Maharaja Education Trust (R), Mysuru

# Maharaja Institute of Technology Mysore

Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

**Approved by AICTE, New Delhi,**
**Affiliated to VTU, Belagavi & Recognized by Government of Karnataka**

# Lecture Notes on

# OPERATING SYSETEM (17CS64)

## Prepared by

## Department of Information Science and Engineering

## Vision/ ಆ ಶ ಯ

"To be recognized as a premier technical and management institution promoting extensive education fostering research, innovation and entrepreneurial attitude"

ಸಂಶೆ ೋಧನ, ಆವಿಷ್ಕಾರ ಹಕಗೂ ಉದ್ಯಮಶೋಲತೆಯನ್ನು ಉತೆತೋಜಿಸನವ ಅ ಗಮುಕನ್ನ ತಕಂತ್ರಕ ಮತ್ನತ ಆಡಳಿತ್ ವಿಜ್ಞಕನ್ ಶಕ್ಷಣ ಕ್ಕಂದ್ರವಕಗಿ
ಗನರನತ್ರಸಿಕೊಳ್ಳಲ್ಪುವಂದ್ನ.

## Mission/ ಧ್ಯೇಯ

- To empower students with indispensable knowledge through dedicated teaching and collaborative learning.

  ಸಮರ್ಪಣ ಕ ಮನೆಂೂಭಹುದ್ ಬೆಂೂಧನೆ ಹಕಗೂ ಸಹಭಕಗಿತ್ವದ ಕಲಿಕಕರಮಗಳಿಂದ್ ವಿದ್ಕರ್ಥಿಗಳನ್ನು ಅತ್ಯುತ್ಕೃಷ್ಟ ಜ್ಞಕನಸಂಪನ್ನರಕಗಿಸನವಂದ್ನ.

- To advance extensive research in science, engineering and management disciplines.

  ವಿಜ್ಞಕನಿಕ, ತಕಂತ್ರರ ಕ ಹಕಗೂ ಆಡಳಿತ್ ವಿಜ್ಞಕನ್ ವಿಭಕಗಗಳಲ್ಲಿ ವಿಸೃತ್ ಸಂಶೆ ೋಧನೆಗಳ ಬೊಡನೆ ಬೆಳ್ವಣಿಗೆ ಹೆಂೂಂದ್ನವಂದ್ನ.

- To facilitate entrepreneurial skills through effective institute - industry collaboration and interaction with alumni.

  ಉದ್ಯಮ ಕ್ಕೃತ್ಯಗಳ ಬೊಡನೆ ಸಹಯೆ ೋಗ, ಸಂಸ್ಥೆಯ ಹಿರಿಯ ವಿದ್ಕರ್ಥಿಗಳ ಬೊಂದಿಗೆ ನಿರಂತರ ಸಂವಹನಗಳಿಂದ್ ವಿದ್ಯರ್ಥಿಗಳಿಗೆ ಉದ್ಯಮಶೆ ೋಲತೆಯ ಕೌಶಲ್ ರಡಿಯಲನ ನಿರವಕಗನವಂದ್ನ.

- To instill the need to uphold ethics in every aspect.

  ಜಿೋವನದ್ದಲಿ ನೆೈತಿಕ ಮೌಲ್ಯಗಳ್ನ್ನು ಅಳ್ವಡಿಸಿಕೊಳ್ಳುವಂದ್ರ ಮಹತ್ವದ್ ಕನರಿತ್ನ ಅರಿವು ಮೂಡಿಸನವಂದ್ನ.

- To mould holistic individuals capable of contributing to the advancement of the society.

  ಸಮ ಹುದ್ ಬೆಳ್ವಣಿಗೆಗೆ ಗಣನಿೋಯ ಕೊಡನಗೆ ನಿೋಡಬಲ ರಿರಕೂಣಪ ವ್ಯಕ್ತತ್ವಳ್ಳು ಸಮರ್ಪ ನಕಗರಿೋಕರನ್ನು ರೂಪಿಸನವಂದ್ನ.

## VISION OF THE DEPARTMENT

To be recognized as the best centre for technical education and research in the field of information science and engineering.

## MISSION OF THE DEPARTMENT

- To facilitate adequate transformation in students through a proficient teaching learning process with the guidance of mentors and all-inclusive professional activities.
- To infuse students with professional, ethical and leadership attributes through industry collaboration and alumni affiliation.
- To enhance research and entrepreneurship in associated domains and to facilitate real time problem solving.
-

## PROGRAM EDUCATIONAL OBJECTIVES:

- Proficiency in being an IT professional, capable of providing genuine solutions to information science problems.
- Capable of using basic concepts and skills of science and IT disciplines to pursue greater competencies through higher education.
- Exhibit relevant professional skills and learned involvement to match the requirements of technological trends.

## PROGRAM SPECIFIC OUTCOME:

Student will be able to

- **PSO1:** Apply the principles of theoretical foundations, data Organizations, networking concepts and data analytical methods in the evolving technologies.
- **PSO2:**Analyse proficient algorithms to develop software and hardware competence in both professional and industrial areas

# Program Outcomes

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

# Maharaja Institute of Technology Mysore

## Department of Information Science and Engineering

# Course Overview

**SUBJECT: OPERATING SYSTEM** **SUBJECT CODE: 17CS64**

Computer operating systems (OS) provide a set of functions needed and used by most application programs on a computer, and the links needed to control and synchronize computer hardware. On the first computers, with no operating system, every program needed the full hardware specification to run correctly and perform standard tasks, and its own drivers for peripheral devices like printers and punched paper card readers. The growing complexity of hardware and application programs eventually made operating systems a necessity for everyday use.

The course introduces the fundamental concepts of operating systems. It covers an introduction which includes the evolution of operating systems (OS), operating system structure, process and threads management, asynchronous concurrent execution, concurrent programming deadlock and indefinite postponement, processor scheduling, virtual memory organization and management; and case studies.

Students are able to understand the concepts and terminology of os they are capable of explaining threading concepts and they are able to analyze the resource management techniques.

# Course Objectives

- Introduce concepts and terminology used in OS
- Explain threading and multithreaded systems
- Illustrate process synchronization and concept of Deadlock
- Introduce Memory and Virtual memory management, File system and storage techniques

# Course Outcomes

| CO's | DESCRIPTION OF THE OUTCOMES |
|---|---|
| 17CS64.1 | Apply the fundamentals of operating systems using suitable algorithms to solve the given problems. |
| 17CS64.2 | Apply suitable techniques for managing the different resources. |
| 17CS64.3 | Analyze CPU performance by the process scheduling and process synchronization |
| 17CS64.4 | Infer the different concepts of OS in platform of usage through case studies. |

# Maharaja Institute of Technology Mysore

## Department of Information Science and Engineering

## Syllabus

**SUBJECT: OPERATING SYSTEM** **SUBJECT CODE: 17CS64**

| Topics Covered as per Syllabus | Teaching Hours |
|---|---|
| **MODULE-1** | |
| **Introduction to operating systems, System structures:** What operating systemsdo; Computer System organization; Computer System architecture; Operating System structure; Operating System operations; Process management; Memory management; Storage management; Protection and Security; Distributed system; Special-purpose systems; Computing environments. Operating System Services; User - Operating System interface; System calls; Types of system calls; System programs; Operating system design and implementation; Operating System structure; Virtual machines; Operating System generation; System boot. **Process Management** Process concept; Process scheduling; Operations on processes; Inter process communication. | **10 Hours** |
| **MODULE-2** | |
| **Multi-threaded Programming**: Overview; Multithreading models; Thread Libraries; Threading issues. Process Scheduling: Basic concepts; Scheduling Criteria; Scheduling Algorithms; Multiple-processor scheduling; Thread scheduling. **Process Synchronization:** Synchronization: The critical section problem; Peterson's solution; Synchronization hardware; Semaphores; Classical problems of synchronization; Monitors. | **10 Hours** |
| **MODULE -3** | |
| **Deadlocks :** Deadlocks; System model; Deadlock characterization; Methods for handling deadlocks; Deadlock prevention; Deadlock avoidance; Deadlock detection and recovery from deadlock. **Memory Management:** Memory management strategies: Background; Swapping; Contiguous memory allocation; Paging; Structure of page table; Segmentation. | **10 Hours** |
| **MODULE-4** | |
| **Virtual Memory Management**: Background; Demand paging; Copy-on-write; Page replacement; Allocation of frames; Thrashing. **File System, Implementation of File System:** File system: File concept; Access methods; Directory structure; File system mounting; File sharing; Protection: Implementing File system: File system structure; File system implementation; Directory implementation; Allocation methods; Free space management. | **10 Hours** |
| **MODULE-5** | |
| **Secondary Storage Structures, Protection:** Mass storage structures; Disk structure; Disk attachment; Disk scheduling; Disk management; Swap space management. Protection: Goals of protection, Principles of protection, Domain of protection, Access matrix, Implementation of access matrix, Access control, Revocation of access rights, Capability- Based systems. **Case Study: The Linux Operating System:** Linux history; Design principles; Kernel modules; Process management; Scheduling; Memory Management; File systems, Input and output;Inter-process communication. | **10 Hours** |
| **List of Text Books** | |
| 1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating System Principles 7th edition, Wiley-India, 2006. | |
| **List of Reference Books** | |
| 1 Ann McHoes Ida M Fylnn, Understanding Operating System, Cengage Learning, 6th Edition<br>2. D.M Dhamdhere, Operating Systems: A Concept Based Approach 3rd Ed, McGraw- Hill, 2013.<br>3. P.C.P. Bhatt, An Introduction to Operating Systems: Concepts and Practice 4th Edition, PHI(EEE), 2014.<br>4. William Stallings Operating Systems: Internals and Design Principles, 6th Edition, Pearson.. | |

# Maharaja Institute of Technology Mysore

## Department of Information Science and Engineering

## Index

**SUBJECT: OPERATING SYSTEM** **SUBJECT CODE: 17CS64**

## Module I

# INTRODUCTION TO OPERATING SYSTEM

**What is an Operating System**?

n operating system is a system software that acts as an intermediary between a user of a computer and the computer hardware.
It is a software that manages the computer hardware.
Os allows the user to execute programs in a convenient and efficient manner.

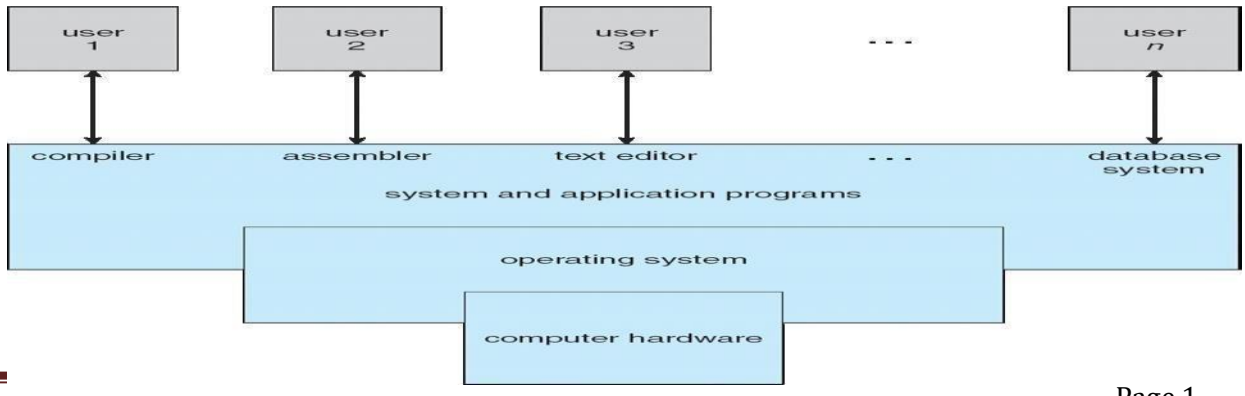
Operating system goals:

- Make the computer system convenient to use. It hides the difficulty in managing the hardware.
- Use the computer hardware in an efficient manner
- Provide and environment in which user can easily interface with computer.
- It is a **resource allocator**

## Computer System Structure (Components of Computer System)

Computer system mainly consists of four components-

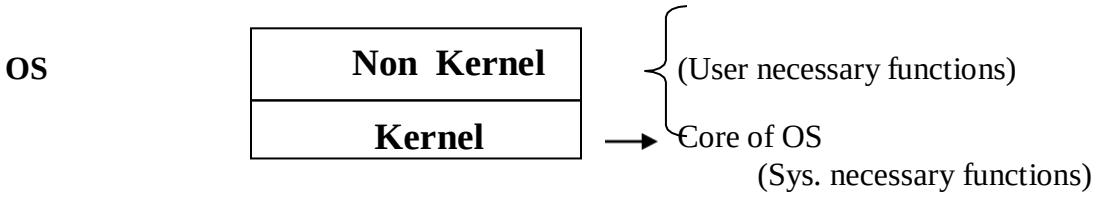
- Hardware – provides basic computing resources
  - ✓ CPU, memory, I/O devices
- Operating system
  - ✓ Controls and coordinates use of hardware among various applications and users
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
  - ✓ Word processors, compilers, web browsers, database systems, video games
- Users
  - ✓ People, machines, other computers



The basic hardware components comprises of CPU, memory, I/O devices. The application program uses these components. The OS controls and co-ordinates the use of hardware, among various application programs (like compiler, word processor etc.) for various users.

The OS allocates the resources among the programs such that the hardware is efficiently used.

The operating system is the program running at all the times on the computer. It is usually called as the kernel.




Kernel functions are used always in system, so always stored in memory. Non kernel functions are stored in hard disk, and it is retrieved whenever required.

**Views of OS**

Operating System can be viewed from two viewpoints–
User views & System views

1. User Views:-
   The user's view of the operating system depends on the type of user.
   i. If the user is using **standalone** system, then OS is designed for ease of use and high performances. Here resource utilization is not given importance.

   ii. If the users are at different **terminals** connected to a mainframe or minicomputers, by sharing information and resources, then the OS is designed to maximize resource utilization. OS is designed such that the CPU time, memory and i/o are used efficiently and no single user takes more than the resource allotted to them.

   iii. If the users are in **workstations**, connected to networks and servers, then the user have a system unit of their own and shares resources and files with other systems. Here the OS is designed for both ease of use and resource availability (files).

   iv. Users of **hand held** systems, expects the OS to be designed for ease of use and performance per amount of battery life.

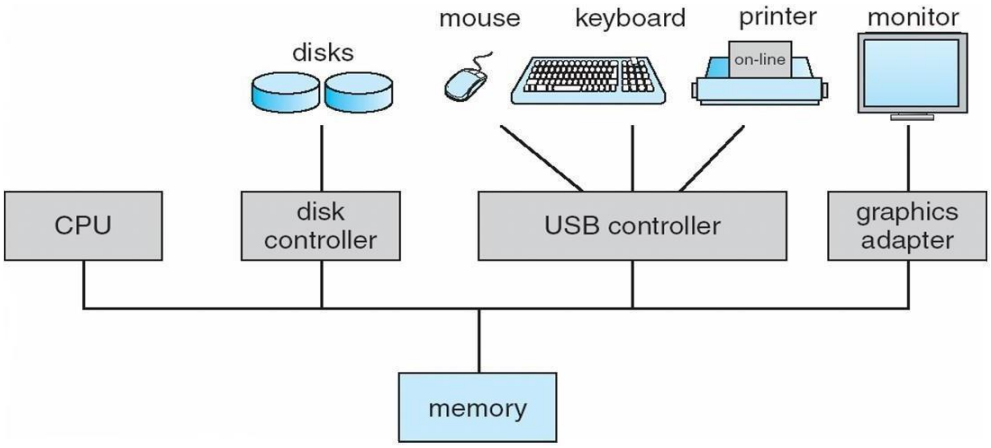
v. Other systems like embedded systems used in home devies (like washing m/c) & automobiles do not have any user interaction. There are some LEDs to show the status of its work.

2. System Views:-
   Operating system can be viewed as a **resource allocator** and **control program**.
   i. **Resource allocator** - The OS acts as a manager of hardware and software resources. CPU time, memory space, file-storage space, I/O devices, shared files etc. are the different resources required during execution of a program. There can be conflicting request for these resources by different programs running in same system. The OS assigns the resources to the requesting program depending on the priority.
   ii. **Control Program** – The OS is a control program and manage the execution of user program to prevent errors and improper use of the computer.

## Computer System Organization

**Computer-system operation**

One or more CPUs, device controllers connect through common bus providing access to shared memory. Each device controller is in-charge of a specific type of device. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory. The CPU and other devices execute concurrently competing for memory cycles. Concurrent execution of CPUs and devices competing for memory cycles




When system is switched on, '**Bootstrap'** program is executed. It is the initial program to run in the system. This program is stored in read-only memory (ROM) or in electrically erasable programmable read-only memory(EEPROM). It initializes the CPU registers, memory, device controllers and other initial setups. The program also locates and loads, the OS kernel to the

memory. Then the OS starts with the first process to be executed (ie. 'init' process) and then wait for the interrupt from the user.

Switch on ⟶ 'Bootstrap' program

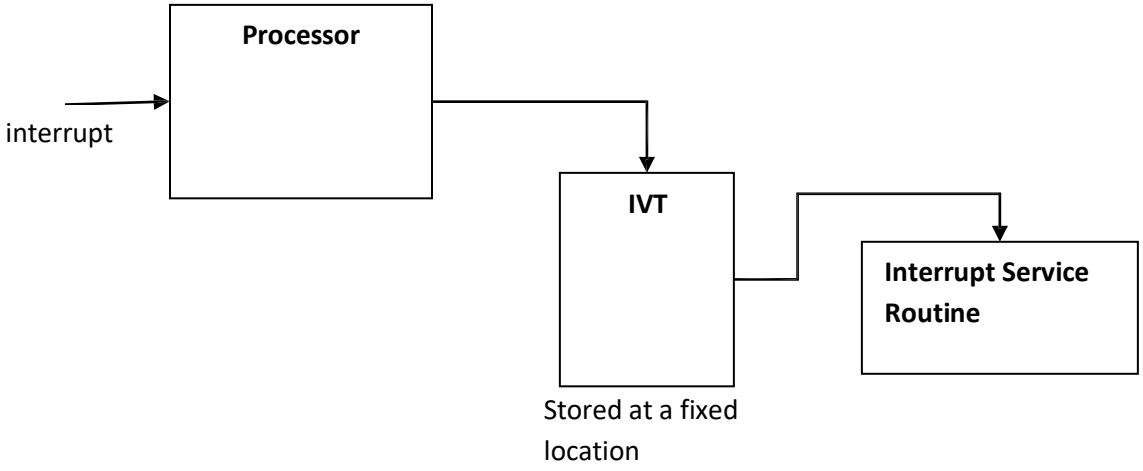
- Initializes the registers, memory and I/O devices
- Locates & loads kernel into memory
- Starts with 'init' process
- Waits for interrupt from user.

**Interrupt handling** –

The occurrence of an event is usually signaled by an **interrupt.** The interrupt can either be from the hardware or the software. Hardware may trigger an interrupt at any time by sending a signal to the CPU. Software triggers 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 (Interrupt Vector Table) contains the starting address where the service routine for the interrupt is located. After the execution of interrupt service routine, the CPU resumes the interrupted computation.

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




**Storage Structure**

Computer programs must be in main memory (**RAM)** to be executed. Main memory is the large memory that the processor can access directly. It commonly is implemented in a semiconductor technology called **dynamic random-access memory (DRAM).** Computers provide Read Only Memory(ROM), whose data cannot be changed.

All forms of memory provide an array of memory words. Each word has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses.

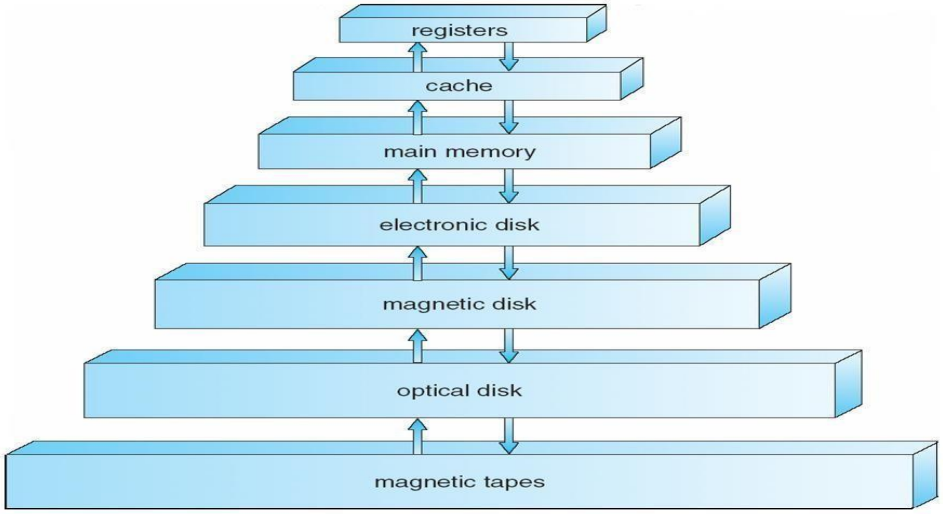
A typical instruction-execution cycle, as executed on a system with a **Von Neumann** architecture, first fetches an instruction from memory and stores that instruction in the **instruction register.** The instruction is then decoded and may cause operands to be fetched from memory and stored in some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

Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible for the following two reasons:

1. Main memory is usually too small to store all needed programs and data permanently.
2. Main memory is a *volatile* storage device that loses its contents when power is turned off.

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

The most common secondary-storage device is a **magnetic disk,** which provides storage for both programs and data. Most programs are stored on a disk until they are loaded into memory. Many programs then use the disk as both a source and a destination of the information for their processing.




The wide variety of storage systems in a computer system can be organized in a hierarchy as shown in the figure, according to speed, cost and capacity. 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 and the capacity of storage generally increases.

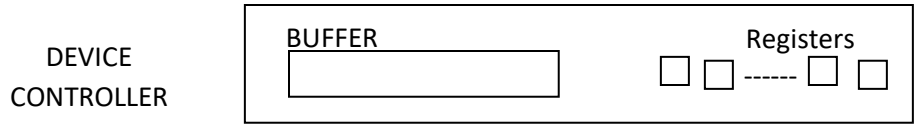
In addition to differing in speed and cost, the various storage systems are either volatile or nonvolatile. V**olatile storage** loses its contents when the power to the device is removed. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safekeeping. In the hierarchy shown in figure, the storage systems above the electronic disk are volatile, whereas those below are nonvolatile.

An **electronic disk** can be designed to be either volatile or nonvolatile. During normal operation, the electronic disk stores data in a large DRAM array, which is volatile. But many electronic-disk devices contain a hidden magnetic hard disk and a battery for backup power. If external power is interrupted, the electronic-disk controller copies the data from RAM to the magnetic disk. Another form of electronic disk is flash memory.

**I/O Structure**

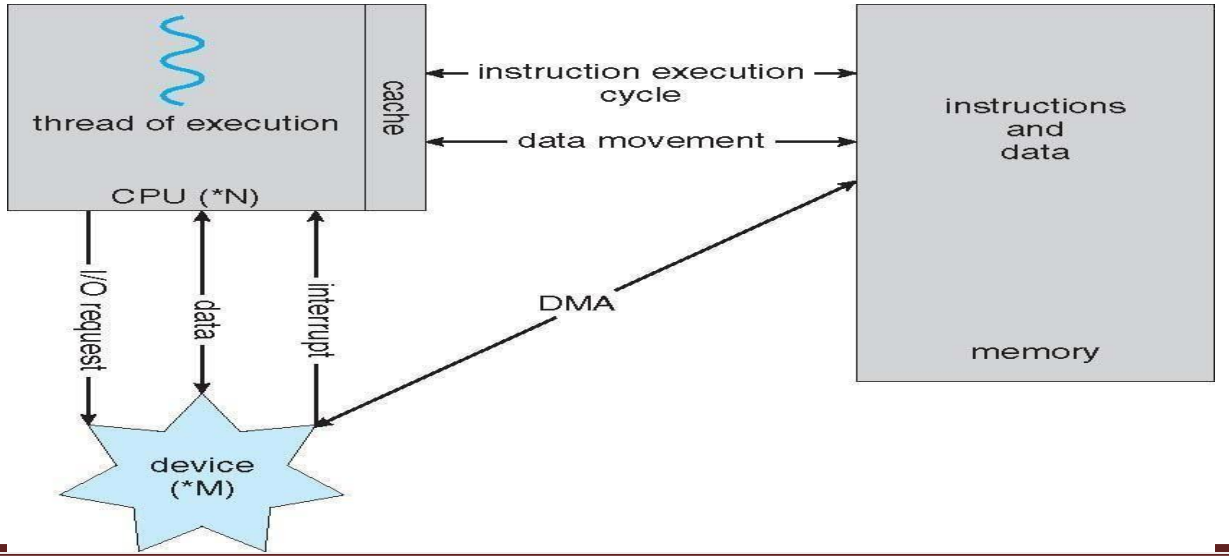
A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

Every device have a device controller, maintains some local buffer and a set of special-purpose registers. The device controller is responsible for moving the data between the peripheral devices. The operating systems have a **device driver** for each device controller.




To start an I/O operation, the device driver loads the registers within the device controller. The device controller, examines the contents of these registers to determine what action to take (such as "read a character from the keyboard"). The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver(OS) via an interrupt that it has finished its operation. The device driver then returns control to the operating system, and also returns the data. For other operations, the device driver returns status information.

This form of interrupt-driven I/O is fine for moving small amounts of data, but very difficult for bulk data movement. To solve this problem, **direct memory access (DMA)** is used.



- DMA is used for high-speed I/O devices, able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte

## Computer System Architecture

Categorized roughly according to the number of general-purpose processors used –

**Single-Processor Systems –**

Most systems use a single processor. The variety of single-processor systems range from PDAs through mainframes. On a single-processor system, there is one main CPU capable of executing instructions from user processes. It contains special-purpose processors, in the form of device-specific processors, for devices such as disk, keyboard, and graphics controllers.

All special-purpose processors run limited instructions and do not run user processes. These are managed by the operating system, the operating system sends them information about their next task and monitors their status.

For example, a disk-controller processor, implements its own disk queue and scheduling algorithm, thus reducing the task of main CPU. Special processors in the keyboard, converts the keystrokes into codes to be sent to the CPU.

The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor. If there is only one general-purpose CPU, then the system is a single-processor system.

**Multiprocessor Systems (parallel systems** or **tightly coupled systems) –**

Systems that have two or more processors in close communication, sharing the computer bus, the clock, memory, and peripheral devices are the multiprocessor systems.

Multiprocessor systems have three main advantages:

1. **Increased throughput -** In multiprocessor system, as there are multiple processors execution of different programs take place simultaneously. Even if the number of processors is increased the performance cannot be simultaneously increased. This is due to the overhead incurred in keeping all the parts working correctly and also due to the competation for the shared resources. The speed-up ratio with *N* processors is not *N*, rather, it is less than N. Thus the speed of the system is not has expected.

2. **Economy of scale -** Multiprocessor systems can cost less than equivalent number of many single-processor systems. As the multiprocessor systems share peripherals, mass storage, and power supplies, the cost of implementing this system is economical. If several processes are working on the same data, the data can also be shared among them.

3. **Increased reliability-** In multiprocessor systems functions are shared among several processors. If one processor fails, the system is not halted, it only slows down. The job of the failed processor is taken up, by other processors.

   Two techniques to maintain 'Increased Reliability' - graceful degradation & fault tolerant

   **Graceful degradation –** As there are multiple processors when one processor fails other process will take up its work and the system goes down slowly.

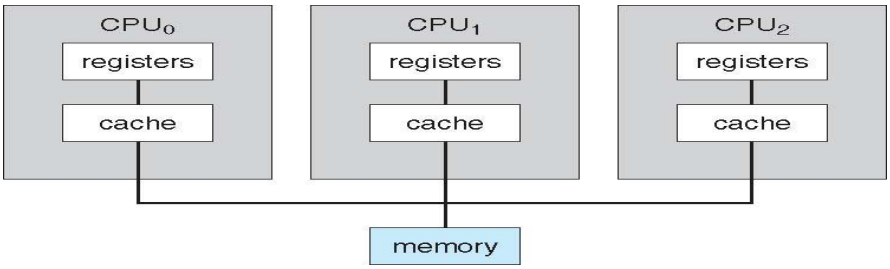
   **Fault tolerant –** When one processor fails, its operations are stopped, the system failure is then detected, diagnosed, and corrected.

   The HP NonStop system uses both hardware and software duplication to ensure continued operation despite faults. The system consists of multiple pairs of CPUs. Both processors in the pair execute same instruction and compare the results. If the results differ, then one CPU of the pair is at fault, and both are halted. The process that was being executed is then moved to another pair of CPUs, and the instruction that failed is restarted. This solution is expensive, since it involves special hardware and considerable hardware duplication.

There are two types of multiprocessor systems –

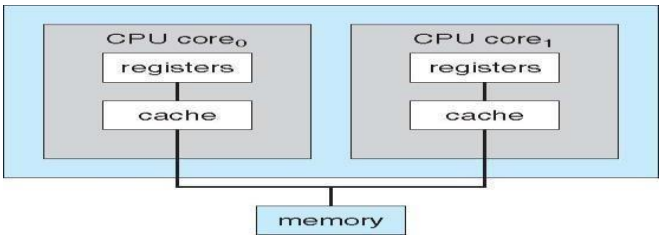
- **Asymmetric multiprocessing**
- **Symmetric multiprocessing**

1) **Asymmetric multiprocessing – (Master/Slave architecture)** Here each processor is assigned a specific task, by the master processor. A master processor controls the other processors in the system. It schedules and allocates work to the slave processors.
2) **Symmetric multiprocessing (SMP) –** All the processors are considered as peers. There is no master-slave relationship. All the processors have its own registers and CPU, only memory is shared.




The benefit of this model is that many processes can run simultaneously. *N* processes can run if there are *N* CPUs—without causing a significant deterioration of performance. Operating systems like Windows, Windows XP, Mac OS X, and Linux—now provide support for SMP.

A recent trend in CPU design is to include multiple compute **cores** on a single chip. The communication between processors within a chip is more faster than communication between two single processors.




**Clustered Systems**

Clustered systems are two or more individual systems connected together via network and sharing software resources. Clustering provides **high-availability** of resources and services. The service will continue even if one or more systems in the cluster fail. High availability is generally obtained by storing a copy of files (s/w resources) in the system.

There are two types of Clustered systems – **asymmetric** and **symmetric**

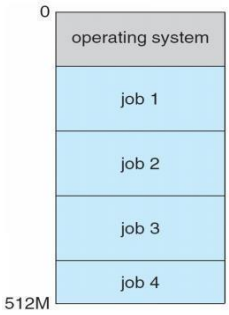
In **asymmetric clustering –** one system is in **hot-stand by mode** while the others are running the applications. The hot-standby host machine does nothing but monitor the active server. If that server fails, the hot-standby host becomes the active server.

In **symmetric clustering –** two or more systems are running applications, and are monitoring each other. This mode is more efficient, as it uses all of the available hardware. If any system fails, its job is taken up by the monitoring system.

Other forms of clusters include parallel clusters and clustering over a wide-area network (WAN). Parallel clusters allow multiple hosts to access the same data on the shared storage. Cluster technology is changing rapidly with the help of **SAN(storage-area networks)**. Using SAN resources can be shared with dozens of systems in a cluster, that are separated by miles.

## Operating-System Structure

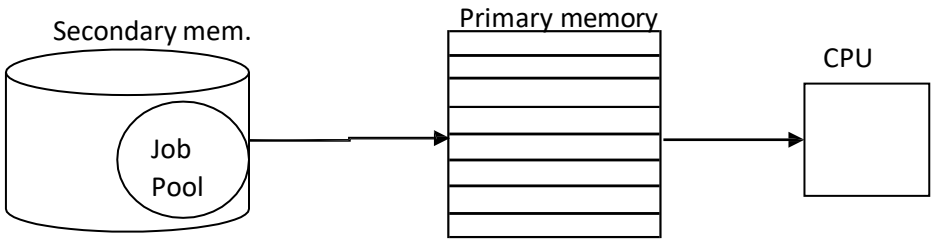
One of the most important aspects of operating systems is the ability to multiprogram. A single user cannot keep either the CPU or the I/O devices busy at all times. **Multiprogramming** increases CPU utilization by organizing jobs, so that the CPU always has one to execute.




The operating system keeps several jobs in memory simultaneously as shown in figure. This set of jobs is a subset of the jobs kept in the job pool. Since the number of jobs that can be kept simultaneously in memory is usually smaller than the number of jobs that can be kept in the job pool(in secondary memory). The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogrammed system, the operating system simply switches to,

and executes, another job. When *that* job needs to wait, the CPU is switched to *another* job, and so on.

Eventually, the first job finishes waiting and gets the CPU back. Thus the CPU is never idle.

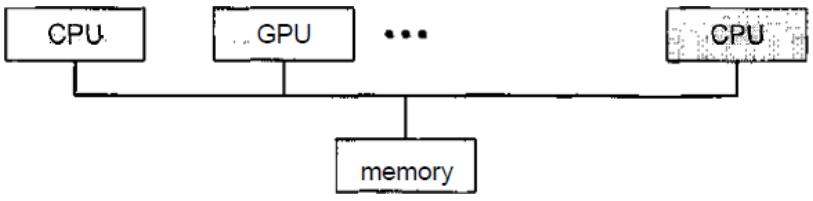



Multiprogrammed systems provide an environment in which the various system resources (for example, CPU, memory, and peripheral devices) are utilized effectively, but they do not provide for user interaction with the computer system.

In **Time sharing** (or **multitasking) systems,** a single CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running. The user feels that all the programs are being executed at the same time. Time sharing requires an **interactive** (or **hands-on) computer system,** which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using a input device such as a keyboard or a mouse, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use only, even though it is being shared among many users.

A **multiprocessor system** is a computer system having two or more CPUs within a single computer system, each sharing main memory and peripherals. Multiple programs are executed by multiple processors parallel.



**Distributed Systems**

Individual systems that are connected and share the resource available in network is called Distributed system. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

A **network** is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. TCP/IP is the most common network protocol. Most operating systems support TCP/IP.

Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide. A **metropolitan-area network** (MAN) links buildings within a city. A **small-area network** connects systems within a several feet using wireless technology. Eg. BlueTooth and 802.11.

The media to carry networks also vary - copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios.

A **network operating system** is an operating system that provides features such as file sharing across the network and that allows different processes on different computers to exchange messages.A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers.

**Operating-System Operations**

Modern operating systems are **interrupt driven.** 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 signaled by the occurrence of an interrupt or a trap. A **trap (or** an **exception)** is a software-generated interrupt. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided that is responsible for dealing with the interrupt.

**a) Dual-Mode Operation**

Since the operating system and the user programs share the hardware and software resources of the computer system, it has to be made sure that an error in a user program cannot cause problems to other programs and the Operating System running in the system.

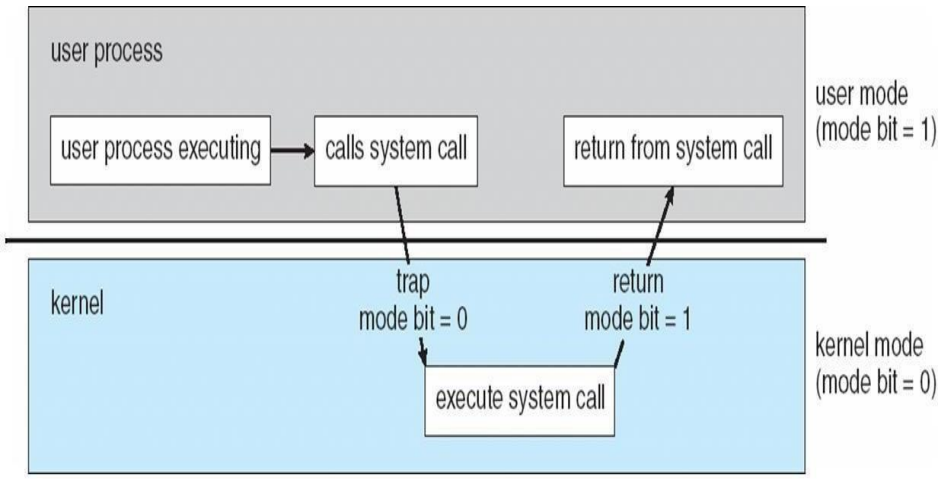
The approach taken is to use a hardware support that allows us to differentiate among various modes of execution.

The system can be assumed to work in two separate **modes** of operation:

- **user mode** and
- **kernel mode** (**supervisor mode, system mode, or privileged mode).**

A hardware bit of the computer, called the **mode bit**, is used to indicate the current mode: kernel (0) or user (1). With the mode bit, we are able to distinguish between a task that is executed by the operating system and one that is executed by the user.

When the computer system is executing a user application, the system is in user mode. When a user application requests a service from the operating system (via a system call), the transition from user to kernel mode takes place.




At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the mode bit from 1 to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode.

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

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

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

**b) Timer**

Operating system uses timer to control the CPU. A user program cannot hold CPU for a long time, this is prevented with the help of timer.

A timer can be set to interrupt the computer after a specified period. The period may be **fixed** (for example, 1/60 second) or **variable** (for example, from 1 millisecond to 1 second).

**Fixed timer** – After a fixed time, the process under execution is interrupted.

**Variable timer –** Interrupt occurs after varying interval. This is implemented using a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs.

Before changing to the user mode, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time.

## Process Management

A program under execution is a process. A process needs resources like CPU time, memory, files, and I/O devices for its execution. These resources are given to the process when it is created or at run time. When the process terminates, the operating system reclaims the resources.

The program stored on a disk is a **passive entity** and the program under execution is an **active entity**. A single-threaded process has one **program counter** specifying the next instruction to execute. The CPU executes one instruction of the process after another, until the process completes. A multithreaded process has multiple program counters, each pointing to the next instruction to execute for a given thread.

The operating system is responsible for the following activities in connection with process management:

- Scheduling process and threads on the CPU
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication

## Memory Management

Main memory is a large array of words or bytes. Each word or byte has its own address. Main memory is the storage device which can be easily and directly accessed by the CPU. As the program executes, the central processor reads instructions and also reads and writes data from main memory.

To improve both the utilization of the CPU and the speed of the computer's response to its users, general-purpose computers must keep several programs in memory, creating a need for memory management.

The operating system is responsible for the following activities in connection with memory management:

- Keeping track of which parts of memory are currently being used by user.
- Deciding which processes and data to move into and out of memory.
- Allocating and deallocating memory space as needed.

## Storage Management

There are three types of storage management i) File system management ii) Mass-storage management iii) Cache management.

**File-System Management**

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. Magnetic disk, optical disk, and magnetic tape are the most common. Each of these media has its own characteristics and physical organization. Each medium is controlled by a device, such as a disk drive or tape drive, that also has its own unique characteristics.

A file is a collection of related information defined by its creator. Commonly, files represent programs and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free-form (for example, text files), or they may be formatted rigidly (for example, fixed fields).

The operating system implements the abstract concept of a file by managing mass storage media. Files are normally organized into directories to make them easier to use. When multiple users have access to files, it may be desirable to control by whom and in what ways (read, write, execute) files may be accessed.

The operating system is responsible for the following activities in connection with file management:

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media

**Mass-Storage Management**

As the main memory is too small to accommodate all data and programs, and as the data that it holds are erased when power is lost, the computer system must provide secondary storage to back up main memory. Most modern computer systems use disks as the storage medium for both programs and data.

Most programs—including compilers, assemblers, word processors, editors, and formatters—are stored on a disk until loaded into memory and then use the disk as both the source and destination of their processing. Hence, the proper management of disk storage is of central importance to a computer system. The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

As the secondary storage is used frequently, it must be used efficiently. The entire speed of operation of a computer may depend on the speeds of the disk. Magnetic tape drives and their tapes, CD, DVD drives and platters are **tertiary storage** devices. The functions that operating systems provides include mounting and unmounting media in devices, allocating and freeing the devices for exclusive use by processes, and migrating data from secondary to tertiary storage.

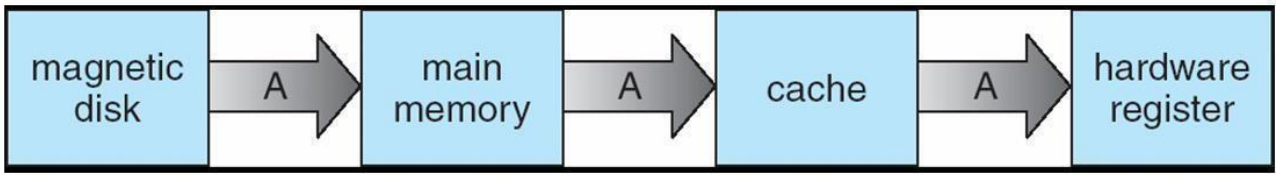
**Caching**

**Caching** is an important principle of computer systems. Information is normally kept in some storage system (such as main memory). As it is used, it is copied into a faster storage system—the cache—as temporary data. When a particular piece of information is required, first we check whether it is in the cache. If it is, we use the information directly from the cache; if it is not in cache, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

Because caches have limited size, **cache management** is an important design problem. Careful selection of the cache size and page replacement policy can result in greatly increased performance.

The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software. For instance, data transfer from cache to CPU and registers is usually a hardware function, with no operating-system intervention. In contrast, transfer of data from disk to memory is usually controlled by the operating system.

In a hierarchical storage structure, the same data may appear in different levels of the storage system. For example, suppose to retrieve an integer A from magnetic disk to the processing program. The operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory. This operation is followed by copying A to the cache and to an internal register. Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register.




In a multiprocessor environment, in addition to maintaining internal registers, each of the CPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, any update done to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency,** and it is usually a hardware problem (handled below the operating-system level).

**I/O Systems**

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. The I/O subsystem consists of several components:

- A memory-management component that includes buffering, caching, and spooling
- A general device-driver interface
- Drivers for specific hardware devices

Only the device driver knows the peculiarities of the specific device to which it is assigned.

**Protection and Security**

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose, there are mechanisms which ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system.

For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU for a long time. Device-control registers are not accessible to users, so the integrity of the various peripheral devices is protected.

**Protection** is a mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means for specification of the controls to be imposed and means for enforcement.

Protection improves reliability. A protection-oriented system provides a means to distinguish between authorized and unauthorized usage. A system can have adequate protection but still be prone to failure and allow inappropriate access.

Consider a user whose authentication information is stolen. Her data could be copied or deleted, even though file and memory protection are working. It is the job of **security** to defend a system from external and internal attacks. Such attacks spread across a huge range and include viruses and worms, denial-of service attacks etc.

Protection and security require the system to be able to distinguish among all its users. Most operating systems maintain 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.

**Distributed Systems**

A distributed system is a collection of systems that are networked to provide the users with access to the various resources in the network. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

A **network** is a communication path between two or more systems. Networks vary by the protocols used(TCP/IP,UDP,FTP etc.), the distances between nodes, and the transport media(copper wires, fiber-optic,wireless).

TCP/IP is the most common network protocol. The operating systems support of protocols also varies. Most operating systems support TCP/IP, including the Windows and UNIX operating systems.

Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a floor, or a building. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide. These networks may run one protocol or several protocols. A **metropolitan-area network** (MAN) connects buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **small-area network** such as might be found in a home.

The transportation media to carry networks are also varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network.

## Special-Purpose Systems

There are different classes of computer systems, whose functions are more limited and specific and it deal with limited computation domains. The systems can be classified as Real-Time Embedded Systems, Multimedia Systems and Handheld Systems.

**Real-Time Embedded Systems**

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to VCRs and microwave ovens. They tend to have very specific tasks. Usually, they have little user interface, and more time is spent for monitoring and managing hardware devices, eg. automobile engines and robotic arms.

The Operating Systems, in these embedded systems vary considerably. Some systems have standard operating systems—such as UNIX—with special-purpose applications. Others have special-purpose embedded operating system providing just the functionality desired.

Embedded systems always run **real-time operating systems.** A real-time system is used when there is restricted time for an operation or for the flow of data. A real-time system functions

correctly only if it returns the correct result within its time constraints. Sensors bring data to the computer. The computer must analyze the data and perform certain action.

Some medical imaging systems, automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are real-time systems. A real-time system has well-defined, fixed time constraints. Processing **mustbe** done within the defined constraints, or the system will fail. For instance, the robot arm should be halted before it has smashed into the car, it was building.

Entire houses can be computerized, so that a computer —can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home.

**Multimedia Systems**

Multimedia data consist of audio and video files as well as conventional files. These data differ from conventional data in that multimedia data—such as frames of video—must be delivered (streamed) according to certain time restrictions (for example, 30 frames per second).

Multimedia describes a wide range of applications like audio files - MP3, DVD movies, video conferencing, and short video clips of movie previews or news. Multimedia applications may also include live webcasts of speeches or sporting events and even live webcams. Multimedia applications can be either audio or video or combination of both. For example, a movie may consist of separate audio and video tracks.

**Handheld Systems**

Handheld systems include personal digital assistants (PDAs), such as Palm and Pocket-PCs, and cellular telephones. Developers of these systems face many challenges, due to the limited memory, slow processors and small screens in such devices.

The amount of physical memory in a handheld depends upon the device, the operating system and applications must manage memory efficiently. This includes returning all allocated memory back to the memory manager when the memory is not being used. A second issue of concern to developers of handheld devices is the speed of the processor used in the devices. Processors for most handheld devices run at faster speed than the processor in a PC. Faster processors require more power and so, a larger battery is required. Another issue is the usage of I/O devices.

Generally, the limitations in the functionality of PDAs are balanced by their convenience and portability. Their use continues to expand as network connections become more available and other options, such as digital cameras and MP3 players, expand their utility.

## Computing Environments

The different computing environments are -

**Traditional Computing**

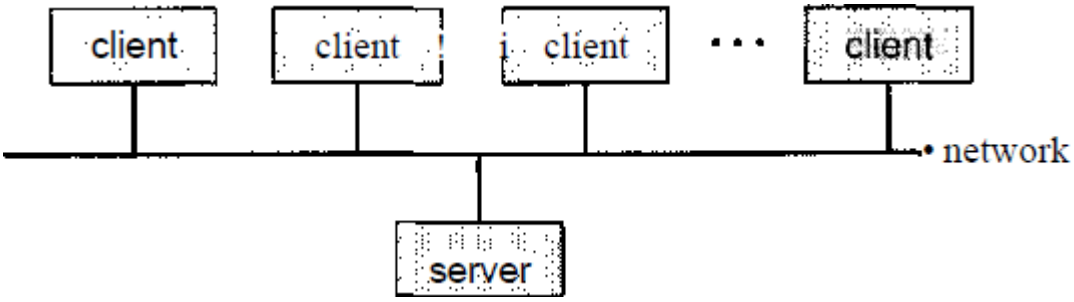
The current trend is toward providing more ways to access these computing environments. Web technologies are stretching the boundaries of traditional computing. Companies establish **portals,** which provide web accessibility to their internal servers. **Network computers** are essentially terminals that understand web-based computing. Handheld computers can synchronize with PCs to allow very portable use of company information. Handheld PDAs can also connect to **wireless networks** to use the company's web portal. The fast data connections are allowing home computers to serve up web pages and to use networks. Some homes even have **firewalls** to protect their networks.

In the latter half of the previous century, computing resources were scarce. Years before, systems were either batch or interactive. Batch system processed jobs in bulk, with predetermined input (from files or other sources of data). Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to rapidly cycle processes through the CPU, giving each user a share of the resources.

Today, traditional time-sharing systems are used everywhere. The same scheduling technique is still in use on workstations and servers, but frequently the processes are all owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time.

**Client-Server Computing**

Designers shifted away from centralized system architecture to - terminals connected to centralized systems. As a result, many of today's systems act as **server systems to** satisfy requests generated by **client systems.** This form of specialized distributed system, called **client-server** system.




General Structure of Client – Server System

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data); in response, the server executes the action and sends back results to the client. A server running a database that responds to client requests for data is an example of such a svstem.
- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running the web browsers.

**Peer-to-Peer Computing**

In this model, clients and servers are not distinguished from one another; here, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service.

In a client-server system, the server is a bottleneck, because all the services must be served by the server. But in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- A peer acting as a client must know, which node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a *discovery protocol* must be provided that allows peers to discover services provided by other peers in the network.

**Web-Based Computing**

Web computing has increased the importance on networking. Devices that were not previously networked now include wired or wireless access. Devices that were networked now have faster network connectivity.

The implementation of web-based computing has given rise to new categories of devices, such as **load balancers,** which distribute network connections among a pool of similar servers. Operating systems like Windows 95, which acted as web clients, have evolved into Linux and Windows XP, which can act as web servers as well as clients. Generally, the Web has increased the complexity of devices, because their users require them to be web-enabled.

The design of an operating system is a major task. It is important that the goals of the new system be well defined before the design of OS begins. These goals form the basis for choices among various algorithms and strategies.

## 2.1 Operating-System Services

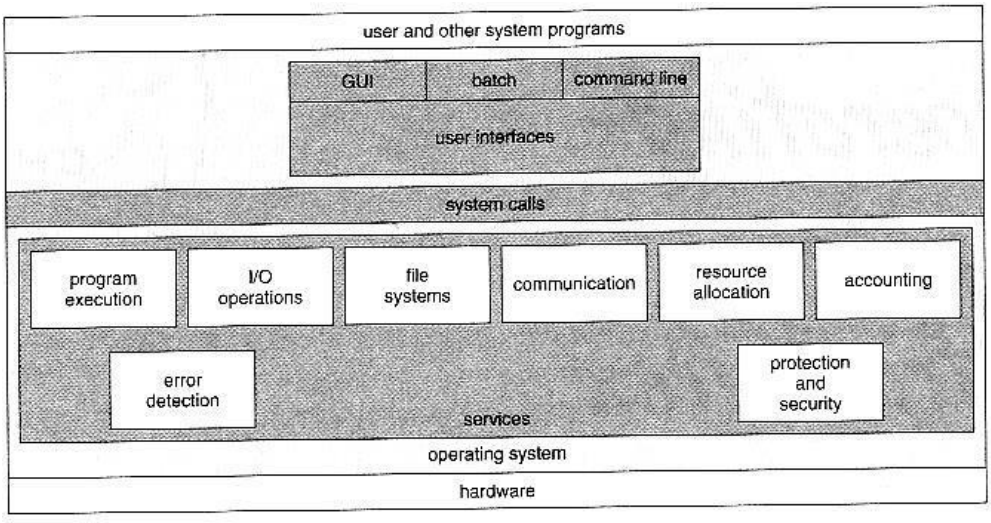



**Figure 2.1** A view of operating system services.

An operating system provides an environment for the execution of programs.

It provides certain services to programs and to the users of those programs.

OS provide services for the users of the system, including:

- **User Interfaces** - Means by which users can issue commands to the system. Depending on the operating system these may be a **command-line interface** ( e.g. sh, csh, ksh, tcsh, etc.), a **Graphical User Interface** (e.g. Windows, X-Windows, KDE, Gnome, etc.), or a **batch command systems**. In Command Line Interface(CLI)- commands are given to the system. In Batch interface – commands and directives to control these commands are put in a file and then the file is executed. In GUI systems- windows with pointing device to get inputs and keyboard to enter the text.
- **Program Execution** - The OS must be able to load a program into RAM, run the program, and terminate the program, either normally or abnormally.
- **I/O Operations** - The OS is responsible for transferring data to and from I/O devices, including keyboards, terminals, printers, and files. For specific devices, special functions are provided(device drivers) by OS.

- **File-System Manipulation** – Programs need to read and write files or directories. The services required to create or delete files, search for a file, list the contents of a file and change the file permissions are provided by OS.
- **Communications** - Inter-process communications, IPC, either between processes running on the same processor, or between processes running on separate processors or separate machines. May be implemented by using the service of OS- like shared memory or message passing.
- **Error Detection** - Both hardware and software errors must be detected and handled appropriately by the OS. Errors may occur in the CPU and memory hardware (such as power failure and memory error), 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).

OS provide services for the efficient operation of the system, including:

- **Resource Allocation** – Resources like CPU cycles, main memory, storage space, and I/O devices must be allocated to multiple users and multiple jobs at the same time.
- **Accounting** – There are services in OS to keep track of system activity and resource usage, either for billing purposes or for statistical record keeping that can be used to optimize future performance.
- **Protection and Security** – The owners of information(file) in multiuser or networked computer system may want to control the use of that information. When several separate processes execute concurrently, one process should not interfere with other or with OS. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders must also be done, by means of a password.

## 2.2 User Operating-System Interface

There are several ways for users to interface with the operating system.

1) Command-line interface, or command interpreter, allows users to directly enter commands to be performed by the operating system.
2) Graphical user interface(GUI), allows users to interface with the operating system using pointer device and menu system.

### Command Interpreter

Command Interpreters are used to give commands to the OS. There are multiple command interpreters known as shells. In UNIX and Linux systems, there are several different shells, like the *Bourne shell, C shell, Bourne-Again shell, Korn shell,* and others.

The main function of the command interpreter is to get and execute the user-specified command. Many of the commands manipulate files: create, delete, list, print, copy, execute, and so on.

The commands can be implemented in two general ways-

1) The command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a particular section of its code that sets up the parameters and makes the appropriate system call.

2) The code to implement the command is in a function in a separate file. The interpreter searches for the file and loads it into the memory and executes it by passing the parameter. Thus by adding new functions new commands can be added easily to the interpreter without disturbing it.

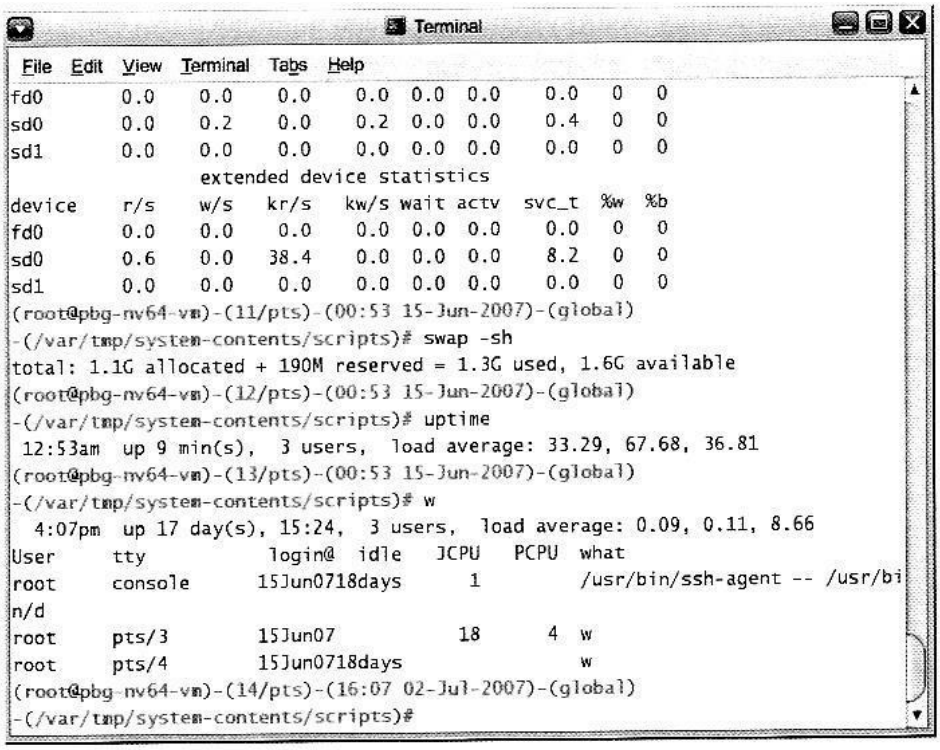



**Figure 2.2** The Bourne shell command interpreter in Solaris 10.

### Graphical User Interface, GUI

Another way of interfacing with the operating system is through a user friendly graphical user interface, or GUI. Here, rather than entering commands directly via a command-line interface, users employ a mouse-based window and menu system. The user moves the mouse to position its

pointer on images, or icons on the screen (the desktop) that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory-known as a folder-or pull down a menu that contains commands.

Graphical user interfaces first appeared on the Xerox Alto computer in 1973.

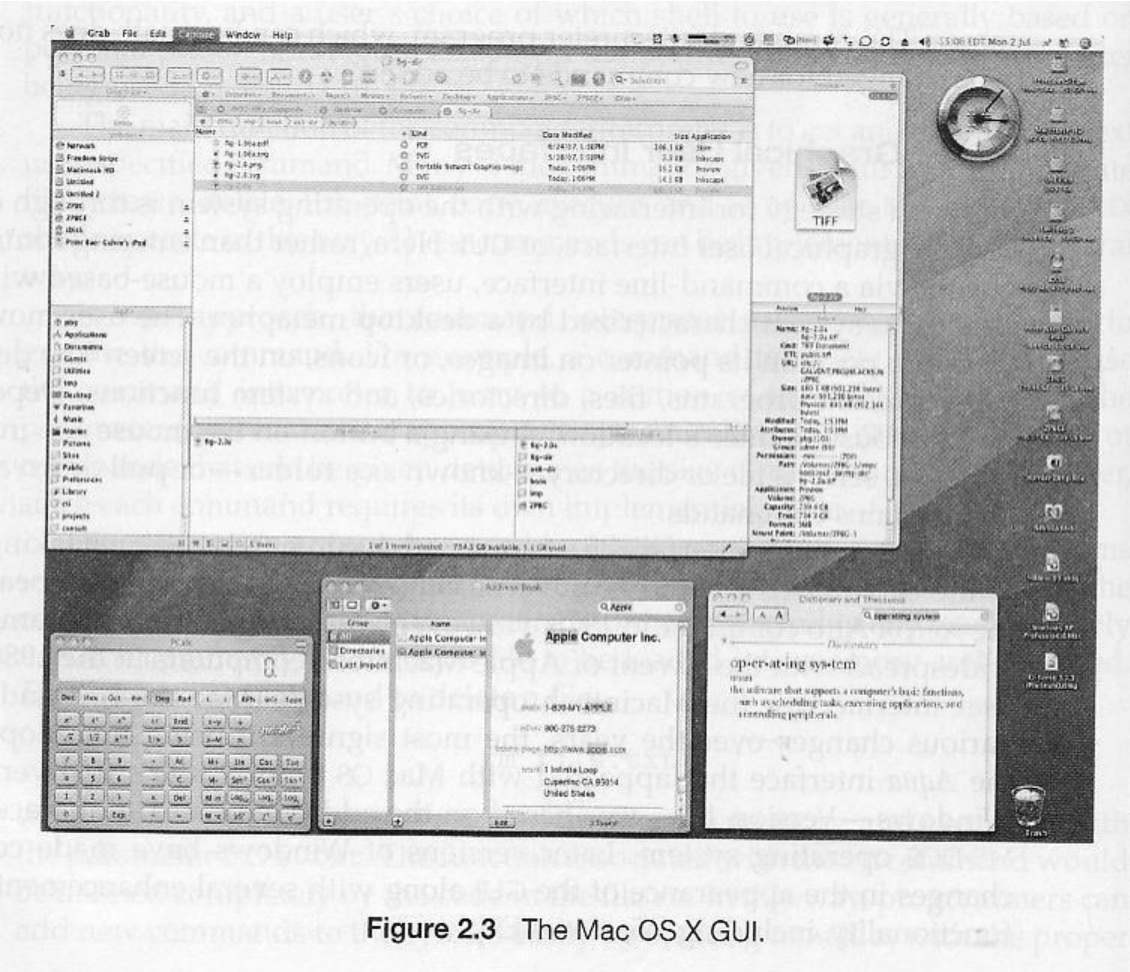



**Figure 2.3** The Mac OS X GUI.

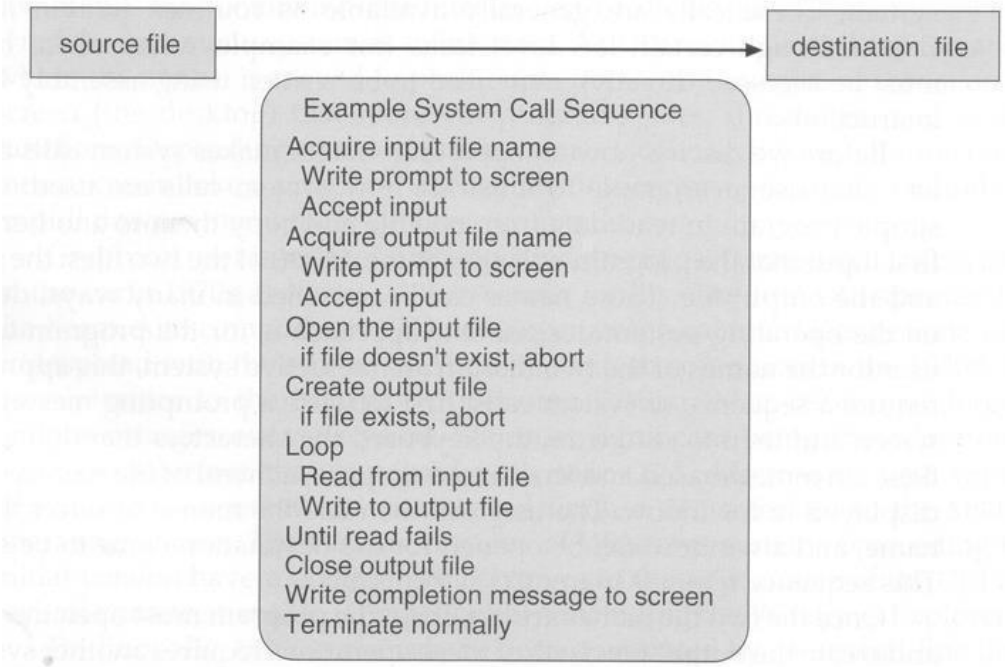
Most modern systems allow individual users to select their desired interface, and to customize its operation, as well as the ability to switch between different interfaces as needed.

## 2.3 System Calls

- System calls is a means to access the services of the operating system.

- Generally written in C or C++, although some are written in assembly for optimal performance.

- The below figure illustrates the sequence of system calls required to copy a file content from one file(input file) to another file (output file).




There are number of system calls used to finish this task. The first system call is to write a message on the screen (monitor). Then to accept the input filename. Then another system call to write message on the screen, then to accept the output filename. When the program tries to open the input file, it may find that there is no file of that name or that the file is protected against access. In these cases, the program should print a message on the console(another system call) and then terminate abnormally (another system call) and create a new one (another system call).

Now that both the fileas are opened, we enter a loop that reads from the input file(another system call) and writes to output file (another system call).

Finally, after the entire file is copied, the program may close both files (another system call), write a message to the console or window(system call), and finally terminate normally (final system call).

- Most programmers do not use the low-level system calls directly, but instead use an "Application Programming Interface", API.

- The APIs instead of direct system calls provides for greater program portability between different systems. The API then makes the appropriate system calls through the system call interface, using a system call table to access specific numbered system calls, as shown in Figure 2.6.
- Each system call has a specific numbered system call. The system call table (consisting of system call number and address of the particular service) invokes a particular service routine for a specific system call.
- The caller need know nothing about how the system call is implemented or what it does during execution.

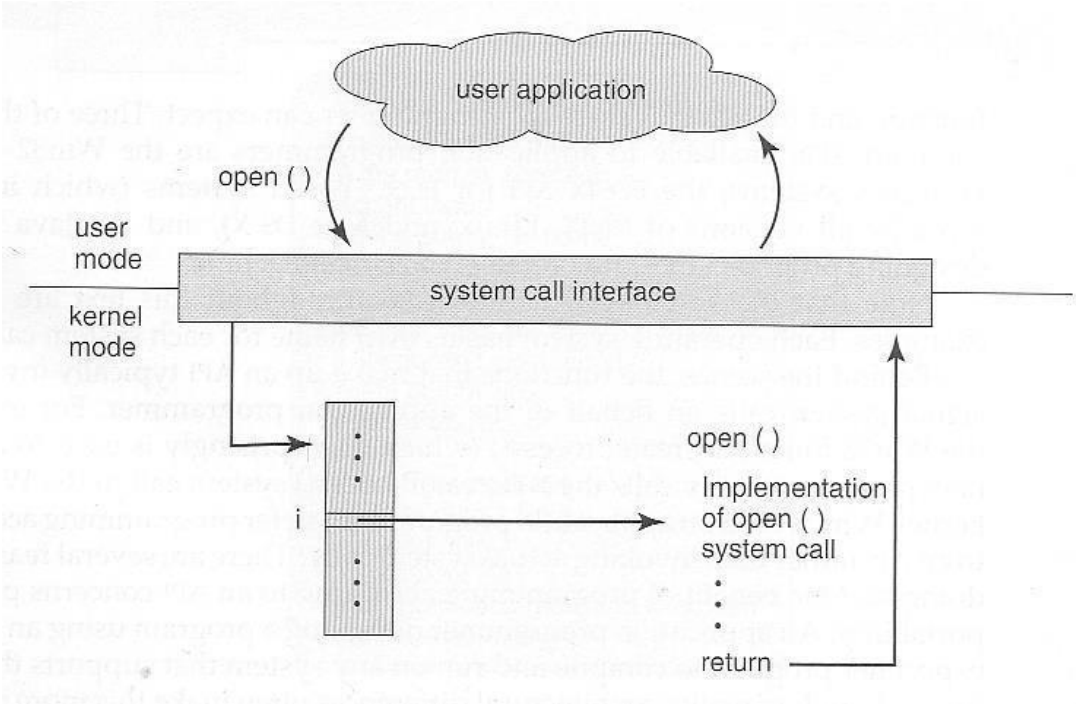



**Figure 2.6 The handling of a user application invoking the open() system call.**

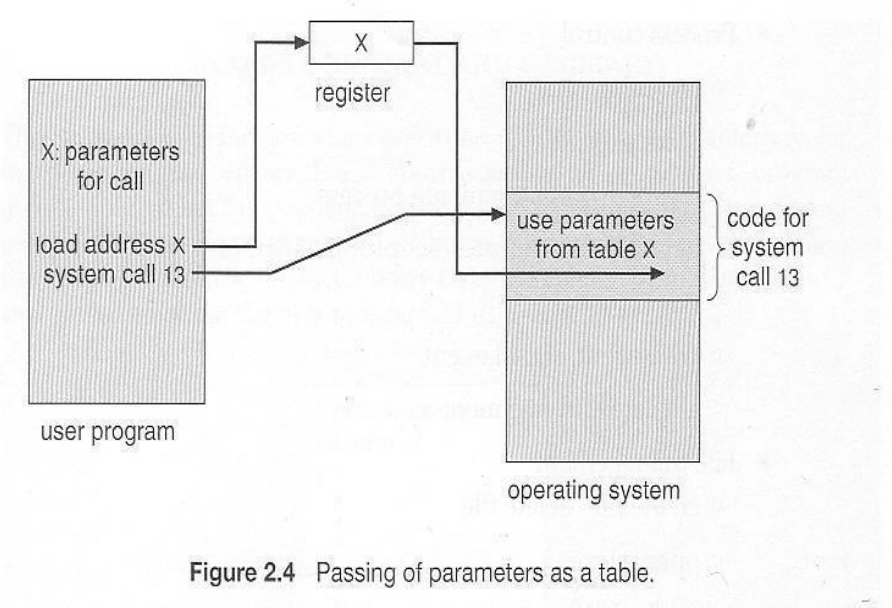



Figure 2.4 Passing of parameters as a table.

Three general methods used to pass parameters to OS are –

- To pass parameters in registers
- If parameters are large blocks, address of block (where parameters are stored in memory) is sent to OS in the register. (Linux & Solaris).
- Parameters can be pushed onto the stack by program and popped off the stack by OS.

### 2.3.1 Types of System Calls

The system calls can be categorized into six major categories:

- Process Control
- File management
- Device management
- Information management
- Communications
- Protection

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

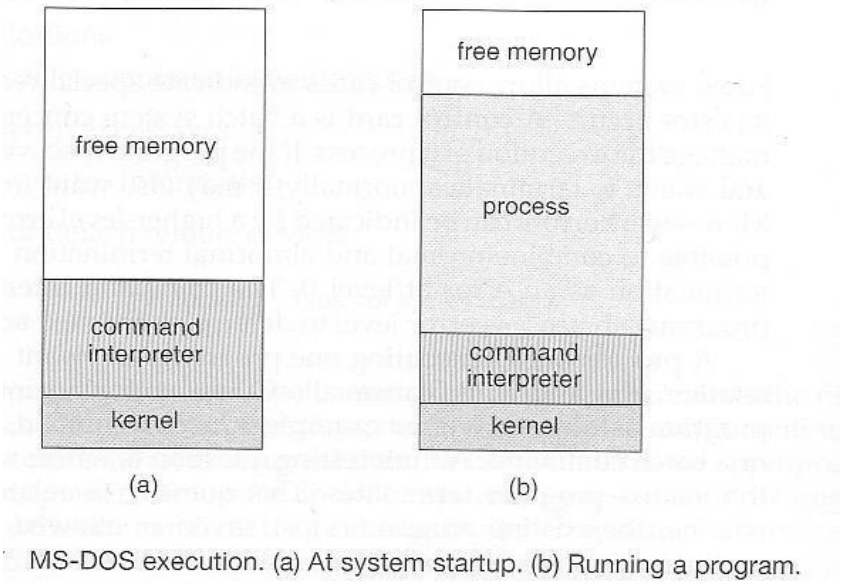
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

**a) Process Control**

- Process control system calls include end, abort, load, execute, create process, terminate process, get/set process attributes, wait for time or event, signal event, and allocate and free memory.
- Processes must be created, launched, monitored, paused, resumed, and eventually stopped.
- When one process pauses or stops, then another must be launched or resumed
- Process attributes like process priority, max. allowable execution time etc. are set and retrieved by OS.
- After creating the new process, the parent process may have to wait (wait time), or wait for an event to occur(wait event). The process sends back a signal when the event has occurred (signal event).

- In DOS, the command interpreter loaded first.Then loads the process and transfers control to it. The interpreter does not resume until the process has completed, as shown in Figure 2.10:




MS-DOS execution. (a) At system startup. (b) Running a program.

**Figure 2.10**

- Because UNIX is a multi-tasking system, the command interpreter remains completely resident when executing a process, as shown in Figure 2.11 below.
  - The user can switch back to the command interpreter at any time, and can place the running process in the background even if it was not originally launched as a background process.
  - In order to do this, the command interpreter first executes a "fork" system call, which creates a second process which is an exact duplicate ( clone ) of the original command interpreter. The original process is known as the parent, and the cloned process is known as the child, with its own unique process ID and parent ID.
  - The child process then executes an "exec" system call, which replaces its code with that of the desired process.
  - The parent ( command interpreter ) normally waits for the child to complete before issuing a new command prompt, but in some cases it can also issue a new prompt right away, without waiting for the child process to complete. ( The child is then said to be running "in the background", or "as a background process". )

**b) File Management**

The file management functions of OS are –

- File management system calls include create file, delete file, open, close, read, write, reposition, get file attributes, and set file attributes.
- After **creating** a file, the file is **opened**. Data is **read** or **written** to a file.
- The file pointer may need to be **repositioned** to a point.
- The file **attributes** like filename, file type, permissions, etc. are set and retrieved using system calls.
- These operations may also be supported for directories as well as ordinary files.

**c) Device Management**

- Device management system calls include **request device**, **release** device, **read**, **write**, **reposition**, **get/set** device attributes, and logically **attach** or **detach** devices.
- When a process needs a resource, a request for resource is done. Then the control is granted to the process. If requested resource is already attached to some other process, the requesting process has to wait.
- In multiprogramming systems, after a process uses the device, it has to be returned to OS, so that another process can use the device.
- Devices may be physical ( e.g. disk drives ), or virtual / abstract ( e.g. files, partitions, and RAM disks ).

**d) Information Maintenance**

- Information maintenance system calls include calls to get/set the time, date, system data, and process, file, or device attributes.
- These system calls care used to transfer the information between user and the OS. Information like current time & date, no. of current users, version no. of OS, amount of free memory, disk space etc. are passed from OS to the user.

**e) Communication**

- Communication system calls create/delete communication connection, send/receive messages, transfer status information, and attach/detach remote devices.
- The **message passing** model must support calls to:
    - Identify a remote process and/or host with which to communicate.
    - Establish a connection between the two processes.
    - Open and close the connection as needed.

- Transmit messages along the connection.
- Wait for incoming messages, in either a blocking or non-blocking state.
- Delete the connection when no longer needed.

- The **shared memory** model must support calls to:
  - Create and access memory that is shared amongst processes (and threads. )
  - Free up shared memory and/or dynamically allocate it as needed.
- Message passing is simpler and easier, ( particularly for inter-computer communications ), and is generally appropriate for small amounts of data. It is easy to implement, but there are system calls for each read and write process.
- Shared memory is faster, and is generally the better approach where large amounts of data are to be shared. This model is difficult to implement, and it consists of only few system calls.

**f) Protection**

- Protection provides mechanisms for controlling which users / processes have access to which system resources.
- System calls allow the access mechanisms to be adjusted as needed, and for non-priveleged users to be granted elevated access permissions under carefully controlled temporary circumstances.

## 2.4 System Programs

A collection os programs that provide a convenient environment for program development and execution (other than OS) are called system programs or system utilities.

- It is not a part of the kernel or command interpreters.
- System programs may be divided into five categories:
  - **File management** - programs to create, delete, copy, rename, print, list, and generally manipulate files and directories.
  - **Status information** - Utilities to check on the date, time, number of users, processes running, data logging, etc. System **registries** are used to store and recall configuration information for particular applications.
  - **File modification** - e.g. text editors and other tools which can change file contents.
  - **Programming-language support** - E.g. Compilers, linkers, debuggers, profilers, assemblers, library archive management, interpreters for common languages, and support for make.
  - **Program loading and execution** - loaders, dynamic loaders, overlay loaders, etc., as well as interactive debuggers.

- **Communications** - Programs for providing connectivity between processes and users, including mail, web browsers, remote logins, file transfers, and remote command execution.

## 2.5 Operating-System Design and Implementation

### 2.5.1 Design Goals

Any system to be designed must have its own goals and specifications. Similarly the OS to be built will have its own goals depending on the type of system in which it will be used, the type of hardware used in the system etc.

- **Requirements** define properties which the finished system must have, and are a necessary steps in designing any large complex system. The requirements may be of two basic groups:

1. User goals (User requirements)
2. System goals (system requirements)

- **User requirements** are features that users care about and understand like system should be convenient to use, easy to learn,reliable, safe and fast.
- **System requirements** are written for the developers, ie. People who design the OS. Their requirements are like easy to design, implement and maintain, flexible, reliable, error free and efficient.

### 2.5.2 Mechanisms and Policies

- Policies determine *what* is to be done. Mechanisms determine *how* it is to be implemented.
- Example: in timer, counter and decrementing counter is the mechanism and deciding how long the time has to be set is the policies.
- Policies change overtime. In the worst case, each change in policy would require a change in the underlying mechanism.
- If properly separated and implemented, policy changes can be easily adjusted without re-writing the code, just by adjusting parameters or possibly loading new data / configuration files.

### 2.5.3 Implementation

- Traditionally OS were written in assembly language.
- In recent years, Os are written in C, or C++. Critical sections of code are still written in assembly language.
- The first OS that was not written in assembly language was the Master Control Program (MCP).
- The advantages of using a higher-level language for implementing operating systems are: The code can be written faster, more compact, easy to port to other systems and is easier to understand and debug.
- The only disadvantages of implementing an operating system in a higher-level language are reduced speed and increased storage requirements.

## 2.7 Operating-System Structure

OS structure must be carefully designed. The task of OS is divided into small components and then interfaced to work together.

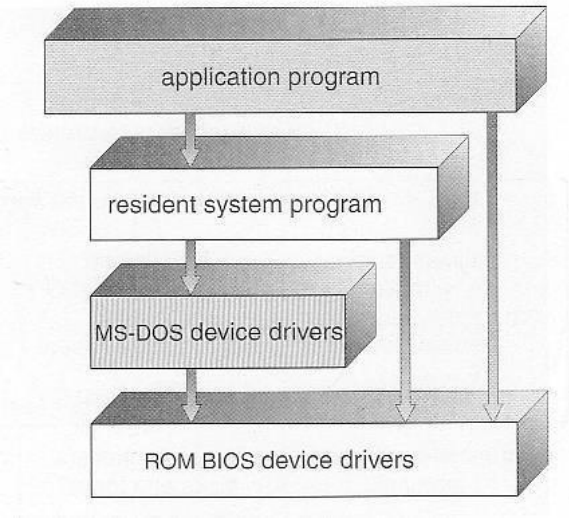
### 2.7.1 Simple Structure

Many operating systems do not have well-defined structures. They started as small, simple, and limited systems and then grew beyond their original scope. Eg: MS-DOS.

In MS-DOS, the interfaces and levels of functionality are not well separated. Application programs can access basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS in bad state and the entire system can crash down when user programs fail.

UNIX OS consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system

calls.




**MS-DOS Layer Structure**

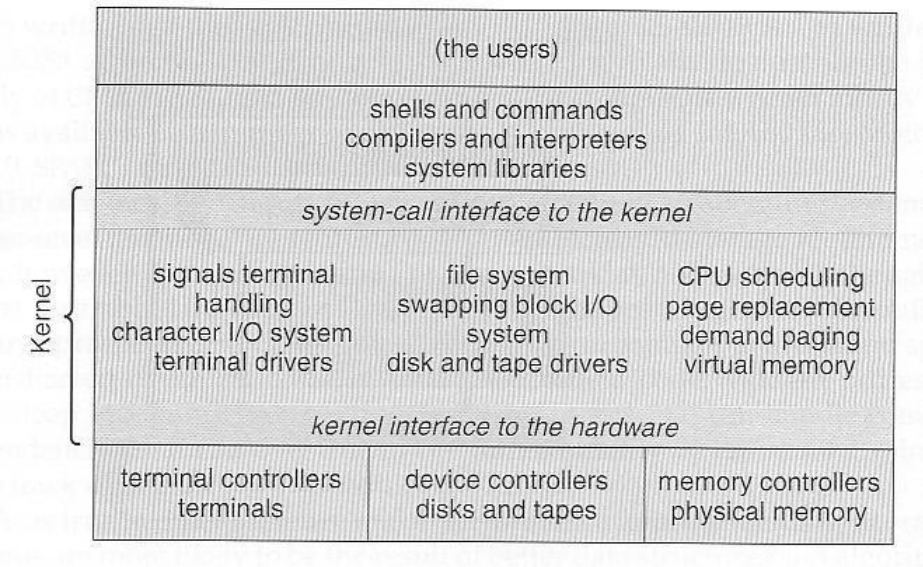



Figure 2.13 UNIX System Structure

### 2.7.2 Layered Approach

- The OS is broken into number of layers (levels). Each layer rests on the layer below it, and relies on the services provided by the next lower layer.

- Bottom layer(layer 0) is the hardware and the topmost layer is the user interface.
- A typical layer, consists of data structure and routines that can be invoked by higher-level layer.

Advantage of layered approach is simplicity of construction and debugging.

The layers are selected so that each uses functions and services of only lower-level layers. So simplifies debugging and system verification. The layers are debugged one by one from the lowest and if any layer doesn't work, then error is due to that layer only, as the lower layers are already debugged. Thus the design and implementation is simplified.

A layer need not know how its lower level layers are implemented. Thus hides the operations from higher layers.

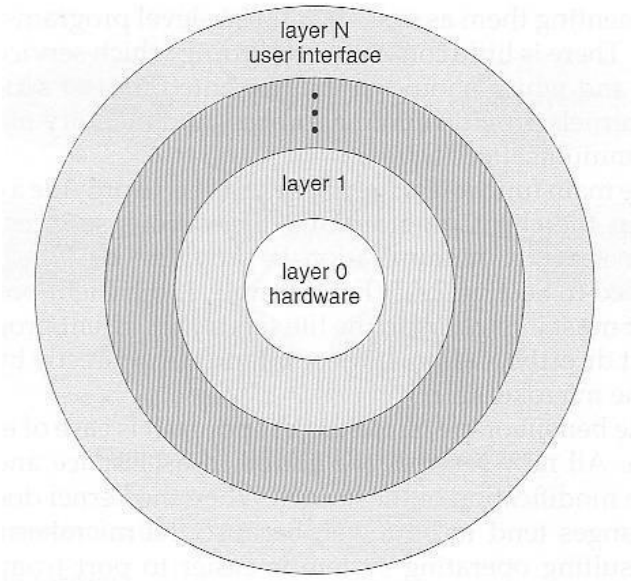



**Figure 2.14 A layered Operating System**

Disadvantages of layered approach:

- The various layers must be appropriately defined, as a layer can use only lower level layers.
- Less efficient than other types, because any interaction with layer 0 required from top layer. The system call should pass through all the layers and finally to layer 0. This is an overhead.

### 2.7.3 Microkernels

- The basic idea behind micro kernels is to **remove all non-essential** services from the kernel, thus making the kernel as small and efficient as possible.
- The removed services are implemented as system applications.
- Most microkernels provide basic process and memory management, and message passing between other services.
- **Benefit** of microkernel - System expansion can also be easier, because it only involves adding more system applications, not rebuilding a new kernel.
- Mach was the first and most widely known microkernel, and now forms a major component of Mac OSX.
- Disadvantage of Microkernel is, it suffers from reduction in performance due to increases system function overhead.

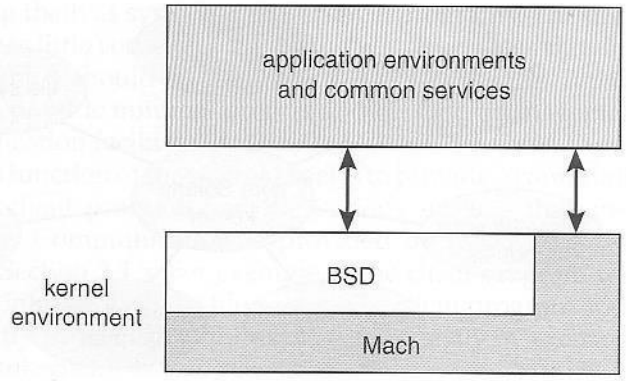



**Figure 2.14** The Mac OS X structure.

### 2.7.4 Modules

- Modern OS development is object-oriented, with a relatively small core kernel and a set of ***modules*** which can be linked in dynamically.
- Modules are similar to layers in that each subsystem has clearly defined tasks and interfaces, but any module is free to contact any other module, eliminating the problems of going through multiple intermediary layers.
- The kernel is relatively small in this architecture, similar to microkernels, but the kernel does not have to implement message passing since modules are free to contact each other directly. Eg: Solaris, Linux and MacOSX.

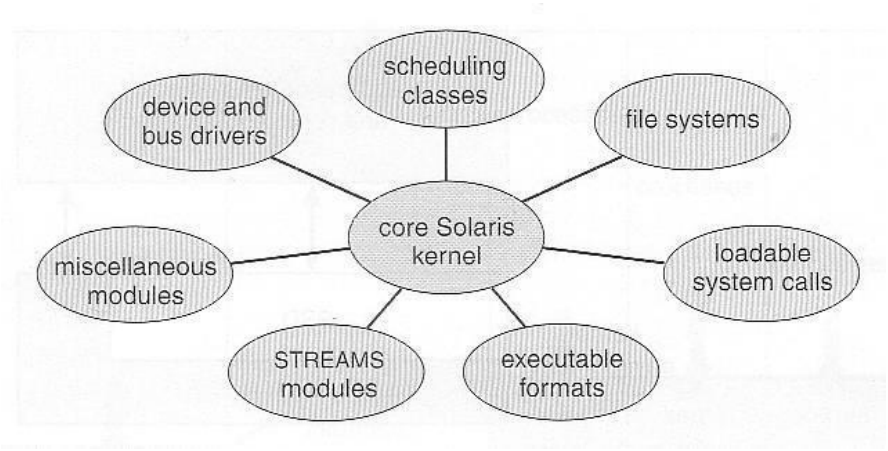



**Figure 2.15 Solaris loadable modules**

- The Max OSX architecture relies on the Mach microkernel for basic system management services, and the BSD kernel for additional services. Application services and dynamically loadable modules ( kernel extensions ) provide the rest of the OS functionality.
- Resembles layered system, but a module can call any other module.
- Resembles microkernel, the primary module has only core functions and the knowledge of how to load and communicate with other modules.

## 2.8 Virtual Machines

The fundamental idea behind a virtual machine is to abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

Creates an illusion that a process has its own processor with its own memory. Host OS is the main OS installed in system and the other OS installed in the system are called guest OS.

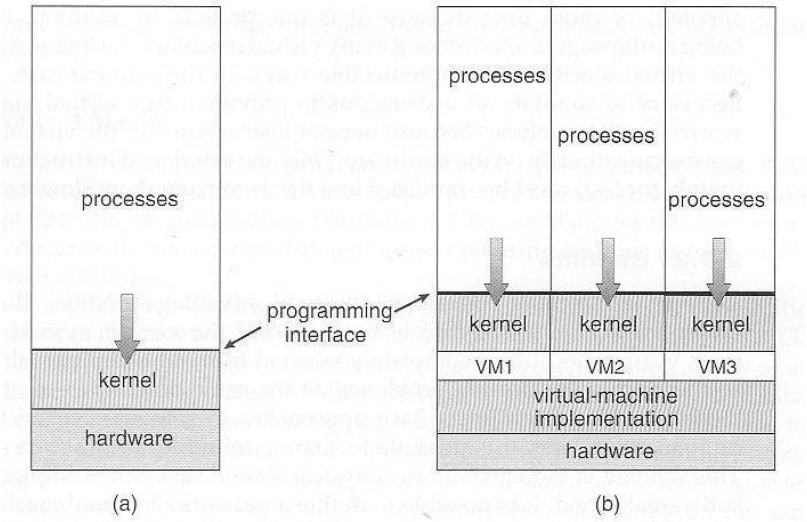



**Figure 2.17 System modes. (A) Nonvirtual machine (b) Virtual machine**

- Virtual machines first appeared as the VM Operating System for IBM mainframes in 1972.

**Benefits**

- Able to share the same hardware and run several different execution environments(OS).
- Host system is protected from the virtual machines and the virtual machines are protected from one another. A virus in guest OS, will corrupt that OS but will not affect the other guest systems and host systems.
- Even though the virtual machines are separated from one another, software resources can be shared among them. Two ways of sharing s/w resource for communication are: a)To share a file system volume(part of memory). b)To develop a virtual communication network to communicate between the virtual machines.
- The operating system runs on and controls the entire machine. Therefore, the current system must be stopped and taken out of use while changes are made and tested. This period is commonly called *system development time.* In virtual machines such problem is eliminated. User programs are executed in one virtual machine and system development is done in another environment.
- Multiple OS can be running on the developer's system **concurrently**. This helps in rapid porting and testing of programmers code in different environments.
- **System consolidation** – two or more systems are made to run in a single system.

**Simulation –**

Here the host system has one system architecture and the guest system is compiled in different architecture. The compiled guest system programs can be run in an emulator that translates each instructions of guest program into native instructions set of host system.

**Para-Virtualization –**

This presents the guest with a system that is similar but not identical to the guest's preferred system. The guest must be modified to run on the para-virtualized hardware.

### 2.8.6 Examples

#### *2.8.6.1 VMware*

VMware is a popular commercial application that abstracts Intel 80X86 hardware into isolated virtual machines. The virtualization tool runs in the user-layer on top of the host OS. The virtual machines running in this tool believe they are running on bare hardware, but the fact is that it is running inside a user-level application.

VMware runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different **guest operating systems** as independent virtual machines.

In below scenario, Linux is running as the host operating system; FreeBSD, Windows NT, and Windows XP are running as guest operating systems. The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems. Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.

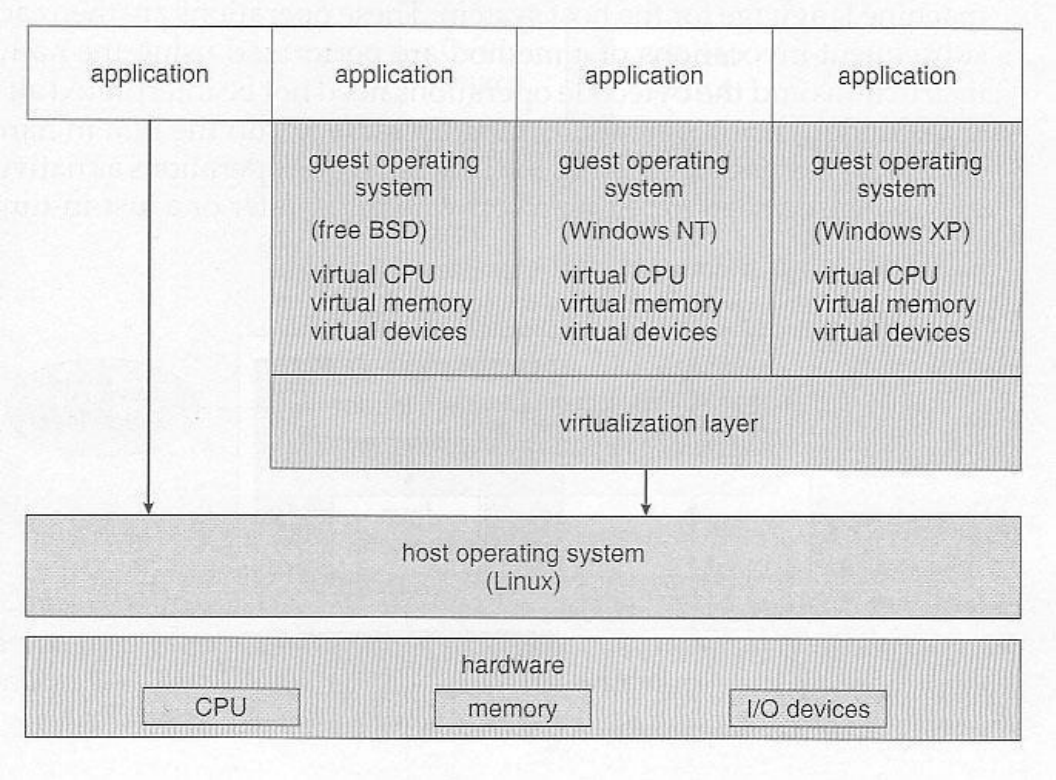



**Figure 2.19 VMware architecture**

### *2.8.6.2 The Java Virtual Machine*

- Java was designed from the beginning to be platform independent, by running Java only on a Java Virtual Machine, JVM, of which different implementations have been developed for numerous different underlying HW platforms.
- Java source code is compiled into Java byte code in .class files. Java byte code is binary instructions that will run on the JVM.
- The JVM implements memory management and garbage collection.
- JVM consists of class loader and Java Interpreter. Class loader loads compiled .class files from both java program and java API for the execution of java interpreter. Then it checks the .class file for validity.

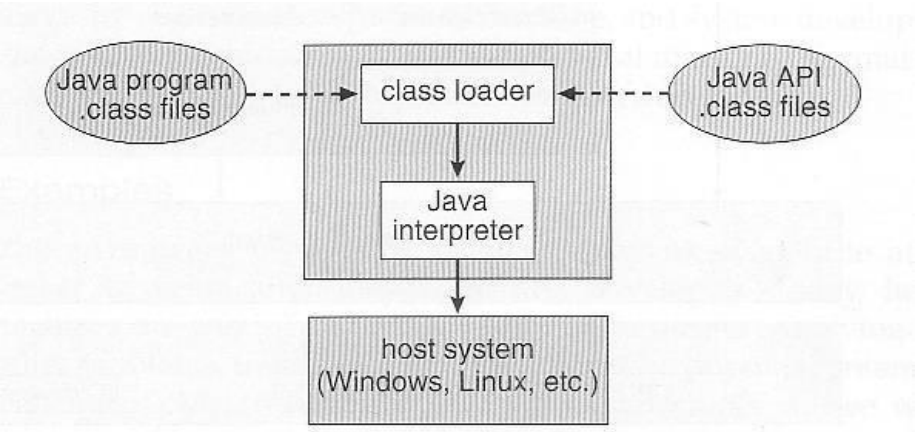



**Figure 2.20 The JVM**

## 2.10 Operating-System Generation

- OSes may be designed and built for a specific HW configuration at a specific site, but more commonly they are designed with a number of variable parameters and components, which are then configured for a particular operating environment.
- Systems sometimes need to be re-configured after the initial installation, to add additional resources, capabilities, or to tune performance, logging, or security.

- At one extreme the OS source code can be edited, re-compiled, and linked into a new kernel.
- More commonly configuration tables determine which modules to link into the new kernel, and what values to set for some key important parameters. This approach may require the configuration of complicated makefiles, which can be done either automatically or through interactive configuration programs; Then make is used to actually generate the new kernel specified by the new parameters.
- At the other extreme a system configuration may be entirely defined by table data, in which case the "rebuilding" of the system merely requires editing data tables.
- Once a system has been regenerated, it is usually required to reboot the system to activate the new kernel. Because there are possibilities for errors, most systems provide some mechanism for booting to older or alternate kernels.

## 2.11 System Boot

The general approach when most computers boot up goes something like this:

- When the system powers up, an interrupt is generated which loads a memory address into the program counter, and the system begins executing instructions found at that address. This address points to the "bootstrap" program located in ROM chips ( or EPROM chips ) on the motherboard.
- The ROM bootstrap program first runs hardware checks, determining what physical resources are present and doing power-on self tests ( POST ) of all HW for which this is applicable. Some devices, such as controller cards may have their own on-board diagnostics, which are called by the ROM bootstrap program.
- The user generally has the option of pressing a special key during the POST process, which will launch the ROM BIOS configuration utility if pressed. This utility allows the user to specify and configure certain hardware parameters as where to look for an OS and whether or not to restrict access to the utility with a password.
    - Some hardware may also provide access to additional configuration setup programs, such as for a RAID disk controller or some special graphics or networking cards.
- Assuming the utility has not been invoked, the bootstrap program then looks for a non-volatile storage device containing an OS. Depending on configuration, it may look for a floppy drive, CD ROM drive, or primary or secondary hard drives, in the order specified by the HW configuration utility.
- Assuming it goes to a hard drive, it will find the first sector on the hard drive and load up the fdisk table, which contains information about how the physical hard drive is divided up into logical partitions, where each partition starts and ends, and which partition is the "active" partition used for booting the system.
- There is also a very small amount of system code in the portion of the first disk block not occupied by the fdisk table. This bootstrap code is the first step that is not built into the hardware, i.e. the first part which might be in any way OS-specific. Generally this code knows just enough to access the hard drive, and to load and execute a ( slightly ) larger boot program.
- For a single-boot system, the boot program loaded off of the hard disk will then proceed to locate the kernel on the hard drive, load the kernel into memory, and then transfer control over to the kernel. There may be some opportunity to specify a particular kernel to be loaded at this stage, which may be useful if a new kernel has just been generated and doesn't work, or if the system has multiple kernels available with different configurations for different purposes. ( Some systems may boot different configurations automatically, depending on what hardware has been found in earlier steps. )
- For dual-boot or multiple-boot systems, the boot program will give the user an opportunity to specify a particular OS to load, with a default choice if the user does not pick a particular OS within a given time frame. The boot program then finds the boot loader for the chosen single-boot OS, and runs that program as described in the previous bullet point.
- Once the kernel is running, it may give the user the opportunity to enter into single-user mode, also known as maintenance mode. This mode launches very few if any system services, and does not enable any logins other than the primary log in on the console. This mode is used primarily for system maintenance and diagnostics.

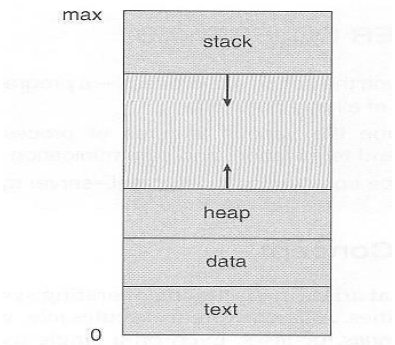
# Processes Concept

- A process is a program under execution.
- Its current activity is indicated by PC(Program Counter) and CPU registers.

## The Process

Process memory is divided into four sections as shown in the figure below:

- The stack is used to store local variables, function parameters, function return values, return address etc.
- The heap is used for dynamic memory allocation.
- The data section stores global and static variables.
- The text section comprises the compiled program code.
- Note that, there is a free space between the stack and the heap. When the stack is full, it grows downwards and when the heap is full, it grows upwards.




ure 3.1 Process in memory.

## Process State

A Process has 5 states. Each process may be in one of the following states –

- **New** - The process is in the stage of being created.
- **Ready** - The process has all the resources it needs to run. It is waiting to be assigned to the processor.
- **Running** – Instructions are being executed..
- **Waiting** - The process is waiting for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.
- **Terminated** - The process has completed its execution.

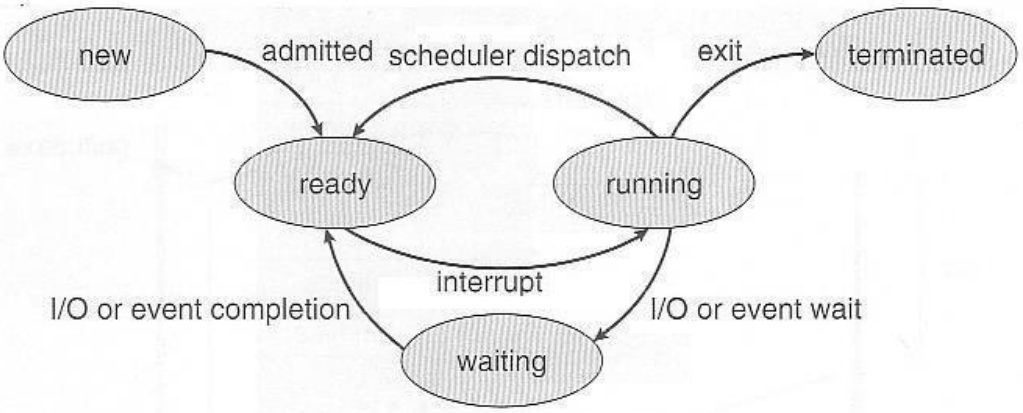



Figure 3.2 Diagram of process state.

## Process Control Block

For each process there is a Process Control Block (PCB), which stores the process-specific information as shown below –

**Process State** – The state of the process may be new, ready, running, waiting, 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. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

**CPU scheduling information**- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

**Memory-management information** – This include information such as the value of the base and limit registers, the page tables, or the segment tables.

**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.

The PCB simply serves as the repository for any information that may vary from

process to process.

## Process Scheduling

Process Scheduler selects an available process for program execution on the CPU. In a multiprocessor system - one process will be under execution and the rest of the processes have to wait until the CPU is free and can be rescheduled.

The main objective of process scheduling is to keep the CPU busy at all times.

## Scheduling Queues

- All processes admitted to the system are stored in the **job queue.**
- Processes in main memory and ready to execute are placed in the **ready queue.**
- Processes waiting for a device to become available are placed in **device queues**. There is generally a separate device queue for each device.

These queues are generally stored as a linked list of PCBs. A queue header will contain two pointers - the **head pointer** pointing to the first PCB and the **tail pointer** pointing to the last PCB in the list. Each PCB has a pointer field that points to the next process in the queue.

When a process is allocated to the **CPU,** it executes for a while and eventually quits, interrupted, or waits for the completion of an I/O request. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk in the device queue.

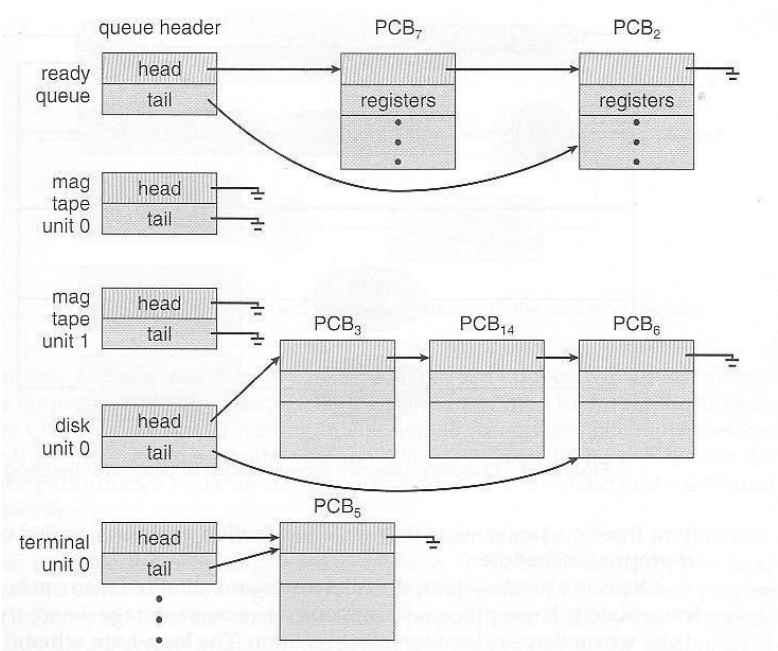



**Figure 3.6** The ready queue and various I/O device queues.

A common representation of process scheduling is a ***queueing diagram***. Each rectangular box in the diagram represents a queue. 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 in the ready queue until it is selected for execution and is given the CPU. 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 subprocess and wait for its termination.
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues.

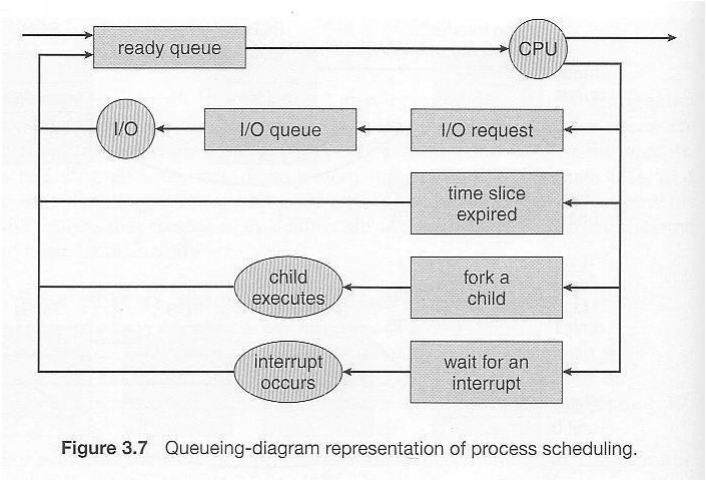



**Figure 3.7** Queueing-diagram representation of process scheduling.

## Schedulers

Schedulers are software which selects an available program to be assigned to CPU.

- A **long-term scheduler** or **Job scheduler** – selects jobs from the job pool (of secondary memory, disk) and loads them into the memory.

  If more processes are submitted, than that can be executed immediately, such processes will be in secondary memory. It runs infrequently, and can take time to select the next process.

- The **short-term scheduler**, or **CPU Scheduler** – selects job from memory and assigns the CPU to it. It must select the new process for CPU frequently.

- The **medium-term scheduler** - selects the process in ready queue and reintroduced into the memory.

Processes can be described as either:
**I/O-bound process** – spends more time doing I/O than computations,
**CPU-bound process** – spends more time doing computations and few I/O operations.

An efficient scheduling system will select a good mix of **CPU-bound** processes and **I/O bound** processes.

- If the scheduler selects **more I/O bound process**, then I/O queue will be full and ready queue will be empty.

- If the scheduler selects **more CPU bound process**, then ready queue will be full and I/O queue will be empty.

Time sharing systems employ a **medium-term scheduler**. It swaps out the process from ready queue and swap in the process to ready queue. When system loads get high, this scheduler will swap one or more processes out of the ready queue for a few seconds, in order to allow smaller faster jobs to finish up quickly and clear the system.

Advantages of medium-term scheduler –

- To remove process from memory and thus reduce the degree of multiprogramming (number of processes in memory).
- To make a proper mix of processes(CPU bound and I/O bound )

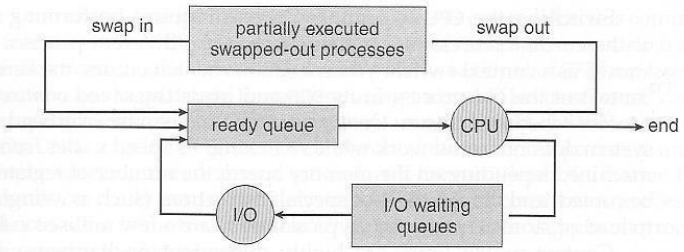



**Figure 3.8** Addition of medium-term scheduling to the queueing diagram.

## Context Switch

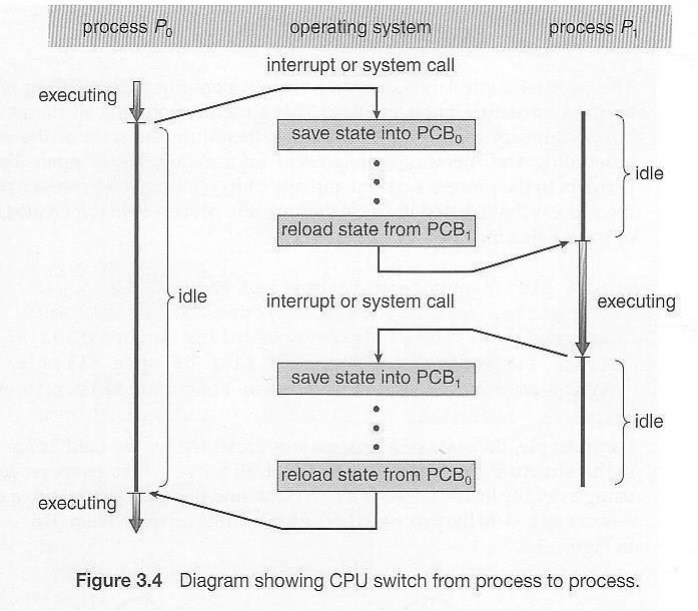



Figure 3.4 Diagram showing CPU switch from process to process.

The task of switching a CPU from one process to another process is called context switching. Context-switch times are highly dependent on hardware support (Number of CPU registers).

Whenever an interrupt occurs (hardware or software interrupt), the state of the currently running process is **saved** into the PCB and the state of another process is **restored** from the PCB to the CPU.

Context switch time is an overhead, as the system does not do useful work while switching.

## Operations on Processes

## Process Creation

A process may create several new processes. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes. Every process has a unique process ID.

On typical Solaris systems, the process at the top of the tree is the '**sched'** process with PID of 0. The **'sched'** process creates several children processes – **init**, **pageout** and **fsflush**. Pageout and fsflush are responsible for managing memory and file systems. The init process with a PID of 1, serves as a parent process for all user processes.

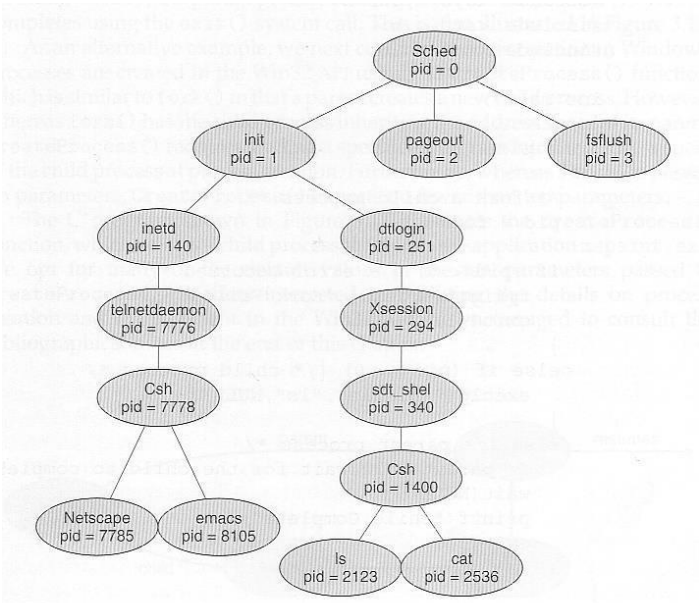



**Figure 3.9** A tree of processes on a typical Solaris system.

A process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, the subprocess may be able to obtain its resources in two ways :

- directly from the operating system
- Subprocess may take the resources of the parent process.

  The resource can be taken from parent in two ways –

  - The parent may have to partition its resources among its children
  - Share the resources among several children.

There are two options for the parent process after creating the child:

- Wait for the child process to terminate and then continue execution. The parent makes a wait( ) system call.
- Run concurrently with the child, continuing to execute without waiting.

Two possibilities for the address space of the child relative to the parent:

- The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behaviour of the **fork** system call in UNIX.
- The child process may have a new program loaded into its address space, with all new code and data segments. This is the behaviour of the **spawn** system calls in Windows.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) {/* error occurred */
      fprintf(stderr, "Fork Failed");
      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);
    }
}
```

**Figure 3.10** C program forking a separate process.

In UNIX OS, a child process can be created by **fork()** system call. The **fork** system call, if successful, returns the PID of the child process to its parents and returns a zero to the child process. If failure, it returns -1 to the parent. Process IDs of current process or its direct parent can be accessed using the getpid( ) and getppid( ) system calls respectively.

The parent waits for the child process to complete with the wait() system call. When the child process completes, the parent process resumes and completes its execution.

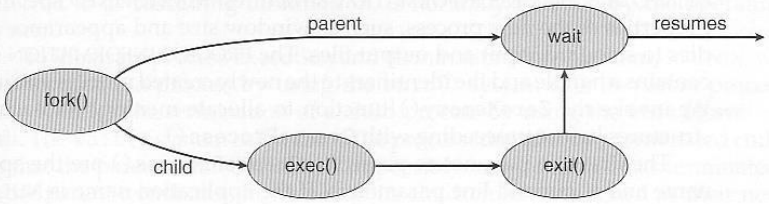



Figure 3.11 Process creation.

In windows the child process is created using the function **createprocess( )**. The createprocess( ) returns 1, if the child is created and returns 0, if the child is not created.

## Process Termination

A process terminates when it finishes executing its last statement and asks the operating system to delete it, by using the **exit**( ) system call. All of the resources assigned to the process like memory, open files, and I/O buffers, are deallocated by the operating system.

A process can cause the termination of another process by using appropriate system call. The parent process can terminate its child processes by knowing of the PID of the child.

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

- The child has exceeded its usage of the resources, it has been allocated.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system terminates all the children. This is called cascading termination.

**Note** : Processes which are trying to terminate but which cannot because their parent is not waiting for them are termed **zombies**. These are eventually inherited by init as orphans and killed off. (Modern UNIX shells do not produce as many orphans and zombies as older systems used to. )

## Interprocess Communication

Processes executing may be either co-operative or independent processes.

- **Independent Processes** – processes that cannot affect other processes or be affected by other processes executing in the system.
- **Cooperating Processes** – processes that can affect other processes or be affected by other processes executing in the system.

Co-operation among processes are allowed for following reasons –

- Information Sharing - There may be several processes which need to access the same file. So the information must be accessible at the same time to all users.
- Computation speedup - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks, which are solved simultaneously ( particularly when multiple processors are involved. )
- Modularity - A system can be divided into cooperating modules and executed by sending information among one another.
- Convenience - Even a single user can work on multiple task by informationsharing.

Cooperating processes require some type of inter-process communication. This is allowed by two models : 1) Shared Memory systems 2)Message Passing systems.

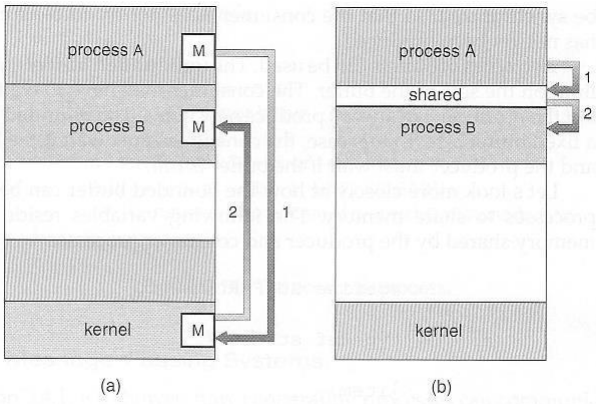
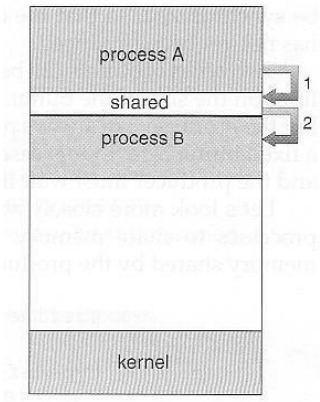



Figure 3.13 Communications models. (a) Message passing. (b) Shared memory.

| Sl. No. | **Shared Memory** | **Message passing** |
|---|---|---|
| 1. | A region of memory is shared by communicating processes, into which the information is written and read | Message exchange is done among the processes by using objects. |
| 2. | Useful for sending large block of data | Useful for sending small data. |

| | | |
|---|---|---|
| 3. | System call is used only to create shared memory | System call is used during every read and write operation. |
| 4. | Message is sent faster, as there are no system calls | Message is communicated slowly. |

- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small.




## Shared-Memory Systems

A region of shared-memory is created within the address space of a process, which needs to communicate. Other processes that needs to communicate uses this shared memory.

The form of data and position of creating shared memory area is decided by the process. Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

The process should take care that the two processes will not write the data to the shared memory at the same time.

## Producer-Consumer Example Using Shared Memory

This is a classic example, in which one process is producing data and another process is consuming the data.

The data is passed via an intermediary buffer (shared memory). The producer puts the data to the buffer and the consumer takes out the data from the buffer. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced.

There are two types of buffers into which information can be put –

- Unbounded buffer
- Bounded buffer

With Unbounded buffer, there is no limit on the size of the buffer, and so on the data produced by producer. But the consumer may have to wait for new items.

With bounded-buffer – As the buffer size is fixed. The producer has to wait if the buffer is full and the consumer has to wait if the buffer is empty.

This example uses shared memory as a circular queue. The **in** and **out** are two pointers to the array. Note in the code below that only the producer changes "in", and only the consumer changes "out".

- First the following data is set up in the shared memory area:

```
#define BUFFER_SIZE 10              //buffer size
Typedef struct {
    . . .
} item;
item buffer[ BUFFER_SIZE
]; int in = 0;
int out = 0;
```

- The producer process –
  Note that the buffer is full when [ (in+1)%BUFFER_SIZE == out ]

```
item nextProduced;
while( true )
{
        /* Produce an item and store it in nextProduced */
        nextProduced = makeNewItem( . . . );

        /* Wait for space to become available */
        while( ( ( in + 1 ) % BUFFER_SIZE ) == out ) //full
            ; /* Do nothing */

        /* And then, if not full store the item */
        buffer[ in ] = nextProduced;
        in = ( in + 1 ) % BUFFER_SIZE;
}
```

- The consumer process –
  Note that the buffer is empty when [ in == out ]

```
item nextConsumed;
while( true )
{
        /* Wait for an item to become available */
        while( in == out )      // buffer empty
            ; /* Do nothing */

        /* Get the next available item */
        nextConsumed = buffer[ out ];
        out = ( out + 1 ) % BUFFER_SIZE;
}
```

## Message-Passing Systems

A mechanism to allow process communication without sharing address space. It is used in distributed systems.

- Message passing systems uses system calls for "send message" and "receive message".
- A communication link must be established between the cooperating processes before messages can be sent.
- There are three methods of creating the link between the sender and the receiver-
  - Direct or indirect communication ( naming )
  - Synchronous or asynchronous communication (Synchronization)
  - Automatic or explicit buffering.

**a) Naming**

The processes that wants to communicate should have a way to refer eachother. ( using some identity)

**Direct communication** the sender and receiver must explicitly know eachothers name. The syntax for send() and receive() functions are as follows-

**send** (*P, message*) – send a message to process P

**receive**(*Q, message*) – receive a message from process Q

Properties of communication link :

- A link is established automatically between every pair of processes that wants to communicate. The processes need to know only each other's identity to communicate.
- A link is associated with exactly one pair of communicating processes
- Between each pair, there exists exactly one link.

Types of addressing in direct communication –

- Symmetric addressing – the above described communication is symmetric communication. Here both the sender and the receiver processes have to name each other to communicate.
- Asymmetric addressing – Here only the sender name is mentioned, but the receiving data can be from any system.

**send(P,** message) --- Send a message to process ***P***

**receive(id,** message). Receive a message from any process

Disadvantages of direct communication – any changes in the identifier of a process, may have to change the identifier in the whole system(sender and receiver), where the messages are sent and received.

**Indirect communication** uses shared mailboxes, or ports.

A mailbox or port is used to send and receive messages. Mailbox is an object into which messages can be sent and received. It has a unique ID. Using this identifier messages are sent and received.

Two processes can communicate only if they have a shared mailbox.

The send and receive functions are –

**send**(*A, message*) – send a message to mailbox A
**receive**(*A, message*) – receive a message from mailbox A

Properties of communication link:

- A link is established between a pair of processes only if they have a shared mailbox
- A link may be associated with more than two processes

- Between each pair of communicating processes, there may be any number of links, each link is associated with one mailbox.

A mail box can be owned by the operating system. It must take steps to –

- ↓ create a new mailbox
- ↓ send and receive messages from mailbox
- ↓ delete mailboxes.

**b) Synchronization**

The send and receive messages can be implemented as either **blocking** or **non-blocking**.

- **Blocking (synchronous) send -** sending process is blocked (waits) until the message is received by receiving process or the mailbox.
- **Non-blocking (asynchronous) send** - sends the message and continues (doesnot wait)

- **Blocking (synchronous) receive -** The receiving process is blocked until a message is available
- **Non-blocking (asynchronous) receive -** receives the message without block. The received message may be a valid message or null.

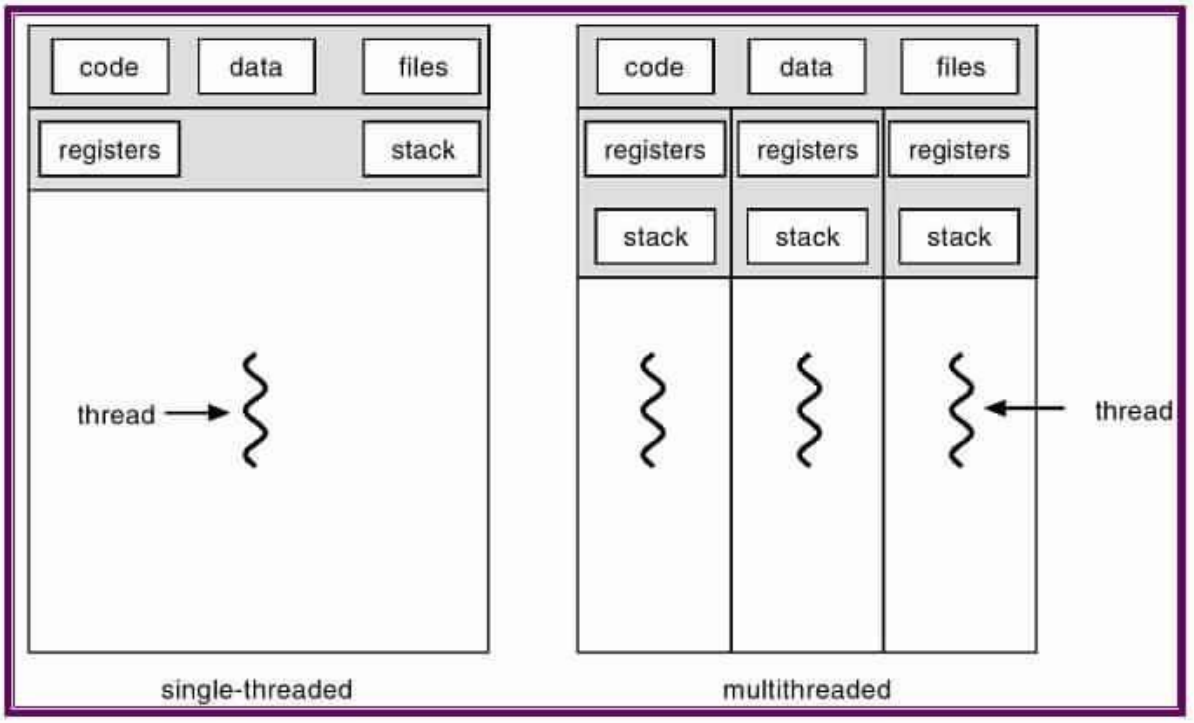
**c) Buffering**

when messages are passed, a temporary queue is created. Such queue can be of three capacities:

- **Zero capacity** – The buffer size is zero (buffer does not exist). Messages are not stored in the queue. The senders must block until receivers accept the messages.
- **Bounded capacity**- The queue is of fixed size(n). Senders must block if the queue is full. After sending 'n' bytes the sender is blocked.
- **Unbounded capacity** - The queue is of infinite capacity. The sender never blocks.

# Module 2 Multi threaded programming

- A ***thread*** is a basic unit of CPU utilization. It consists of a thread ID, program counter, a stack, and a set of registers.
- Traditional processes have a single thread of control. It is also called as **heavyweight process**. There is one program counter, and one sequence of instructions that can be carried out at any given time.
- A multi-threaded application have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files. Such process are called as **lightweight process**.




## Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to

allow the other tasks to proceed without blocking.

- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input ( keystrokes ), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- In a web server - Multiple threads allow for multiple requests to be served simultaneously. A thread is created to service each request; meanwhile another thread listens for more client request.
- In a web browser – one thread is used to display the images and another thread is used to retrieve data from the network.

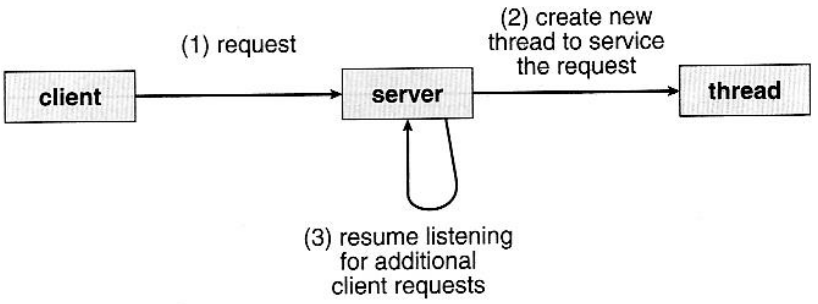



**Figure 4.2** Multithreaded server architecture.

## Benefits

The four major benefits of multi-threading are:

1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
   Multi threading allows 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** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. **Economy** - Creating and managing threads is much faster than performing the same tasks for processes. Context switching between threads takes less time.
4. **Scalability**, i.e. **Utilization of multiprocessor architectures** – Multithreading can be greatly utilized in a multiprocessor architecture. A single threaded process can make use of only one CPU, whereas the execution of a multi-threaded application may be split among the available processors. Multithreading on a multi-CPU machine increases concurrency. In a single processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time.

## Multicore Programming

- A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.
- A multi-threaded application running on a traditional single-core chip, would have to execute the threads one after another. On a multi-core chip, the threads could be spread across the available cores, allowing true parallel processing.

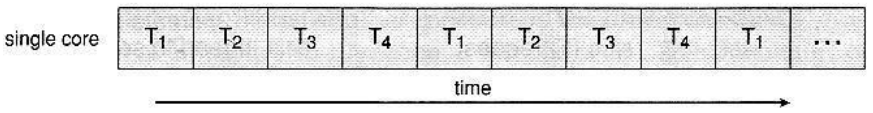



**Figure 4.3** Concurrent execution on a single-core system.

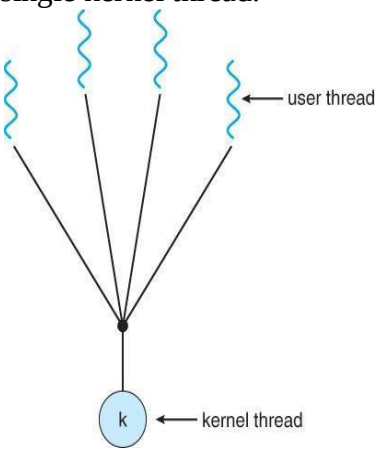
- For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.
- For application programmers, there are five areas where multi-core chips present new challenges:
    1. Dividing activities - Examining applications to find activities that can be performed concurrently.
    2. Balance - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.
    3. Data splitting - To prevent the threads from interfering with one another.
    4. Data dependency - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.
    5. Testing and debugging - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

## Multithreading Models

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- User threads are the threads that application programmers would put into their programs. They are supported above the kernel, without kernel support.
- Kernel threads are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple tasks simultaneously.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following models.

### a) Many-To-One Model

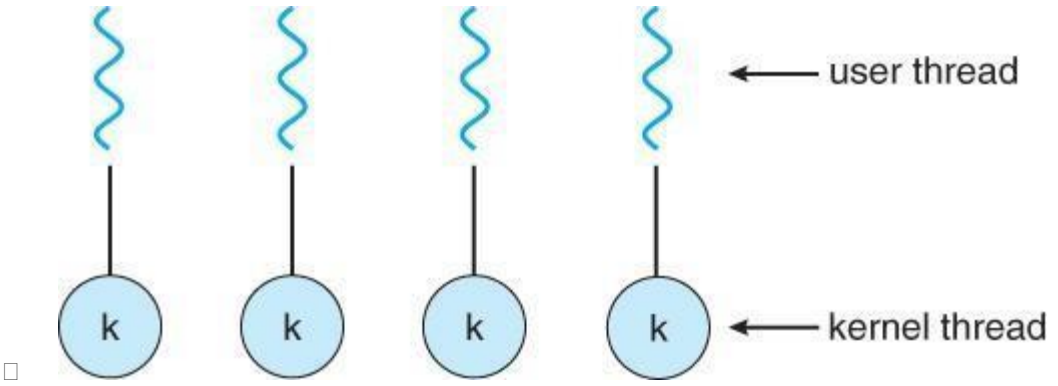
In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.




- Thread management is handled by the thread library in user space, which is very efficient.
- If a blocking system call is made by one of the threads, then the entire process blocks. Thus blocking the other user threads from continuing the execution.
- Only one user thread can access the kernel at a time, as there is only one kernel thread. Thus the threads are unable to run in parallel on multiprocessors.
- Green threads of Solaris and GNU Portable Threads implement the many-to-one model.

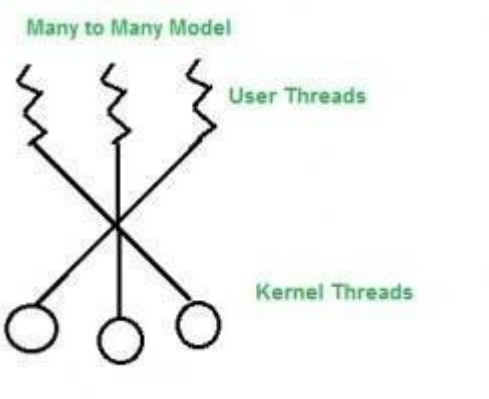
### b) One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- This model places a limit on the number of threads created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.


user thread
k
k
k
k
kernel thread

**Many-To-Many Model**

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.




- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.
- This model is also called as two-tier model.
- It is supported by operating system such as IRIX, HP-UX, and Tru64 UNIX.

## Threading Issues

### a) The fork( ) and exec( ) System Calls

The fork( ) system call is used to create a separate, duplicate process.
When a thread program calls fork( ),

- The new process can be a copy of the parent, with all the threads
- The new process is a copy of the single thread only (that invoked the process)

If the thread invokes the exec( ) system call, the program specified in the parameter to exec( ) will be executed by the thread created.

### b) Cancellation

Terminating the thread before it has completed its task is called thread cancellation. The thread to be cancelled is called **target thread**.

Example : Multiple threads required in loading a webpage is suddenly cancelled, if the browser window is closed.

Threads that are no longer needed may be cancelled in one of two ways:

1. **Asynchronous Cancellation** - cancels the thread immediately.
2. **Deferred Cancellation** – the target thread periodically check whether it has to terminate, thus gives an opportunity to the thread, to terminate itself in an orderly fashion.

   In this method, the operating system will reclaim all the resources before cancellation.

**c) Signal Handling**

A signal is used to notify a process that a particular event has occurred.

All signals follow same path-

1) A signal is generated by the occurrence of a particular event.
2) A generated signal is delivered to a process.
3) Once delivered, the signal must be handled.

A signal can be invoked in 2 ways : synchronous or asynchronous.

**Synchronous signal** – signal delivered to the same program. Eg – illegal memory access, divide by zero error.

**Asynchronous signal** – signal is sent to another program. Eg – Ctrl C

In a single-threaded program, the signal is sent to the same thread. But, in multi-threaded environment, the signal is delivered in variety of ways, depending on the type of signal –

- Deliver the signal to the thread, to which the signal applies.
- Deliver the signal to every threads in the process.
- Deliver the signal to certain threads in the process.
- Deliver the signal to specific thread, which receive all the signals.

A signal can be handled by one of the two ways –

Default signal handler - signal is handled by OS.

User-defined signal handler - User overwrites the OS handler.

**d) Thread Pools**

In multithreading process, thread is created for every service. Eg – In web server, thread is created to service every client request.

Creating new threads every time, when thread is needed and then deleting it when it is done can be inefficient, as –

Time is consumed in creation of the thread.

A limit has to be placed on the number of active threads in the system. Unlimited thread creation may exhaust system resources.

An alternative solution is to create a number of threads when the process first starts, and put those threads into a ***thread pool***.

- Threads are allocated from the pool when a request comes, and returned to the pool when no longer needed(after the completion of request).
- When no threads are available in the pool, the process may have to wait until one becomes available.

Benefits of Thread pool –

- Thread creation time is not taken. The service is done by the thread existing in the pool. Servicing a request with an existing thread is faster than waiting to create a thread.
- The thread pool limits the number of threads in the system. This is important on systems that cannot support a large number of concurrent threads.

The ( maximum ) number of threads available in a thread pool may be determined by parameters like the number of CPUs in the system, the amount of memory and the expected number of client request.

**e) Thread-Specific Data**

- Data of a thread, which is not shared with other threads is called thread specific data.
- Most major thread libraries ( pThreads, Win32, Java ) provide support for thread-specific data.

Example – if threads are used for transactions and each transaction has an ID. This unique ID is a specific data of the thread.

**f) Scheduler Activations**

Scheduler Activation is the technique **used** for communication between the user-thread library and the kernel.
It works as follows:

- ↓ the kernel must inform an application about certain events. This procedure is known as an **upcall.**
- ↓ Upcalls are handled by the thread library with an **upcall handler,** and upcall handlers must run on a virtual processor.

Example - The kernel triggers an upcall occurs when an application thread is about to block. The kernel makes an upcall to the thread library informing that a thread is about to block and also informs the specific ID of the thread.

The upcall handler handles this thread, by saving the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.

The upcall handler then schedules another thread that is eligible to run on the virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. Thus assigns the thread to the available virtual processor.

**Thread Libraries**

- Thread libraries provide an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space.
- There are three main thread libraries in use –
    1. POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
    2. Win32 threads - provided as a kernel-level library on Windows systems.
    3. Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.
- The following sections will demonstrate the use of threads in all three systems for calculating the sum of integers from 0 to N in a separate thread, and storing the result in a variable "sum".

**4.3.1 Pthreads**

- The POSIX standard ( IEEE 1003.1c ) defines the *specification* for pThreads, not the *implementation*.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function, in this example the runner( ) function.
- Pthread_create() function is used to create a thread.

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

**Figure 4.6** Multithreaded C program using the Pthreads API.

Figure 4.9

### 4.3.2 Win32 Threads

- Similar to pThreads. Examine the code example to see the differences, which are mostly syntactic & nomenclature.
- Here summation() function is used to perform the separate thread function.
- CreateThread() is the function to create a thread.

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

   // create the thread
   ThreadHandle = CreateThread(
      NULL, // default security attributes
      0, // default stack size
      Summation, // thread function
      &Param, // parameter to thread function
      0, // default creation flags
      &ThreadId); // returns the thread identifier

   if (ThreadHandle != NULL) {
      // now wait for the thread to finish
      WaitForSingleObject(ThreadHandle,INFINITE);

      // close the thread handle
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

**Figure 4.7** Multithreaded C program using the Win32 API.

### **4.3.3** Java Threads

- ALL Java programs use Threads.

```
class Sum
{
  private int sum;

  public int getSum() {
    return sum;
  }

  public void setSum(int sum) {
    this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }

  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
      sum += i;
    sumValue.setSum(sum);
  }
}

public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else {
        // create the object to be shared
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
          thrd.join();
          System.out.println
            ("The sum of "+upper+" is "+sumObject.getSum());
        } catch (InterruptedException ie) { }
      }
    }
    else
      System.err.println("Usage: Summation <integer value>"); }
}
```

**Figure 4.8** Java program for the summation of a non-negative integer.

- The creation of new Threads requires to implement the Runnable Interface, which contains a built-in method "public void run( )" . The Thread class will have to overwrite the built-in function run( ), in which the thread code should be written.
- Creating a Thread Object does not start the thread running - To start the thread, the built-in start( ) method should be invoked, which in turn call the run() method(where statements to be executed by thread are written).

# CPU SCHEDULING

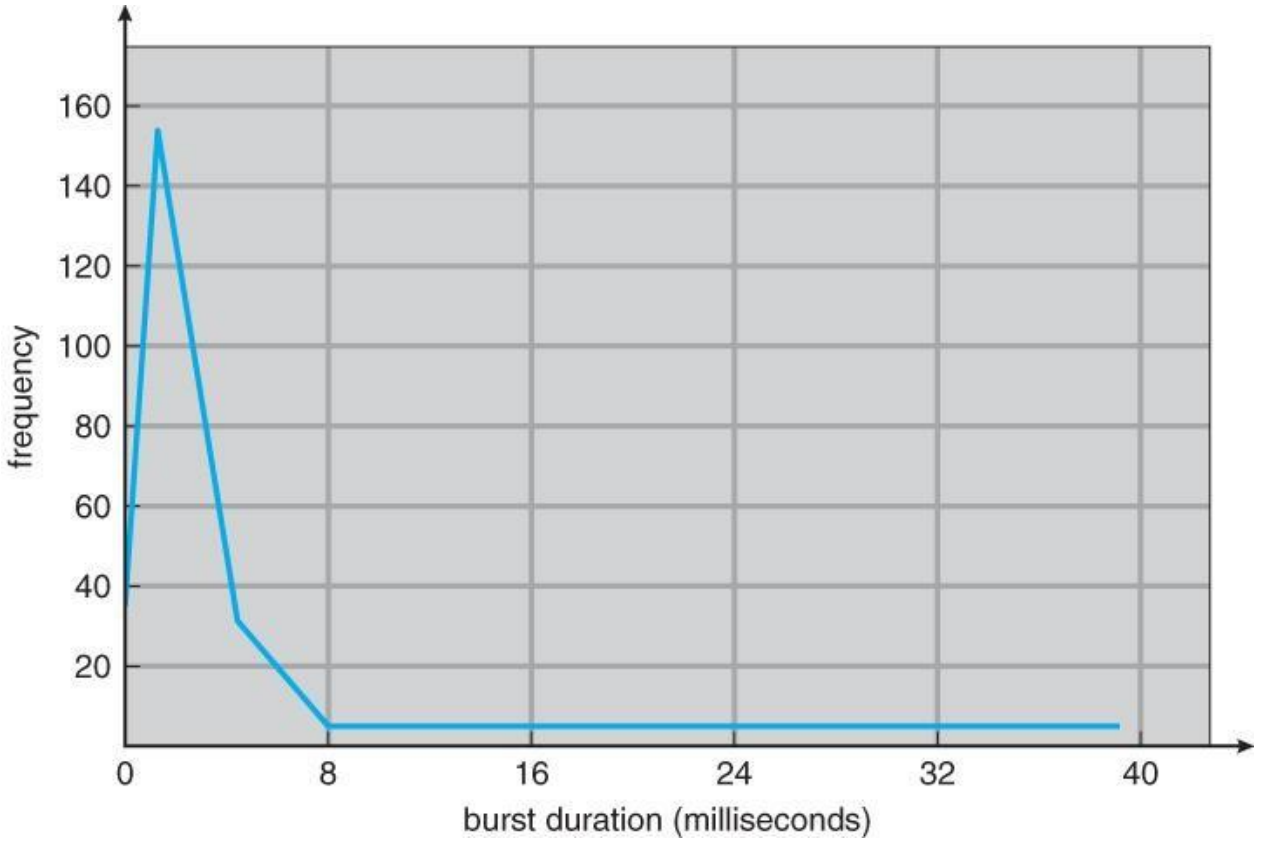
## 3.1 BASIC CONCEPTS

In a single-processor system, only one process can run at a time; other processes must wait until the CPU is free. The objective of multiprogramming is to have some process running at all times in processor, to maximize CPU utilization.

In multiprogramming, several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU. Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is one of the primary computer resources. Thus, its scheduling is central to operating-system design.

### CPU-I/O Burst Cycle

Process execution consists of a cycle of CPU execution and I/O wait. The state of process under execution is called **CPU burst** and the state of process under I/O request & its handling is called **I/O burst**. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution as shown in the following figure:




**CPU Scheduler**

Whenever the CPU becomes idle, the operating system must select one of the processes from the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

A ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. All the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

**Non - Preemptive Scheduling –** once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.

**Preemptive Scheduling –** The process under execution, may be released from the CPU, in the middle of execution due to some inconsistent state of the process.

**Dispatcher**

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

**SCHEDULING CRITERIA**

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU utilization** - The CPU must be kept as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 to 90 percent .
- **Throughput** - If the CPU is busy executing processes, then work is done fast. One measure of work is the number of processes that are completed per time unit, called throughput.
- **Turnaround time -** From the point of view of a particular process, the important criterion is how long it takes to execute that process. 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.

  Time spent waiting (to get into memory + ready queue + execution + I/O)
- **Waiting time -** The total amount of time the process spends waiting in the ready queue.
- **Response time -** The time taken from the submission of a request until the first response is produced is called the response time. It is the time taken to start responding. In interactive system, response time is given criterion.

It is desirable to **maximize** CPU utilization and throughput and to **minimize** turnaround time, waiting time, and response time.

**SCHEDULING ALGORITHMS**

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

### 1. First-Come, First-Served Scheduling

Other names of this algorithm are:

- First-In-First-Out (FIFO)
- Run-to-Completion
- Run-Until-Done

First-Come-First-Served algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. This algorithm is always nonpreemptive, once a process is assigned to CPU, it runs to completion.

Advantages :

- more predictable than other schemes since it offers time
- code for FCFS scheduling is simple to write and understand

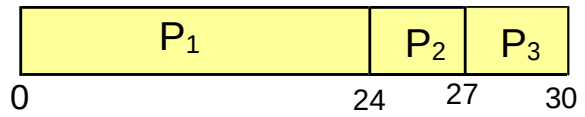
Disadvantages:

- Short jobs(process) may have to wait for long time
- Important jobs (with higher priority) have to wait
- cannot guarantee good response time
- average waiting time and turn around time is often quite long
- lower CPU and device utilization.

Example:-

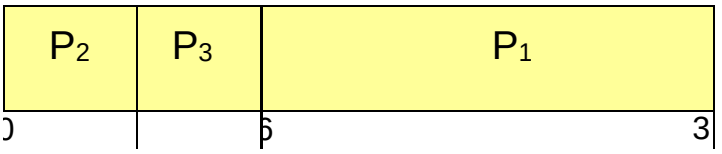
| Process | Burst Time |
|---|---|
| *P1* | 24 |
| *P2* | 3 |
| *P3* | 3 |

Suppose that the processes arrive in the order: *P1*, *P2* , *P3*
The Gantt Chart for the schedule is:




Waiting time for *P1* = 0; *P2* = 24; *P3* = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Suppose that the processes arrive in the order *P2* , *P3* , *P1*
The Gantt chart for the schedule is:


P2
P3
P1
0
6
3

Waiting time for *P1* = 6*; P2* = 0*; P3* = 3
Average waiting time: (6 + 0 + 3)/3 = 3
Much better than previous case

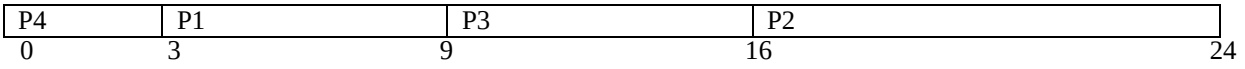
Here, there is a ***Convoy effect***, as all the short processes wait for the completion of one big process. Resulting in lower CPU and device utilization.

### 3.3.2 Shortest-Job-First Scheduling

This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

| Process | Burst Time |
|---|---|
| pl | 6 |
| p2 | 8 |
| p3 | 7 |
| p4 | 3 |



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

The waiting time is 3 milliseconds for process $P_1$, 16 milliseconds for process $P_2$, 9 milliseconds for process $P_3$, and 0 milliseconds for process $P_4$. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.

The SJF scheduling algorithm is provably optimal, in that it gives the minimum average waiting time for a given set of processes. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst. The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts. Let $t_n$ be the length of the nth CPU burst, and let $t_{n+1}$ be our predicted value for the next CPU burst. Then, for $\alpha$, $0 \leq \alpha \leq 1$, define

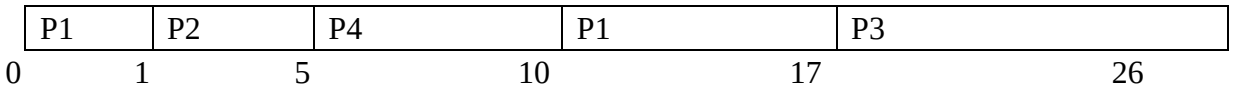
$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remaining-time-first scheduling.

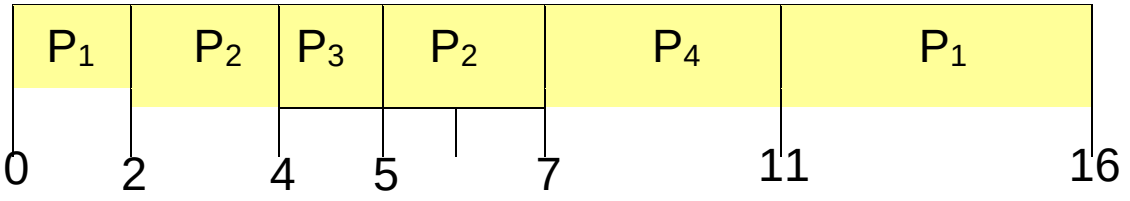
As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

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

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chatt




->SJF (preemptive)




->Average waiting time = (9 + 1 + 0 +2)/4 = 3

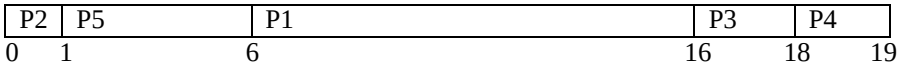
### 3.3.3 Priority Scheduling

The SJF algorithm is a special case of the general priority scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

As an example, consider the following set of processes, assumed to have arrived at time 0, in the order $P_1$, $P_2$, … , $P_5$, with the length of the CPU burst given in milliseconds:

| Process | Burst Time | priority |
|---|---|---|
| pl | 10 | 0 |
| p2 | 1 | 1 |
| p3 | 2 | 4 |
| p4 | 1 | 5 |
| P5 | 5 | 2 |

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




The average waiting time is 8.2 milliseconds.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low- priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

A solution to the problem of indefinite blockage of low-priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

### 3.3.4 Round-Robin Scheduling

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is

treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

If we use a time quantum of 4 milliseconds, then process $P_1$ gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process $P_2$. Since process $P_2$ does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process $P_3$. Once each process has received 1 time quantum, the CPU is returned to process $P_1$ for an additional time quantum. The resulting RR schedule is

The average waiting time is 17/3 = 5.66 milliseconds.

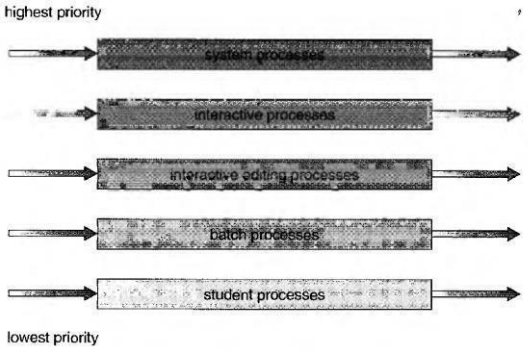
In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row . If a process's CPU burst exceeds 1 time quantum, that process is preempted arid is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

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. Each process must wait no longer than (n-1) x q time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

### 3.3.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues (Figure). The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.




In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. For example, the foreground queue may have absolute priority over the background queue.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

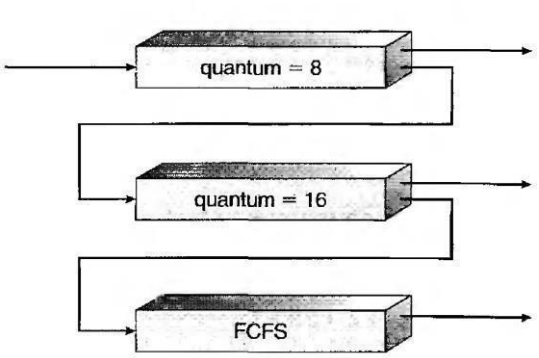
Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground-background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

### 3.3.6 Multilevel Feedback-Queue Scheduling

Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system. The multilevel feedback-queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 (Figure). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.




A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst. Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes. Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

In general, a multilevel feedback-queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher-priority queue
- The method used to determine when to demote a process to a lower-priority queue
- The method used to determine which queue a process will enter when that process needs service

## 3.4 MULTIPLE-PROCESSOR SCHEDULING

### 3.4.1 Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code. This asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.

A second approach uses symmetric multiprocessing (SMP), where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes. Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute.

### 3.4.2 Processor Affinity

Consider what happens to cache memory when a process has been running on a specific processor: The data most recently accessed by the process populates the cache for the

processor; and as a result, successive memory accesses by the process are often satisfied in cache memory. Now, if the process migrates to another processor, the contents of cache memory must be invalidated for the processor being migrated from, and the cache for the processor being migrated to must be re-populated. Because of the high cost of invalidating and re-populating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**, meaning that a process has an affinity for the processor on which it is currently running.

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so— we have a situation known as soft affinity. Here, it is possible for a process to migrate between processors. Some systems —such as Linux—also provide system calls that support hard affinity, thereby allowing a process to specify that it is not to migrate to other processors.

### 3.4.3 Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads along with lists of processes awaiting the CPU. Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system.

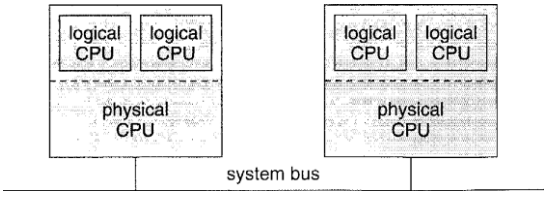
Load balancing is typically only necessary on systems where each processor has its own private queue of eligible processes to execute. On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue.

There are two general approaches to load balancing: push migration and pull migration. With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors. Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.

### 3.4.4 Symmetric Multithreading

SMP systems allow several threads to run concurrently by providing multiple physical processors. An alternative strategy is to provide multiple logical—rather than physical—processors. Such a strategy is known as symmetric multithreading (or SMT).

The idea behind SMT is to create multiple logical processors on the same physical processor, presenting a view of several logical processors to the operating system, even on a system with only a single physical processor. Each logical processor has its own architecture state, which includes general-purpose and machine-state registers. Furthermore, each logical processor is responsible for its own interrupt handling, meaning that interrupts are delivered to—and handled by—logical processors rather than physical ones. Otherwise, each logical processor shares the resources of its physical processor, such as cache memory and buses. The following figure illustrates a typical SMT architecture with two physical processors, each housing two logical processors. From the operating system's perspective, four processors are available for work on this system.



## 3.5 THREAD SCHEDULING

On operating systems that support user-level and kernel-level threads, it is kernel-level threads—not processes—that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).

### 3.5.1 Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP, a scheme known as process-contention scope (PCS), since competition for the CPU takes place among threads belonging to the same process. To decide which kernel thread to schedule onto a CPU, the kernel uses system-contention scope (SCS). Competition for the CPU with SCS scheduling takes place among all threads in the system.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer. PCS will typically preempt the thread currently running in favor of a higher-priority thread.

### 3.5.2 Pthread Scheduling

Pthreads identifies the following contention scope values:

- PTHREAD_SCOPE_PROCESS schedules threads using PCSscheduling.
- PTHREAD_SCOPE_SYSTEM schedules threads using SCSscheduling.

On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs. The number of LWPs is maintained by the thread library, perhaps using scheduler activations. The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

The Pthread IPC provides the following two functions for getting—and setting—the contention scope policy;

- pthread_attr_setscope (pthread_attr_t *attr, int scope)
- pthread_attr_getscope (pthread_attr_t *attr, int *scope)

The first parameter for both functions contains a pointer to the attribute set for the thread. The second parameter for the pthread_attr_setscope () function is passed either the THREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set. In the case of pthread_attr_getscope(), this second parameter contains a pointer to an int value that is set to the current value of the contention scope. If an error occurs, each of these functions returns non-zero values.

# Process Synchronization

## 5.1 Background

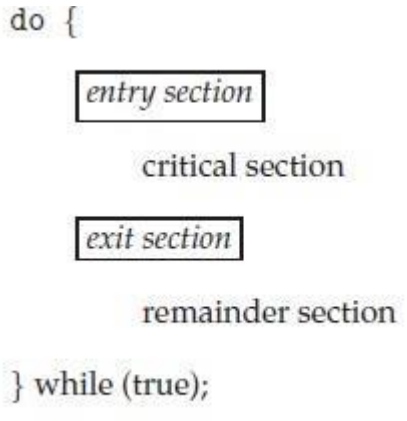
Since processes frequently needs to communicate with other processes therefore, there is a need for a well- structured communication, without using interrupts, among processes.

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**

To guard against the race condition ensure only one process at a time can be manipulating the variable or data. To make such a guarantee processes need to be synchronized in some way

## 5.2 The Critical-Section Problem

Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on. when one process is executing in its critical section, no other process is allowed to execute in its critical section. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. The general structure of a typical process Pi is shown in Figure




General structure of a typical process $P_i$.

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

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

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

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

Two general approaches are used to handle critical sections in operating systems:

- **Preemptive kernels:** A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- **Nonpreemptive kernels**.. A nonpreemptive 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.

### 5.3 Peterson's Solution

A classic software-based solution to the critical-section problem known as **Peterson's solution**. It addresses the requirements of mutual exclusion, progress, and bounded waiting.

- It Is two process solution.
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:

  int turn;

  Boolean flag[2]

  The variable turn indicates whose turn it is to enter the critical section. The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready.
- The structure of process *Pi* in Peterson's solution:

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

- It proves that
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

## 5.4 Synchronization Hardware

Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Simple hardware instructions can be used effectively in solving the critical-section problem. These solutions are based on the **locking** —that is, protecting critical regions through the use of locks.

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

Solution to Critical Section problem using locks

- Modern machines provide special atomic hardware instructions

  Atomic = non-interruptable
- Either test memory word and set value(TestAndSet( )) Or swap contents of two memory words(Swap( )).
- The definition of the test and set() instruction

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

- Using test and set() instruction, mutual exclusion can be implemented by declaring a boolean variable lock, initialized to false. The structure of process *Pi* is shown in Figure:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Mutual-exclusion implementation with test and set().

- Using Swap() instruction, mutual exclusion can be provided as : A global Boolean variable lock is declared and is initialized to *false* and each process has a local Boolean variable *key*.

```
void Swap (boolean *a, boolean *b)
{
      boolean temp = *a;
      *a = *b;
      *b = temp:
}
```

The definition of the Swap() instruction

```
do {
      key = TRUE;
      while ( key == TRUE)
            Swap (&lock, &key );

                  //    critical section

      lock = FALSE;

                  //      remainder section

} while (TRUE);
```

Mutual exclusion implementation with Swap() instruction

- Test and Set() instruction & Swap() Instruction do not satisfy the bounded-waitingrequirement.

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

. Bounded-waiting mutual exclusion with test and set()

### 5.5 Semaphores

The hardware-based solutions to the critical-section problem are complicated as well as generally inaccessible to application programmers. So operating-systems designers build software tools to solve the critical-section problem, and this synchronization tool called as Semaphore.

- Semaphore *S is an* integer variable
- Two standard operations modify S: wait() and signal()
  Originally called P() and V()
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
    while S <= 0
            ; // no-op
     S--;
}
signal (S) {
   S++;
}
```

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time.

**Usage:**

Semaphore classified into:

- Counting semaphore: Value can range over an unrestricted domain.

- Binary semaphore(Mutex locks): Value can range only between from 0 & 1. It provides mutual exclusion.

```
Semaphore mutex;    // initialized to 1
do {
    wait (mutex);
       // Critical Section
    signal (mutex);
         // remainder section
} while (TRUE);
```

- Consider 2 concurrently running processes:

  *S*1;

  signal(synch);

  In process *P*1, and the statements

  wait(synch);

  *S*2;

  Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal(synch), which is after statement *S*1 has been executed.

**Implementation:**

The disadvantage of the semaphore is **busy waiting** i.e While a process is in critical section, any other process that tries to enter its critical section must loop continuously in the entry code. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spin lock** because the process spins while waiting for the lock.

**Solution for Busy Waiting problem:**

Modify the definition of the wait() and signal()operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which

changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

To implement semaphores under this definition, define a semaphore as follows:

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

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

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

and the signal() semaphore operation can be defined as

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

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P.

### 5.6.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes, these processes are said to be deadlocked.

Consider below example: a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and 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 *P*0 executes wait(S) and then *P*1 executes wait(Q).When *P*0 executes wait(Q), it must wait until *P*1 executes signal(Q). Similarly, when *P*1 executes wait(S), it must wait until *P*0 executes signal(S). Since these signal() operations cannot be executed, *P*0 and *P*1 are deadlocked.

Another problem related to deadlocks is **indefinite blocking** or **starvation**.

## 5.7 Classic Problems of Synchronization

### 5.7.1 The Bounded-Buffer Problem:

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

Code for producer is given below:

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

Code for consumer is given below:

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

### 5.7.2 The Readers–Writers Problem

- A data set is shared among a number of concurrent processes
  - ✓ Readers – only read the data set; they do **not** perform any updates
  - ✓ Writers– can both read and write
- Problem – allow multiple readers to read at the same time
  - ✓ Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities.
  - ✓ *First* variation – no reader kept waiting unless writer has permission to use shared object
  - ✓ *Second* variation- Once writer is ready, it performs asap.
- Shared Data
  - ✓ Data set
  - ✓ Semaphore mutex initialized to 1
  - ✓ Semaphore wrt initialized to 1
  - ✓ Integer readcount initialized to 0

The structure of writer process:

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

The structure of reader process:

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

### 5.7.3 The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.


A philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

**Solution:** One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She

releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1. The structure of philosopher *i* is shown in Figure

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

Several possible remedies to the deadlock problem are replaced by:

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

### 5.8 Monitors

**Incorrect use of semaphore operations:**

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

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

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

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

In this case, a deadlock will occur.

- Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

**Solution:**

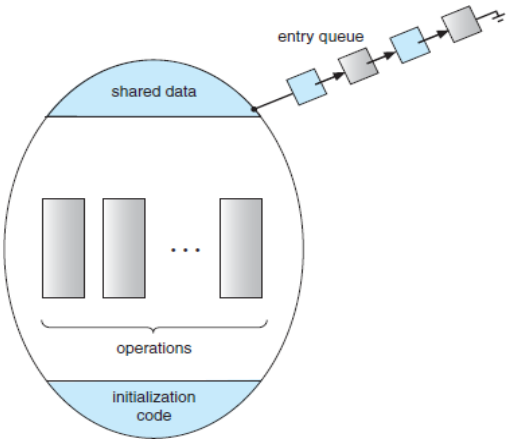
**Monitor:** An **abstract data type**—or **ADT**—encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT.

A ***monitor type*** is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The monitor construct ensures that only one process at a time is active within the monitor.

The syntax of a monitor type is shown in Figure:

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Schematic view of a monitor:




To have a powerful Synchronization schemes a *condition* construct is added to the Monitor. So synchronization scheme can be defined with one or more variables of type *condition.*

```
condition x, y;
```

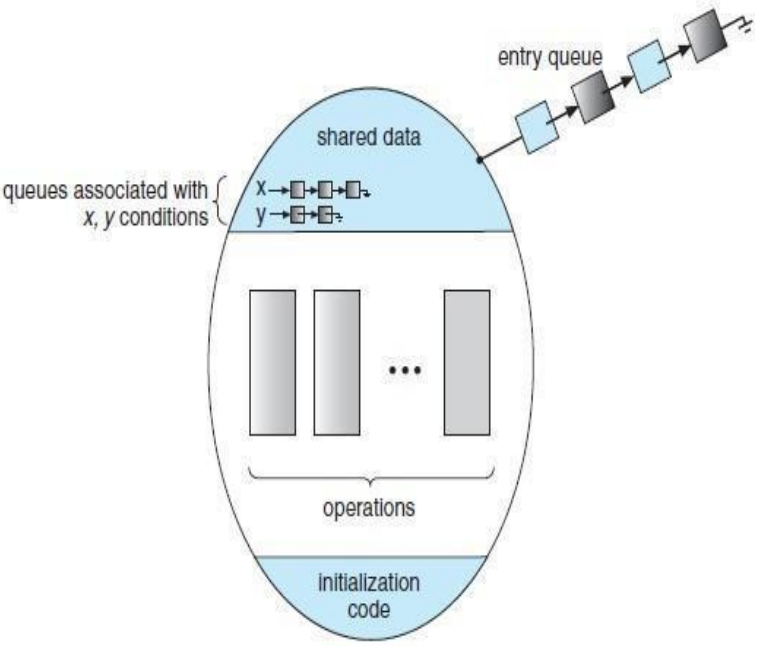
The only operations that can be invoked on a condition variable are wait() and signal(). The operation

```
x.wait();
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal();
```

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed. Contrast this operation with the signal() operation associated with semaphores, which always affects the state of the semaphore.




### 5.8.2 Dining-Philosophers Solution Using Monitors

A deadlock-free solution to the dining-philosophers problem using monitor concepts. This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

Consider following data structure:

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

Philosopher *i* can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).

And also declare:

Condition self[5];

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

A monitor solution to the dining-philosopher problem:

```
monitor DiningPhilosophers
  {
    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;
      }
}
```

### 5.8.3 Implementing a Monitor Using Semaphores

For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus, each external function F is replaced by

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

Mutual exclusion within a monitor is ensured.

For each condition x, we introduce a semaphore x sem and an integer variable x count, both initialized to 0. The operation x.wait() can now be implemented as

```
x_count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x_count--;
```

The operation x.signal() can be implemented as

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

### 5.8.4 Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then to determine which of the suspended processes should be resumed next, one simple solution is to use a first-come, first-served (FCFS) ordering, so that the process that has been waiting the longest is resumed first. For this purpose, the **conditional-wait** construct can be used. This construct has the form

```
x.wait(c);
```

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

The ResourceAllocator monitor shown in the above Figure, which controls the allocation of a single resource among competing processes.

A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
...
access the resource;
...
R.release();
```

where R is an instance of type ResourceAllocator.

The monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

# Module 3 DEAD LOCKS

## 3.3.1. System Model

● For the purposes of deadlock discussion, a system can be modeled as a collection of limited resources, which can be partitioned into different categories, to be allocated to a number of processes, each having different needs.

● Resource categories may include memory, printers, CPUs, open files, tape drives, CD-ROMS, etc.

● By definition, all the resources within a category are equivalent, and a request of this category can

be equally satisfied by any one of the resources in that category. If this is not the case ( i.e. if there is some difference between the resources within a category ), then that category needs to be further divided into separate categories. For example, "printers" may need to be separated into "laser printers" and "color inkjet printers".

● Some categories may have a single resource.

● In normal operation a process must request a resource before using it, and release it when it is done, in the following sequence:

1. Request - If the request cannot be immediately granted, then the process must wait until the resource(s) it needs become available. Example: system calls open( ), malloc( ), new( ), and request( ).

2. Use - The process uses the resource.

Example: prints to the printer or reads from the file.

3. Release - The process relinquishes the resource. so that it becomes available for other processes.

Example: close( ), free( ), delete( ), and release( ).

● For all kernel-managed resources, the kernel keeps track of what resources are free and which are

allocated, to which process they are allocated, and a queue of processes waiting for this resource to become available. Application-managed resources can be controlled using mutexes or wait( ) and signal( ) calls, ( i.e. binary or counting semaphores. )

● A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set ( and which can only be released when that other waiting process makes progress. )

3.3.2. Deadlock Characterization

Necessary Conditions:

There are four conditions that are necessary to achieve deadlock:

Mutual Exclusion - At least one resource must be held in a non-sharable mode; If any other process

requests this resource, then that process must wait for the resource to be released.

Hold and Wait - A process must be simultaneously holding at least one resource and waiting for at least

one resource that is currently being held by some other process.

No preemption - Once a process is holding a resource ( i.e. once its request has been granted ), then that

resource cannot be taken away from that process until the process voluntarily releases it.

Circular Wait - A set of processes { P0, P1, P2, . . ., PN } must exist such that every P[ i ] is waiting for

P[ ( i + 1 ) % ( N + 1 ) ]. ( Note that this condition implies the hold-and-wait condition, but it is easier to

deal with the conditions if the four are considered separately. )

Resource-Allocation Graph

In some cases deadlocks can be understood more clearly through the use of Resource-Allocation Graphs, having the following properties:

*A set of resource categories, { R1, R2, R3, . . ., RN }, which appear as square nodes on the

graph. Dots inside the resource nodes indicate specific instances of the resource. ( E.g. two dots might

represent two laser printers. )

*A set of processes, { P1, P2, P3, . . ., PN }

*Request Edges - A set of directed arcs from Pi to Rj, indicating that process Pi has requested Rj,

and is currently waiting for that resource to become available.

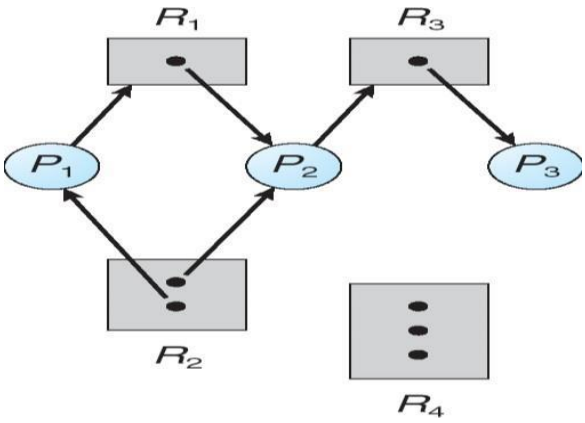
*Assignment Edges - A set of directed arcs from Rj to Pi indicating that resource Rj has been

allocated to process Pi, and that Pi is currently holding resource Rj.

Note that a request edge can be converted into an assignment edge by reversing the direction of the arc

when the request is granted. ( However note also that request edges point to the category box, whereas

assignment edges emanate from a particular instance dot within the box. )




*If a resource-allocation graph contains no cycles, then the system is not deadlocked. ( When looking for

cycles, remember that these are directed graphs. ) See the example in Figure 7.2 above.

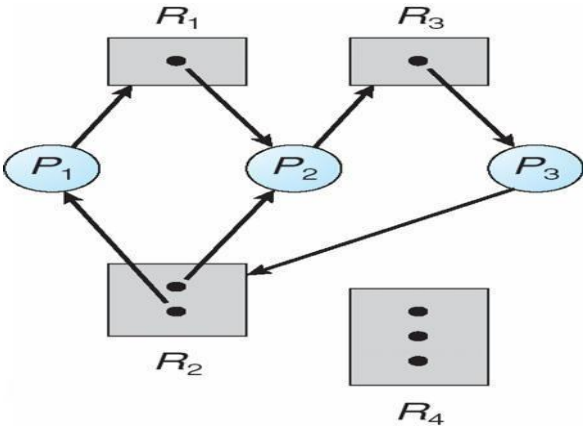
*If a resource-allocation graph does contain cycles AND each resource category contains only a single

instance, then a deadlock exists.

*If a resource category contains more than one instance, then the presence of a cycle in the resourceallocation

graph indicates the possibility of a deadlock, but does not guarantee one. Consider, for

example, Figures 7.3 and 7.4 below:




3.3.3. Methods for Handling Deadlocks

Generally speaking there are three ways of handling deadlocks:

Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.

Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are

detected.

Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply

let them happen and reboot as necessary than to incur the constant overhead and system performance

penalties associated with deadlock prevention or detection. This is the approach that both Windows and

UNIX take.

In order to avoid deadlocks, the system must have additional information about all processes. In

particular, the system must know what resources a process will or may request in the future. ( Ranging

from a simple worst-case maximum to a complete resource request and release plan for each process,

depending on the particular algorithm. )

Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or preempting resources, neither of which is an attractive alternative.

If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

3.3.4. Deadlock Prevention

Deadlocks can be prevented by preventing at least one of the four required conditions:

Mutual Exclusion

Shared resources such as read-only files do not lead to deadlocks.

Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

Hold and Wait

To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:

i. Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.

ii. Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.

iii. Either of the methods described above can lead to starvation if a process requires one or more

popular resources.

No Preemption

Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.

i. One approach is that if a process is forced to wait when requesting a new resource, then all other

resources previously held by this process are implicitly released, ( preempted ), forcing this process to reacquire

the old resources along with the new resources in a single request, similar to the previous

discussion.

ii. Another approach is that when a resource is requested and not available, then the system looks to see

what other processes currently have those resources and are themselves blocked waiting for some other

resource. If such a process is found, then some of their resources may get preempted and added to the list

of resources for which the process is waiting.

iii. Either of these approaches may be applicable for resources whose states are easily saved and restored,

such as registers and memory, but are generally not applicable to other devices such as printers and tape

drives.

Circular Wait

i. One way to avoid circular wait is to number all resources, and to require that processes request

resources only in strictly increasing ( or decreasing ) order.

ii. In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.

iii. One big challenge in this scheme is determining the relative ordering of the different resources

Deadlock Avoidance

The general idea behind deadlock avoidance is to prevent deadlocks from ever happening, by preventing

at least one of the aforementioned conditions.

This requires more information about each process, AND tends to lead to low device utilization. ( I.e. it is In some algorithms the scheduler only needs to know the maximum number of each resource that a

process might potentially use. In more complex algorithms the scheduler can also take advantage of the

schedule of exactly what resources may be needed in what order.

When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks,

then that process is just not started or the request is not granted.

A resource allocation state is defined by the number of available and allocated resources, and the

maximum requirements of all processes in the system.

Safe State

i. A state is safe if the system can allocate all resources requested by all processes ( up to their stated

maximums ) without entering a deadlock state.

ii. More formally, a state is safe if there exists a safe sequence of processes { P0, P1, P2, ..., PN } such

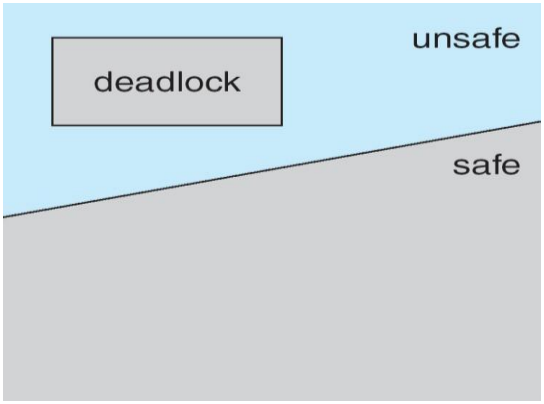
that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all

processes Pj where j < i. ( I.e. if all the processes prior to Pi finish and free up their resources, then Pi will

be able to finish also, using the resources that they have freed up. )

iii. If a safe sequence does not exist, then the system is in an unsafe state, which MAY lead to deadlock. (

All safe states are deadlock free, but not all unsafe states lead to deadlocks. )




For example, consider a system with 12 tape drives, allocated as follows. Is this a safe state? What is the

safe sequence?

| | Maximum needs | Current allocation |
|---|---|---|
| P0 | 10 | 5 |
| P1 | 4 | 2 |
| P2 | 9 | 2 |

ii. Key to the safe state approach is that when a request is made for resources, the request is granted only

if the resulting allocation state is a safe one.

Resource-Allocation Graph Algorithm

i. If resource categories have only single instances of their resources, then deadlock states can be detected

by cycles in the resource-allocation graphs.

ii. In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph

with claim edges, noted by dashed lines, which point from a process to a resource that it may request in

the future.

iii. In order for this technique to work, all claim edges must be added to the graph for any particular

process before that process is allowed to request any resources. ( Alternatively, processes may only make

requests for resources for which they have already established claim edges, and claim edges cannot be

added to any process that is currently holding resources. )

iv. When a process makes a request, the claim edge Pi->Rj is converted to a request edge. Similarly when

a resource is released, the assignment reverts back to a claim edge.

v. This approach works by denying requests that would produce cycles in the resource-allocation graph,

taking claim edges into effect.

Consider for example what happens when process P2 requests resource R2:

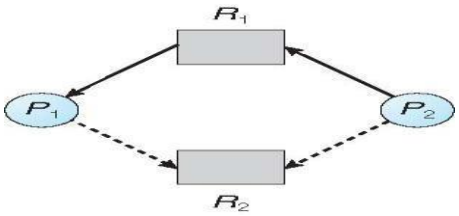



Fig: Resource allocation graph for dead lock avoidance

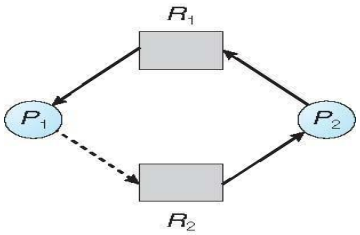



Fig: An Unsafe State in a Resource Allocation Graph

Banker's Algorithm

i. For resource categories that contain more than one instance the resource-allocation graph

method does not work, and more complex ( and less efficient ) methods must be chosen.

ii. The Banker's Algorithm gets its name because it is a method that bankers could use to assure

that when they lend out resources they will still be able to satisfy all their clients. ( A banker

won't loan out a little money to start building a house unless they are assured that they will later

be able to loan out the rest of the money to finish the house. )

iii. When a process starts up, it must state in advance the maximum allocation of resources it may

request, up to the amount available on the system.

iv. When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types. N

Available: Vector of length m. If available [j] = k, there are k instances of resource type Rj available n

Max: n x m matrix. If Max [i,j] = k, then process Pi may request at most k instances of resource type Rjn

Allocation: n x m matrix. If Allocation[i,j] = k then Pi is currently allocated k instances of Rjn

Need: n x m matrix. If Need[i,j] = k, then Pi may need k more instances of Rj to complete its task

Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

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

Initialize: Work = Available Finish [i] = false for i = 0, 1, …, n- 1

2. Find and i such that both:

(a) Finish [i] = false

(b) Needi <= Work

If no such i exists, go to step 4

3. 3. Work = Work + Allocationi

Finish[i] = true go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state

Resource-Request Algorithm for Process Pi

type Rj

1. If Requesti <= Needi

go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim

2. If Requesti £ Available, go to step 3. Otherwise Pi must wait, since resources are not available

3. Pretend to allocate requested resources to Pi by modifying the state as follows:

Available = Available – Request;

Allocationi = Allocationi + Requesti;

Needi = Needi – Requesti;

lIf safe Þ the resources are allocated to Pi

lIf unsafe Þ Pi must wait, and the old resource-allocation state is restored

| | *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 | |

| | *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 < P1, P3, P4, P2, P0> satisfies safety criteria.

Example: P1 Request (1,0,2) Check that Request £ Available (that is, (1,0,2) <= (3,3,2) Þ true

| *Allocation* | *Need* | *Available* |
|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 4 3 | 2 3 0 |
| $P_1$ | 3 0 2 | 0 2 0 | |
| $P_2$ | 3 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 1 | |

Executing safety algorithm shows that sequence < P1, P3, P4, P0, P2> satisfies safety requirement

3.3.6. Deadlock Detection

i. If deadlocks are not avoided, then another approach is to detect when they have occurred and recover

somehow.

ii. In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in

place for recovering from deadlocks, and there is potential for lost work when processes must be aborted

or have their resources preempted.

6.1 Single Instance of Each Resource Type

i. If each resource category has a single instance, then we can use a variation of the resource-allocation

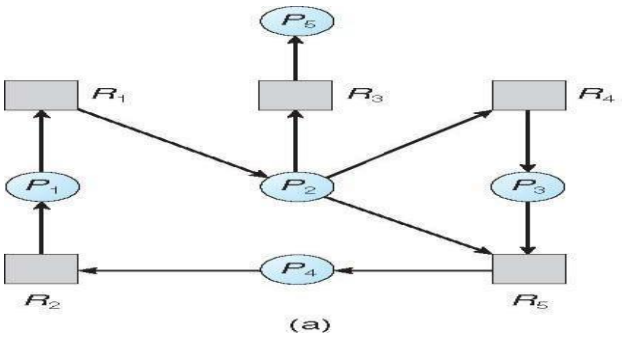
graph known as a wait-for graph.

ii. A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and

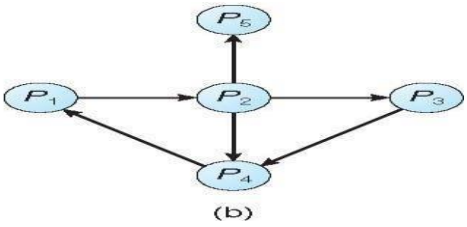
collapsing the associated edges, as shown in the figure below.

iii. An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process

Pj is currently holding.




Resource-Allocation Graph




Resource-Allocation Graph

This algorithm must maintain the wait-for graph, and periodically search it for cycles.

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 [ij] = k, then process

Pi is requesting k more instances of resource type. Rj .

Detection Algorithm

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

(a) Work = Available(b) For i = 1,2, …, n,

if Allocationi != 0, then

Finish[i] = false;

otherwise,

Finish[i] = true

2. Find an index i such that both:

(a) Finish[i] == false

(b) Requesti <=Work

If no such i exists, go to step 4

3. 3. Work = Work + Allocationi

Finish[i] = true go to step 2

4. If Finish[i] == false, for some i, 1 <= i <=n, then the system is in deadlock state.

Moreover, if Finish[i] == false, then Pi is deadlocked

Algorithm requires an order of O(m x n2) operations to detect whether the system is in deadlocked

state.

Example of Detection Algorithm

Five processes P0 through P4; three resource types A (7 instances), B (2 instances), and C (6 instances

| | *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 | |

Now suppose that process P2 makes a request for an additional instance of type C, yielding the state

shown below. Is the system now deadlocked?

| | *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 1 | |
| $P_3$ | 2 1 1 | 1 0 0 | |
| $P_4$ | 0 0 2 | 0 0 2 | |

i. When should the deadlock detection be done? Frequently, or infrequently?

The answer may depend on how frequently deadlocks are expected to occur, as well as the possible

consequences of not catching them immediately. ( If deadlocks are not removed immediately when they

occur, then more and more processes can "back up" behind the deadlock, making the eventual task of

unblocking the system more difficult and possibly damaging to more processes. )

ii. There are two obvious approaches, each with trade-offs:

1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has

the advantage of detecting the deadlock right away, while the minimum number of processes are involved

in the deadlock. ( One might consider that the process whose request triggered the deadlock condition is

the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for

the resulting deadlock. ) The down side of this approach is the extensive overhead and performance hit

caused by checking for deadlocks so frequently.

2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when

CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is

done much less frequently, but the down side is that it becomes impossible to detect the processes

i. When should the deadlock detection be done? Frequently, or infrequently?

The answer may depend on how frequently deadlocks are expected to occur, as well as the possible

consequences of not catching them immediately. ( If deadlocks are not removed immediately when they

occur, then more and more processes can "back up" behind the deadlock, making the eventual task of

unblocking the system more difficult and possibly damaging to more processes. )

ii. There are two obvious approaches, each with trade-offs:

1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has

the advantage of detecting the deadlock right away, while the minimum number of processes are involved

in the deadlock. ( One might consider that the process whose request triggered the deadlock condition is

the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for

the resulting deadlock. ) The down side of this approach is the extensive overhead and performance hit

caused by checking for deadlocks so frequently.

2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when

CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is

done much less frequently, but the down side is that it becomes impossible to detect the processes

**7. Recovery From Deadlock**

There are three basic approaches to recovery from deadlock:

i. Inform the system operator, and allow him/her to take manual intervention.

ii. Terminate one or more processes involved in the deadlock

iii. Preempt resources.

Process Termination

1. Two basic approaches, both of which recover resources allocated to terminated processes:

i. Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.

ii. Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.

2. In the latter case there are many factors that can go into deciding which processes to terminate next:

i. Process priorities.

ii. How long the process has been running, and how close it is to finishing.

iii. How many and what type of resources is the process holding. ( Are they easy to preempt and restore? )

iv. How many more resources does the process need to complete.

v. How many processes will need to be terminated

vi. Whether the process is interactive or batch.

Resource Preemption

When preempting resources to relieve deadlock, there are three important issues to be addressed:

Selecting a victim - Deciding which resources to preempt from which processes involves many of the

same decision criteria outlined above.

Rollback - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. ( I.e. abort the process and make it start over. )

Starvation - How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.

## 3.1 MEMORY MANAGEMENT

Memory management is concerned with managing the primary memory. Memory consists of

array of bytes or words each with their own address. The instructions are fetched from the memory

by the CPU based on the value program counter.

Functions of memory management:

- Keeping track of status of each memory location.
- Determining the allocation policy.
- Memory allocation technique.
- De-allocation technique.

Address Binding:

Programs are stored on the secondary storage disks as binary executable files. When the programs

are to be executed they are brought in to the main memory and placed within a process. The collection of

processes on the disk waiting to enter the main memory forms the input queue. One of the processes

which are to be executed is fetched from the queue and placed in the main memory. During the execution

it fetches instruction and data from main memory. After the process terminates it returns back the

memory space. During execution the process will go through different steps and in each step the address

is represented in different ways. In source program the address is symbolic. The compiler converts the

symbolic address to re-locatable address. The loader will convert this re-locatable address to absolute

address.

Binding of instructions and data can be done at any step along the way:

Compile time:-If we know whether the process resides in memory then absolute code can be

generated. If the static address changes then it is necessary to re-compile the code from the beginning.

Load time:-If the compiler doesn't know whether the process resides in memory then it generates

the re-locatable code. In this the binding is delayed until the load time.

Execution time:-If the process is moved during its execution from one memory segment to

another then the binding is delayed until run time. Special hardware is used for this. Most of the general

purpose operating system uses this method.

Logical versus physical address:

The address generated by the CPU is called logical address or virtual address. The address seen

by the memory unit i.e., the one loaded in to the memory register is called the physical address. Compile

time and load time address binding methods generate some logical and physical address. The execution

time addressing binding generate different logical and physical address. Set of logical address space

generated by the programs is the logical address space. Set of physical address corresponding to these

logical addresses is the physical address space. The mapping of virtual address to physical address during

run time is done by the hardware device called memory management unit (MMU). The base register is

also called re-location register. Value of the re-location register is added to every address generated by the

user process at the time it is sent to memory.

Dynamic re-location using a re-location registers

The above figure shows that dynamic re-location which implies mapping from virtual addresses

space to physical address space and is performed by the hardware at run time. Re-location is performed

by the hardware and is invisible to the user dynamic relocation makes it possible to move a partially

executed process from one area of memory to another without affecting.

Dynamic Loading:

For a process to be executed it should be loaded in to the physical memory. The size of the

process is limited to the size of the physical memory. Dynamic loading is used to obtain better memory

utilization. In dynamic loading the routine or procedure will not be loaded until it is called. Whenever a

routine is called, the calling routine first checks whether the called routine is already loaded or not. If it is

not loaded it cause the loader to load the desired program in to the memory and updates the programs

address table to indicate the change and control is passed to newly called routine.

Advantage: Gives better memory utilization. Unused routine is never loaded. Do not need special

operating system support. This method is useful when large amount of codes are needed to handle in

frequently occurring cases.

Dynamic linking and Shared libraries:

Some operating system supports only the static linking. In dynamic linking only the main

program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded

and the link is established at the time of references. This linking is postponed until the execution time.

With dynamic linking a “stub” is used in the image of each library referenced routine. A “stub” is a piece

of code which is used to indicate how to locate the appropriate memory resident library routine or how to

load library if the routine is not already present. When “stub” is executed it checks whether the routine is

present is memory or not. If not it loads the routine in to the memory. This feature can be used to update

libraries i.e., library is replaced by a new version and all the programs can make use of this library. More

than one version of the library can be loaded in memory at a time and each program uses its version of the

library. Only the programs that are compiled with the new version are affected by the changes

incorporated in it. Other programs linked before new version is installed will continue using older

libraries this type of system is called “shared library”.

3.1.1 Swapping

Swapping is a technique of temporarily removing inactive programs from the memory of the

system. A process can be swapped temporarily out of the memory to a backing store and then brought

back in to the memory for continuing the execution. This process is called swapping.

Eg:-In a multi-programming environment with a round robin CPU scheduling whenever the time quantum

expires then the process that has just finished is swapped out and a new process swaps in to the memory

for execution.

A variation of swap is priority based scheduling. When a low priority is executing and if a high

priority process arrives then a low priority will be swapped out and high priority is allowed for execution.

This process is also called as Roll out and Roll in.

Normally the process which is swapped out will be swapped back to the same memory space that

is occupied previously. This depends upon address binding.

If the binding is done at load time, then the process is moved to same memory location. If the

binding is done at run time, then the process is moved to different memory location. This is because the operating system support. This method is useful when large amount of codes are needed to handle in

frequently occurring cases.

Dynamic linking and Shared libraries:

Some operating system supports only the static linking. In dynamic linking only the main

program is loaded in to the memory. If the main program requests a procedure, the procedure is loaded

and the link is established at the time of references. This linking is postponed until the execution time.

With dynamic linking a “stub” is used in the image of each library referenced routine. A “stub” is a piece

of code which is used to indicate how to locate the appropriate memory resident library routine or how to

load library if the routine is not already present. When “stub” is executed it checks whether the routine is

present is memory or not. If not it loads the routine in to the memory. This feature can be used to update

libraries i.e., library is replaced by a new version and all the programs can make use of this library. More

than one version of the library can be loaded in memory at a time and each program uses its version of the

library. Only the programs that are compiled with the new version are affected by the changes

incorporated in it. Other programs linked before new version is installed will continue using older

libraries this type of system is called “shared library”.

3.1.1 Swapping

Swapping is a technique of temporarily removing inactive programs from the memory of the

system. A process can be swapped temporarily out of the memory to a backing store and then brought

back in to the memory for continuing the execution. This process is called swapping.

Eg:-In a multi-programming environment with a round robin CPU scheduling whenever the time quantum

expires then the process that has just finished is swapped out and a new process swaps in to the memory

for execution.

A variation of swap is priority based scheduling. When a low priority is executing and if a high

priority process arrives then a low priority will be swapped out and high priority is allowed for execution.

This process is also called as Roll out and Roll in.

Normally the process which is swapped out will be swapped back to the same memory space that

is occupied previously. This depends upon address binding.

If the binding is done at load time, then the process is moved to same memory location. If the

binding is done at run time, then the process is moved to different memory location. This is because the physical address is computed during run time.

Swapping requires backing store and it should be large enough to accommodate the copies of all

memory images. The system maintains a ready queue consisting of all the processes whose memory

images are on the backing store or in memory that are ready to run. Swapping is constant by other factors:

To swap a process, it should be completely idle. A process may be waiting for an i/o operation. If the i/o

is asynchronously accessing the user memory for i/o buffers, then the process cannot be swapped

3.1.2 Contiguous memory allocation:

One of the simplest method for memory allocation is to divide memory in to several fixed

partition. Each partition contains exactly one process. The degree of multi-programming depends on the

number of partitions. In multiple partition method, when a partition is free, process is selected from the

input queue and is loaded in to free partition of memory. When process terminates, the memory partition

becomes available for another process. Batch OS uses the fixed size partition scheme.

The OS keeps a table indicating which part of the memory is free and is occupied. When

the process enters the system it will be loaded in to the input queue. The OS keeps track of the memory

requirement of each process and the amount of memory available and determines which process to

allocate the memory. When a process requests, the OS searches for large hole for this process, hole is a

large block of free memory available. If the hole is too large it is split in to two. One part is allocated to

the requesting process and other is returned to the set of holes. The set of holes are searched to determine

which hole is best to allocate. There are three strategies to select a free hole:

 First bit:-Allocates first hole that is big enough. This algorithm scans memory from the

beginning and selects the first available block that is large enough to hold the process.

 Best bit:-It chooses the hole i.e., closest in size to the request. It allocates the smallest

hole i.e., big enough to hold the process.

 Worst fit:-It allocates the largest hole to the process request. It searches for the largest

hole in the entire list.

Operating Systems

First fit and best fit are the most popular algorithms for dynamic memory allocation. First fit is

generally faster. Best fit searches for the entire list to find the smallest hole i.e., large enough. Worst fit

reduces the rate of production of smallest holes. All these algorithms suffer from fragmentation.

Memory Protection:

Memory protection means protecting the OS from user process and protecting process

from one another. Memory protection is provided by using a re-location register, with a limit register.

Re-location register contains the values of smallest physical address and limit register contains range of

logical addresses.

(Re-location = 100040 and limit = 74600). The logical address must be less than the limit

register; the MMU maps the logical address dynamically by adding the value in re-location register.

When the CPU scheduler selects a process for execution, the dispatcher loads the re-location and limit

register with correct values as a part of context switch. Since every address generated by the CPU is

checked against these register we can protect the OS and other users programs and data from being

modified.

Fragmentation:

Memory fragmentation can be of two types: Internal Fragmentation External Fragmentation

Internal Fragmentation there is wasted space internal to a portion due to the fact that block of data

loaded is smaller than the partition. Eg:-If there is a block of 50kb and if the process requests 40kb and if

the block is allocated to the process then there will be 10kb of memory left.

External Fragmentation exists when there is enough memory space exists to satisfy the request,

but it not contiguous i.e., storage is fragmented in to large number of small holes.

External Fragmentation may be either minor or a major problem.

One solution for over-coming external fragmentation is compaction. The goal is to move all the

free memory together to form a large block. Compaction is not possible always. If the relocation is static

and is done at load time then compaction is not possible. Compaction is possible if the re-location is

dynamic and done at execution time.

Another possible solution to the external fragmentation problem is to permit the logical address

space of a process to be non-contiguous, thus allowing the process to be allocated physical memory

whenever the latter is available.

3.1.3 Segmentation

Most users do not think memory as a linear array of bytes rather the users thinks memory as a collection of variable sized segments which are dedicated to a particular use such as code, data, stack,

heap etc. A logical address is a collection of segments. Each segment has a name and length. The address

specifies both the segment name and the offset within the segments. The users specify address by using

two quantities: a segment name and an offset. For simplicity the segments are numbered and referred by

a segment number. So the logical address consists of <segment number, offset>.

Hardware support:

We must define an implementation to map 2D user defined address in to 1D physical address.

This mapping is affected by a segment table. Each entry in the segment table has a segment base and

segment limit. The segment base contains the starting physical address where the segment resides and

limit specifies the length of the segment.

The use of segment table is shown in the above figure: Logical address consists of two parts:

segment number‘s’ and an offset‘d’ to that segment. The segment number is used as an index to segment

table. The offset’d’ must be in between 0 and limit, if not an error is reported to OS. If legal the offset is

added to the base to generate the actual physical address. The segment table is an array of base limit

register pairs.

Protection and Sharing:

A particular advantage of segmentation is the association of protection with the segments. The

memory mapping hardware will check the protection bits associated with each segment table entry to

prevent illegal access to memory like attempts to write in to read-only segment. Another advantage of segmentation involves the sharing of code or data. Each process has a segment table associated with it.Segments are shared when the entries in the segment tables of two different processes points to samephysical location. Sharing occurs at the segment table. Any information can be shared at the segmentlevel. Several segments can be shared so a program consisting of several segments can be shared. We can also share parts of a program.

Advantages: Eliminates fragmentation. x Provides virtual growth. Allows dynamic segment growth.

Assist dynamic linking. Segmentation is visible.

Fig refer to txt

Differences between segmentation and paging:-

Segmentation:

• Program is divided in to variable sized segments. x User is responsible for dividing the program

in to segments.

• Segmentation is slower than paging.

• Visible to user.

• Eliminates internal fragmentation.

• Suffers from external fragmentation.

• Process or user segment number, offset to calculate absolute address.

Paging:

• Programs are divided in to fixed size pages.

• Division is performed by the OS.

• Paging is faster than segmentation.

• Invisible to user.

• Suffers from internal fragmentation.

• No external fragmentation.

• Process or user page number, offset to calculate absolute address.

3.1.4 Paging

Paging is a memory management scheme that permits the physical address space of a process to

be non-contiguous. Support for paging is handled by hardware. It is used to avoid external fragmentation.

Paging avoids the considerable problem of fitting the varying sized memory chunks on to the backing

store. When some code or date residing in main memory need to be swapped out, space must be found on backing store.

Basic Method:

Physical memory is broken in to fixed sized blocks called frames (f). Logical memory is broken

in to blocks of same size called pages (p). When a process is to be executed its pages are loaded in to

available frames from backing store. The blocking store is also divided in to fixed-sized blocks of same Logical address generated by the CPU is divided in to two parts: page number (p) and page offset

(d). The page number (p) is used as index to the page table. The page table contains base address of each

page in physical memory. This base address is combined with the page offset to define the physical

memory i.e., sent to the memory unit. The page size is defined by the hardware. The size of a power of 2,

varying between 512 bytes and 10Mb per page. If the size of logical address space is 2^m address unit

and page size is 2^n, then high order m-n designates the page number and n low order bits represents page

offset.

Eg:-To show how to map logical memory in to physical memory consider a page size of 4 bytes

and physical memory of 32 bytes (8 pages). Logical address 0 is page 0 and offset 0. Page 0 is in

frame 5. The logical address 0 maps to physical address 20. [(5*4) + 0]. logical address 3 is page 0 and

offset 3 maps to physical address 23 [(5*4) + 3]. Logical address 4 is page 1 and offset 0 and page 1 is

mapped to frame 6. So logical address 4 maps to physical address 24 [(6*4) + 0]. Logical address 13 is

page 3 and offset 1 and page 3 is mapped to frame 2. So logical address 13 maps to physical address 9

[(2*4) + 1].

Hardware Support for Paging:

The hardware implementation of the page table can be done in several ways:

The simplest method is that the page table is implemented as a set of dedicated registers. These registers

must be built with very high speed logic for making paging address translation. Every accessed memory must go through paging map. The use of registers for page table is satisfactory if the page table is small.

If the page table is large then the use of registers is not visible. So the page table is kept in the main

memory and a page table base register [PTBR] points to the page table. Changing the page table requires

only one register which reduces the context switching type. The problem with this approach is the time

required to access memory location. To access a location [i] first we have to index the page table using

PTBR offset. It gives the frame number which is combined with the page offset to produce the actual

address. Thus we need two memory accesses for a byte.

The only solution is to use special, fast, lookup hardware cache called translation look aside buffer

[TLB] or associative register. LB is built with associative register with high speed memory. Each register

contains two paths a key and a value.

When an associative register is presented with an item, it is compared with all the key values, if

found the corresponding value field is return and searching is fast. TLB is used with the page table as

follows: TLB contains only few page table entries. When a logical address is generated by the CPU, its

page number along with the frame number is added to TLB. If the page number is found its frame

memory is used to access the actual memory. If the page number is not in the TLB (TLB miss) the

memory reference to the page table is made. When the frame number is obtained use can use it to access

the memory. If the TLB is full of entries the OS must select anyone for replacement. Each time a new

page table is selected the TLB must be flushed [erased] to ensure that next executing process do not use

wrong information. The percentage of time that a page number is found in the TLB is called HIT ratio.

Protection:

Memory protection in paged environment is done by protection bits that are associated with each

frame these bits are kept in page table. x One bit can define a page to be read-write or read-only. To find the correct frame number every reference to the memory should go through page table. At the same time

physical address is computed. The protection bits can be checked to verify that no writers are made to

read-only page. Any attempt to write in to read-only page causes a hardware trap to the OS. This

approach can be used to provide protection to read-only, read-write or execute-only pages. One more bit

is generally added to each entry in the page table: a valid-invalid bit.

**Refer to txt for fig**

**A valid bit indicates that associated page is in the processes logical address space and thus it is a legal or**

**valid page.**

**If the bit is invalid, it indicates the page is not in the processes logical addressed space and illegal. Illegal**

**addresses are trapped by using the valid-invalid bit.**

**The OS sets this bit for each page to allow or disallow accesses to that page.**

**3.1.5 Structure Of Page Table**

**a. Hierarchical paging:**

**Recent computer system support a large logical address apace from 2^32 to 2^64. In this system**

**the page table becomes large. So it is very difficult to allocate contiguous main memory for page table.**

**One simple solution to this problem is to divide page table in to smaller pieces. There are several ways to**

**accomplish this division.**

**One way is to use two-level paging algorithm in which the page table itself is also paged. Eg:-In a 32 bit**

machine with page size of 4kb. A logical address is divided in to a page number consisting of 20 bits and

a page offset of 12 bit. The page table is further divided since the page table is paged, the page number is

further divided in to 10 bit page number and a 10 bit offset.

**b.** Hashed page table:

Hashed page table handles the address space larger than 32 bit. The virtual page number is used

as hashed value. Linked list is used in the hash table which contains a list of elements that hash to the

same location.

Each element in the hash table contains the following three fields: Virtual page number x Mapped

page frame value x Pointer to the next element in the linked list

Working:

Virtual page number is taken from virtual address. Virtual page number is hashed in to hash table.

Virtual page number is compared with the first element of linked list. Both the values are

matched, that value is (page frame) used for calculating the physical address. If not match then entire

linked list is searched for matching virtual page number. Clustered pages are similar to hash table but one

difference is that each entity in the hash table refer to several pages.

**c.** Inverted Page Tables:

Since the address spaces have grown to 64 bits, the traditional page tables become a problem.

Even with two level page tables. The table can be too large to handle. An inverted page table has only

entry for each page in memory. Each entry consisted of virtual address of the page stored in that read-only

location with information about the process that owns that page.

Each virtual address in the Inverted page table consists of triple <process-id , page number ,

offset >. The inverted page table entry is a pair <process-id , page number>. When a memory reference is

made, the part of virtual address i.e., <process-id , page number> is presented in to memory sub-system.

The inverted page table is searched for a match. If a match is found at entry I then the physical address <i

, offset> is generated. If no match is found then an illegal address access has been attempted. This scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed

to search the table when a page reference occurs. If the whole table is to be searched it takes too long.

Advantage:

• Eliminates fragmentation.

• Support high degree of multiprogramming.

• Increases memory and processor utilization.

• Compaction overhead required for the re-locatable partition scheme is also eliminated.

Disadvantage:

• Page address mapping hardware increases the cost of the computer.

• Memory must be used to store the various tables like page tables, memory map table etc.

• Some memory will still be unused if the number of available block is not sufficient for the

address space of the jobs to be run.

**d.** Shared Pages:

Another advantage of paging is the possibility of sharing common code. This is useful in timesharing

environment. Eg:-Consider a system with 40 users, each executing a text editor. If the text editor is of

150k and data space is 50k, we need 8000k for 40 users. If the code is reentrant it can be shared. Consider

the following figure

If the code is reentrant then it never changes during execution. Thus two or more processes can

execute same code at the same time. Each process has its own copy of registers and the data of two

processes will vary. Only one copy of the editor is kept in physical memory. Each user's page table maps

to same physical copy of editor but date pages are mapped to different frames. So to support 40 users we

need only one copy of editor (150k) plus 40 copies of 50k of data space i.e., only 2150k instead of 8000k. refer to txt for fig

# Module IV
# Virtual-Memory Management

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

- In practice, most real processes do not need all their pages, or at least not all at once, for several reasons:
  1. Error handling code is not needed unless that specific error occurs, some of which are quite rare.
  2. Certain features of certain programs are rarely used.
- The ability to load only the portions of processes that are actually needed has several benefits:
  - Programs could be written for a much larger address space (virtual memory space ) than physically exists on the computer.
  - Because each process is only using a fraction of their total address space, there is more memory left for other programs, improving CPU utilization and system throughput.
  - Less I/O is needed for swapping processes in and out of RAM, speeding things up.

The figure below shows the general layout of ***virtual memory***, which can be much larger than physical memory:

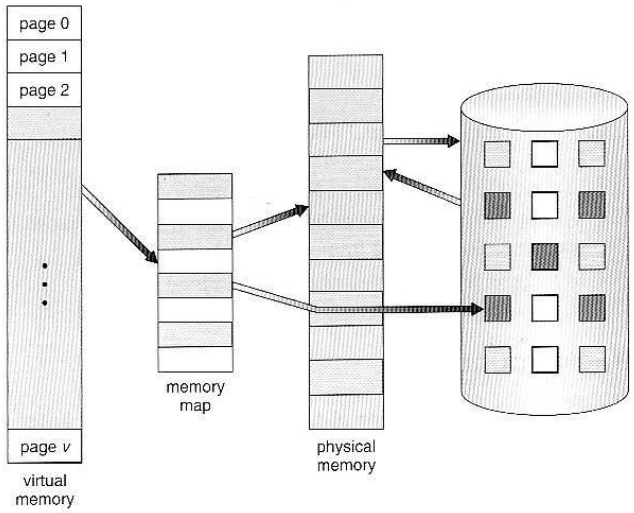



**Figure 9.1** Diagram showing virtual memory that is larger than physical memory.

- The figure below shows ***virtual address space***, which is the programmer's logical view of process memory storage. The actual physical layout is controlled by the process's page table.

- Note that the address space shown in Figure 9.2 is ***sparse*** - A great hole in the middle of the address space is never used, unless the stack and/or the heap grow to fill the hole.

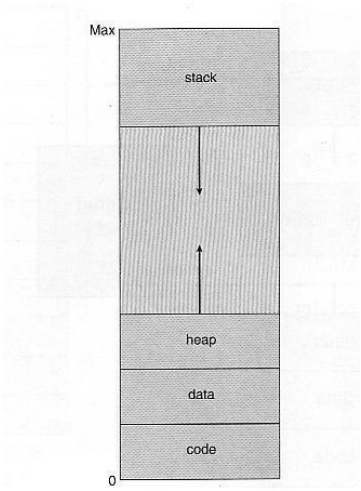



**Figure 9.2** Virtual address space.

- Virtual memory also allows the sharing of files and memory by multiple processes, with several benefits:
- System libraries can be shared by mapping them into the virtual address space of more than one process.
- Processes can also share virtual memory by mapping the same block of memory to more than one process.
- Process pages can be shared during a fork( ) system call, eliminating the need to copy all of the pages of the original ( parent ) process.

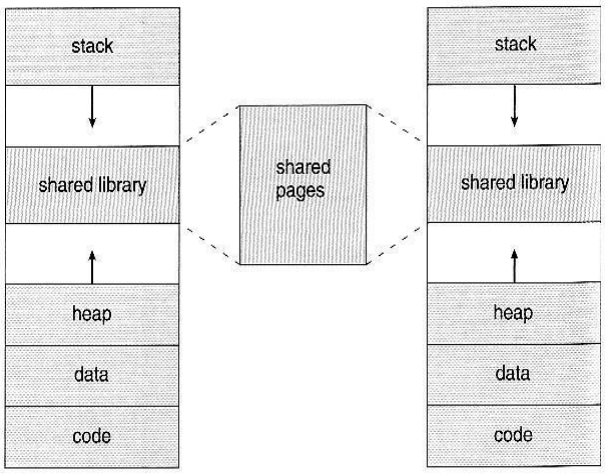



**Figure 9.3** Shared library using virtual memory.

## 9.2 Demand Paging

- The basic idea behind ***demand paging*** is that when a process is swapped in, its pages are not swapped in all at once. Rather they are swapped in only when the process needs them. ( on demand. ) This is termed as ***lazy swapper***, although a ***pager*** is a more accurate term.

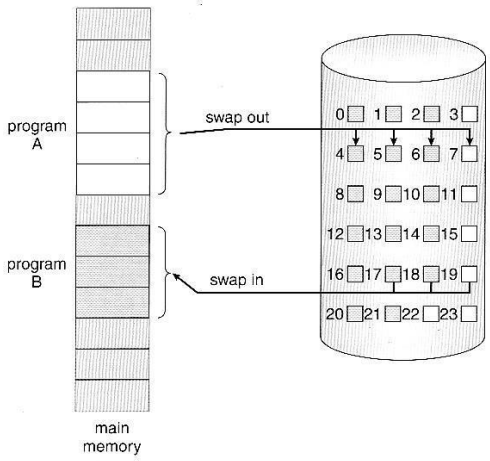



**Figure 9.4** Transfer of a paged memory to contiguous disk space.

- The basic idea behind demand paging is that when a process is swapped in, the pager only loads into memory those pages that is needed presently.
- Pages that are not loaded into memory are marked as invalid in the page table, using the invalid bit. Pages loaded in memory are marked as valid.
- If the process only ever accesses pages that are loaded in memory ( ***memory resident*** pages ), then the process runs exactly as if all the pages were loaded in to memory.

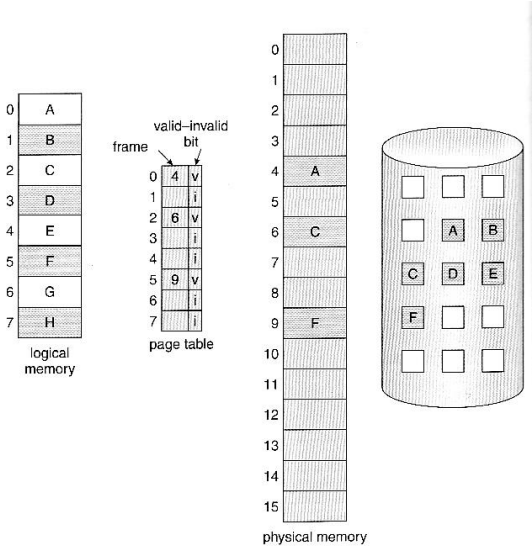



**Figure 9.5** Page table when some pages are not in main memory.

- On the other hand, if a page is needed that was not originally loaded up, then a ***page fault trap*** is generated, which must be handled in a series of steps:
  1. The memory address requested is first checked, to make sure it was a valid memory request.
  2. If the reference is to an invalid page, the process is terminated. Otherwise, if the page is not present in memory, it must be paged in.
  3. A free frame is located, possibly from a free-frame list.
  4. A disk operation is scheduled to bring in the necessary page from disk.
  5. After the page is loaded to memory, the process's page table is updated with the new frame number, and the invalid bit is changed to indicate that this is now a valid page reference.
  6. The instruction that caused the page fault must now be restarted from the beginning.

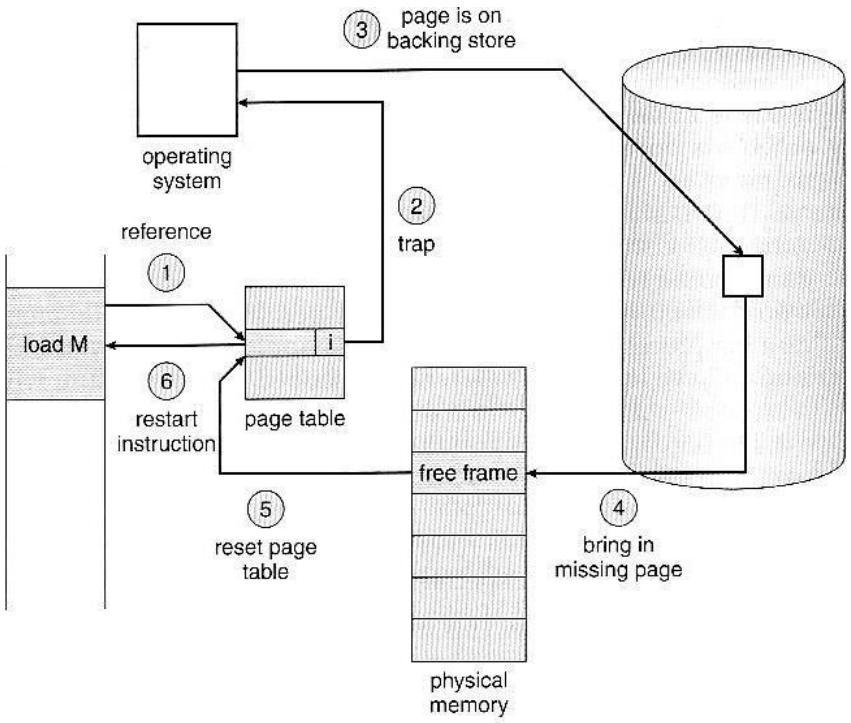



**Figure 9.6** Steps in handling a page fault.

- In an extreme case, the program starts execution with zero pages in memory. Here NO pages are swapped in for a process until they are requested by page faults. This is known as ***pure demand paging.***
- The **hardware necessary** to support demand paging is the same as for paging and swapping: A page table and secondary memory.

**Performance of Demand Paging**

- There is some slowdown and performance hit whenever a page fault occurs(as the required page is not available in memory) and the system has to go get it from memory.
- There are many steps that occur when servicing a page fault and some of the steps are optional or variable. But just for the sake of discussion, suppose that a normal memory access requires 200 nanoseconds, and that servicing a page fault takes 8 milliseconds. ( 8,000,000 nanoseconds, or 40,000 times a normal memory access. ) With a ***page fault rate*** of p, ( on a scale from 0 to 1 ), the effective access time is now:

Effective access time = p * time taken to access memory in page fault+ (1-p)* time taken to access memory

= p * 8000000 + ( 1 - p ) * ( 200 )

= 200 + 7,999,800 * p

Even if only one access in 1000 causes a page fault, the effective access time drops from 200 nanoseconds to 8.2 microseconds, a slowdown of a factor of 40 times. In order to keep the slowdown less than 10%, the page fault rate must be less than 0.0000025, or one in 399,990 accesses.

## 9.3 Copy-on-Write

- The idea behind a copy-on-write is that the pages of a parent process is shared by the child process, until one or the other of the processes changes the page. Only when a process changes any page content, that page is copied for the child.

- Only pages that can be modified need to be labeled as copy-on-write. Code segments can simply be shared.
- Some systems provide an alternative to the fork( ) system call called a ***virtual memory fork, vfork( )***. In this case the parent is suspended, and the child uses the parent's memory pages. This is very fast for process creation, but requires that the child not modify any of the shared memory pages

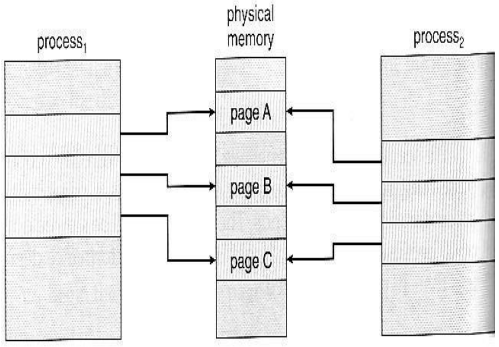



Figure 9.7 Before process 1 modifies page C.

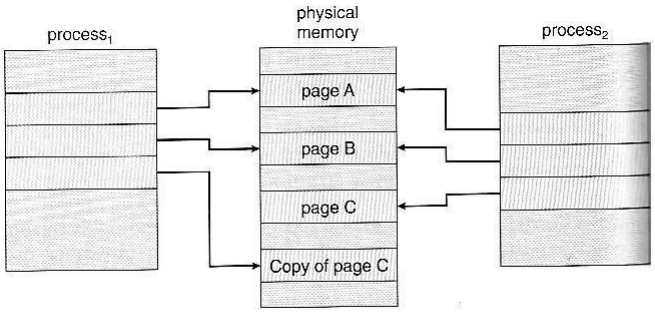



Figure 9.8 After process 1 modifies page C.

before performing the exec( ) system call.

## 9.4 Page Replacement

- In order to make the most use of virtual memory, we load several processes into memory at the same time. Since we only load the pages that are actually needed by each process at any given time, there are frames to load many more processes in memory.
- If some process suddenly decides to use more pages and there aren't any free frames available. Then there are several possible solutions to consider:
    1. Adjust the memory used by I/O buffering, etc., to free up some frames foruser processes.
    2. Put the process requesting more pages into a wait queue until some free frames become available.
    3. Swap some process out of memory completely, freeing up its page frames.
    4. Find some page in memory that isn't being used right now, and swap that page only out to disk, freeing up a frame that can be allocated to the process requesting it. This is known as ***page replacement***, and is the most common solution. There are many different algorithms for page replacement.

The previously discussed page-fault processing assumed that there would be free frames available on the free-frame list. Now the page-fault handling must be modified to free up a frame if necessary, as follows:

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select an existing frame to be replaced, known as the ***victim frame***.
    c. Write the victim frame to disk. Change all related page tables to indicate that this page is no longer in memory.
3. Read in the desired page and store it in the frame. Change the entries in page table.
4. Restart the process that was waiting for this page.

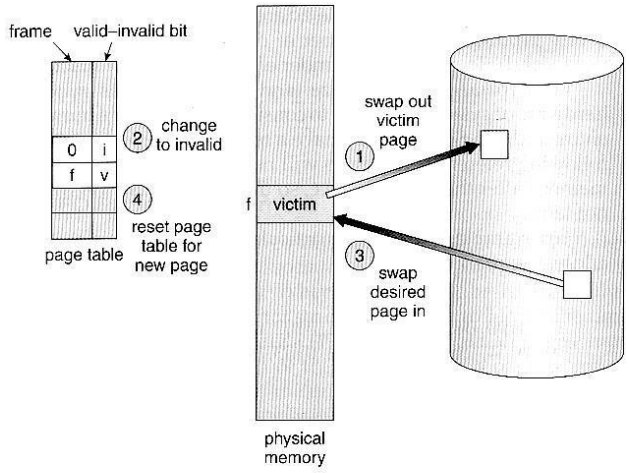



**Figure 9.10** Page replacement.

- Note that step 2c adds an extra disk write to the page-fault handling, thus doubling the time required to process a page fault. This can be reduced by assigning a ***modify bit,*** or ***dirty bit*** to each page in memory, indicating whether or not it has been changed since it was last loaded in from disk. If the page is not modified the bit is not set. If the dirty bit has not been set, then the page is unchanged, and does not need to be written out to disk. Many page replacement strategies specifically look for pages that do not have their dirty bit set.
- There are two major requirements to implement a successful demand paging system.

  A ***frame-allocation algorithm*** and a ***page-replacement algorithm.*** The former centers around how many frames are allocated to each process, and the latter deals with how to select a page for replacement when there are no free frames available.

- The overall goal in selecting and tuning these algorithms is to generate the fewest number of overall page faults. Because disk access is so slow relative to memory access, even slight improvements to these algorithms can yield large improvements in overall system performance.
- Algorithms are evaluated using a given string of page accesses known as a ***reference string.***

## Few Page Replacement algorithms –

### a) FIFO Page Replacement

- A simple and obvious page replacement strategy is ***FIFO***, i.e. first-in-first-out.
- This algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

- Or a FIFO queue can be created to hold all pages in memory. As new pages are brought in, they are added to the tail of a queue, and the page at the head of the queue is the next victim.
- In the following example, a reference string is given and there are 3 free frames. There are 20 page requests, which results in 15 page faults.

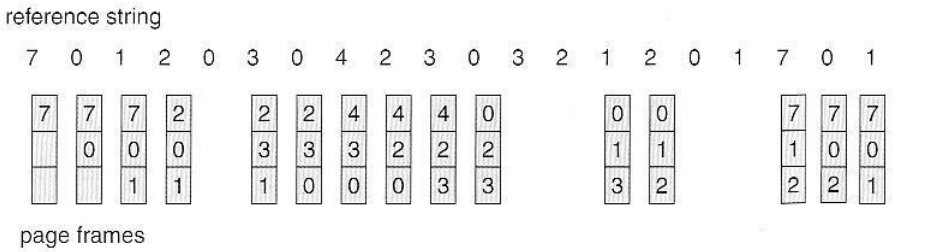



**Figure 9.12** FIFO page-replacement algorithm.

- Although FIFO is simple and easy to understand, it is not always optimal, or even efficient.
- ***Belady's anomaly*** tells that for some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.

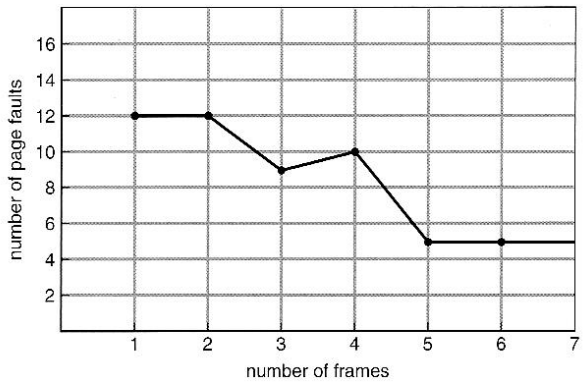



**Figure 9.13** Page-fault curve for FIFO replacement on a reference string.

## b) Optimal Page Replacement

- The discovery of Belady's anomaly lead to the search for an ***optimal page-replacement algorithm***, which is simply that which yields the lowest of all possible page-faults, and which does not suffer from Belady's anomaly.
- Such an algorithm does exist, and is called ***OPT or MIN.*** This algorithm is "Replace the page that will not be used for the longest time in the future."
- The same reference string used for the FIFO example is used in the example below, here the minimum number of possible page faults is 9.

- Unfortunately OPT cannot be implemented in practice, because it requires the knowledge of future string, but it makes a nice benchmark for the comparison and evaluation of real proposed new algorithms.

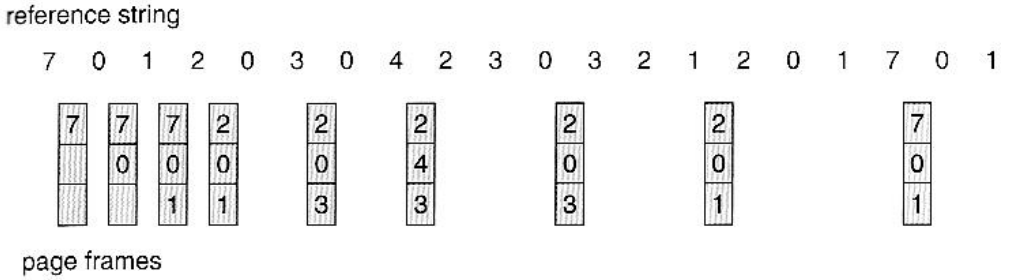



**Figure 9.14** Optimal page-replacement algorithm.

### c) LRU Page Replacement

- The ***LRU (Least Recently Used)*** algorithm, predicts that the page that has not been used in the longest time is the one that will not be used again in the near future.
- Some view LRU as analogous to OPT, but here we look backwards in time instead of forwards.
- Figure 9.15 illustrates LRU for our sample string, yielding 12 page faults, ( as compared to 15 for FIFO and 9 for OPT. )

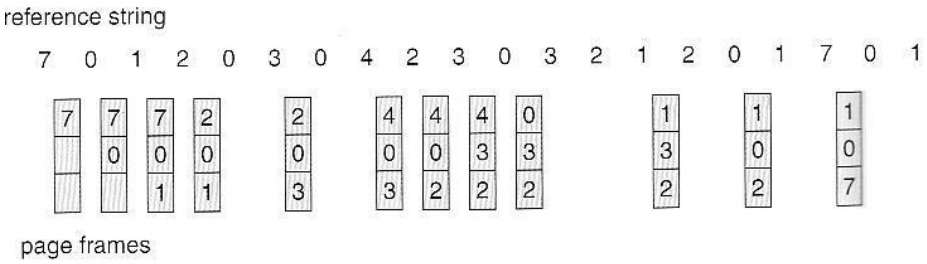



**Figure 9.15** LRU page-replacement algorithm.

- LRU is considered a good replacement policy, and is often used. There are two simple approaches commonly used to implement this:
  1. **Counters.** With each page-table entry a time-of-use field is associated. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access.

2. **Stack.** Another approach is to use a stack, and whenever a page is accessed, pull that page from the middle of the stack and place it on the top. The LRU page will always be at the bottom of the stack. Because this requires removing objects from the middle of the stack, a doubly linked list is the recommended data structure.

- Neither LRU or OPT exhibit Belady's anomaly. Both belong to a class of page-replacement algorithms called ***stack algorithms,*** which can never exhibit Belady's anomaly.

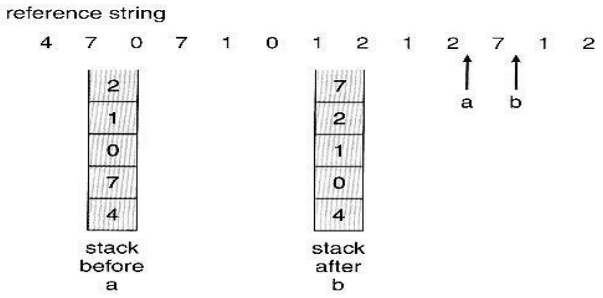



**Figure 9.16** Use of a stack to record the most recent page references.

### d) LRU-Approximation Page Replacement

- Many systems offer some degree of hardware support, enough to approximate LRU.
- In particular, many systems provide a ***reference bit*** for every entry in a page table, which is set anytime that page is accessed. Initially all bits are set to zero, and they can also all be cleared at any time. One bit distinguishes pages that have been accessed since the last clear from those that have not been accessed.

### d.1 Additional-Reference-Bits Algorithm

- An 8-bit byte(reference bit) is stored for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
- If the shift register contains 00000000, then the page has not been used for eight time periods.
- A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

### d.2 Second-Chance Algorithm

- The ***second chance algorithm*** is a FIFO replacement algorithm, except the reference bit is used to give pages a second chance at staying in the page table.
- When a page must be replaced, the page table is scanned in a FIFO ( circular queue ) manner.

- If a page is found with its reference bit as '0', then that page is selected as the next victim.
- If the reference bit value is '1', then the page is given a second chance and its reference bit value is cleared( assigned as'0').

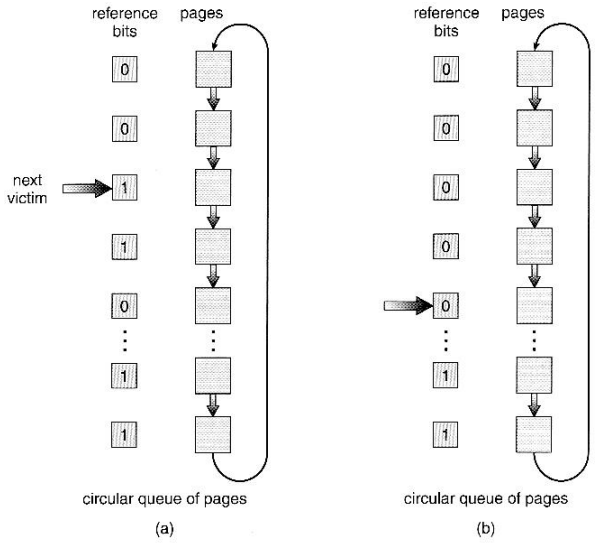



**Figure 9.17** Second-chance (clock) page-replacement algorithm.

- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often, then it sets its reference bit again.
- This algorithm is also known as the ***clock*** algorithm.

One way to implement the second-chance algorithm is as a circular queue. *A* pointer indicates which page is *to* be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

### d.3 Enhanced Second-Chance Algorithm

- The **enhanced second chance algorithm** looks at the reference bit and the modify bit ( dirty bit ) as an ordered page, and classifies pages into one of four classes:
  1. ( 0, 0 ) - Neither recently used nor modified.
  2. ( 0, 1 ) - Not recently used, but modified.
  3. ( 1, 0 ) - Recently used, but clean.
  4. ( 1, 1 ) - Recently used and modified.

- This algorithm searches the page table in a circular fashion, looking for the first page it can find in the lowest numbered category. i.e. it first makes a pass looking for a ( 0, 0 ), and then if it can't find one, it makes another pass looking for a ( 0, 1 ), etc.
- The main difference between this algorithm and the previous one is the preference for replacing clean pages if possible.

**e) Counting-Based Page Replacement**

- There are several algorithms based on counting the number of references that have been made to a given page, such as:
  - ***Least Frequently Used, LFU:*** Replace the page with the lowest reference count. A problem can occur if a page is used frequently initially and then not used any more, as the reference count remains high. A solution to this problem is to right-shift the counters periodically, yielding a time-decaying average reference count.
  - ***Most Frequently Used, MFU:*** Replace the page with the highest reference count. The logic behind this idea is that pages that have already been referenced a lot have been in the system a long time, and we are probably done with them, whereas pages referenced only a few times have only recently been loaded, and we still need them.

**f) Page-Buffering Algorithms**

- Maintain a certain minimum number of free frames at all times. When a page-fault occurs, go ahead and allocate one of the free frames from the free list first, so that the requesting process is in memory as early as possible, and then select a victim page to write to disk and free up a frame.
- Keep a list of modified pages, and when the I/O system is idle, these pages are written to disk, and then clear the modify bits, thereby increasing the chance of finding a "clean" page for the next potential victim and page replacement can be done much faster.

## 9.5 Allocation of Frames

- The absolute minimum number of frames that a process must be allocated is dependent on system architecture.
- The maximum number is defined by the amount of available physical memory.

**Allocation Algorithms**

After loading of OS, there are two ways in which the allocation of frames can be done to the processes.

- **Equal Allocation -** If there are m frames available and n processes to share them, each process gets m / n frames, and the leftovers are kept in a free-frame buffer pool.
- **Proportional Allocation -** Allocate the frames proportionally depending on the size of the process. If the size of process i is $S_i$, and S is the sum of size of all processes in the system, then the allocation for process $P_i$ is $a_i = m * S_i / S$. where m is the free frames available in the system.

Consider a system with a 1KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames.

With proportional allocation, we would split 62 frames between two processes, as follows-

m=62, S = (10+127)=137
Allocation for process 1 = 62 X 10/137 ~ 4
Allocation for process 2 = 62 X 127/137 ~57

Thus allocates 4 frames and 57 frames to student process and database respectively.

- Variations on proportional allocation could consider priority of processrather than just their size.

### Global versus Local Allocation

- Page replacement can occur both at local or global level.
- With local replacement, the number of pages allocated to a process is fixed, and page replacement occurs only amongst the pages allocated to this process.
- With global replacement, any page may be a potential victim, whether it currently belongs to the process seeking a free frame or not.
- Local page replacement allows processes to better control their own page fault rates, and leads to more consistent performance of a given process over different system load levels.
- Global page replacement is overall more efficient, and is the more commonly used approach.

### Non-Uniform Memory Access ( New )

- Usually the time required to access all memory in a system is equivalent.
- This may not be the case in multiple-processor systems, especially where each CPU is physically located on a separate circuit board which also holds some portion of the overall system memory.
- In such systems, CPUs can access memory that is physically located on the sameboard much faster than the memory on the other boards.
- The basic solution is akin to processor affinity - At the same time that we try to schedule processes on the same CPU to minimize cache misses, we also try to allocate memory for those processes on the same boards, to minimize access times.

## 9.6 Thrashing

Thrashing is the state of a process where there is **high paging activity**. A process that is spending more time paging than executing is said to be ***thrashing.***

### 9.6.1 Cause of Thrashing

- When memory is filled up and processes starts spending lots of time waiting for their pages to page in, then CPU utilization decreases(Processes are not executed as they are waiting for some pages), causing the scheduler to add in even more processes and increase the degree of multiprogramming even more. Thrashing has occurred, and system throughput plunges. No work is getting done, because the processes are spending all their time paging.
- In the graph given below , CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

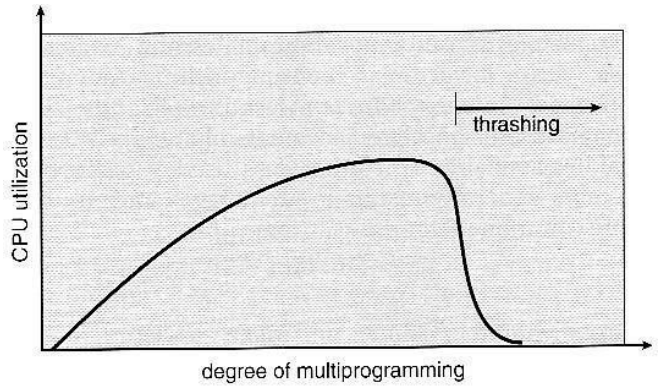



**Figure 9.18** Thrashing.

- Local page replacement policies can prevent thrashing process from taking pages away from other processes, but it still tends to clog up the I/O queue.

### 9.6.2 Working-Set Model

- The ***working set model*** is based on the concept of locality, and defines a ***working set window***, of length ***delta.*** Whatever pages are included in the most recent delta page references are said to be in the processes working set window, and comprise its current working set, as illustrated in Figure 9.20:

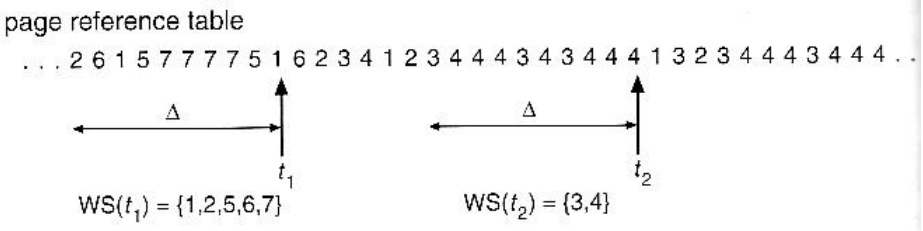



**Figure 9.20** Working-set model.

- The selection of delta is critical to the success of the working set model - If it is too small then it does not encompass all of the pages of the current locality, and if it is too large, then it encompasses pages that are no longer being frequently accessed.
- The total demand of frames, D, is the sum of the sizes of the working sets for all processes ($D=WSS_i$ ). If D exceeds the total number of available frames, then at least one process is thrashing, because there are not enough frames available to satisfy its minimum working set. If D is significantly less than the currently available frames, then additional processes can be launched.
- The hard part of the working-set model is keeping track of what pages are in the current working set, since every reference adds one to the set and removes one older page.

### 9.6.3 Page-Fault Frequency

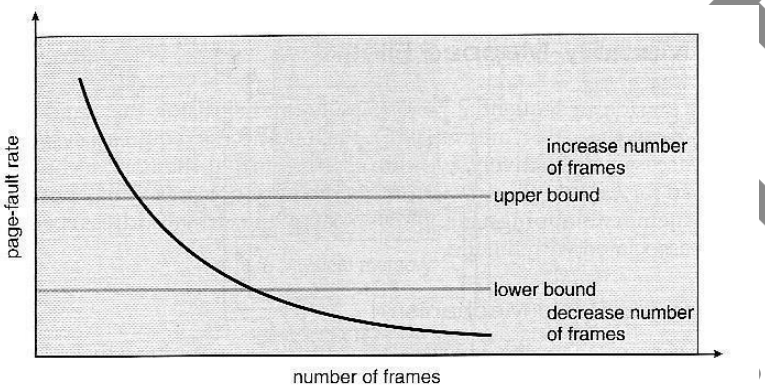



**Figure 9.21** Page-fault frequency.

- When page-fault rate is too high, the process needs more frames and when it is too low, the process may have too many frames.
- The upper and lower bounds can be established on the page-fault rate. If the actual page-fault rate exceeds the upper limit, allocate the process another frame or suspend the process. If the page-fault rate falls below the lower limit, remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.

# Module IV

# Implementation of File System

- **File Concept**
  - **File Attributes**
  - **File Operations**
  - **File Types**
  - **File Structure**
  - **Internal File Structure**
- **Access Methods**
  - **Sequential Access**
  - **Direct Access**
  - **Other Access Methods – Indexed Access**
- **Directory and Disk Structure**
  - **Storage Structure**
  - **Directory Overview**
  - **Single-Level Directory**
  - **Two-Level Directory**
  - **Tree –Structured Directories**
  - **Acyclic –Graph Directories**
  - **General Graph Directory**
- **File System Mounting**
- **File Sharing**
  - **Multiple Users**
  - **Remote File Systems**
    - **The Client-Server Model**
    - **Distributed Information Systems**
    - **Failure Modes**
- **Consistency Semantics**
  - **Unix Semantics**
  - **Session Semantics**
  - **Immutable – Shared Files Semantics**
- **Protection – Types, Access Control, other approaches**
- **File-System Structure**
- **File-System Implementation**
- **Directory Implementation**

- **Allocation Methods**
- **Free-Space Management**

**File Concepts**

The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system.

A file is a collection of related information that is recorded on secondary storage. A file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

**a) File Attributes**

A file is named, for the convenience of its human users, and is referred to by its name. A file's attributes vary from one operating system to another .The attributes of a file are p-

- **Name -** The symbolic file name is the only information kept in humanreadable form. Some special significance is given to names, and particularly extensions ( .exe, .txt, etc. ).
- **Identifier** – It is a unique number, that identifies the file within the file system.
- **Type** – Type of the file like text, executable, other binary, etc.
- **Location** - . location of the file on that device.
- **Size** - The current size of the file (in bytes, words, or blocks)
- **Protection -** Access-control information ( reading, writing, executing).
- **Time, date, and user identification** –These data can be useful for protection, security, and usage monitoring.

**b) File Operations**

The operating system provides system calls to create, write, read, reposition, delete, and truncate files.

- Creating a file - Two steps are necessary to create a file
  - Find space in the file system for the file.
  - Make an entry for the new file in the directory.
- Writing a file - To write a file, the system call consists of both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
- Reading a file - To read from a file, the system call that specifies the name of the file and where the next block of the file should be put. The directory is searched for the file, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated.
- Repositioning within a file - The directory is searched for the file, and the file pointer is repositioned to a given value. This file operation is also known as a file seek.
- Deleting a file – To delete a file, search the directory for the file. Release all file space, so that it can be reused by other files, and erase the directory entry.

- Truncating a file - The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged –except for file length. The file size is reset to zero.
- Information about currently open files is stored in an **open file table**. It contains informations like:
  - **File pointer** - records the current position in the file, for the next read or write access.
  - **File-open count** - How many times has the current file been opened by different processes, at the same time and not yet closed? When this counter reaches zero the file can be removed from the table.
  - **Disk location of the file –** The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each opearation.
  - **Access rights –** The file access permissions are stored on the per-process table so that the operating system can allow or deny subsequent I/O requests.
- Some systems provide support for **file locking.**
  - A **shared lock** is for reading only.
  - A **exclusive lock** is for writing as well as reading.
  - An **advisory lock** , it is up to the software developers to ensure that locks are acquired or realeased.
  - A **mandatory lock** , prevents any other process from accessing the locked file.. ( A truly locked door. )
    - UNIX uses advisory locks, and Windows uses mandatory locks.

### c) File Types

| file type | usual extension | function |
|---|---|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, pas, asm, a | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| text | txt, doc | textual data, documents |
| word processor | wp, tex, rtf, doc | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | ps, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | arc, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, rm, mp3, avi | binary file containing audio or A/V information |

**Figure 10.2** Common file types.

- File name consists of two parts: name and extension
- The user and the operating system can identify the type of a file using the name.
- Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension. Example : resume.doc, threads.c etc.
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.
- For instance, only a file with a ".corn", ".exe", or ".bat", can be executed.

**d) File Structure**

- The study of different ways of storing files in secondary memory such that they can be easily accessed.
- File types can be used to indicate the internal structure of the file. Certain files must be in a particular structure that is understood by the operating system.
- For example, the operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and the location of the first instruction.
- UNIX treats all files as sequences of bytes, with no further consideration of the internal structure.
- Macintosh files have two **forks** - a **resource fork**, and a **data fork**. The resource fork contains information relating to the UI, such as icons and button images. The data fork contains the traditional file contents-program code or data.

**e) Internal File Structure**

- Disk systems typically have a well-defined block size determined by the size of a sector. A group of sectors form a group
- All disk I/0 is performed in units of one block, and all blocks are the same size.
- Logical records may even vary in length. Padding a number of logical records into physical blocks is a common solution to this problem.
- The packing can be done either by the user's application program or by the operating system. In either case, the file may be considered a sequence of blocks.
- All the basic I/O functions operate in terms of blocks.
- Disk space is always allocated in terms of blocks. Some portion of last block(while storing a file) is always wasted. This is called internal fragmentation.

## 10.2 Access Methods

The file information is accessed and read into computer memory. The information in the file can be accessed in several ways.

**a) Sequential Access**

- Here information in the file is processed in order, one record after the other.
- This mode of access is a common method; for example, editors and compilers usually access files in this fashion.
- A sequential access file emulates magnetic tape operation, and generally supports a few operations:
  - read next - read a record and advance the file pointer to the next position.
  - write next - write a record to the end of file and advance the file pointer to the next position.
  - skip n records - May or may not be supported. 'n' may be limited to positive numbers, or may be limited to +/- 1.

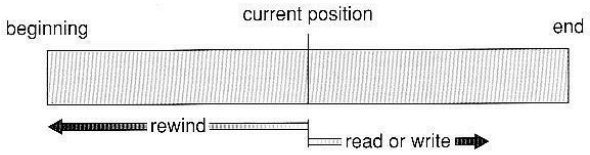



**Figure 10.3** Sequential-access file.

**b) Direct Access**

A file is made up of fixed-length logical records that allow programs to read and write records randomly. The records can be rapidly accessed in any order.

Direct access are of great use for immediate access to large amount of information.
Eg : Database file. When a query occurs, the query is computed and only the selected rows are access directly to provide the desired information.

Operations supported include:

- read n - read record number n. (position the cursor to n and then read the record)
- write n - write record number n. (position the cursor to n and then write the record)
- jump to record n – move to nth record (n- could be 0 or the end of file)
- If the record length is *L*, there is a request for record 'N'. Then the direct access to the starting byte of record 'N' is at L*(N-1)

  Eg: if 3rd record is required and length of each record(L) is 50, then the starting position of 3rd record is L*(N-1)

  Address = 50*(3-1) = 100.

**c) Other Access Methods(Indexed method)**

- These methods generally involve the construction of an ***index*** for the file called **index file**.

- The index file is like an index page of a book, which contains key and address. To find a record in the file, we first search the index and then use the pointer to access the record directly and find the desired record.
- An indexed access scheme can be easily built on top of a direct access system.
- For very large files, the index file itself is very large. The solution to this is to create an index for index file. i.e. multi-level indexing.

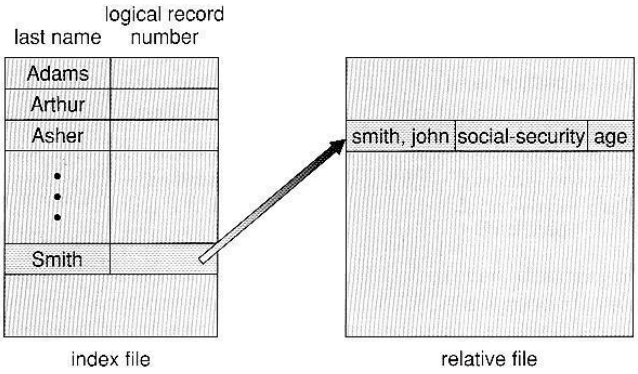



**Figure 10.5** Example of index and relative files.

## 10.3 Directory Structure

Directory is a structure which contains filenames and information about the files like location, size, type etc. The files are put in different directories. Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses.

Partitions are also known as *slices* or *minidisks*. A file system can be created on each of these parts of the disk. Any entity containing a file system is generally known as a *volume*.

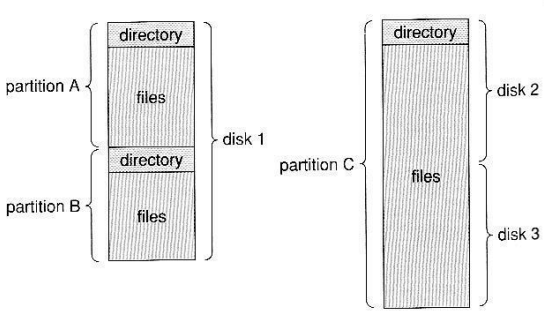



**Figure 10.6** A typical file-system organization.

**Directory Overview**

The directory can be viewed as a symbol table that translates file names into their directory entries.

Directory operations to be supported include:

- Search for a file - search a directory structure to find the entry for a particular file.
- Create a file – create new files and add to the directory
- Delete a file - When a file is no longer needed, erase it from the directory
- List a directory - list the files in a directory and the contents of the directory entry.
- Rename a file – Change the name of the file. Renaming a file may also allow its position within the directory structure to be changed.
- Traverse the file system - Access every directory and every file within a directory structure.

**Directory Structures -**

**a) Single-Level Directory**

- It is the simplest directory structure.
- All files are contained in the same directory, which is easy to support and understand.

The limitations of this structure is that -

- All files are in the same directory must have unique names.
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

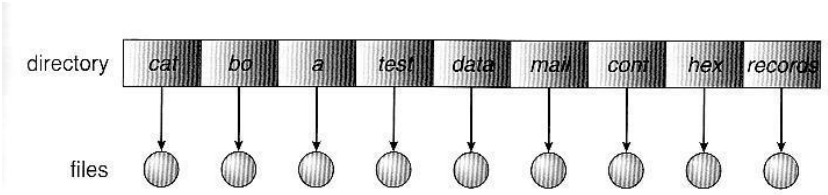



**Figure 10.7** Single-level directory.

**b) Two-Level Directory**

- Each user gets their own directory space - ***user file directory(UFD)***
- File names only need to be unique within a given user's directory.
- A **master file directory(MFD)** is used to keep track of each users directory, and must be maintained when users are added to or removed from the system.
- When a user refers to a particular file, only his own UFD is searched.
- All the files within each UFD are unique.
- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists.

- To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name. The user directories themselves must be created and deleted as necessary.
- This structure **isolates** one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files.

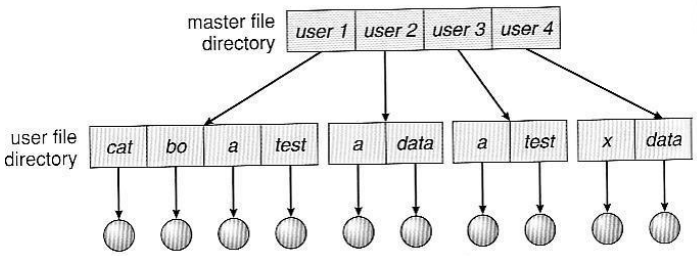



**Figure 10.8** Two-level directory structure.

## c) Tree-Structured Directories

- A tree structure is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories.
- 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* and *relative.* An absolute path begins at the root and follows a down to the specified file, giving the directory names on the path. A relative path defines a path from the current directory.
- For example, in the tree-structured file system of figure below if the current directory is *root/spell/mail*, then the relative path name is *prt/first* and the files absolute path name *root/spell/mail/prt/jirst.*

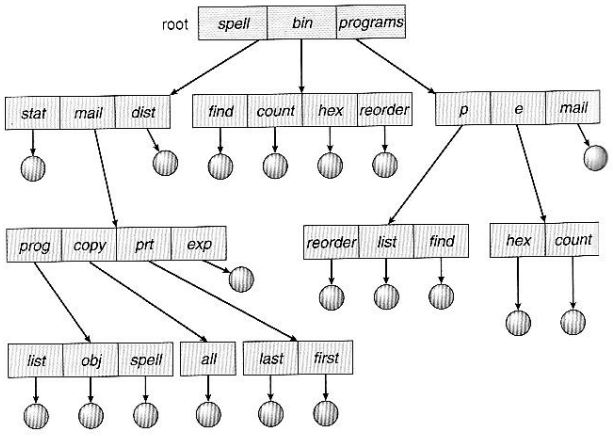



**Figure 10.9** Tree-structured directory structure.

- Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.
- One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.

**d) Acyclic-Graph Directories**

- When the same files need to be accessed in more than one place in the directory structure ( e.g. because they are being shared by more than one user), it can be useful to provide an acyclic-graph structure. ( Note the **directed** arcs from parent to child. )
  - UNIX provides two types of **links** (pointer to another file)for implementing the acyclic-graph structure.
  - A **hard link** ( usually just called a link ) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same filesystem.
  - A **symbolic link**, that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other filesystems, as well as ordinary files in the current filesystem.
- Windows only supports symbolic links, termed **shortcuts.**
- Hard links require a **reference count**, or **link count** for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.

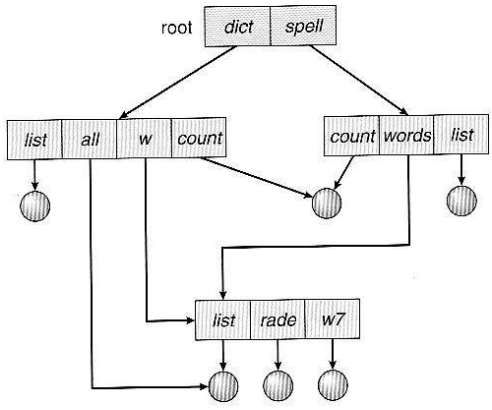



**Figure 10.10** Acyclic-graph directory structure.

- For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:
  - One option is to find all the symbolic links and adjust them also.

- Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.
- What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?

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.
When a link or a copy of the directory entry is established, a new entry is added to the file-reference list. When a link or directory entry is deleted, we remove its entry on the list. The file is deleted when its file-reference list is empty.

**e) General Graph Directory**

- If cycles are allowed in the graphs, then several problems can arise:
  - Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. (Or not to follow symbolic links, and to only allow symbolic links to refer to directories)
  - Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem. (chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either system. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely deleted. )

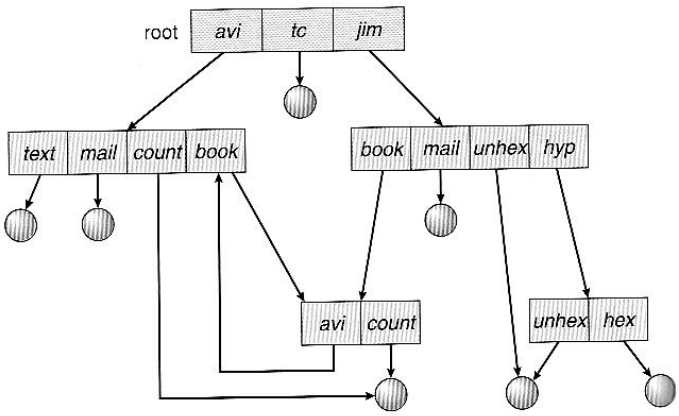



**Figure 10.11** General graph directory.

## 10.4 File-System Mounting

- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a filesystem to mount and a **mount point** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.
- Any files ( or sub-directories ) that had been stored in the mount point directory prior to mounting the new filesystem are now hidden by the mounted filesystem, and are no longer available. For this reason some systems only allow mounting onto empty directories.
- Filesystems can only be mounted by root, unless root has previously configured certain filesystems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy filesystems to /mnt or something like it) Anyone can run the mount command to see what filesystems are currently mounted.
- Filesystems may be mounted read-only, or have other restrictions imposed.

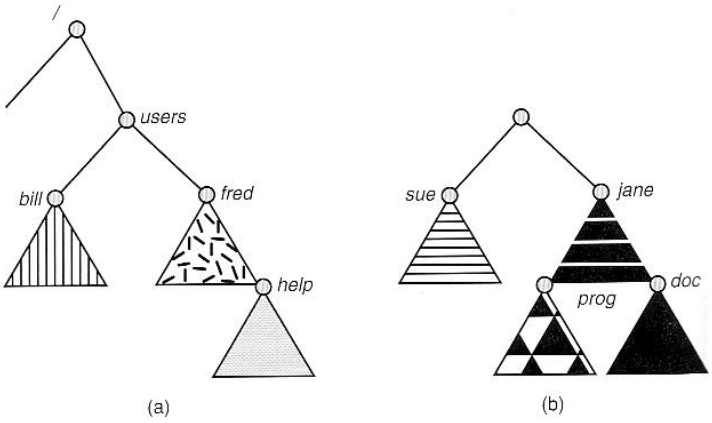



**Figure 10.12** File system. (a) Existing system. (b) Unmounted volume.

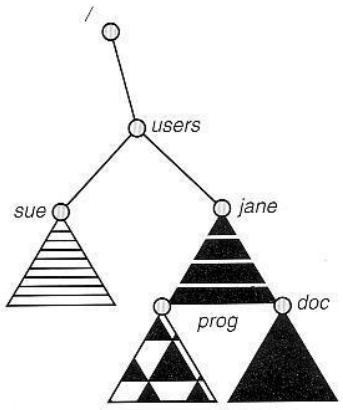



**Figure 10.13** Mount point.

The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.

- Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.
- More recent Windows systems allow filesystems to be mounted to any directory in the filesystem, much like UNIX.

## 10.5 File Sharing

### 10.5.1 Multiple Users

- On a multi-user system, more information needs to be stored for each file:
  - The owner ( user ) who owns the file, and who can control its access.
  - The group of other user IDs that may have some special access to the file.
  - What access rights are afforded to the owner ( **U**ser ), the **G**roup, and to the rest of the world ( the universe, a.k.a. **O**thers. )
  - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

### 10.5.2 Remote File Systems

- The advent of the Internet introduces issues for accessing files stored on remote computers
  - The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account and password controlled, or **anonymous**, not requiring any user name or password.
  - Various forms of **distributed file systems** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. ( The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism. )
    - The WWW has made it easy once again to access files on remote systems without mounting their filesystems, generally using ( anonymous ) ftp as the underlying file transport mechanism.

#### a) The Client-Server Model

- When one computer system remotely mounts a filesystem that is physically located on another system, the system which physically owns the files acts as a **server**, and the system which mounts them is the **client.**
- User IDs and group IDs must be consistent across both systems for the system to work properly. ( I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users. )
- The same computer can be both a client and a server. ( E.g. cross-linked file systems. )
- There are a number of security concerns involved in this model:
  - Servers commonly restrict mount permission to certain trusted systems only. Spoofing ( a computer pretending to be a different computer ) is a potential security risk.
  - Servers may restrict remote access to read-only.
  - Servers restrict which filesystems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.

- The NFS ( Network File System ) is a classic example of such a system.

**b) Distributed Information Systems**

- The **Domain Name System, DNS,** provides for a unique naming system across all of the Internet.
- Domain names are maintained by the **Network Information System, NIS**, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.
- Microsoft's **Common Internet File System, CIFS**, establishes a **network login** for each user on a networked system with shared file access. Older Windows systems used **domains**, and newer systems ( XP, 2000 ), use **active directories.** User names must match across the network for this system to be valid.
- A newer approach is the **Lightweight Directory-Access Protocol, LDAP,** which provides a **secure single sign-on** for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

**c) Failure Modes**

- When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.
- However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system ( or the network ) will come back up eventually.

### 10.5.3 Consistency Semantics

- **Consistency Semantics** deals with the consistency between the views of shared files on a networked system. When one user changes the file, when do other users see the changes?
- The series of accesses between the open() and close() operations of a file is called the **file session**.

Examples of consistency semantics -

**a) UNIX Semantics**

- The UNIX file system uses the following semantics:
  - Writes to an open file are immediately visible to any other user who has the file open.

- One implementation uses a shared location pointer, which is adjusted for all sharing users.

- There is a single copy of the file, which may delay some accesses.

**b) Session Semantics**

- The Andrew File System, AFS uses the following semantics:
  - Writes to an open file are not immediately visible to other users.
  - When a file is closed, any changes made become available only to users who open the file at a later time.
- According to these semantics, a file can be associated with multiple ( possibly different ) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.
- AFS file systems may be accessible by systems around the world. Access control is maintained through (somewhat) complicated access control lists, which may grant access to the entire world ( literally ) or to specifically named users accessing the files from specifically named remote environments.

**c) Immutable-Shared-Files Semantics**

- Under this system, when a file is declared as **shared** by its creator, then the name cannot be re-used by any other process and it cannot be modified.

## 10.6 Protection

The information in a computer system must be stored safely without any physical damage (the issue of *reliability)* and improper access (the issue of *protection).*

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically copy disk files to tape at regular intervals (once per day or week or month). The damage could be due to hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism.

### 10.6.1 Types of Access

Systems that do not permit access to the files of other users do not need protection. Thus, we could provide complete protection by prohibiting access. Alternatively, we could provide free access with no protection. Several different types of operations may be controlled:

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

### 10.6.2 Access Control

To make access to files depending on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

- Constructing such a list may be a tedious, if we do not know in advance the list of users in the system.
- The directory entry, must be of variable size, as the list grows, resulting in more complicated space management.

These problems can be resolved by use of a condensed version of the access list. 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.

The most common recent approach is to combine access-control lists with owner, group, and universe access control scheme.

### 10.6.3 Other Protection Approaches

Associate a password with each file.
Using of password is effective, but has a few disadvantages:

- The number of passwords that a user needs to remember maybe large
- If one password is used, then once the password is discovered, all the files can be accessed.

Some system allow users to associate a password to a subdirectory, rather than only to file.

## 10.7 File-System Structure

Disks provide the bulk of secondary storage on which a file system is maintained. The two characteristics that make them a convenient medium for storing multiple files:

1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
2. A disk can access directly any given block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of *blocks*. Block sizes may range from 512 bytes to 4K or larger.( Rather than transferring a byte at a time,)
- A file system poses two quite different design problems.
  - The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
  - The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

- File systems organize storage on disk drives, and can be viewed as a layered design:
  - At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
  - Lowest level***, I/O Control*** consists of ***device drivers***, which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card ( device ) on a system has a different set of addresses ( registers, ***ports*** ) that it listens to, and a unique set of command codes and results codes that it understands.
  - The ***basic file system*** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block.
  - The ***file organization module*** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
  - The ***logical file system*** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the meta data as well as block number information for finding the data on the disk.
- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be filesystem specific.

- When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems.
- Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 .

**10.8 File-System Implementation**

On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

File systems store several important data structures on the disk:

- A ***boot-control bloc*k**, ( per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume In UFS, it is called the **boot block**; in NTFS, it is the partition **boot sector.**
- A ***volume control block,*** ( per volume ) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, freeblock count and free-block pointers, and free FCB count and FCB pointers. In UFS, this is called a **superblock;** in NTFS, it is stored in. the **master file table.**
    - A directory structure ( per file system ), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a ***master file table.***
    - The ***File Control Block, FCB,*** ( per file ) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

- There are also several key data structures stored in memory:
  - An in-memory mount table contains information about each mounted volume..
  - An in-memory directory cache of recently accessed directoryinformation.
  - ***A system-wide open file table***, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
  - ***A per-process open file table,*** containing a pointer to the system open file table as well as some other information. ( For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not. )

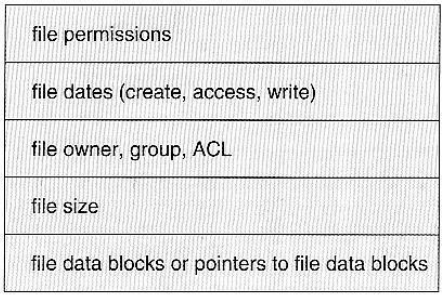



**Figure 11.2** A typical file-control block.

- Figure 11.3 illustrates some of the interactions of file system components when files are created and/or used:
  - When a new file is created, a new FCB is allocated and filled out with important information regarding the new file.
  - When a file is accessed during a program, the open( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a ***file descriptor***, and Windows refers to it as a ***file handle***.
  - If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
  - When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

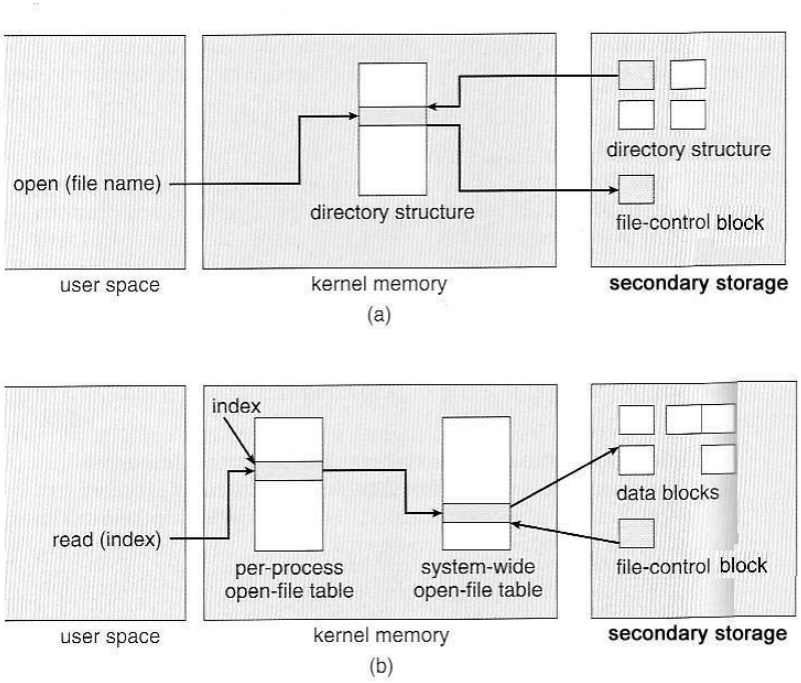



**Figure 11.3** In-memory file-system structures. (a) File open. (b) File read.

### 10.8.1 Partitions and Mounting

- Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.
- Partitions can either be used as raw devices ( with no structure imposed upon them ), or "cooked;' containing a file system. they can be formatted to hold a filesystem ( i.e. populated with FCBs and initial directory structures as appropriate. ) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have file-system device drivers loaded and therefore cannot interpret the file-system format.
- The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. Modern boot programs understand multiple OSes and filesystem formats, and can give the user a choice of which of several available systems to boot.
- The ***root partition*** contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. ( Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach.

Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary. )

- Continuing with the boot process, additional filesystems get mounted, adding their information into the appropriate mount table structure. As a part of the mounting process the file systems may be checked for errors or inconsistencies, either because they are flagged as not having been closed properly the last time they were used, or just for general principals. Filesystems may be mounted either automatically or manually. In UNIX a mount point is indicated by setting a flag in the in-memory copy of the inode, so all future references to that inode get re-directed to the root directory of the mounted filesystem.

### 10.8.2 Virtual File Systems

- ***Virtual File Systems, VFS***, provide a common interface to multiple different filesystem types. In addition, it provides for a unique identifier ( vnode ) for files across the entire space, including across all filesystems of different types. (UNIX inodes are unique only across a single filesystem, and certainly do not carry across networked file systems.)
- The VFS in Linux is based upon four key object types:
  - The ***inode*** object, representing an individual file
  - The ***file*** object, representing an open file.
  - The ***superblock*** object, representing a filesystem.
  - The ***dentry*** object, representing an individual directory entry.
- Linux VFS provides a set of common functionalities for each filesystem, using function pointers accessed through a table. The same functionality is accessed through the same table position for all filesystem types, though the actual functions pointed to by the pointers may be filesystem-specific. Common operations provided include open( ), read( ), write( ), and mmap( ).

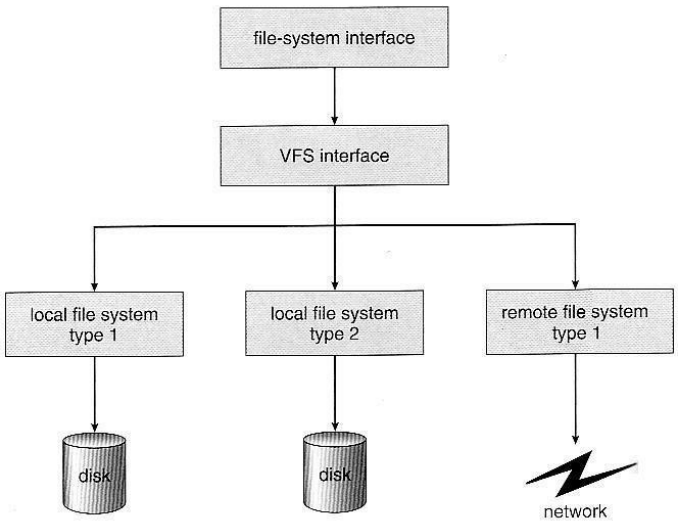



**Figure 11.4** Schematic view of a virtual file system.

Figure 11.4. The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors. The second layer is called the virtual file system (VFS) layer; it serves two important functions:

1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally.
2. The VFS provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode, that contains a numerical designator for a network-wide unique file. (UNIX inodes are unique within only a single file system.) This network-wide uniqueness is required for support of network file systems.

The kernel maintains one vnode structure for each active node (file or directory).

## 10.9 Directory Implementation

The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system. Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

### a) Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- The disadvantage of a linear list of directory entries is that finding a file requires a linear search.
- To overcome this, a software cache is implemented to store the recently accessed directory structure.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files,
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

### b) Hash Table

- With 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.

- Here **collisions** may occur. Collision is the situation where two file names hash to the same location.
- Alternatively, a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.
- The major disadvantage with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63, for instance, by using the remainder of a division by 64. If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.

## 10.10 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. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed.

### a) Contiguous Allocation

- ***Contiguous Allocation*** requires that all blocks of a file be kept together contiguously.
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.
- Storage allocation is done by using one of the algorithms(first fit, best fit, worst fit).
- The allocation of blocks contiguous leads to external fragmentation.
- Problems can arise when files grow, or if the exact size of a file is unknown at creation time:
  - Over-estimation of the file's final size increases external fragmentation and wastes disk space.
  - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.
  - If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.
- A variation is to allocate file space in large contiguous chunks, called ***extents.*** When a file outgrows its original extent, then an additional one block is allocated. A pointer points from last block of contiguous memory allocation to the extended chunk.

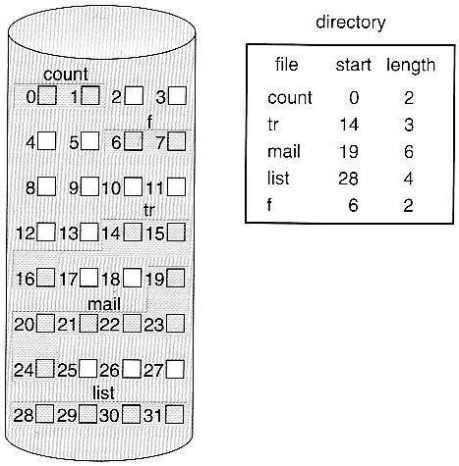



**Figure 11.5** Contiguous allocation of disk space.

## b) Linked Allocation

- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. ( E.g. a block may be 508 bytes instead of 512. )
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.
- Allocating ***clusters*** of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

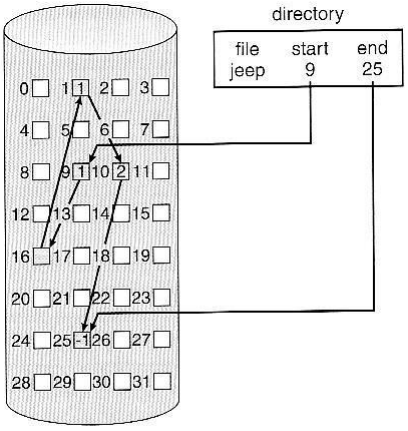



**Figure 11.6** Linked allocation of disk space.

- The ***File Allocation Table, FAT,*** used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.

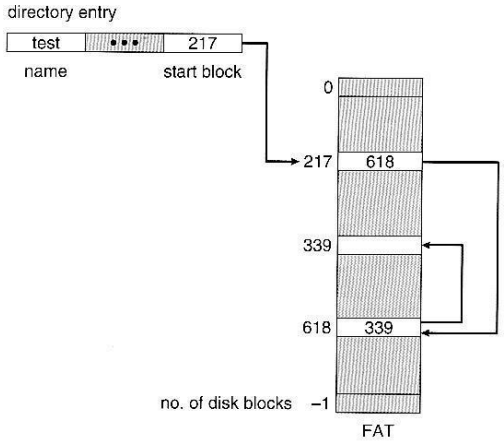



**Figure 11.7** File-allocation table.

## c) Indexed Allocation

- ***Indexed Allocation*** combines all of the indexes(block numbers) for accessing each file into a common block ( for that file ).
- Each file will have a common block called the index block.

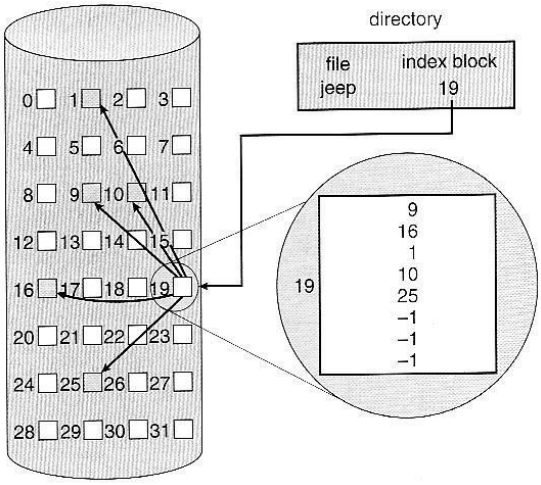



**Figure 11.8** Indexed allocation of disk space.

- Some disk space is wasted ( relative to linked lists or FAT tables ) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:

- **Linked Scheme** - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.
- **Multi-Level Index** - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
- **Combined Scheme** - This is the scheme used in UNIX inodes, in which the first 12 entries data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. The advantage of this scheme is that for small files ( files stored in less than 12 blocks ), the data blocks are readily accessible ( up to 48K with 4K block sizes ); files up to about 4144K ( using 4K blocks ) are accessible with only a single indirect block ( which can be cached ), and huge files are still accessible using a relatively small number of disk accesses ( larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers. )

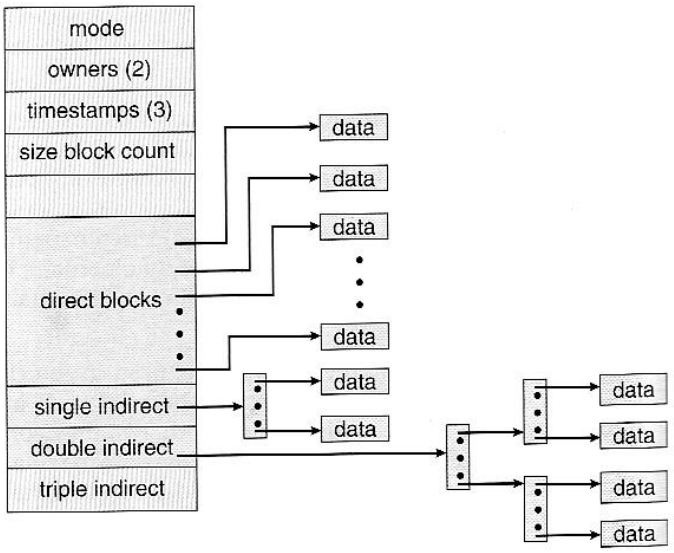



**Figure 11.9** The UNIX inode.

## Performance

- The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.
- Some systems support more than one allocation method, which may require specifying how the file is to be used ( sequential or random access ) at the time it is allocated. Such systems also provide conversion utilities.
- Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.

- And of course some systems adjust their allocation schemes ( e.g. block sizes ) to best match the characteristics of the hardware for optimum performance.

## 10.11 Free-Space Management:

The space created after deleting the files can be reused. Another important aspect of disk management is keeping track of free space in memory. The list which keeps track of free space in memory is called the free-space list. To create a file, search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free-space list, is implemented in different ways as explained below.

### a) Bit Vector

- Fast algorithms exist for quickly finding contiguous blocks of a given size
- One simple approach is to use a ***bit vector***, in which each bit represents a disk block, set to 1 if free or 0 if allocated.

For example, consider a disk where blocks 2,3,4,5,8,9, 10,11, 12, 13, 17and 18 are free, and the rest of the blocks are allocated. The free-space bit map would be

0011110011111100011

- Easy to implement and also very efficient in finding the first free block or 'n' consecutive free blocks on the disk.
- The down side is that a 40GB disk requires over 5MB just to store the bitmap.

### b) Linked List

- A linked list can also be used to keep track of all free blocks.
- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
- The FAT table keeps track of the free list as just one more linked list on the table.

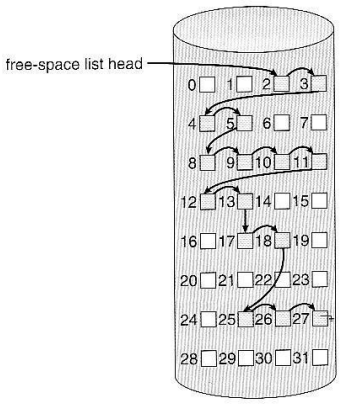



**Figure 11.10** Linked free-space list on disk.

**c) Grouping**

- A variation on linked list free lists. It stores the addresses of n free blocks in the first free block. The first n-1 blocks are actually free. The last block contains the addresses of another n free blocks, and so on.
- The address of a large number of free blocks can be found quickly.

**d) Counting**

- When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks.
- Rather than keeping al list of n free disk addresses, we can keep the address of first free block and the number of free contiguous blocks that follow the first block.
- Thus the overall space is shortened. It is similar to the extent method of allocating blocks.

**e) Space Maps ( New )**

- Sun's ZFS file system was designed for huge numbers and sizes of files, directories, and even file systems.
- The resulting data structures could be inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into ( hundreds of) ***metaslabs*** of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

# Module V Secondary - Storage Structure

Magnetic disks provide a bulk of secondary storage. Disks come in various sizes and speed. Here the information is stored magnetically. Each disk platter has a flat circular shape like CD. The two surfaces of a platter are covered with a magnetic material. The surface of a platter is logically divided into circular **tracks,** which are subdivided into **sectors.** Sector is the basic unit of storage. The set of tracks that are at one arm position makes up a **cylinder.**

The number of cylinders in the disk dirve equals the number of tracks in each platter. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of disk drives is measured in gigabytes.

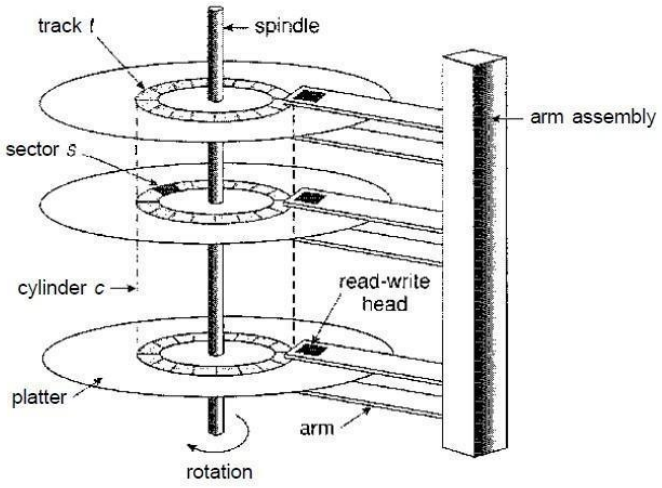



**Figure 12.1** Moving-head disk mechanism.

The head moves from the inner track of the disk to the outer track. When the disk drive is operating the disks is rotating at a constant speed.

To read or write the head must be positioned at the desired track and at the beginning of the desired sector on that track.

- Seek Time:-Seek time is the time required to move the disk arm to the required track.
- Rotational Latency(Rotational Delay):-Rotational latency is the time taken for the disk to rotate so that the required sector comes under the r/w head.
- Positioning time or random access time is the summation of seek time and rotational delay.
- Disk Bandwidth:-Disk bandwidth is the total number of bytes transferred divided by total time between the first request for service and the completion of last transfer.
- Transfer rate is the rate at which data flow between the drive and the computer.

As the disk head flies on an extremely thin cushion of air, the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash.**

**Magnetic Tapes**

Magnetic tape is a secondary-storage medium. It is a permanent memory and can hold large quantities of data. The time taken to access data (access time) is large compared with that of magnetic disk, because here data is accessed sequentially. When the nth data has to be read, the tape starts moving from first and reaches the nth position and then data is read from nth position. It is not possible to directly move to the nth position. So tapes are used mainly for backup, for storage of infrequently used information.

**DISK STRUCTURE**

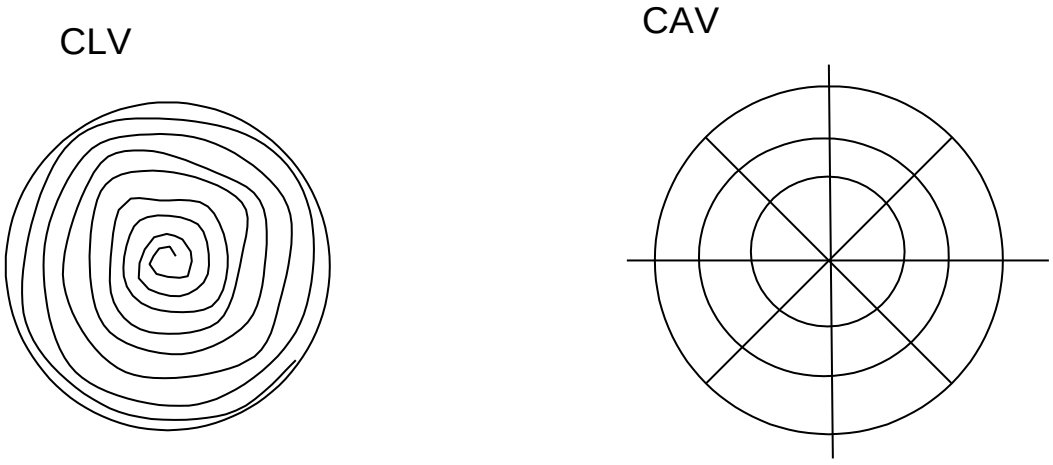
Each disk platter is divided into number of tracks and each track is divided into number of sectors. Sectors is the basic unit for read or write operation in the disk.

Modern disk drives are addressed as a large one-dimensional array. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

The disk structure (architecture) can be of two types –

i) **Constant Linear Velocity** (CLV)
ii) **Constant Angular Velocity** (CAV)

i) CLV - The density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. This architecture is used in CD-ROM and DVD-ROM.
ii) CAV – There is same number of sectors in each track. The sectors are densely packed in the inner tracks. The density of bits decreases from inner tracks to outer tracks to keep the data rate constant.



**DISK ATTACHMENT**

Computers can access data in two ways.

i) via I/O ports (or host-attached storage)
ii) via a remote host in a distributed file system( or network-attached storage)

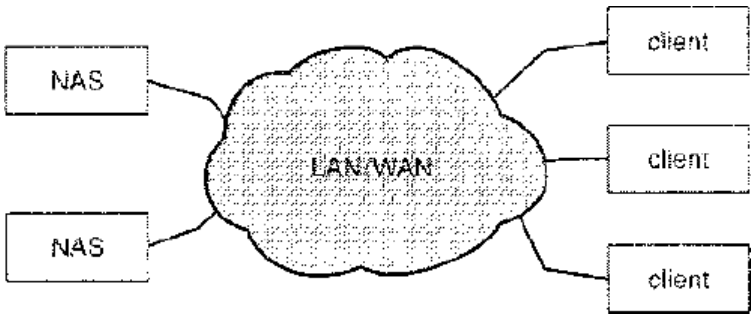
**i) Host-Attached Storage**

Host-attached storage is storage accessed through local I/O ports. Example : the typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. The other cabling systems are – **SATA**(Serially Attached Technology Attachment), **SCSI**(Small Computer System Interface) and **fiber channel** (FC).

SCSI is a bus architecture. Its physical medium is usually a ribbon cable. FC is a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. An improved version of this architecture is the basis of **storage-area networks** (SANs).

**ii) Network-Attached Storage**

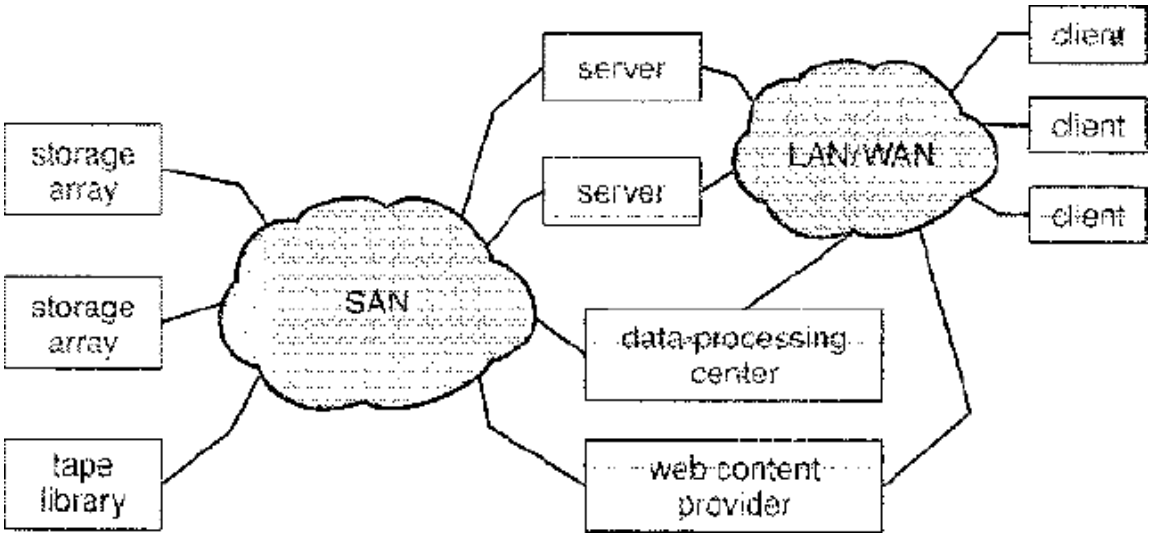
A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a network as shown in the figure. Clients access network-attached storage via a remote-procedure-call interface. The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—-usually the same local-area network (LAN) carries all data traffic to the clients.




Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage files.

**ii) Storage Area Network(SAN)**

A storage-area network (SAN) is a private network connecting servers and storage units. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. A SAN switch allows or prohibits access between the hosts and the storage. Fiber Chanel is the most common SAN interconnect.

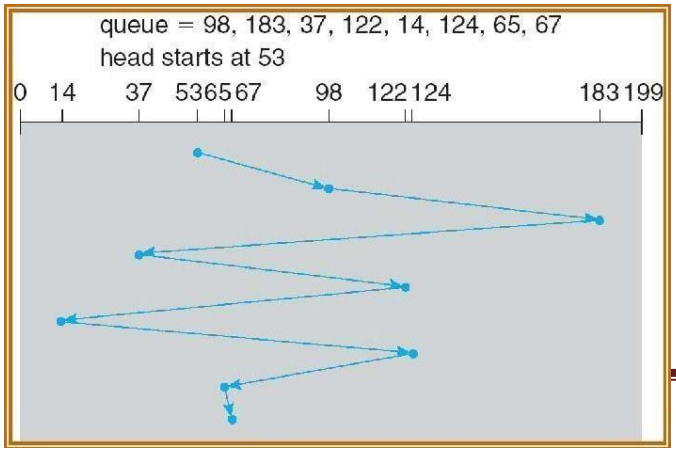



**DISK SCHEDULING**

Different types of disk scheduling algorithms are as follows:

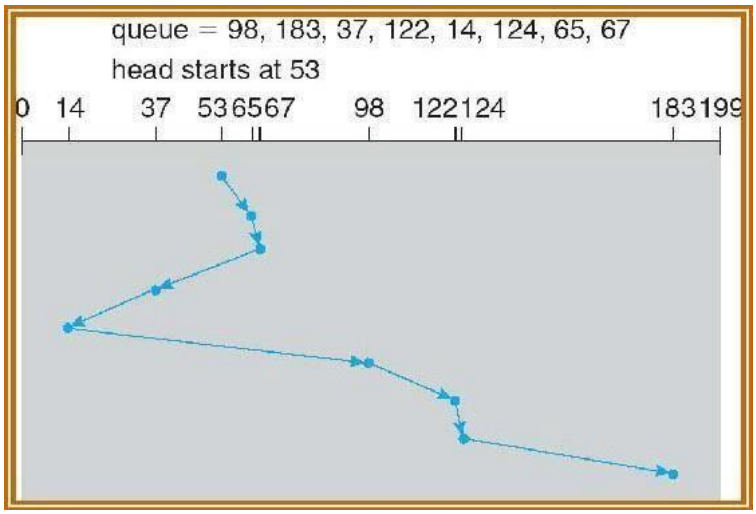
- FCFS (First Come First Serve)
- SSTF(Shortest Seek Time First)
- SCAN (Elecvator)
- C-SCAN
- LOOK
- C-LOOK

i) **FCFS scheduling algorithm**: This is the simplest form of disk scheduling algorithm. This services the request in the order they are received. This algorithm is fair but do not provide fastest service. It takes no special care to minimize the overall seek time. *Eg:*-consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67




If the disk head is initially at 53, it will first move from 53 to 98 then to 183 and then to 37, 122, 14, 124, 65, 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.

i) **SSTF** ( Shortest Seek Time First) **algorithm**: This selects the request with minimum seek time from the current head position. SSTF chooses the pending request closest to the current head position. *Eg*:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

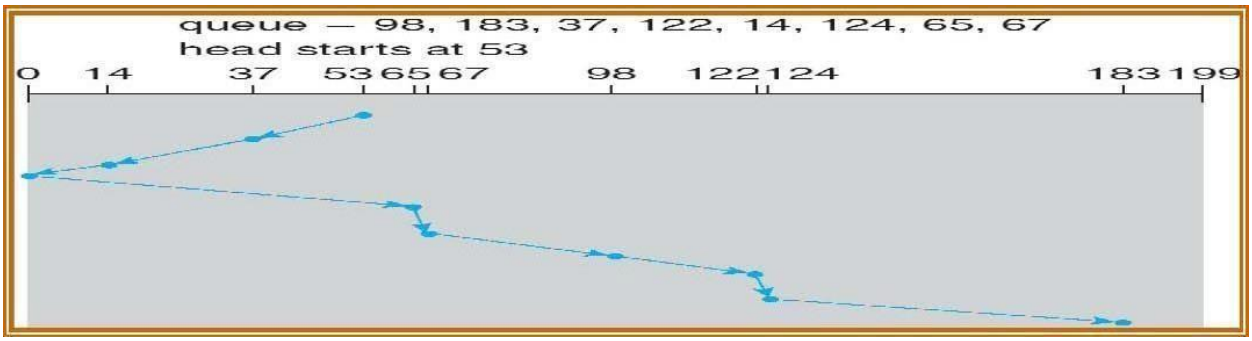



If the disk head is initially at 53, the closest is at cylinder 65, then 67, then 37 is closer then 98 to 67. So it services 37, continuing we service 14, 98, 122, 124 and finally 183. The total head movement is only 236 cylinders. SSTF is a substantial improvement over FCFS, it is not optimal.

ii) **SCAN algorithm**: In this the disk arm starts moving towards one end, servicing the request as it reaches each cylinder until it gets to the other end of the disk. At the other end, the direction of the head movement is reversed and servicing continues. The initial direction is chosen depending upon the direction of the head.

*Eg*:-:-consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and183. At cylinder 199 the arm will reverse and will move towards the other end of the disk servicing 37 and then 14. The SCAN is also called as **elevator** algorithm.

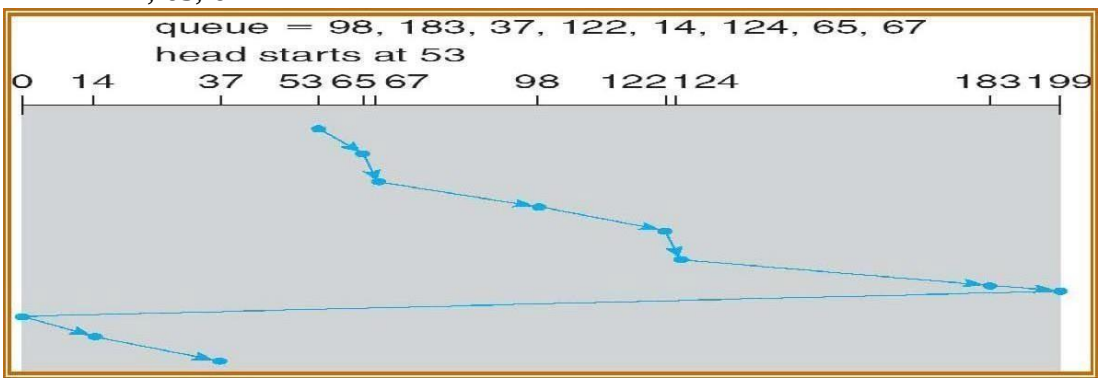



If the disk head is initially at 53 and if the head is moving towards **0th track**, it services 37 and then 14. At cylinder 0 the arm will reverse and will move towards the other end of the disk servicing 65, 67, 98, 122, 124 and 183.

iv) **C-SCAN** (Circular scan) **algorithm**:

C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from end of the disk to the other servicing the request along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any request on the return.

*Eg*:- consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67




If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At cylinder 199 the arm will reverse and will move immediately towards the other end of the disk, then changes the direction of head and serves 14 and then 37.

Note: If the disk head is initially at 53 and if the head is moving towards track 0, it services 37 and 14 first. At cylinder 0 the arm will reverse and will move immediately towards the other end of the disk servicing 65, 67, 98, 122, 124 and183.

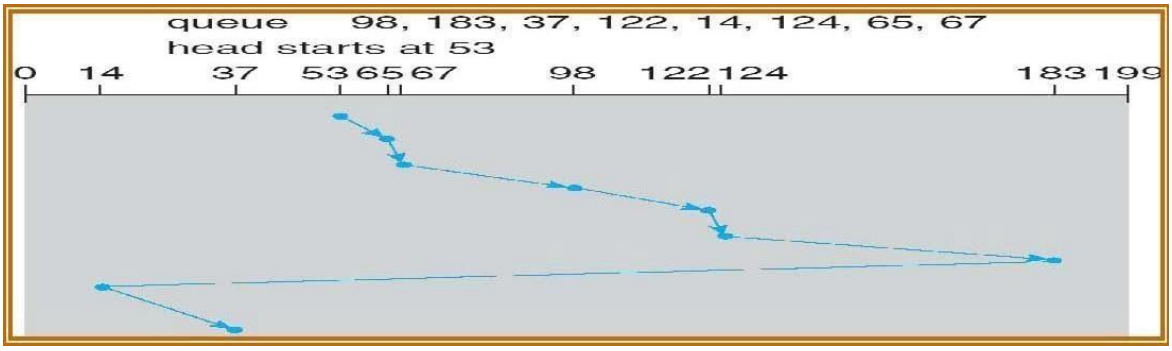
v) **Look Scheduling algorithm**:

Look and C-Look scheduling are different version of SCAN and C-SCAN respectively. Here the arm goes only as far as the final request in each direction. Then it reverses, without going all the way to the end of the disk. The Look and C-Look scheduling look for a request before continuing to move in a given direction.

*Eg*:-:-consider a disk queue with request for i/o to blocks on cylinders. 98, 183, 37, 122, 14, 124, 65, 67

If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and183. At the final request 183, the arm will reverse and will move towards the first request 14 and then serves 37.

**vi) C-Look Scheduling algorithm:**




If the disk head is initially at 53 and if the head is moving towards the outer track, it services 65, 67, 98, 122, 124 and 183. At the last request, the arm will reverse and will move immediately towards the first request 14 and then serves 37.

**Selection of a Disk-Scheduling Algorithm**
SSTF is commonly used and it increases performance over FCFS.
SCAN and C-SCAN algorithm is better for a heavy load on disk.
SCAN and C-SCAN have less starvation problem.
Disk scheduling algorithm should be written as a separate module of the operating system.
SSTF or Look is a reasonable choice for a default algorithm.

SSTF is commonly used algorithms has it has a less seek time when compared with other algorithms. SCAN and C-SCAN perform better for systems with a heavy load on the disk, (ie. more read and write operations from disk).

Selection of disk scheduling algorithm is influenced by the file allocation method, if contiguous file allocation is choosen, then FCFS is best suitable, because the files are stored in contiguous blicks and there will be limited head movements required. A linked or indexed file, in contrast, may include blocks that are widely scattered on the disk, resulting in greater head movement.

The location of directories and index blocks is also important. Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently. Suppose that a directory entry is on the first cylinder and a file's data are on the final cylinder. The disk head has to move the entire width of the disk. If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width. Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.

Because of these complexities, the disk-scheduling algorithm is very important and is written as a separate module of the operating system.

**Disk Management**

**Disk Formatting**
The process of dividing the disk into sectors and filling the disk with a special data structure is called **low-level formatting**. Sector is the smallest unit of area that is read/written by the disk controller. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size) and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an error-correcting code (ECC).

When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When a sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.

Most hard disks are low-level-forniatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk.

When the disk controller is instructed for low-level-formatting of the disk, the size of datablock of all sector sit can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is of sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track; but it also means that fewer headers and trailers are written on each track and more space is available for user data.

The operating system needs to record its own data structures on the disk. It does so in two steps. **partition** and **logical formatting.**
**Partition** – is to partition the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system's executable code, while another holds user files.
**logical formatting** (or creation of a file system) - Now, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space (a FAT or modes) and an initial empty directory.

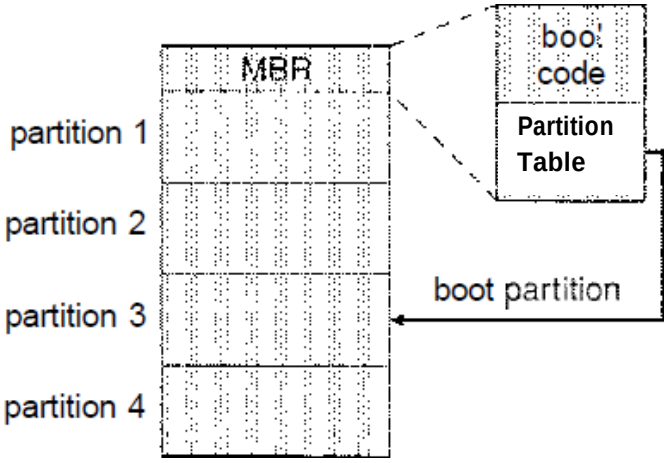
To increase efficiency, most file systems group blocks together into larger chunks, frequently called **clusters.**

**Boot Block**
When a computer is switched on or rebooted—it must have an initial program to run. This is called the *bootstrap* program. The bootstrap program –

- ↓ initializes the CPU registers, device controllers, main memory, and then starts the operating system.
- ↓ Locates and loads the operatingsystem from the disk
- ↓ jumps to beginning the operating-system execution.

The bootstrap is stored in read-only memory (ROM). Since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM, hardware chips. So most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in "the boot blocks" at a fixed location on the disk. A disk that has a boot partition is called a **boot disk** or **system disk.**

The Windows 2000 system places its boot code in the first sector on the hard disk (**master boot record,** or MBR). The code directs the system to read the boot code from, the MBR. In addition to containing boot code, the MBR contains a table listing the partitions for the hard disk and a flag indicating which partition the system is to be booted from.




**Bad Blocks**

Disk are prone to failure of sectors due to the fast movement of r/w head. Sometimes the whole disk will be changed. Such group of sectors that are defective are called as bad blocks. Different ways to overcome bad blocks are -

- ↓ Some bad blocks are handled manually, eg. In MS-DOS.
- ↓ Some controllers replace each bad sector logically with one of the spare sectors(extra sectors). The schemes used are **sector sparing** or **forwarding** and **sector slipping**.

In MS-DOS format command, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block.

In SCSI disks , bad blocks are found during the low-level formatting at the factory and is updated over the life of the disk. Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding.**

A typical bad-sector transaction might be as follows:

- The operating system tries to read logical block 87.
- The controller finds that the sector is bad. It reports this finding to the operating system.
- The next time the system is rebooted, a special, command is run to tell the SCSI controller to replace the bad sector with a spare.
- After that, whenever the system requests logical block 87, the request is translated into the replacement sector's (spare) address by the controller.

Some controllers replace bad blocks by **sector slipping**. Here is an example: Suppose that logical block 17 becomes defective and the first available spare follows sector 202. Then, sector slipping remaps all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 is copied into the spare, then sector 201 into 202, 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.

## SWAP SPACE MANAGEMENT

The amount of swap space needed on a system can vary depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used. It can range from a few megabytes of disk space to gigabytes.

The swap space can overestimated or underestimated. It is safer to overestimate than to underestimate the amount of swap space required. If a system runs out of swap space due to underestimation of space, it may be forced to abort processes or may crash entirely. Overestimation wastes disk space that could otherwise be used for files, but it does no other harm.

### Swap-Space Location

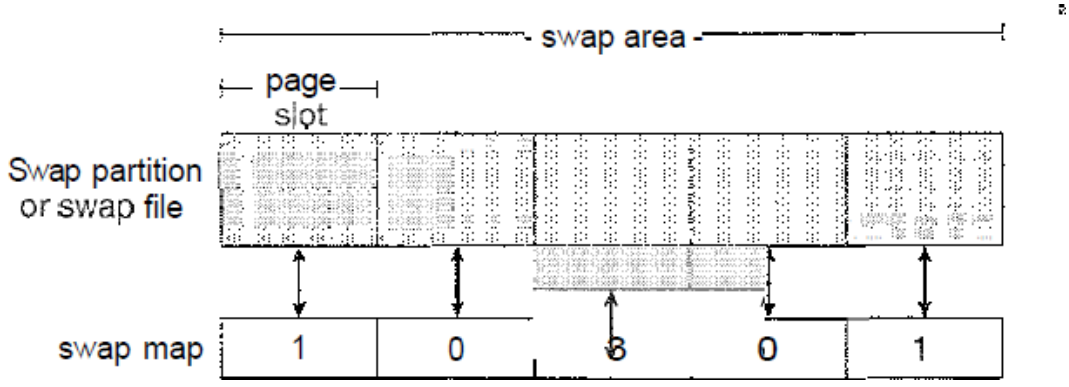
A swap space can reside in one of two places: It can be carved out of the normal **file system**, or it can be in a separate **disk partition**. If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image. We can improve performance by caching the block location information in physical memory.

Alternatively, swap space can be created in a separate raw partition. A separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.

### Swap-Space Management: An Example

Solaris allocates swap space only when a page is forced out of physical memory, rather than when the virtual memory page is first created.

Linux is similar to Solaris in that swap space is only used for anonymous memory or for regions of memory shared by several processes. Linux allows one or more swap areas to be



established. A swap area may be in either a swap file on a regular file system or a raw swap partition. Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages. Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the swap area. If the value of a counter is 0, the corresponding page slot is available. Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page; for example, a value of 3 indicates that the swapped page is mapped to three different processes. The data structures for swapping on Linux systems are shown in below figure.

# PROTECTION:

## GOALS OF PROTECTION

Protection is a mechanism for **controlling the access** of programs, processes, or users to the resources defined by a computer system. Protection ensures that only processes that have gained proper authorization from the operating system can operate on the files, memory segments, CPU, and other resources of a system.

Protection is required to prevent mischievous, intentional violation of an access restriction by a user.

## PRINCIPLES OF PROTECTION

A key, time-tested guiding principle for protection is the 'principle of least privilege'. It dictates that programs, users, and even systems be given just enough privileges to perform their tasks. An operating system provides mechanisms to enable privileges when they are needed and to disable them when they are not needed.

## DOMAIN OF PROTECTION

A computer system is a collection of processes and objects. O*bjects* are both **hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives) and **software objects** (such as files, programs, and semaphores). Each object (resource) has a unique name that differentiates it from all other objects in the system.

The operations that are possible may depend on the object. For example, a CPU can only be executed on. Memory segments can be read and written, whereas a CD-ROM or DVD-ROM can only be read. Tape drives can be read, written, and rewound. Data files can be created, opened, read, written, closed, and deleted; program files can be read, written, executed, and deleted.

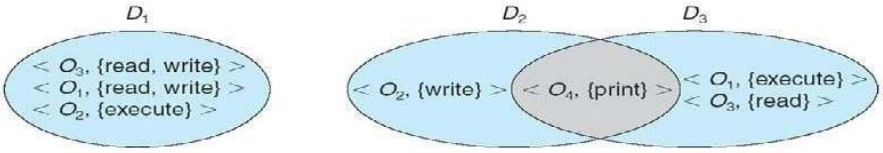
A process should be allowed to access only those resources

a) for which it has authorization
b) currently requires to complete process

**Domain Structure**

A domain is a set of objects and types of access to these objects. Each domain is an ordered pair of <object-name,rights-set>. Example, if domain D has the access right <*file F*, {read,write}>, then all process executing in domain D can both read and write file *F*, and cannot perform any other operation on that object.

Domains do not need to be disjoint; they may share access rights. For example, in below figure, we have three domains: D1 D2, and D3. The access right < *O4*, (print}> is shared by D2 and *D3,it* implies that a process executing in either of these two domains can print object O4.




A domain can be realized in different ways, it can be a user, process or a procedure.
ie. each user as a domain,
each process as a domain or
each procedure as a domain.

**ACCESS MATRIX**

Our model of protection can be viewed as a matrix, called an **access matrix.** It is a general model of protection that provides a mechanism for protection without imposing a particular protection policy. The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. The entry access(i,j) defines the set of operations that a process executing in domain *Di* can invoke on object *Oj*.

| object / domain | $F_1$ | $F_2$ | $F_3$ | printer |
|---|---|---|---|---|
| $D_1$ | read | | read | |
| $D_2$ | | | | print |
| $D_3$ | | read | execute | |
| $D_4$ | read write | | read write | |

In the above diagram, there are four domains and four objects—three files (F1, F2, F3) and one printer. A process executing in domain D1 can read files F1 and F3. A process executing in domain D4 has the same privileges as one executing in domain *D1;* but in addition, it can also write onto files F1 and F3.
When a user creates a new object Oj, the column Oj is added to the access matrix with the appropriate initialization entries, as dictated by the creator.

The process executing in one domain and be **switched** to another domain. When we switch a process from one domain to another, we are executing an operation (**switch**) on an object (the domain). Domain switching from domain Di to domain *Dj* is allowed if and only if the access right switch € access(i,j). Thus, in the given figure, a process executing in domain D2 can switch to domain D3 or to domain D4. A process in domain D4 can switch to D1, and one in domain *D1* can switch to domain D2.

| object / domain | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch |
| $D_3$ | | read | execute | | | | | |
| $D_4$ | read write | | read write | | switch | | | |

Allowing controlled change in the contents of the access-matrix entries requires three additional operations: **copy, owner, and control**.

| object / domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | execute | | write* |
| $D_2$ | execute | read* | execute |
| $D_3$ | execute | | |

(a)

| object / domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | execute | | write* |
| $D_2$ | execute | read* | execute |
| $D_3$ | execute | read | |

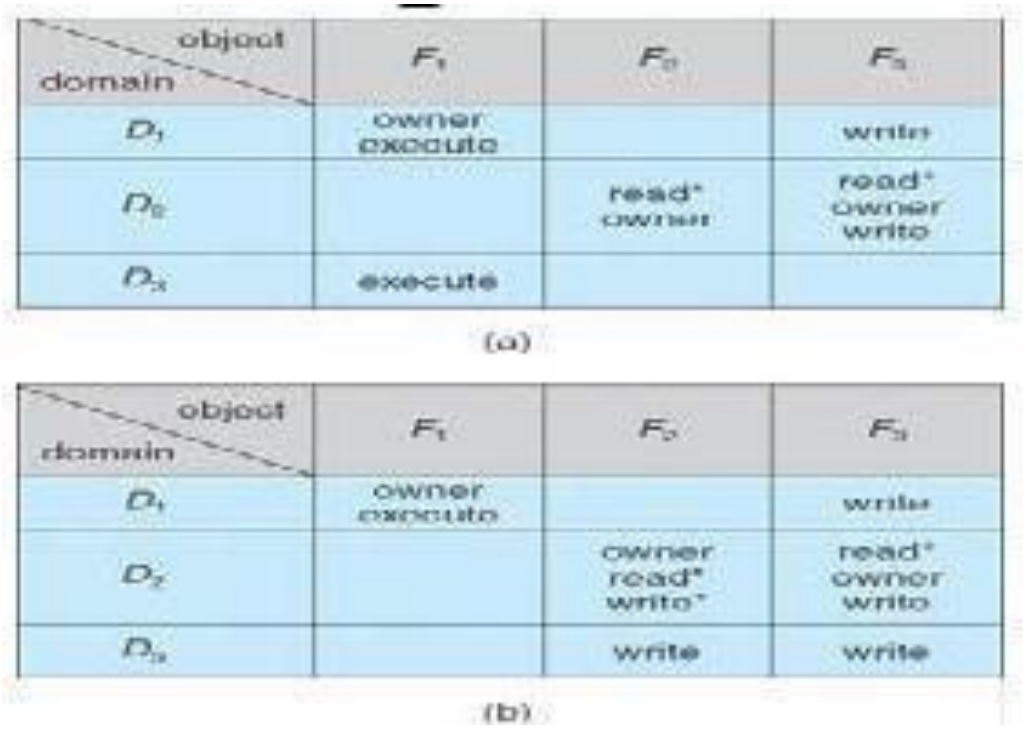
**(b)**

The ability to **copy** an access right from one domain (or row) of the access matrix to another is denoted by an asterisk (*) appended to the access right. The *copy* right allows the copying of the access right only within the column for which the right is defined. In the below figure, a process executing in domain *D2* can copy the read operation into any entry associated with file F2. Hence, the access matrix of figure (a) can be modified to the access matrix shown in figure (b).

This scheme has two variants:

1) A right is copied from access(i,j) to access(k,j); it is then removed from access(i,j). This action is a *transfer* of a right, rather than a copy.
2) Propagation of the *copy* right- limited copy. Here, when the right *R** is copied from access(i,j) to access(k,j), only the right *R* (not R*) is created. A process executing in domain *Dk* cannot further copy the right *R.*

We also need a mechanism to allow **addition** of new rights and **removal** of some rights. The ***owner*** right controls these operations. If access(i,j) includes the ***owner*** right, then a process executing in domain Di, can add and remove any right in any entry in column j. For example, in below figure (a), domain D1 is the owner of F1, and thus can add and delete any valid right in column F1. Similarly, domain *D2* is the owner of *F2* and F3 and thus can add and remove any valid right within these two columns. Thus, the access matrix of figure(a) can be modified to the access matrix shown in figure(b) as follows.



| object / domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | owner execute | | write |
| $D_2$ | | read* owner | read* owner write |
| $D_3$ | execute | | |

(a)

| object / domain | $F_1$ | $F_2$ | $F_3$ |
|---|---|---|---|
| $D_1$ | owner execute | | write |
| $D_2$ | | owner read* write* | read* owner write |
| $D_3$ | | write | write |

(b)

A mechanism is also needed to change the entries in a row. If access(i,j) includes the ***control*** right, then a process executing in domain *Di*, can remove any access right from row j. For example, in figure, we include the *control* right in access(D3, D4). Then, a process executing in domain D3 can modify domain D4.

| object / domain | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch |
| $D_3$ | | read | execute | | | | | Control |
| $D_4$ | read write | | read write | | switch | | | |

| object / domain | $F_1$ | $F_2$ | $F_3$ | laser printer | $D_1$ | $D_2$ | $D_3$ | $D_4$ |
|---|---|---|---|---|---|---|---|---|
| $D_1$ | read | | read | | | switch | | |
| $D_2$ | | | | print | | | switch | switch control |
| $D_3$ | | read | execute | | | | | |
| $D_4$ | write | | write | | switch | | | |

**Implementation of Access Matrix**

Different methods of implementing the access matrix (which is sparse).

- Global Table
- Access Lists for Objects
- Capability Lists for Domains
- Lock-Key Mechanism

**Global Table**

This is the simplest implementation of access matrix. A set of ordered triples <*domain*, *object*, *rights-set*> is maintained in a file. Whenever an operation *M* is executed on an object Oj, within domain *Di*, the table is searched for a triple <*Di*, Oj, *Rk*>. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

Drawbacks -

The table is usually large and thus cannot be kept in main memory.

Additional I/O is needed

**Access Lists for Objects**

Each column in the access matrix can be implemented as an access list for one object. The empty entries are discarded. The resulting list for each object consists of ordered pairs *<domain, rights-set>*. It defines all domains access right for that object. When an operation M is executed on object Oj in Di, search the access list for object Oj, look for an entry *<Di,* Rj > with M e Kj. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.

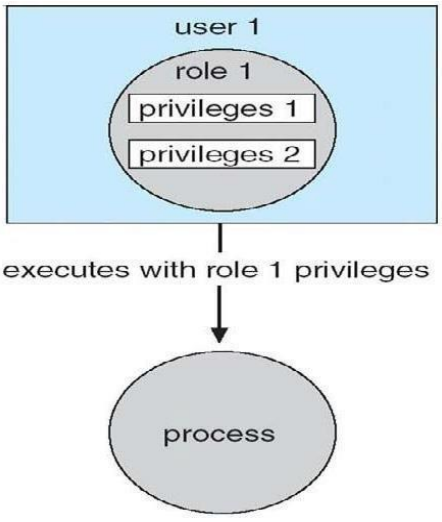
**Capability Lists for Domains**

A capability list for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its name or address, called a **capability**. To execute operation M on object Oj, the process executes the operation *M,* specifying the capability for object Oj as a parameter. Simple possession of the capability means that access is allowed.

Capabilities are usually distinguished from other data in one of two ways:
Each object has a tag to denote its type either as a capability or as accessible data.
Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system.

**A Lock-Key Mechanism**

The lock-key scheme is a compromise between access lists and capability lists. Each object has a list of unique bit patterns, called locks. Similarly, each domain has a list of unique bit patterns, called keys. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.




**Access Control**

Each file and directory are assigned an owner, a group, or possibly a list of users, and for each of those entities, access-control information is assigned.

Solaris 10 advances the protection available in the Sun Microsystems operating system by explicitly adding the principle of least privilege via **role-based access control (RBAC).** This facility revolves around privileges. A privilege is the right to execute a system call or to use an option within that system call (such as opening a file with write access). Privileges can be assigned to processes, limiting them to exactly the access they need to

perform their work. Privileges and programs can also be assigned to **roles.** Users are assigned roles or can take roles based on passwords to the roles. In this way, a user can take a role that enables a privilege, allowing the user to run a program to accomplish a specific task, as depicted in Figure 14.8. This implementationof privileges decreases the security risk associated with superusers and setuid programs.

**Revocation of Access Rights**

Since the capabilities are distributed throughout the system, we must find them before we can revoke them. Schemes that implement revocation for capabilities include the following:

• **Reacquisition.** Periodically, all capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.

• **Back-pointers.** A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary.

• **Indirection**. The capabilities point indirectly to the objects. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it. Then, when an access is attempted, the capability is found to point to an illegal table entry.

• **Keys**. A key is a unique bit pattern that can be associated with a capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process owning the capability. A master key is associated with each object; it can be defined or replaced with the **set-key** operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared with the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised.

In key-based schemes, the operations of defining keys, inserting them into lists, and deleting them from lists should not be available to all users.

**CAPABILITY-BASED SYSTEM**

Here, survey of two capability-based protection systems is done.

**1) An Example: Hydra**

Hydra is a capability-based protection system that provides considerable flexibility. A fixed set of possible access rights is known to and interpreted by the system. These rights include such basic forms of access as the right to read, write, or execute a memory segment. In addition, a user (of the protection system) can declare other rights.

Operations on objects are defined procedurally. The procedures that implement such operations are themselves a form of object, and they are accessed indirectly by capabilities. The names of user-defined procedures must be identified to the protection system if it is to deal with

objects of the user defined type. When the definition of an object is made known to Hydra, the names of operations on the type become auxiliary rights.

Hydra also provides rights amplification. This scheme allows a procedure to be certified as *trustworthy* to act on a formal parameter of a specified type on behalf of any process that holds a right to execute the procedure. The rights held by a trustworthy procedure are independent of, and may exceed, the rights held by the calling process.

When a user passes an object as an argument to a procedure, we may need to ensure that the procedure cannot modify the abject. We can implement this restriction readily by passing an access right that does not have the modification (write) right.

The procedure-call mechanism of Hydra was designed as a direct solution to the *problem of mutually suspicious subsystems.*

A Hydra subsystem is built on top of its protection kernel and may require protection of its own components. A subsystem interacts with the kernel through calls on a set of kernel-defined primitives that define access rights to resources defined by the subsystem.

**2) An Example: Cambridge CAP System**

A different approach to capability-based protection has been taken in the design of the Cambridge CAP system. CAP's capability system is simpler and superficially less powerful than that of Hydra. It can be used to provide secure protection of user-defined objects. CAP has two kinds of capabilities.

The ordinary kind is called a **data capability. It** can be used to provide access to objects, but the only rights provided are the standard read, write, and execute of the individual storage segments associated with the object.

The second kind of capability is the **software capability,** which is protected, but not interpreted, by the CAP microcode. It is interpreted by a *protected* (that is, a privileged) procedure, which may be written by an application programmer as part of a subsystem. A particular kind of rights amplification is associated with a protected procedure.

# CASE STUDY: THE LINUX OPERATING SYSTEM

## History

- Linux is a modem, free operating system based on UNIX standards.
- First developed as a small but self-contained kernel in 1991 by Linus Torvalds, with the major design goal of UNIX compatibility.
- Its history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet.
- It has been designed to run efficiently and reliably on common PC hardware, but also runs on a variety of other platforms.
- The core Linux operating system kernel is entirely original, but it can run much existing free UNIX software, resulting in an entire UNIX-compatible operating system free from proprietary code.

## The Linux Kernel

- Version 0.01 (May 1991) had no networking, ran only on 80386-compatible Intel processors and on PC hardware, had extremely limited device-drive support, and supported only the Minix file system.
- Linux 1.0 (March 1994) included these new features:
  - Support for UNIX's standard TCP/IP networking protocols
  - BSD-compatible socket interface for networking programming
  - Device-driver support for running IP over an Ethernet
  - Enhanced file system
  - Support for a range of SCSI controllers for high-performance disk access
  - Extra hardware support
- Version 1.2 (March 1995) was the final PC-only Linux kernel.

## Linux 2.0

- Released in June 1996, 2.0 added two major new capabilities:
  - Support for multiple architectures, including a fully 64-bit native Alpha port.

- Support for multiprocessor architectures

- Other new features included:
  - Improved memory-management code
  - Improved TCP/IP performance
  - Support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand.
  - Standardized configuration interface
- Available for Motorola 68000-series processors, Sun Sparc systems, and for PC and PowerMac systems.

# The Linux System

- Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.
- The min system libraries were started by the GNU project, with improvements provided by the Linux community.
- Linux networking-administration tools were derived from 4.3BSD code; recent BSD derivatives such as Free BSD have borrowed code from Linux in return.
- The Linux system is maintained by a loose network of developers collaborating over the Internet, with a small number of public ftp sites acting as de facto standard repositories.

# Linux Distributions

- Standard, precompiled sets of packages, or *distributions*, include the basic Linux system, system installation and management utilities, and ready-to-install packages of common UNIX tools.
- The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places; modern distributions include advanced package management.
- Early distributions included SLS and Slackware. *Red Hat* and *Debian* are popular distributions from commercial and noncommercial sources, respectively.

- The RPM Package file format permits compatibility among the various Linux distributions.

# Linux Licensing

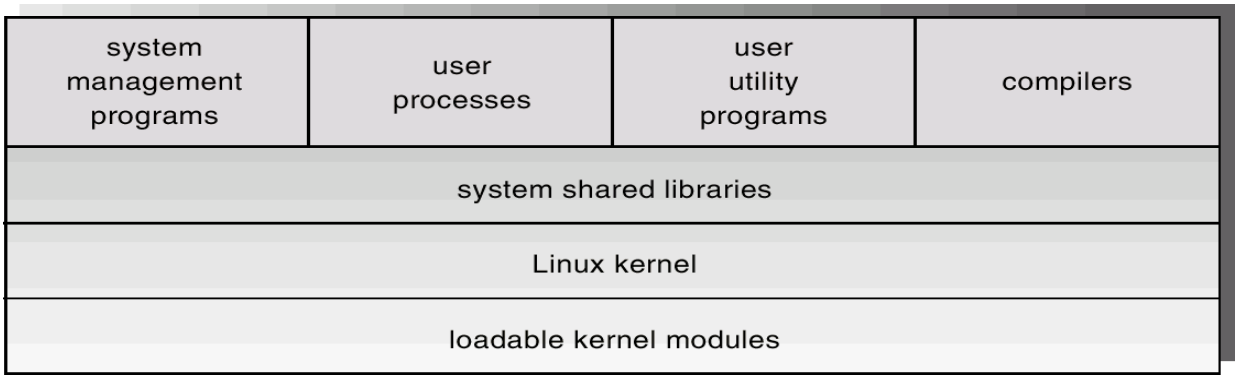
- The Linux kernel is distributed under the GNU General Public License (GPL), the terms of which are set out by the Free Software Foundation.
- Anyone using Linux, or creating their own derivative of Linux, may not make the derived product proprietary; software released under the GPL may not be redistributed as a binary-only product.

# Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools..
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model.
- Main design goals are speed, efficiency, and standardization.
- Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.
- The Linux programming interface adheres to the SVR4 UNIX semantics, rather than to BSD behavior

# Components of a Linux System

- Like most UNIX implementations, Linux is composed of three main bodies of code; the most important distinction between the kernel and all other components.
- The **kernel** is responsible for maintaining the important abstractions of the operating system.
  - Kernel code executes in *kernel mode* with full access to all the physical resources of the computer.
  - All kernel code and data structures are kept in the same single address space.




- The **system libraries** define a standard set of functions through which applications interact with the kernel, and which implement much of the operating-system functionality that does not need the full privileges of kernel code.
- The **system utilities** perform individual specialized management tasks.

# Kernel Modules

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel.
- A kernel module may typically implement a device driver, a file system, or a networking protocol.
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.
- Three components to Linux module support:
  - module management
  - driver registration
  - conflict resolution

# Module Management

- Supports loading modules into memory and letting them talk to the rest of the kernel.
- Module loading is split into two separate sections:

- Managing sections of module code in kernel memory
- Handling symbols that modules are allowed to reference

- The module requestor manages loading requested, but currently unloaded, modules; it also regularly queries the kernel to see whether a dynamically loaded module is still in use, and will unload it when it is no longer actively needed.

# Driver Registration

- Allows modules to tell the rest of the kernel that a new driver has become available.
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time.
- Registration tables include the following items:
  - Device drivers
  - File systems
  - Network protocols
  - Binary format

# Conflict Resolution

- A mechanism that allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver
- The conflict resolution module aims to:
  - Prevent modules from clashing over access to hardware resources
  - Prevent *autoprobes* from interfering with existing device drivers
  - Resolve conflicts with multiple drivers trying to access the same hardware

**Process Management**

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The **fork** system call creates a new process.

- A new program is run after a call to **execve**.

- Under UNIX, a process encompasses all the information that the operating system must maintain *t* track the context of a single execution of a single program.
- Under Linux, process properties fall into three groups: the process's identity, environment, and context.

# Process Identity

_ **Process ID** (PID). The unique identifier for the process; used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process.

_ **Credentials**. Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files.

_ **Personality**. Not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can slightly modify the semantics of certain system calls. Used primarily by emulation libraries to request that system calls be compatible with certain specific flavors of UNIX.

# Process Environment

- The process's environment is inherited from its parent, and is composed of two null-terminated vectors:
  - The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
  - The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values.
- Passing environment variables among processes and inheriting variables by a process's children are flexible means of passing information to components of the usermode systemsoftware.

- The environment-variable mechanism provides a customization of the operating system that can be set on a per-process basis, rather than being configured for the system as a whole.

# Process Context

- The (constantly changing) state of a running program at any point in time.
- The **scheduling context** is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process.
- The kernel maintains **accounting** information about the resources currently being consumed by each process, and the total resources consumed by the process in its lifetime so far.
- The **file table** is an array of pointers to kernel file structures. When making file I/O system calls, processes refer to files by their index into this table.
- Whereas the file table lists the existing open files, the **file-system context** applies to requests to open new files. The current root and default directories to be used for new file searches are stored here.
- The **signal-handler table** defines the routine in the process's address space to be called when specific signals arrive.
- The **virtual-memory context** of a process describes the full contents of the its private address space.

# Processes and Threads

- Linux uses the same internal representation for processes and threads; a thread is simply a new process that happens to share the same address space as its parent.
- A distinction is only made when a new thread is created by the **clone** system call.
  - **fork** creates a new process with its own entirely new process context
  - **clone** creates a new process with its own identity, but that is allowed to share the data structures of its parent

- Using **clone** gives an application fine-grained control over exactly what is shared between two threads.

# Scheduling

- The job of allocating CPU time to different tasks within an operating system.
- While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks.
- Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

# Kernel Synchronization

- A request for kernel-mode execution can occur in two ways:
  - A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs.
  - A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.
- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section
- Linux uses two techniques to protect critical sections:

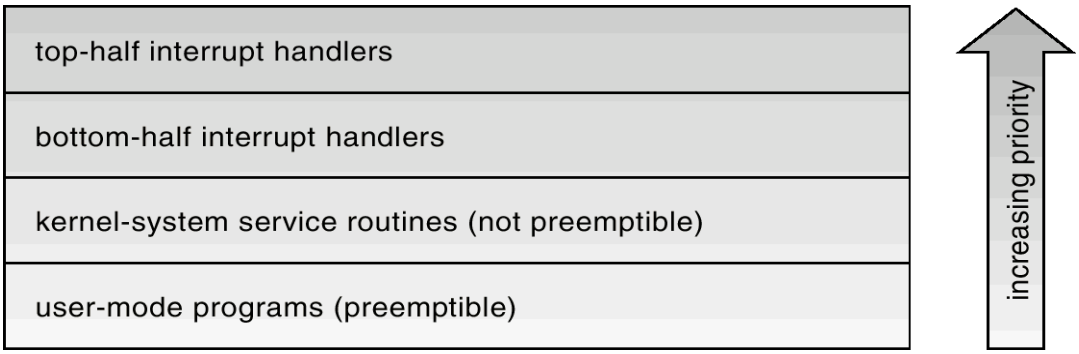
  1. Normal kernel code is nonpreemptible
     - ✓ when a time interrupt is received while a process is executing a kernel system service routine, the kernel's **need_resched** flag is set so that the scheduler will run once the system call has completed and control is about to be returned to user mode.

  2. The second technique applies to critical sections that occur in an interrupt service routines.
- By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures.

- To avoid performance penalties, Linux's kernel uses a synchronization architecture that allows long critical sections to run without having interrupts disabled for the critical section's entire duration.
- Interrupt service routines are separated into a *top half* and a *bottom half*.
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled.
  - The bottom half is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves.
  - This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code.

# Interrupt Protection Levels

- Each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level.
- User processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.




**Process Scheduling**

- Linux uses two process-scheduling algorithms:
  - A time-sharing algorithm for fair preemptive scheduling between multiple processes
  - A real-time algorithm for tasks where absolute priorities are more important than fairness
- A process's scheduling class defines which algorithm to apply.

- For time-sharing processes, Linux uses a prioritized, credit based algorithm.
  - The crediting rule factors in both the process's history and its priority.
  - This crediting system automatically prioritizes interactive or I/O-bound processes.
- Linux implements the FIFO and round-robin real-time scheduling classes; in both cases, each process has a priority in addition to its scheduling class.
  - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the longest-waiting one
  - FIFO processes continue to run until they either exit or block
  - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round robing processes of equal priority automatically time-share between themselves.

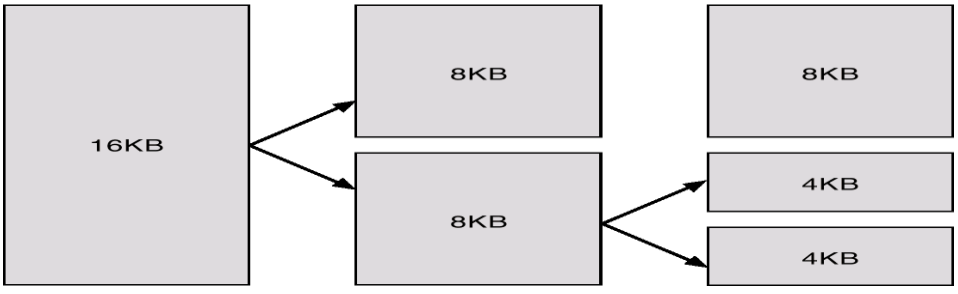
# Symmetric Multiprocessing

- Linux 2.0 was the first Linux kernel to support SMP hardware; separate processes or threads can execute in parallel on separate processors.
- To preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code.

# Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory.
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes.

**Splitting of Memory in a Buddy Heap**



**Managing Physical Memory**

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request.
- The allocator uses a *buddy-heap* algorithm to keep track of available physical pages.
  - Each allocatable memory region is paired with an adjacent partner.
  - Whenever two allocated partner regions are both freed up they are combined to form a larger region.
  - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request.
- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator).

# Virtual Memory

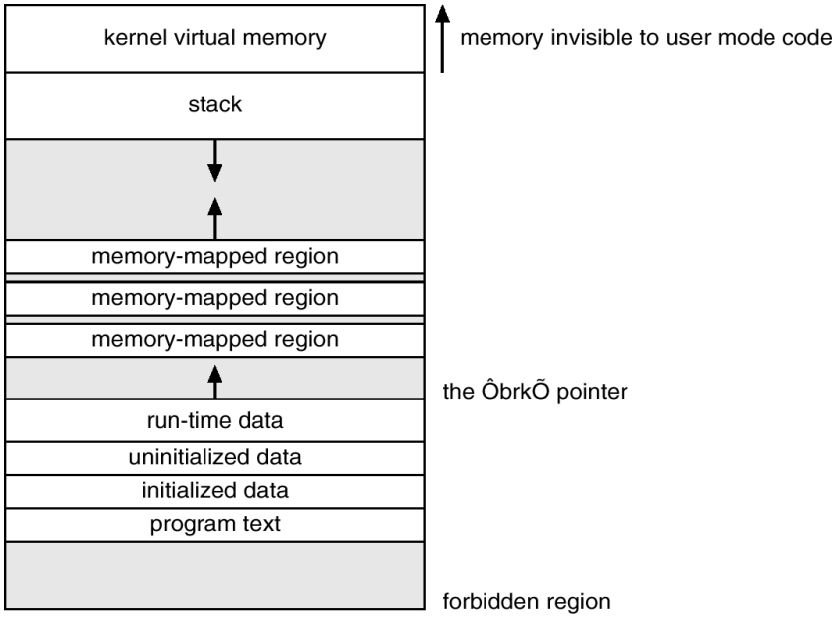
- The VM system maintains the address space visible to each process: It creates pages of virtual memory on demand, and manages the loading of those pages from disk or their swapping back out to disk as required.
- The VM manager maintains two separate views of a process's address space:
  - A logical view describing instructions concerning the layout of the address space. The address space consists of a set of nonoverlapping regions, each representing a continuous, page-aligned subset of the address space.
  - A physical view of each address space which is stored in the hardware page tables for the process.
- Virtual memory regions are characterized by:
  - The backing store, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (*demand-zero* memory)
  - The region's reaction to writes (page sharing or copy-onwrite).

- The kernel creates a new virtual address space
  1. When a process runs a new program with the **exec** system call
  2. Upon creation of a new process by the **fork** system call
- On executing a new program, the process is given a new, completely empty virtual-address space; the program loading routines populate the address space with virtual memory regions.
- Creating a new process with **fork** involves creating a complete copy of the existing process's virtual address space.
  - The kernel copies the parent process's VMA descriptors, then creates a new set of page tables for the child.
  - The parent's page tables are copies directly into the child's, with the reference count of each page covered being incremented.
  - After the fork, the parent and child share the same physical pages of memory in their address spaces.
- The VM paging system relocates pages of memory from physical memory out to disk when the memory is needed for something else.
- The VM paging system can be divided into two sections:
  - The pageout-policy algorithm decides which pages to write out to disk, and when.
  - The paging mechanism actually carries out the transfer, and pages data back into physical memory as needed.
- The Linux kernel reserves a constant, architecture dependent region of the virtual address space of every process for its own internal use.
- This kernel virtual-memory area contains two regions:
  - A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code.
  - The reminder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory.

**Executing and Loading User Programs**

- Linux maintains a table of functions for loading programs; it gives each function the opportunity to try loading the given file when an exec system call is made.
- The registration of multiple loader routines allows Linux to support both the ELF and a.out binary formats.
- Initially, binary-file pages are mapped into virtual memory; only when a program tries to access a given page will a page fault result in that page being loaded into physical memory.
- An ELF-format binary file consists of a header followed by several page-aligned sections; the ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

**Memory Layout for** ELF **Programs**




**Static and Dynamic Linking**

- A program whose necessary library functions are embedded directly in the program's executable binary file is *statically* linked to its libraries.
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions.

- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once.

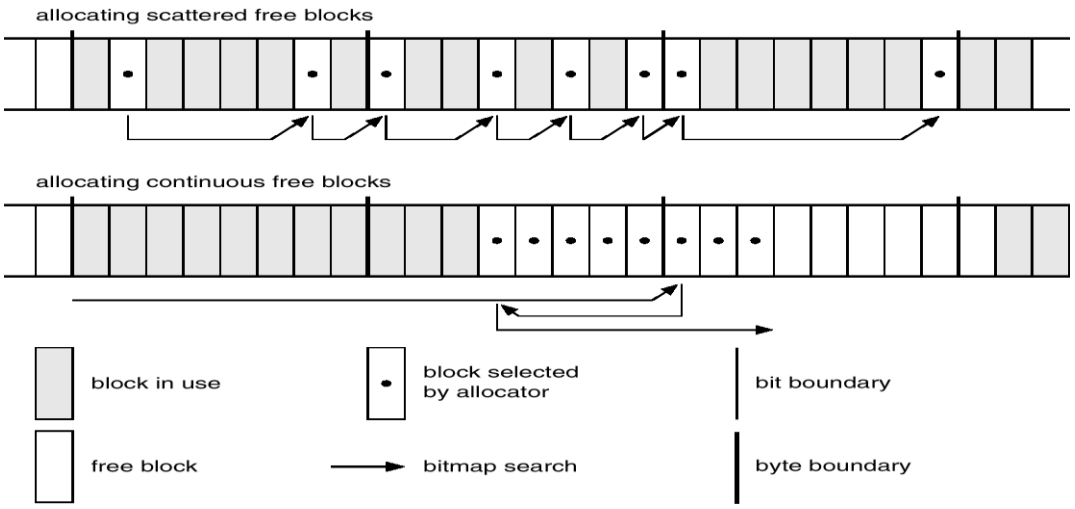
# File Systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics.
- Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the *virtual file system (VFS)*.
- The Linux VFS is designed around object-oriented principles and is composed of two components:
  - A set of definitions that define what a file object is allowed to look like
    - ✔The *inode-object* and the *file-object* structures represent individual files
    - ✔the *file system object* represents an entire file system
  - A layer of software to manipulate those objects.

**The Linux Ext2fs File System**

- Ext2fs uses a mechanism similar to that of BSD Fast File System (ffs) for locating data blocks belonging to a specific file.
- The main differences between ext2fs and ffs concern their disk allocation policies.
  - In ffs, the disk is allocated to files in blocks of 8Kb, with blocks being subdivided into fragments of 1Kb to store small files or partially filled blocks at the end of a file.
  - Ext2fs does not use fragments; it performs its allocations in smaller units. The default block size on ext2fs is 1Kb, although 2Kb and 4Kb blocks are also supported.
  - Ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

**Ext2fs Block-Allocation Policies**




**The Linux Proc File System**

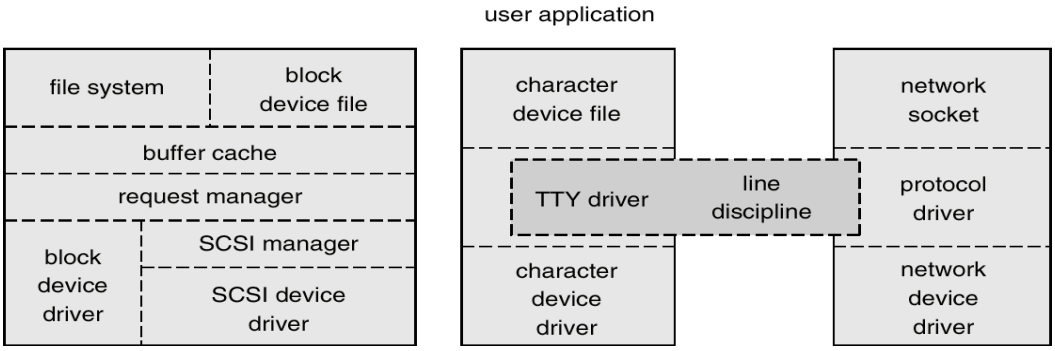
- The **proc** file system does not store data, rather, its contents are computed on demand according to user file I/O requests.
- **proc** must implement a directory structure, and the file contents within; it must then define a unique and persistent inode number for each directory and files it contains.
  - It uses this inode number to identify just what operation is required when a user tries to read from a particular file inode or perform a lookup in a particular directory inode.
  - When data is read from one of these files, **proc** collects the appropriate information, formats it into text form and places it into the requesting process's read buffer.

# Input and Output

- The Linux device-oriented file system accesses disk storage through two caches:
  - Data is cached in the page cache, which is unified with the virtual memory system
  - Metadata is cached in the buffer cache, a separate cache indexed by the physical disk block.
- Linux splits all devices into three classes:
  - *block devices* allow random access to completely independent, fixed size blocks of data

- *character devices* include most other devices; they don't need to support the functionality of regular files.
- *network devices* are interfaced via the kernel's networking Subsystem

**Device-driver Block Structure**




# Block Devices

- Provide the main interface to all disk devices in a system.
- The *block buffer* cache serves two main purposes:
  - it acts as a pool of buffers for active I/O
  - it serves as a cache for completed I/O
- The *request manager* manages the reading and writing of buffer contents to and from a block device driver.

**Character Devices**

- A device driver which does not offer random access to fixed blocks of data.
- A character device driver must register a set of functions which implement the driver's various file I/O operations.
- The kernel performs almost no preprocessing of a file read or write request to a character device, but simply passes on the request to the device.
- The main exception to this rule is the special subset of character device drivers which implement terminal devices, for which the kernel maintains a standard interface.

**Interprocess Communication**

Like UNIX, Linux informs processes that an event has occurred via signals.

- There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process.

- The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and **wait.queue** structures.

**Passing Data Between Processes**

- he pipe mechanism allows a child process to inherit a communication channel to its parent, data written to one end of the pipe can be read a the other.
- Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space.
- To obtain synchronization, however, shared memory must be used in conjunction with another Interprocess communication mechanism.

**Shared Memory Object**

- The shared-memory object acts as a backing store for shared-memory regions in the same way as a file can act as backing store for a memory-mapped memory region.
- Shared-memory mappings direct page faults to map in pages from a persistent shared-memory object.
- Shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

# Network Structure

- Networking is a key area of functionality for Linux.
  - It supports the standard Internet protocols for UNIX to UNIX communications.
  - It also implements protocols native to nonUNIX operating systems, in particular, protocols used on PC networks, such as Appletalk and IPX.
- Internally, networking in the Linux kernel is implemented by three layers of software:
  - The socket interface
  - Protocol drivers
  - Network device drivers

- The most important set of protocols in the Linux networking system is the internet protocol suite.
  - It implements routing between different hosts anywhere on the network.
  - On top of the routing protocol are built the UDP, TCP and ICMP protocols

# Security

- The *pluggable authentication modules (PAM)* system is available under Linux.
- PAM is based on a shared library that can be used by any system component that needs to authenticate users.
- Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers (**uid** and **gid**).
- Access control is performed by assigning objects a *protections mask*, which specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access.
- Linux augments the standard UNIX **setuid** mechanism in two ways:
  - It implements the POSIX specification's saved *user-id* mechanism, which allows a process to repeatedly drop and reacquire its effective uid.
  - It has added a process characteristic that grants just a subset of the rights of the effective uid.
- Linux provides another mechanism that allows a client to selectively pass access to a single file to some server process without granting it any other privileges.

**Device-driver Block Structure**