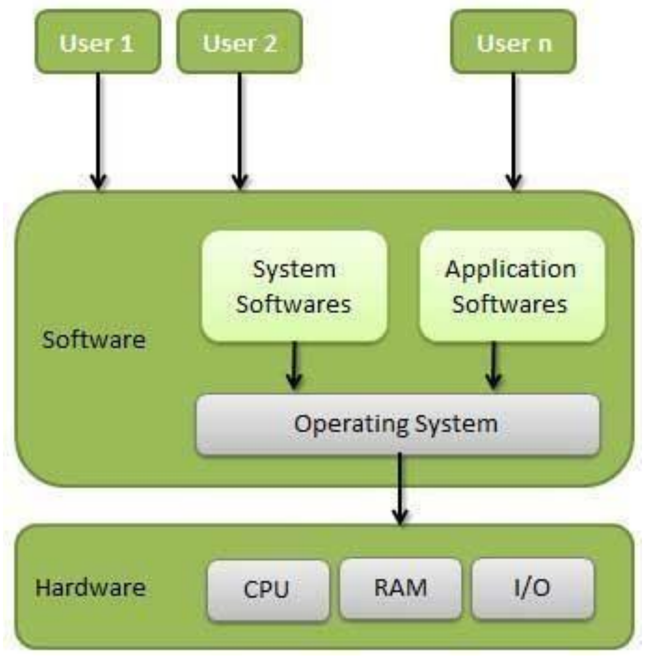
Error Handling:

- An operating system that uses protected memory can guard against many kinds of hardware and application errors.
- OS can recover from disk read, device unavailable, transient write failures
- Most return an error number or code when I/O request fails
- System error logs hold problem reports.

UNIX V CASE STUDIES

5.1 The Linux System

- An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs. The Linux open source operating system, or Linux OS, is a freely distributable, cross-platform operating system based on UNIX.
- The Linux consist of a kernel and some system programs. There are also some application programs for doing work. The kernel is the heart of the operating system which provides a set of tools that are used by system calls.
- The defining component of Linux is the Linux kernel, an operating system kernel first released on 5 October 1991 by *Linus Torvalds.*



- A Linux-based system is a modular Unix-like operating system. It derives much of its basic design from principles established in UNIX. Such a system uses a monolithic kernel which handles process control, networking, and peripheral and file system access.

5.2 Important features of Linux Operating System

- **Portable** - Portability means software can work on different types of hardware in same way. Linux kernel and application programs supports their installation on any kind of hardware platform.
- **Open Source** - Linux source code is freely available and it is community based development project.
- **Multi-User** & **Multiprogramming** - Linux is a multiuser system where multiple users can access system resources like memory/ ram/ application programs at same time. Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** - Linux provides a standard file structure in which system files/ user filesar

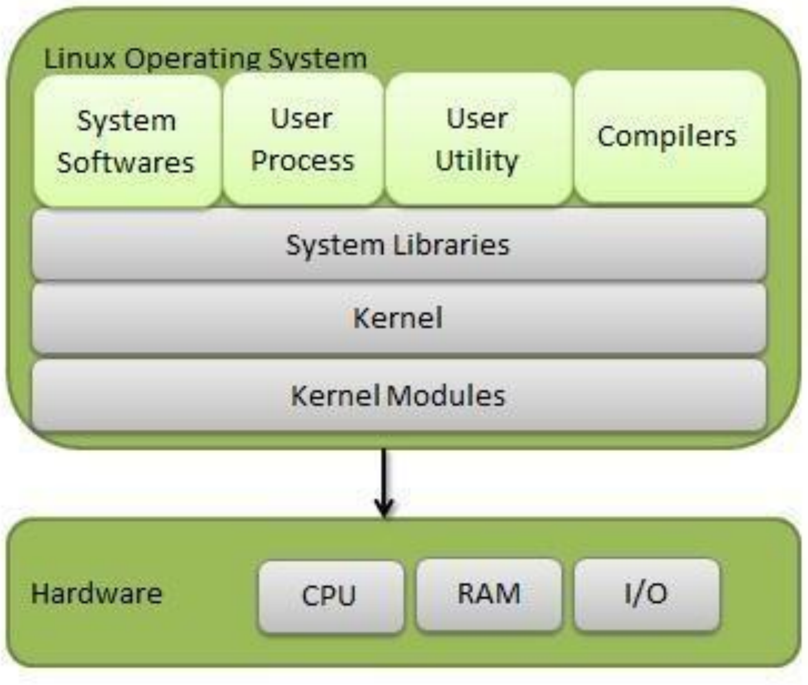
arranged.

- **Shell** - Linux provides a special interpreter program which can be used to execute commands of the operating system.
- **Security** - Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

5.3 Components of Linux System

Linux Operating System has primarily three components

- **Kernel** - Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It is consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.
- **System Library** - System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implements most of the functionalities of the operating system and do not requires kernel module's code access rights.
- **System Utility** - System Utility programs are responsible to do specialized, individual level tasks

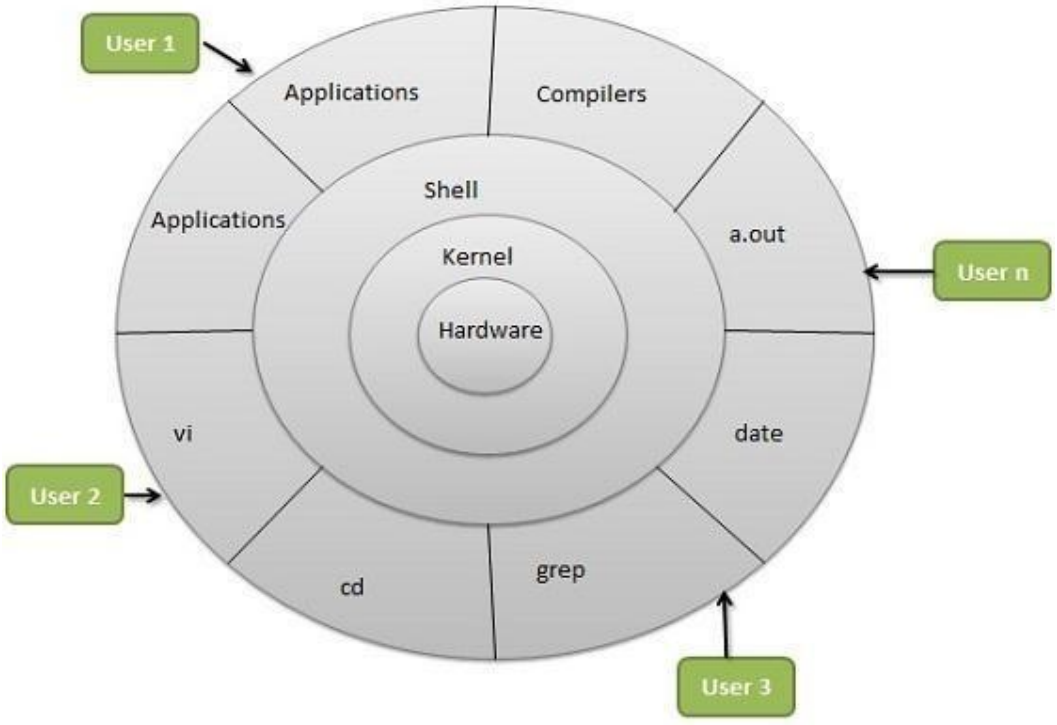


Installed components of a Linux system include the following:

- A **bootloader** is a program that loads the Linux kernel into the computer's main memory, by being executed by the computer when it is turned on and after the firmware initialization is performed.
- An **init** program is the first process launched by the Linux kernel, and is at the root of the process tree.
- **Software libraries**, which contain code that can be used by running processes. The most commonly used software library on Linux systems, the GNU C Library (glibc), C standard library and Widget toolkits.
- **User interface programs** such as command shells or windowing environments. The user

interface, also known as the shell, is either a command-line interface (CLI), a graphical user interface (GUI), or through controls attached

5.4 Architecture



Linux System Architecture is consists of following layers

1. **Hardware layer** - Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).
2. **Kernel** - Core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
3. **Shell** - An interface to kernel, hiding complexity of kernel's functions from users. Takes commands from user and executes kernel's functions.
4. **Utilities** - Utility programs giving user most of the functionalities of an operating systems.

5.5 Modes of operation

- **Kernel Mode:**
  - Kernel component code executes in a special privileged mode called *kernel mode* with full access to all resources of the computer.
  - This code represents a single process, executes in single address space and do not require any context switch and hence is very efficient and fast.
  - Kernel runs each processes and provides system services to processes, provides protected access to hardware to processes.
- **User Mode:**
  - The system programs use the tools provided by the kernel to implement the various services required from an operating system. System programs, and all other programs, run `on top of the kernel', in what is called the user mode.
  - Support code which is not required to run in kernel mode is in System Library.

- User programs and other system programs work in User Mode which has no access to system hardware and kernel code.
- User programs/ utilities use System libraries to access Kernel functions to get system's low level tasks.

5.6 Major Services provided by LINUX System

1. **Initialization (init)**

   The single most important service in a LINUX system is provided by **init** program. The ***init*** is started as the first process of every LINUX system, as the last thing the kernel does when it boots. When init starts, it continues the boot process by doing various startup chores (checking and mounting file systems, starting daemons, etc).

2. **Logins from terminals (getty)**

   Logins from terminals (via serial lines) and the console are provided by the **getty** program. **init** starts a separate instance of **getty** for each terminal upon which logins are to be allowed. Getty reads the username and runs the login program, which reads the password. If the username and password are correct, login runs the shell.

3. **Logging and Auditing (syslog)**

   The kernel and many system programs produce error, warning, and other messages. It is often important that these messages can be viewed later, so they should be written to a file. The program doing this logging operation is known as **syslog**.

4. **Periodic command execution (cron & at)**

   Both users and system administrators often need to run commands periodically. For example, the system administrator might want to run a command to clean the directories with temporary files from old files, to keep the disks from filling up, since not all programs clean up after themselves correctly.

   - The **cron** service is set up to do this. Each user can have a *crontab* file, where the lists the commands wish to execute and the times they should be executed.
   - The **at** service is similar to cron, but it is once only: the command is executed at the given time, but it is not repeated.

5. **Graphical user interface**

   - UNIX and Linux don't incorporate the user interface into the kernel; instead, they let it be implemented by user level programs. This applies for both text mode and graphical environments. This arrangement makes the system more flexible.

- The graphical environment primarily used with Linux is called the X Window System (X for short) that provides tools with which a GUI can be implemented. Some popular window managers are blackbox and windowmaker. There are also two popular desktop managers, KDE and Gnome.

6. Network logins (telnet, rlogin & ssh)

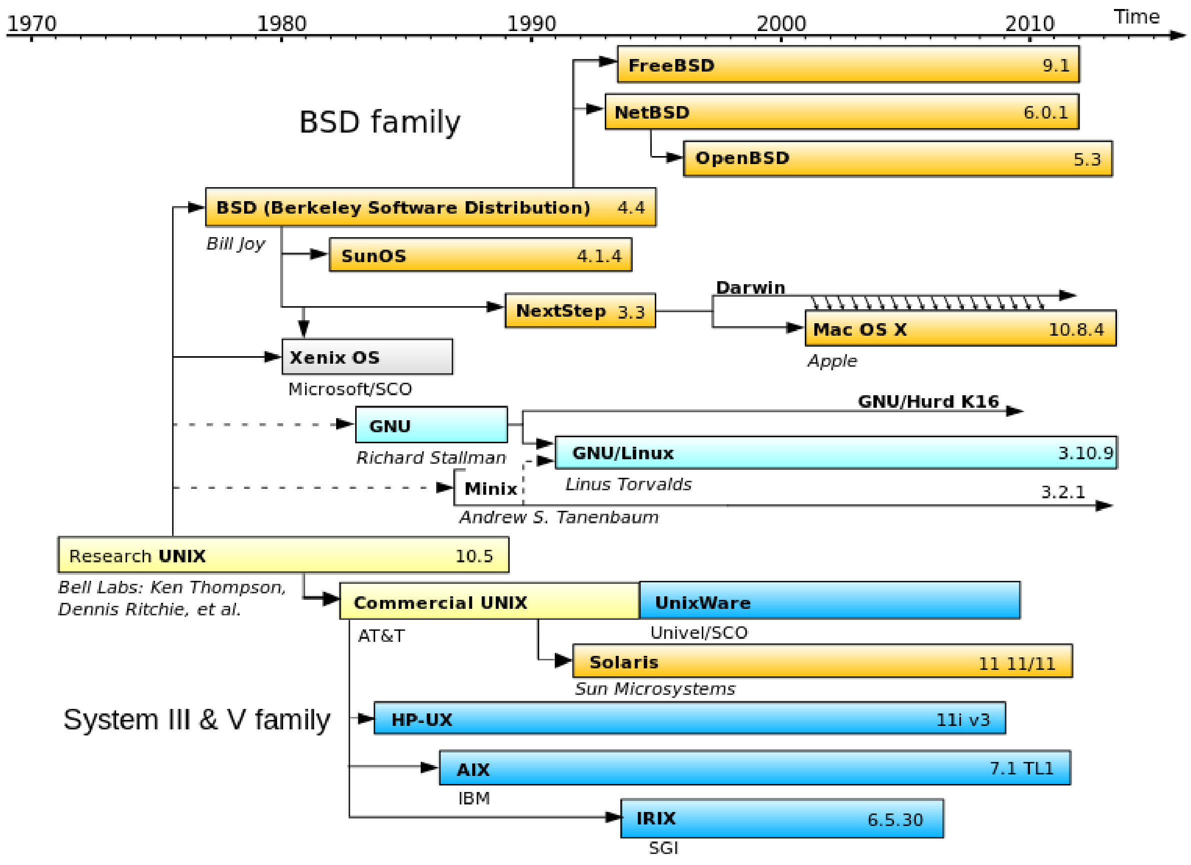
Network logins work a little differently than normal logins. For each person logging in via the network there is a separate virtual network connection. It is therefore not possible to run a separate getty for each virtual connection. There are several different ways to log in via a network, **telnet** and **ssh** being the major ones in TCP/IP networks.

Most of Linux system administrators consider telnet and rlogin to be insecure and prefer ssh, the ``secure shell'', which encrypts traffic going over the network, thereby making it far less likely that the malicious can ``sniff'' the connection and gain sensitive data like usernames and passwords.

7. Network File System (NFS & CIFS)

One of the more useful things that can be done with networking services is sharing files via a network file system. Depending on your network this could be done over the Network File System (NFS), or over the Common Internet File System (CIFS).

NFS is typically a 'UNIX' based service. In Linux, NFS is supported by the kernel. CIFS however is not. In Linux, CIFS is supported by **Samba**. With a network file system any file operations done by a program on one machine are sent over the network to another computer.



5.7 SYSTEM ADMINISTRATOR

- A system administrator is a person who is responsible for the configuration and reliable operation of computer systems, especially multi-user computers, such as servers.
- The system administrator seeks to ensure that the uptime, performance, resources, and security of the computers without exceeding the budget.
- To meet these needs, a system administrator may acquire, install, or upgrade computer components and software, provide routine automation, maintain security policies AND troubleshoot.

5.7.1 Responsibilities of a System Administrator

A system administrator's responsibilities might include:

- Installing and configuring new hardware and software.
- Applying operating system updates, patches, and configuration changes.
- Analyzing system logs and identifying potential issues with computer systems.
- Introducing and integrating new technologies into existing data center environments and configuring, adding, and deleting file systems.
- Performing routine audit of systems and software.
- Adding, removing, or updating user account information, resetting passwords, etc.
- Responsibility for security and documenting the configuration of the system.
- Troubleshooting any reported problems.
- System performance tuning.

5.7.2 Various System Administrator Roles

In a larger company, these may all be separate positions within a computer support or Information Services (IS) department. In a smaller group they may be shared by a few sysadmins, or even a single person.

- A **database administrator** (DBA) maintains a database system, and is responsible for the integrity of the data and the efficiency and performance of the system.
- A **network administrator** maintains network infrastructure such as switches and routers, and diagnoses problems with these or with the behaviour of network-attached computers.
- A **security administrator** is a specialist in computer and network security, including the administration of security devices such as firewalls, as well as consulting on general security measures.

- A **web administrator** maintains web server services (such as Apache or IIS) that allow for internal or external access to web sites. Tasks include managing multiple sites, administering security, and configuring necessary components and software.
- A **computer operator** performs routine maintenance and upkeep, such as changing backup tapes or replacing failed drives in a redundant array of independent disks (RAID).
- A **postmaster** administers a mail server.
- A **Storage Administrator (SAN)** can create, provision, add or remove Storage to/from Computer systems. Storage can be attached locally to the system or from a storage area network (SAN) or network-attached storage (NAS).

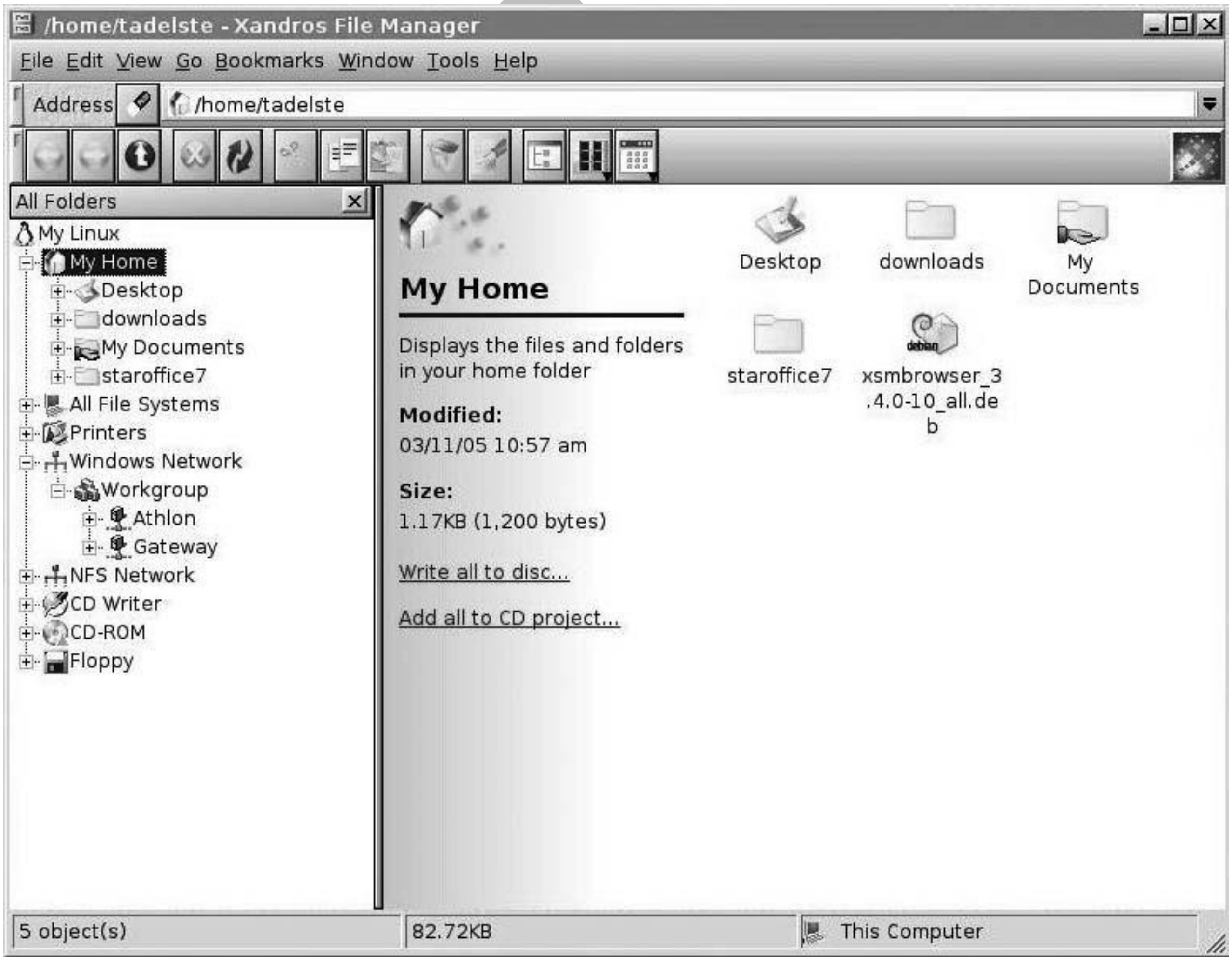
5.7.3 Requirements for LINUX system administrator

1. While specific knowledge is a boon, system administrator should possess basic knowledge about all aspects of Linux. For example, a little knowledge about Solaris, BSD, nginx or various flavors of Linux.
2. Knowledge in at least one of the upper tier scripting language such as Python, Perl, Ruby or more.
3. To be a system administrator, he/she at least needs to have some hands-on experience of system management, system setup and managing Linux or Solaris based servers as well as configuring them.
4. Knowledge in shell programming such as Buorne or Korn and architecture.
5. Knowledge about storage technologies like FC, NFS or iSCSI is great, while knowledge regarding backup technologies is a must for a system administrator.
6. Knowledge in testing methodologies like Subversion or Git is great, while knowledge of version control is also an advantage.
7. Knowledge about basics of configuration management tools like Puppet and Chef.
8. Skills with system and application monitoring tools like SNMP or Nagios are also important, as they show your ability as an administrator in a team setting.
9. Knowledge about how to operate virtualized VMWare or Xen Server, Multifunction Server and Samba
10. An ITIL Foundation certification for Linux system administrator.

5.8 SETTING UP A LINUX MULTIFUNCTION SERVER

A Linux machine can be configured as a server either by compiling several well-defined scripts and off-line downloaded packages or through on-line installation method. Setting up a multifunction server, the system administrator should have knowledge about a series of shell commands. A Linux machine can be configured as any of following application servers such as,

- A Web Server (Apache 2.0.x)
- A Mail Server (Postfix)
- A DNS Server (BIND 9)
- An FTP Server (ProFTPD)
- Mail Delivery Agents (POP3/POP3s/IMAP/IMAPs)
- Webalizer for web site statistics



My Home

Displays the files and folders in your home folder

Modified:
03/11/05 10:57 am

Size:
1.17KB (1,200 bytes)

5 object(s) | 82.72KB

Files and directories shared by Linux system, as viewed from a Windows PC

5.8.1 Server Requirements

To set up a Linux Internet server, we will need a connection to the Internet and a static IP address. The system can also be setup with the address leased by ISP and configure it statically.

Computer with at least a Pentium III CPU, a minimum of 256 MB of RAM, and a 10 GB hard drive is preferred. Obviously, a newer CPU and additional memory will provide better performance. This chapter is based on Debian's stable version. We strongly suggest using a CD with the Netinstall kernel. The Debian web site provides downloadable CD images.

5.8.2 Installing & Configuring Network Services

Administrator should log into the server from a remote console on desktop. It is recommended to do further administration from another system (even a laptop), because a secure server normally runs in what is called headless mode—that is, it has no monitor or keyboard.

Get used to administering the server like this. A SSH client on the remote machine is needed which virtually all Linux distributions have and which can be downloaded for other operating systems as well.

Configuring the Network

If DHCP is used during the Debian installation, Server with a static IP address should be configured as follows,

1. To change the settings to use a static IP address, you'll need to become root and edit the file /etc/network/interfaces to suit your needs. As an example, we'll use the IP address 70.153.258.42.
2. To add the IP address 70.153.258.42 to the interface eth0, we must change the file to look like this (you'll have to obtain some of the information from your ISP):

```
auto      eth0
iface     eth0 inet static
address   70.153.258.42
netmask   255.255.255.248
network   70.153.258.0
broadcast 70.153.258.47
gateway   70.153.258.46
```

3. After editing the /etc/network/interfaces file, restart the network by entering:

```
# /etc/init.d/networking restart
```

4. To edit /etc/resolv.conf and add nameservers to resolve Internet hostnames to their corresponding IP addresses. At this point, we will simply set up a minimal DNS
   server. Our ***resolv.conf*** looks as follows:

```
search    server
nameserver 70.153.258.42
nameserver 70.253.158.45
nameserver 151.164.1.8
```

5. Now edit /etc/hosts and add your IP addresses:

```
127.0.0.1 localhost.localdomain  localhost server1
70.153.258.42  server1.centralsoft.org  server1
```

6. Now, to set the hostname, enter these commands:

```
# echo server1.centralsoft.org > /etc/hostname
# /bin/hostname -F /etc/hostname
```

7. verify that you configured your hostname correctly by running the *hostname* command:

```
~$ hostname -f
server1.centralsoft.org
```

5.9 Providing Domain Name Services (BIND - the ubiquitous DNS server)

- Debian provides a stable version of BIND in its repositories. BIND can be installed, setup and secure it in a *chroot* environment, meaning it won't be able to see or access files outside its own directory tree. This is an important security technique.
- The term *chroot* refers to the trick of changing the root filesystem (the /directory) that a process sees, so that most of the system is effectively inaccessible to it.
- The BIND server also can be configured to run as a non-root user. That way, if someone gains access to BIND, he/she won't gain root privileges or be able to control other processes.

1. To install BIND on your Debian server, run this command:

```
# apt-get install bind9
```

Debian downloads and configures the file as an Internet service and the status can be seen on the console:

```
Setting up bind9 (9.2.4-1)
Adding group `bind' (104) - Done.
Adding system user `bind'
Adding new user `bind' (104) with group `bind'.
Not creating home directory.
Starting domain name service: named.
```

2. To put BIND in a secured environment, create a directory where the service can run unexposed to other processes. First stop the service by running the following command:

```
# /etc/init.d/bind9 stop
```

3. Edit the file /etc/default/bind9 so that the daemon will run as the unprivileged user bind, chrooted to /var/lib/named. Change the line:

   ```
   OPTS="-u bind"
   ```

   So that it reads:

   ```
   OPTIONS="-u bind -t /var/lib/named"
   ```

4. To provide a complete environment for running BIND, create the necessary directories under /var/lib:

   ```
   # mkdir -p /var/lib/named/etc
   # mkdir /var/lib/named/dev
   # mkdir -p /var/lib/named/var/cache/bind
   # mkdir -p /var/lib/named/var/run/bind/run
   ```

   Then move the config directory from /etc to /var/lib/named/etc:

   ```
   # mv /etc/bind /var/lib/named/etc
   ```

   Next, create a symbolic link to the new config directory from the old location, to avoid problems when BIND is upgraded in the future:

   ```
   # ln -s /var/lib/named/etc/bind /etc/bind
   ```

   Make null and random devices for use by BIND, and fix the permissions of the directories:

   ```
   # mknod /var/lib/named/dev/null c 1 3
   # mknod /var/lib/named/dev/random c 1 8
   ```

   Then change permissions and ownership on the files:

   ```
   # chmod 666 /var/lib/named/dev/null
               /var/lib/named/dev/random
   # chown -R bind:bind /var/lib/named/var/*
   # chown -R bind:bind /var/lib/named/etc/bind
   ```

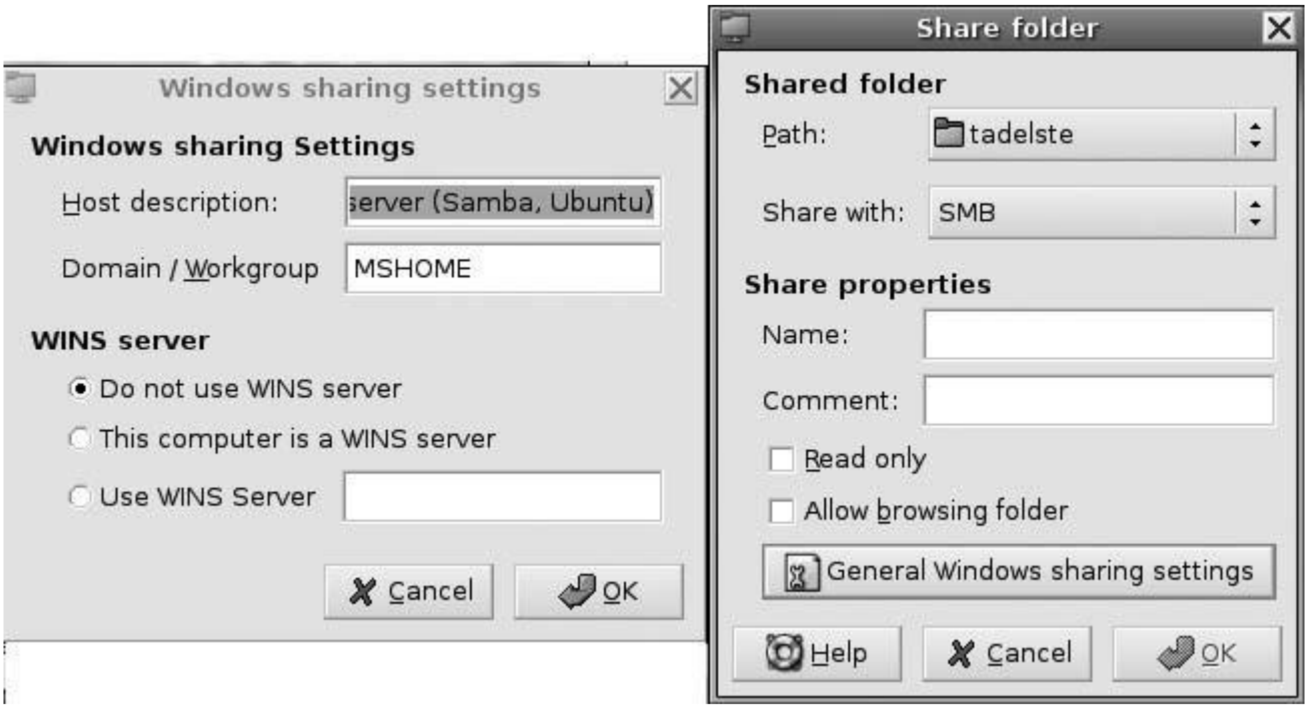
5. Finally, start BIND:

   ```
   # /etc/init.d/bind9 start
   ```

6. To check whether named is functioning without any trouble. Execute this command:

   ```
   server1:/home/admin# rndc status
   number of zones: 6
   debug level: 0
   xfers running: 0
   xfers deferred: 0
   soa queries in progress: 0
   query logging is OFF
   ```

```
server is up and running
server1:/home/admin#
```



Windows sharing settings

Windows sharing Settings

Host description: server (Samba, Ubuntu)

Domain / Workgroup: MSHOME

WINS server

● Do not use WINS server

○ This computer is a WINS server

○ Use WINS Server

Share folder

Shared folder

Path: tadelste

Share with: SMB

Share properties

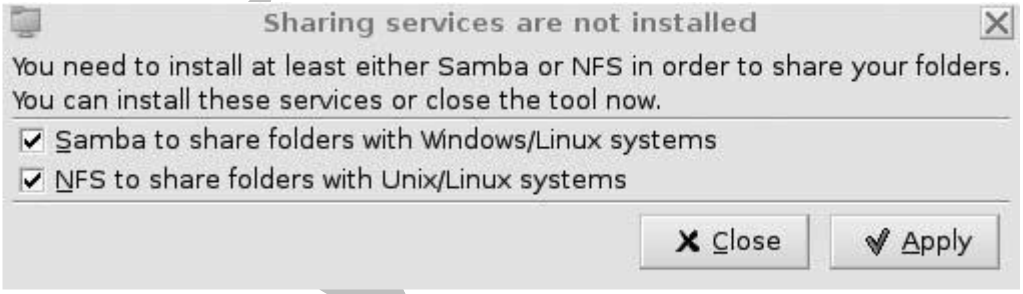
Name:

Comment:

☐ Read only

☐ Allow browsing folder

Setting up Ubuntu shares in a Windows environment



Sharing services are not installed

You need to install at least either Samba or NFS in order to share your folders.
You can install these services or close the tool now.

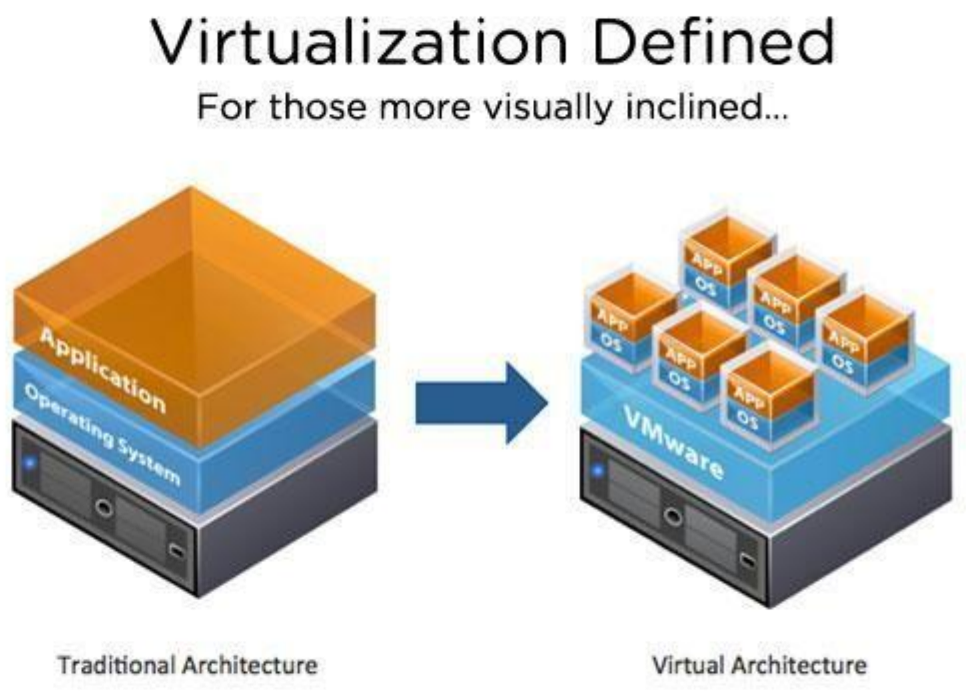
☑ Samba to share folders with Windows/Linux systems

☑ NFS to share folders with Unix/Linux systems

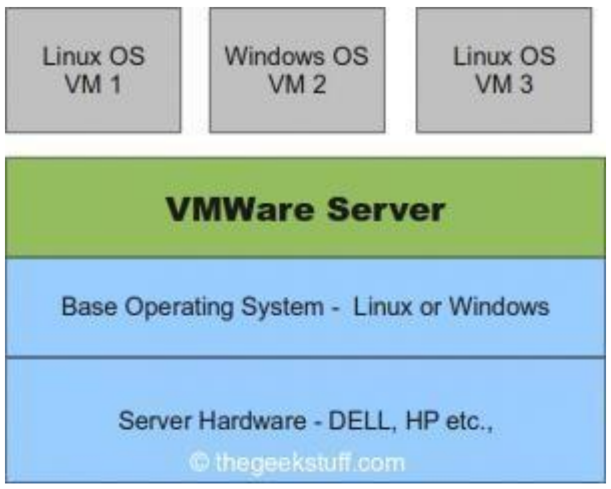
Ubuntu's setup screen for file-sharing services

5.10 Virtualization

- Virtualization refers to the act of creating a virtual (rather than actual) version of something, including a virtual computer hardware platform, operating system (OS), storage device, or computer network resources.



Traditional Architecture vs. Virtual Architecture



Linux OS VM 1
Windows OS VM 2
Linux OS VM 3
VMWare Server
Base Operating System - Linux or Windows
Server Hardware - DELL, HP etc.,
© thegeekstuff.com

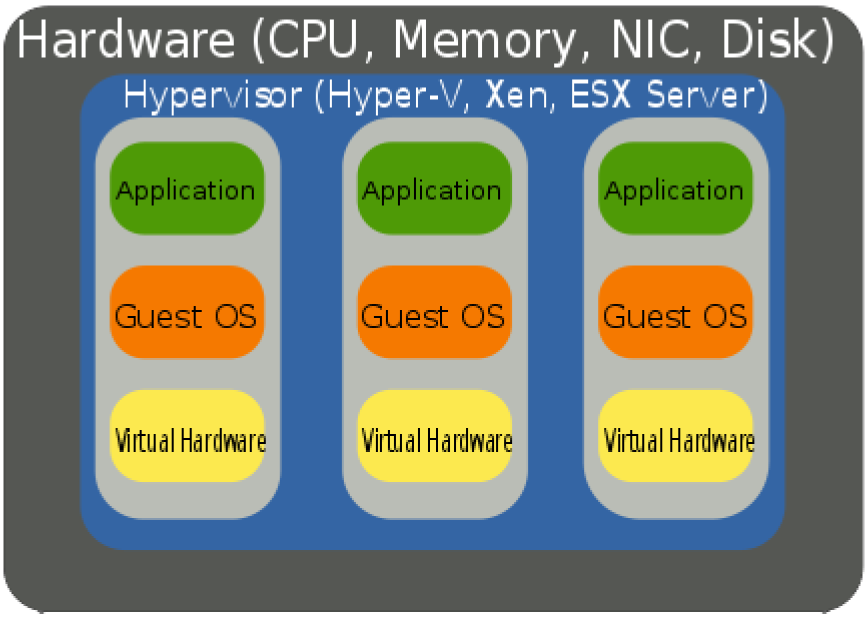
Virtual Machine Server – A Layered Approach

- *Hardware virtualization* or *platform virtualization* refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is separated from the underlying hardware resources.

- *Hardware virtualization* hides the physical characteristics of a computing platform from

material

- For example, a computer that is running Microsoft Windows may host a virtual machine that looks like a computer with the Ubuntu Linux operating system; Ubuntu-based software can be run on the virtual machine.



Hardware Virtualization

Benefits of Virtualization

1. Instead of deploying several physical servers for each service, only one server can be used. Virtualization let multiple OSs and applications to run on a server at a time. Consolidate hardware to get vastly higher productivity from fewer servers.
2. If the preferred operating system is deployed as an image, so we needed to go through the installation process only once for the entire infrastructure.
3. **Improve business continuity:** Virtual operating system images allow us for instant recovery in case of a system failure. The crashed system can be restored back by coping the virtual image.
4. **Increased uptime:** Most server virtualization platforms offer a number of advanced features that just aren't found on physical servers which increases servers' uptime. Some of features are live migration, storage migration, fault tolerance, high availability, and distributed resource scheduling.
5. **Reduce capital and operating costs:** Server consolidation can be done by running multiple virtual machines (VM) on a single physical server. Fewer servers means lower capital and operating costs.

Architecture - Virtualization

The heart of virtualization is the “virtual machine” (VM), a tightly isolated software container with an operating system and application inside. Because each virtual machine is completely separate and independent, many of them can run simultaneously on a single computer. A thin layer of software called a hypervisor decouples the virtual machines from the host and dynamically allocates computing resources to each virtual machine as needed.

This architecture redefines your computing equation and delivers:

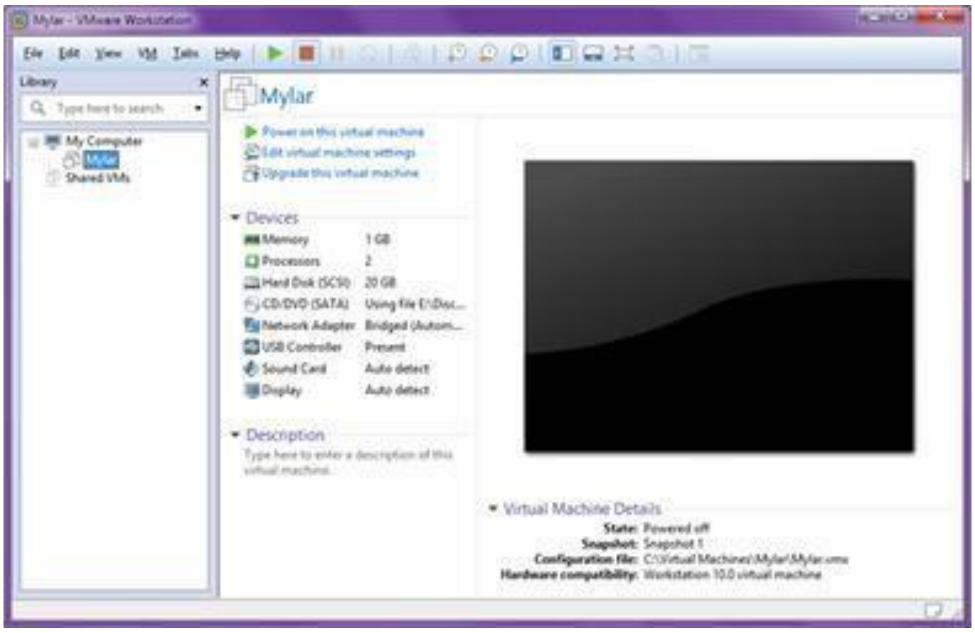
- **Many applications on each server:** As each virtual machine encapsulates an entire machine, many applications and operating systems can run on a single host at the same time.
- **Maximum server utilization, minimum server count**: Every physical machine is used to its full capacity, allowing you to significantly reduce costs by deploying fewer servers overall.
- **Faster, easier application and resource provisioning:** As self-contained software files, virtual machines can be manipulated with copy-and-paste ease. Virtual machines can even be transferred from one physical server to another while running, via a process known as live migration.

5.10.1 Setting up a VMware Workstation

5.10.2 VMware Workstation is developed and sold by VMware, Inc., a division of EMC Corporation. VMware Workstation is a hypervisor that runs on x86 or x86-64 computers; it enables users to set up one or more virtual machines (VMs) on a single physical machine, and use them simultaneously along with the actual machine.

Each virtual machine can execute its own operating system, including versions of Microsoft Windows, Linux, BSD, and MS-DOS. VMware Workstation supports bridging existing host network adapters and share physical disk drives and USB devices with a virtual machine. In addition, it can simulate disk drives. It can mount an existing ISO image file into a virtual optical disc drive so that the virtual machine sees it as a real one. Likewise, virtual hard disk drives are made via ***.vmdk*** files.

VMware Workstation can save the state of a virtual machine (a "snapshot") at any instant. These snapshots can later be restored, effectively returning the virtual machine to the saved state.


Mylar - VMware Workstation
File Edit View VM Tabs Help
Library
Type here to search
My Computer
Mylar
Shared VMs
Mylar
Power on this virtual machine
Edit virtual machine settings
Upgrade this virtual machine
Devices
Description
Type here to enter a description of this virtual machine.
Virtual Machine Details

Memory	1 GB
Processors	2
Hard Disk (SCSI)	20 GB
CD/DVD (SATA)	Using file E:\Doc...
Network Adapter	Bridged (Autom...
USB Controller	Present
Sound Card	Auto detect
Display	Auto detect

State: Powered off
Snapshot: Snapshot 1
Configuration file: C:\Virtual Machines\Mylar\Mylar.vmx
Hardware compatibility: Workstation 10.0 virtual machine

VMware Workstation

VMware Workstation includes the ability to designate multiple virtual machines as a team which can then be powered on, powered off, suspended or resumed as a single object, making it particularly useful for testing client-server environments.

VMWare Player

The VMware Player, a virtualization package of basically similar, but reduced, functionality, is also available, and is free of charge for non-commercial use, or for distribution or other use by written agreement.

VMware Player is a virtualization software package supplied free of charge by VMware, Inc. VMware Player can run existing virtual appliances and create its own virtual machines. It uses the same virtualization core as VMware Workstation, a similar program with more features, but not free of charge. VMware Player is available for personal non-commercial use, or for distribution or other use by written agreement.

VMware claims the Player offers better graphics, faster performance, and tighter integration for running Windows XP under Windows Vista or Windows 7 than Microsoft's Windows XP Mode running on Windows Virtual PC, which is free of charge for all purposes.

VMware Tools

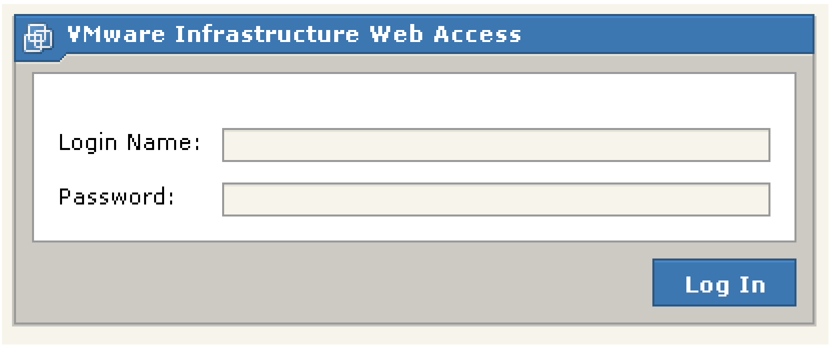
VMware Tools is a package with drivers and other software that can be installed in guest operating systems to increase their performance. It has several components, including the following drivers for the emulated hardware:

- VESA-compliant graphics for the guest machine to access high screen resolutions

- Mouse integration, Drag-and-drop file support
- Clipboard sharing between host and guest
- Time synchronization capabilities (guest syncs with host machine's clock)
- Support for Unity, a feature that allows seamless integration of applications with the host desktop

Installing and Configuring VMWare

1. Download VMware Server 2. VMware management console on a remote Ubuntu desktop behind a firewall at a remote location. Run the following command:
```
     $gksu vmware-server-console
```
2. Install the VMware Server 2.0.2 rpm as shown below.
```
     # rpm -ivh VMware-server-2.0.2-203138.i386.rpm
Preparing...
     1:VMware-server
########################################### [100%]
```
The installation of VMware Server 2.0.2 for Linux completed successfully. You can decide to remove this software from your system at any time by invoking the following command:
```
     rpm -e VMware-server
```
Before running VMware Server for the first time, you need to configure it for your running kernel by invoking the following command:
```
     /usr/bin/vmware-config.pl
```
3. Configure VMware Server 2 using *vmware-config.pl*. Execute the vmware-config.pl as shown below. Accept default values for everything. Partial output of the vmware- config.pl is shown below.
```
     # /usr/bin/vmware-config.pl
```
4. Go to VMware Infrastructure Webaccess. Go to **https://{host-os-ip}:8333/ui** to access the VMware Infrastructure web access console.



VMware Web Access Login

Installing a VMware Guest OS

1. **Start VMware Workstation**

   Windows host: Double-click the VMware Workstation icon on your desktop or use the Start menu (Start > Programs > VMware > VMware Workstation).

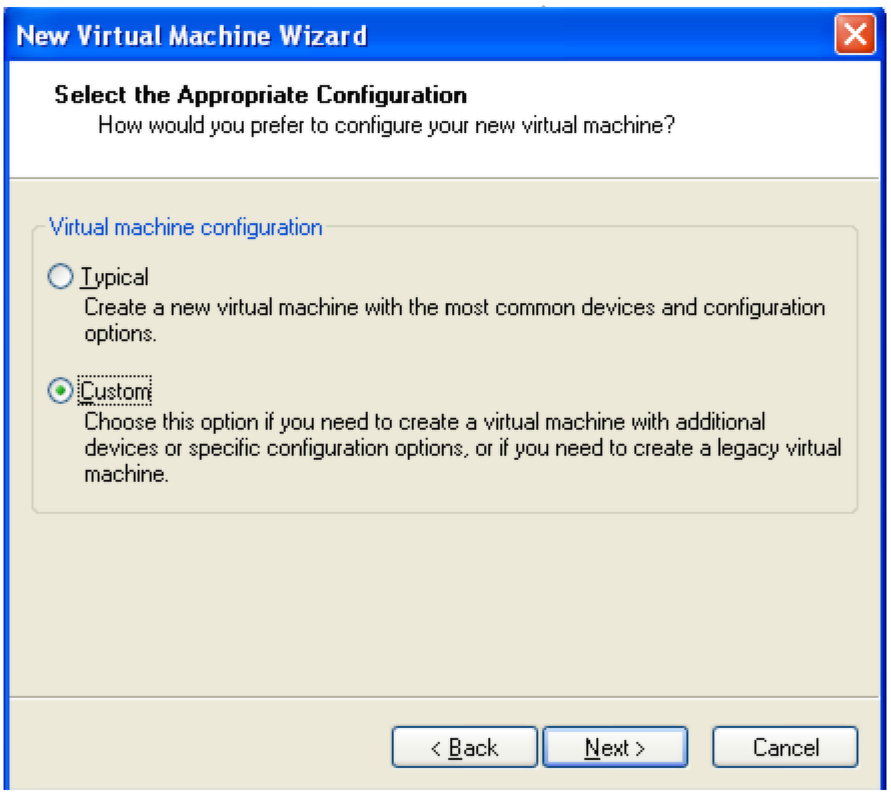
   Linux host: In a terminal window, enter the command

   ```
   vmware &
   ```

2. **Start the New Virtual Machine Wizard**

   When you start VMware Workstation, you can open an existing virtual machine or create a new one. Choose File > New > Virtual Machine to begin creating your virtual machine.

3. Select the method you want to use for configuring your virtual machine.



New Virtual Machine Wizard

Select the Appropriate Configuration

How would you prefer to configure your new virtual machine?

Virtual machine configuration

○ Typical
Create a new virtual machine with the most common devices and configuration options.

◉ Custom
Choose this option if you need to create a virtual machine with additional devices or specific configuration options, or if you need to create a legacy virtual machine.

If you select ***Typical***, the wizard prompts you to specify or accept defaults for the following choices:

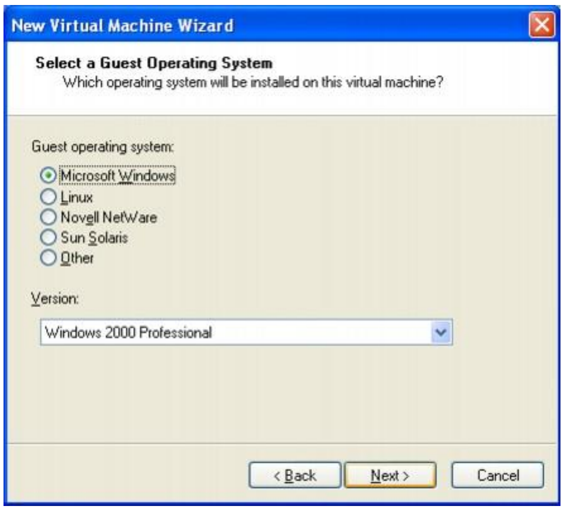
- The guest operating system
- The virtual machine name and the location of the virtual machine's files
- The network connection type
- Whether to allocate all the space for a virtual disk at the time you create it
- Whether to split a virtual disk into 2GB files

If you select ***Custom***, the wizard prompts you to specify or accept defaults for the following choices:

- Make a legacy virtual machine that is compatible with Workstation 4.x, GSX Server 3.x, ESX Server 2.x and VMware ACE 1.x.
- Use an IDE virtual disk for a guest operating system that would otherwise have a SCSI virtual disk created by default
- Use a physical disk rather than a virtual disk and Set memory options that are different from the defaults

4. Select a guest operating system and type a name and folder for the virtual machine.

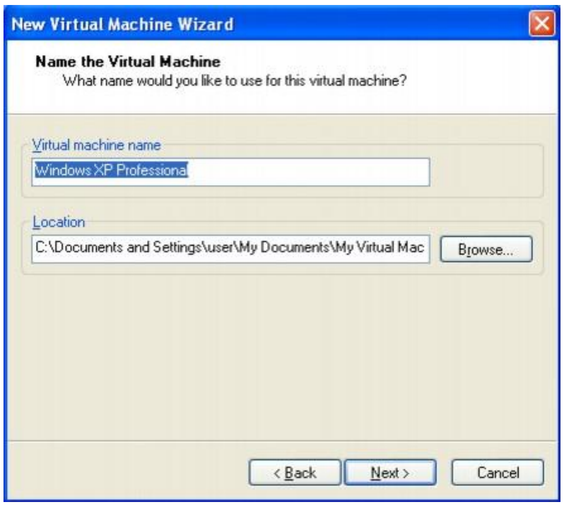
   Linux hosts: The default location for this Windows XP Professional virtual machine is `<homedir>/vmware/winXPPro`, where `<homedir>` is the home directory of the user who is currently logged on.


New Virtual Machine Wizard

Select a Guest Operating System
Which operating system will be installed on this virtual machine?

Guest operating system:
[✓ Microsoft Windows] Linux Novell NetWare Sun Solaris Other

Version: Windows 2000 Professional

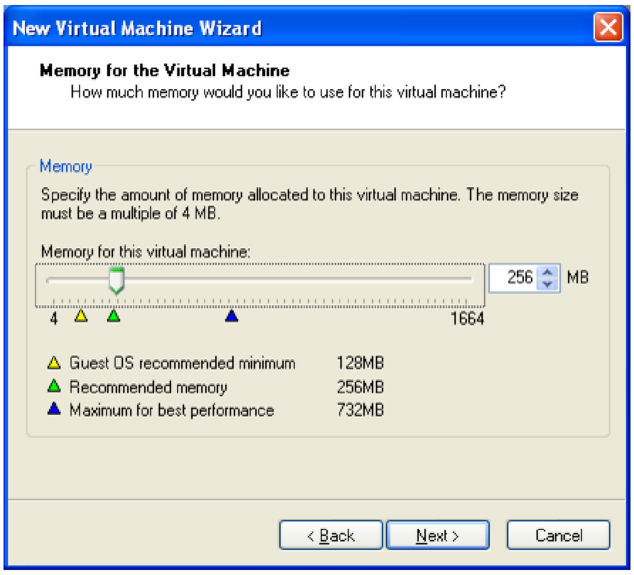

New Virtual Machine Wizard

Name the Virtual Machine
What name would you like to use for this virtual machine?

Virtual machine name: Windows XP Professional

Location: C:\Documents and Settings\user\My Documents\My Virtual Mac

5. Specify the number of processors for the virtual machine. The setting Two is supported only for host machines with at least two logical processors.


New Virtual Machine Wizard

Memory for the Virtual Machine
How much memory would you like to use for this virtual machine?

Memory

Specify the amount of memory allocated to this virtual machine. The memory size must be a multiple of 4 MB.

Memory for this virtual machine: 256 MB

4 1664

△ Guest OS recommended minimum	128MB
▲ Recommended memory	256MB
▲ Maximum for best performance	732MB

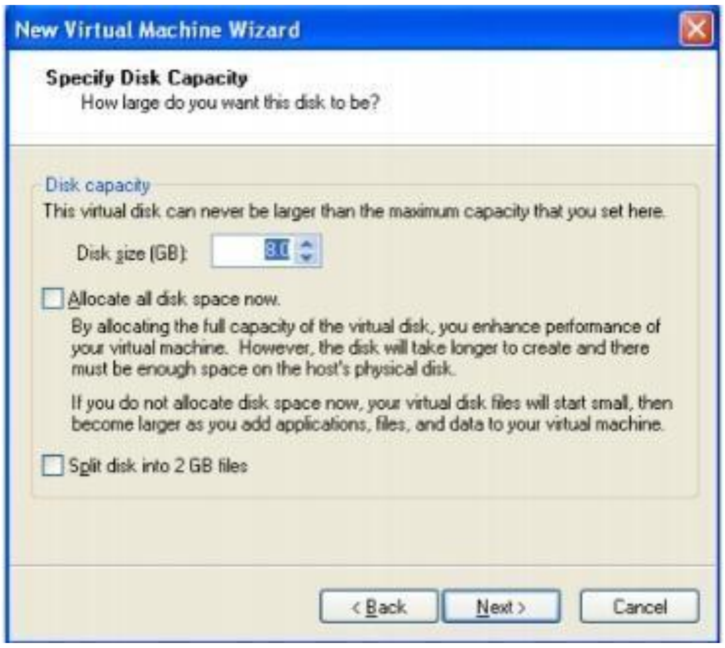
If you selected **Custom** as your configuration path, you may adjust the memory settings or accept the defaults, then click Next to continue.

6. Configure the networking capabilities of the virtual machine.

    If you selected *Typical* as your configuration path, click Finish and the wizard sets up the files needed for your virtual machine.

    If you selected *Custom* as your configuration path, continue with the steps below to configure a disk for your virtual machine.

7. Select whether to create an IDE or SCSI disk and specify the capacity of the virtual disk.



Specify Disk Capacity
How large do you want this disk to be?

Disk capacity
This virtual disk can never be larger than the maximum capacity that you set here.
Disk size (GB): 8.0

☐ Allocate all disk space now.
By allocating the full capacity of the virtual disk, you enhance performance of your virtual machine. However, the disk will take longer to create and there must be enough space on the host's physical disk.
If you do not allocate disk space now, your virtual disk files will start small, then become larger as you add applications, files, and data to your virtual machine.

☐ Split disk into 2 GB files

8. Click Finish. The wizard sets up the files needed for your virtual machine.

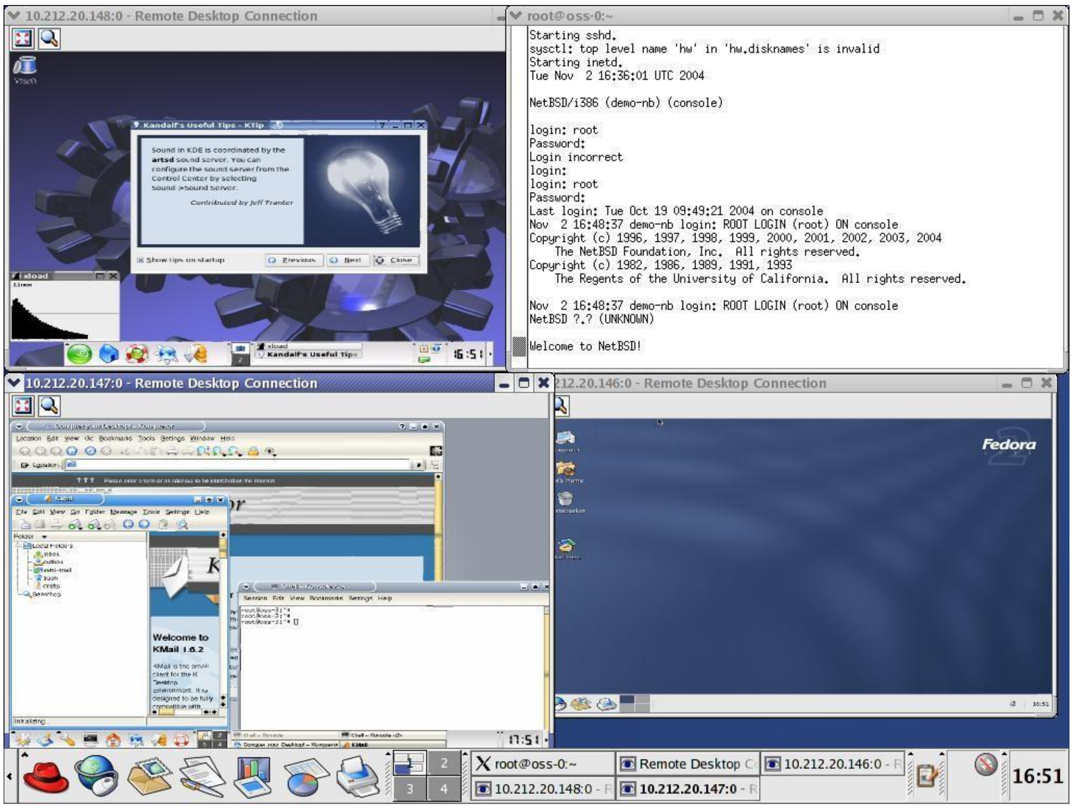
5.10.3 Setting up a XEN Workstation XEN Workstation

Xen is a hypervisor using a microkernel design, providing services that allow multiple computer operating systems to execute on the same computer hardware concurrently.

The University of Cambridge Computer Laboratory developed the first versions of Xen. The Xen community develops and maintains Xen as free and open-source software, subject to the requirements of the GNU General Public License (GPL), version 2. Xen is currently available for the IA-32, x86-64 and ARM instruction sets.

XenServer runs directly on server hardware without requiring an underlying operating system, which results in an efficient and scalable system. XenServer works by abstracting elements from the physical machine (such as hard drives, resources and ports) and allocating

them to the virtual machines running on it.


10.212.20.148:0 - Remote Desktop Connection
root@oss-0:~
10.212.20.147:0 - Remote Desktop Connection
212.20.146:0 - Remote Desktop Connection
Fedora
Welcome to KMail 1.6.2
16:51

```
Starting sshd.
sysctl: top level name 'hw' in 'hw.disknames' is invalid
Starting inetd.
Tue Nov  2 16:36:01 UTC 2004

NetBSD/i386 (demo-nb) (console)

login: root
Password:
Login incorrect
login:
login: root
Password:
Last login: Tue Oct 19 09:49:21 2004 on console
Nov  2 16:48:37 demo-nb login: ROOT LOGIN (root) ON console
Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004
    The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
    The Regents of the University of California.  All rights reserved.

Nov  2 16:48:37 demo-nb login: ROOT LOGIN (root) ON console
NetBSD ?.? (UNKNOWN)

Welcome to NetBSD!
```

XEN Environment

Responsibilities of the hypervisor include memory management and CPU scheduling of all virtual machines, and for launching the most privileged domain - the only virtual machine which by default has direct access to hardware. From the dom0 the hypervisor can be managed and unprivileged domains can be launched.

Benefits of Using XenServer

1. **Using XenServer reduces costs by:**
   - Consolidating multiple VMs onto physical servers
   - Reducing the number of separate disk images that need to be managed
   - Allowing for easy integration with existing networking and storage infrastructures
2. **Using XenServer increases flexibility by:**
   - Allowing you to schedule zero downtime maintenance by using XenMotion to live migrate VMs between XenServer hosts
   - Increasing availability of VMs by using High Availability to configure policies that restart VMs on another XenServer host if one fails
   - Increasing portability of VM images, as one VM image will work on a range of deployment infrastructures

Administering XenServer

- There are two methods by which to administer XenServer: XenCenter and the XenServer Command-Line Interface (CLI).
- **XenCenter** is a graphical, Windows-based user interface. XenCenter allows you to manage XenServer hosts, pools and shared storage, and to deploy, manage and monitor VMs from your Windows desktop machine.
- The XenCenter on-line Help is a useful resource for getting started with XenCenter and for context-sensitive assistance.

Installing and Configuring XenServer

1. Type the following command to get information about xen server package

```
# yum info xen
```

2. Run the system-config-securitylevel program or edit /etc/selinux/config to looks as follows:

```
SELINUX=Disabled
SELINUXTYPE=targeted
```

   If you changed the SELINUX value from enforcing, you'll need to reboot Fedora before proceeding.

3. This command will install the Xen hypervisor, a Xen-modified Fedora kernel called *domain 0*, and various utilities:

```
# yum install kernel-xen0
```

4. To make the Xen kernel the default, change this line:

```
default=1
```

   to

```
default=0
```

5. Now you can reboot. Xen should start automatically, but let's check:

```
# /usr/sbin/xm list
Name        ID   Mem(MiB)   VCPUs      State      Time(s)
Domain-0    0    880        1          r-----     20.5
```

```
The output should show that Domain-0 is running. Domain 0 controls
all the guest operating systems that run on the processor,
similarly to how the kernel controls processes in an operating
system.
```

Installing a Xen Guest OS from the Command-line

1. **Preparing the System for virt-install**

   Fedora Linux does not install VNC by default. To verify whether VNC is installed, run the following command from a Terminal Window:

If rpm reports that VNC is not installed, it may be installed from root as follows:

```
yum   install   vnc
```

2. **Running virt-install to Build the Xen Guest System**

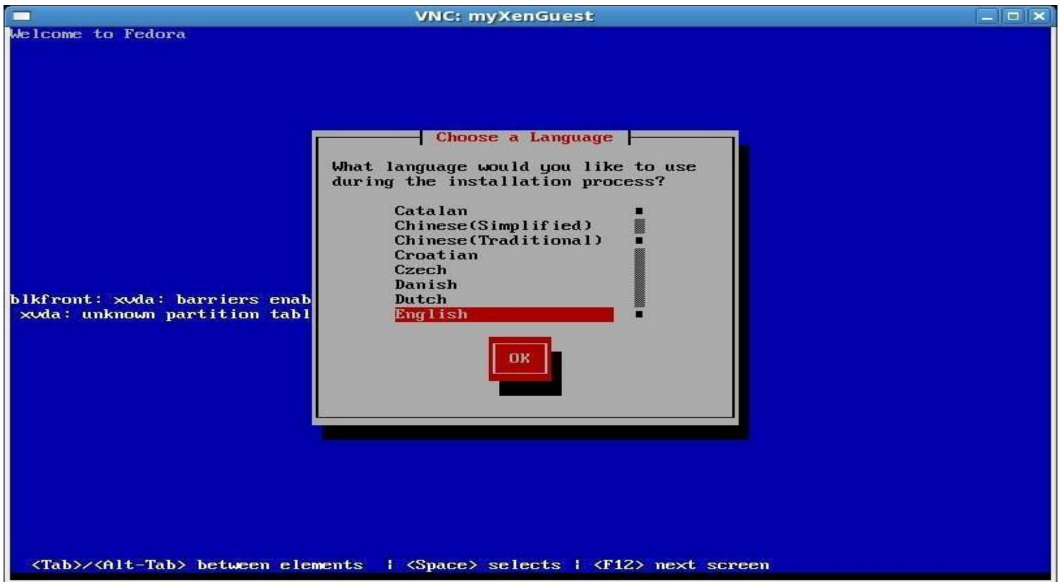
   virt-install must be run as root and, once invoked, will ask a number of questions before creating the guest system. The question are as follows:

   i. *What is the name of your virtual machine and install location?*
   ii. *How much RAM should be allocated (in megabytes)?*
   iii. *What would you like to use as the disk (path)?*
   iv. *Would you like to enable graphics support? (yes or no)*

   The following transcript shows a typical virt-install session:

```
# virt-install
```

3. Once the guest system has been created, the vncviewer screen will appear containing the operating system installer:



```
VNC: myXenGuest
Welcome to Fedora

blkfront: xvda: barriers enab
 xvda: unknown partition tabl

Choose a Language

What language would you like to use
during the installation process?

Catalan
Chinese(Simplified)
Chinese(Traditional)
Croatian
Czech
Danish
Dutch
English

OK

<Tab>/<Alt-Tab> between elements  | <Space> selects | <F12> next screen
```

Installing a Xen Guest OS (Fedora Core 5)

1. Fedora Core 5 has a Xen guest installation script that simplifies the process, although it installs only FC5 guests. The script expects to access the FC5 install tree via FTP, the Web, or NFS; for some reason, you can't specify a directory or file.

```
# mkdir /var/www/html/dvd
# mount -t iso9660 /dev/dvd /var/www/html/dvd
# apachectl start
```

   Now we'll run the installation script and answer its questions:

```
# xenguest-install.py
```

2. Xen does not start the guest operating system automatically. You need to type this command on the host:

3. To prove that both servers are running, try these commands:

```
# xm list
# xentop
```

4. To start Xen domains automatically, use these commands:

```
# /sbin/chkconfig --level 345 xendomains on
# /sbin/service xendomains start
```

5. To Edit A Xen Guest Configuration File, Which Is A Text File (Actually, A Python Script) In The

   /Etc/Xen Directory.

```
        # man xmdomain.cfg
```

And edit as follows,

```
# Automatically generated Xen config file
name = "guest1"
memory = "256"
disk = [ 'file:/xenguest,xvda,w' ]
vif = [ 'mac=00:16:3e:63:c7:76' ]
uuid = "bc2c1684-c057-99ea-962b-de44a038bbda"
bootloader="/usr/bin/pygrub"
on_reboot = 'restart'
on_crash = 'restart'
```

6. Once you have a guest configuration file, create the Xen guest with this command:

```
        # xm create -c guest_name
```

where

guest_name can be a full pathname or a relative filename (in which case Xen places

it in /etc/xen/guest_name).

Xen will create the guest domain and try to boot it from the given file or device. The **-c** option attaches a console to the domain when it starts, so you can answer the installation questions that appear.