OPERATING SYSTEMS

Subject Code: 10CS53	**I.A. Marks : 25**
Hours/Week : 04	**Exam Hours: 03**
Total Hours : 52	**Exam Marks: 100**

PART – A

UNIT – 1 **6 Hours**

Introduction to Operating Systems, System structures: What operating systems do; 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.

UNIT – 2 **7 Hours**

Process Management: Process concept; Process scheduling; Operations on processes; Inter-process communication. Multi-Threaded Programming: Overview; Multithreading models; Thread Libraries; Threading issues. Process Scheduling: Basic concepts; Scheduling criteria; Scheduling algorithms; Multiple-Processor scheduling; Thread scheduling.

UNIT – 3 **7 Hours**

Process Synchronization : Synchronization: The Critical section problem; Peterson's solution; Synchronization hardware; Semaphores; Classical problems of synchronization; Monitors.

UNIT – 4 **6 Hours**

Deadlocks: Deadlocks: System model; Deadlock characterization; Methods for handling deadlocks; Deadlock prevention; Deadlock avoidance; Deadlock detection and recovery from deadlock.

PART – B

UNIT – 5 **7 Hours**

Memory Management: Memory Management Strategies: Background; Swapping; Contiguous memory allocation; Paging; Structure of page table; Segmentation. Virtual Memory Management: Background; Demand paging; Copy-on-write; Page replacement; Allocation of frames; Thrashing.

UNIT – 6 **7 Hours**

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

UNIT – 7 **6 Hours**

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.

UNIT – 8 **6 Hours**

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.

Text Books:

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne: Operating System Principles, 7th edition, Wiley India, 2006.
   (Chapters: 1, 2, 3.1 to 3.4 , 4.1 to 4.4, 5.1 to 5.5, 6.1 to 6.7, 7, 8.1 to 8.6, 9.1 to 9.6, 10, 11.1 to 11.5, 12.1 to 12.6, 17.1 to 17.8, 21.1 to 21.9)

TABLE OF CONTENTS

UNIT 1: INTRODUCTION

Operating System

- An OS is a program that acts as an intermediary between
  - → computer-user and
  - → computer-hardware.
- It also provides a basis for application-programs
- Goals of OS:
  - ➢ To execute programs.
  - ➢ To make solving user-problems easier.
  - ➢ To make the computer convenient to use.
- The OS (also called kernel) is the one program running at all times on the computer.
- Different types of OS:
  - ➢ Mainframe OS is designed to optimize utilization of hardware.
  - ➢ Personal computer (PC) OS supports complex game, business application.
  - ➢ Handheld computer OS is designed to provide an environment in which a user can easily interface with the computer to execute programs.

What Operating Systems do?

- Four components of a computer (Figure 1.1):
  1. Hardware
  2. OS
  3. Application programs and
  4. Users

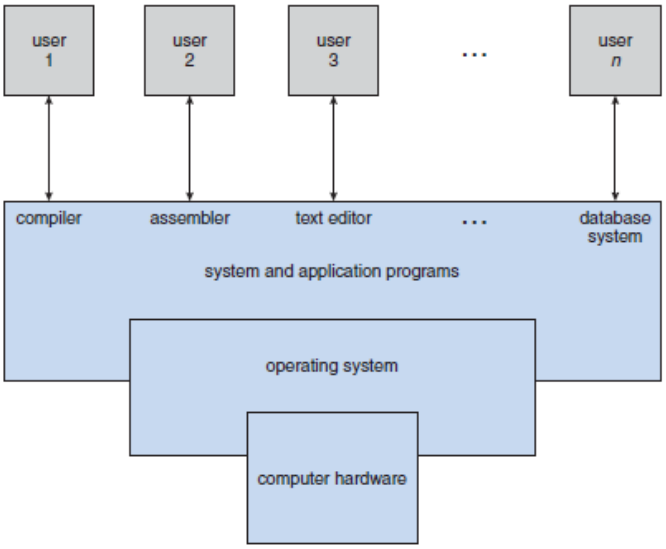


Figure 1.1 Abstract view of the components of a computer system

- Hardware provides basic computing-resources:
  - → CPU
  - → memory and
  - → I/O devices.
- Application-program defines how the resources are used to solve computing-problems of the users.
  - Ex: word processors, spread sheets, compilers.
- The OS controls & co-ordinates the use of hardware among various application-program for various users.
- Two views of OS:
  - 1) User and 2) System.

User View

• The user's view of the computer depends on the interface being used:

1 Most users use a PC consisting of a monitor, keyboard and system-unit.
- The OS is designed mostly for ease of use.
- Some attention is paid to performance.

No attention is paid to resource utilization**.**
- The OS is optimized for the single-user experience.

2. Some users use a terminal connected to a mainframe or (a minicomputer).
- The OS is designed

→ to maximize resource utilization.
→ to assure that no individual user takes more than her fair share.

3. Some users use a workstation connected to network.
- The users have dedicated resources such as networking and servers.
- The OS is designed to compromise between

→ individual usability and
→ resource utilization.

4. Some users use a handheld computer.
- The OS is designed mostly for individual usability.
- Performance per unit of battery life is a very important factor.

System View

1. An OS as a resource allocator
- Resources used to solve a computing-problem:

→ CPU time
→ memory-space
→ file-storage space and
→ I/0 devices.
- The OS manages and allocates the above resources to programs and the users.

2. An OS is a control program
- The OS is needed to control:

→ operations of I/0 devices and
→ execution of user-programs to prevent errors.

Computer System Organization

• A computer consists of
→ one or more CPUs and
→ no. of device-controllers (Figure 1.2).
• Controller is in charge of a specific type of device (for ex: audio devices).
• CPU and controllers can execute concurrently.
• A memory-controller is used to synchronize access to the shared-memory.
• Following events occur for a computer to start running:

1. Bootstrap program is an initial program that runs when computer is powered-up.
2. Bootstrap program
   → initializes all the system from registers to memory-contents and
   → loads OS into memory.
3. Then, OS
   → starts executing the first process (such as "init") and
   → waits for some event to occur.
4. The occurrence of an event is signaled by an interrupt from either the hardware or the software (Figure 1.3).
   i) Hardware may trigger an interrupt by sending a signal to the CPU.
   ii) Software may trigger an interrupt by executing a system-call.
5. When CPU is interrupted, the CPU
   → stops current computation and
   → transfers control to ISR (interrupt service routine).
6. Finally, the ISR executes; on completion, the CPU resumes the interrupted computation.

Common Functions of Interrupts

• Interrupt transfers control to the ISR generally, through the interrupt-vector, which contains the addresses of all the service routines.
• Interrupt architecture must save the address of the interrupted-instruction.
• Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt.
• A *trap* is a software-generated interrupt caused either by an error or a user request.
• A modern OS is interrupt-driven.

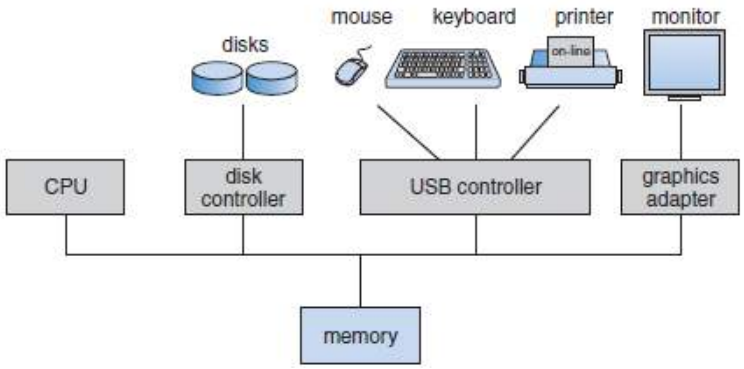


Figure 1.2 A modern computer system

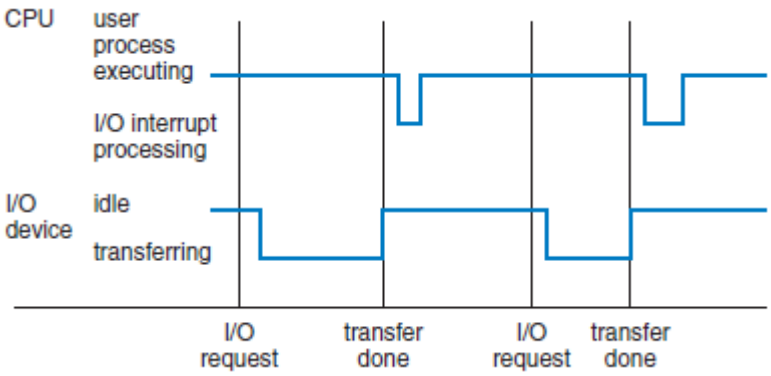


Figure 1.3 Interrupt time line for a single process doing output

Storage Structure

- Programs must be in main-memory (also called RAM) to be executed.
- Interaction with memory is done through a series of *load* or *store* instructions.
  - ***1. Load Instruction***
    - Moves a word from main-memory to an internal register within the CPU.
  - ***2. Store Instruction***
    - Moves the content of a register to main-memory.
- Also, the CPU automatically loads instructions from main-memory for execution.
- Ideally, we want the programs & data to reside in main-memory permanently. This is not possible for 2 reasons:
  1. Main-memory is small.
  2. Main-memory is volatile i.e. it loses its contents when powered-off.
- Most computers provide secondary-storage as an extension of main-memory.
  For ex: magnetic disk.
- Main requirement:
  The secondary-storage must hold large amount of data permanently.
- The wide range of storage-systems can be organized in a hierarchy (Figure 1.4).
- The higher levels are expensive, but they are fast.
  The lower levels are inexpensive, but they are slow.

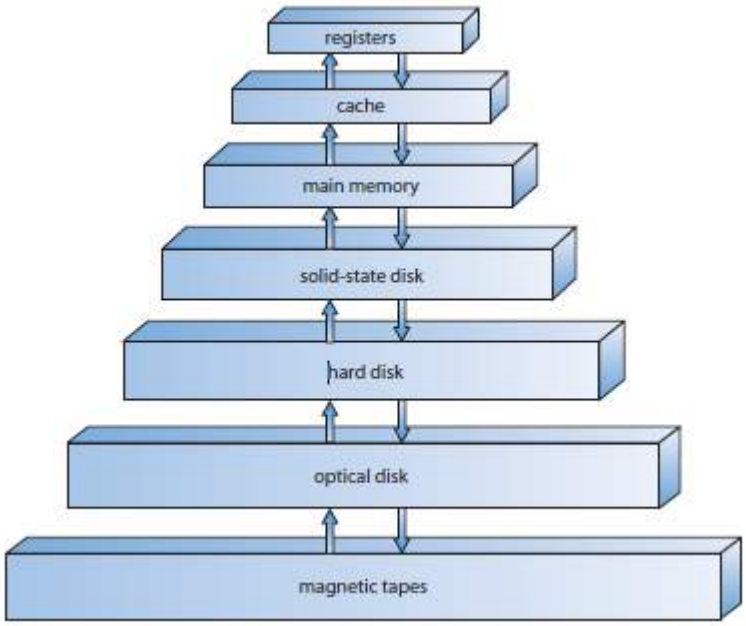


Figure 1.4 Storage-device hierarchy

I/O Structure

- A computer consists of CPUs and multiple device controllers (Figure 1.5).
- A controller is in charge of a specific type of device.
- The controller maintains
  - → some local buffer and
  - → set of special-purpose registers.
- Typically, OS has a *device-driver* for each controller.
- Interrupt-driven I/O:
  1. Driver loads the appropriate registers within the controller.
  2. Controller examines the contents of registers to determine what action to take.
  3. Controller transfers data from the device to its local buffer.
  4. Controller informs the driver via an interrupt that it has finished its operation.
  5. Driver then returns control to the OS.
- Problem: Interrupt-driven I/O produces high overhead when used for bulk data-transfer.
  Solution: Use DMA (direct memory access).
- In DMA, the controller transfers blocks of data from buffer-storage directly to main memory without CPU intervention.

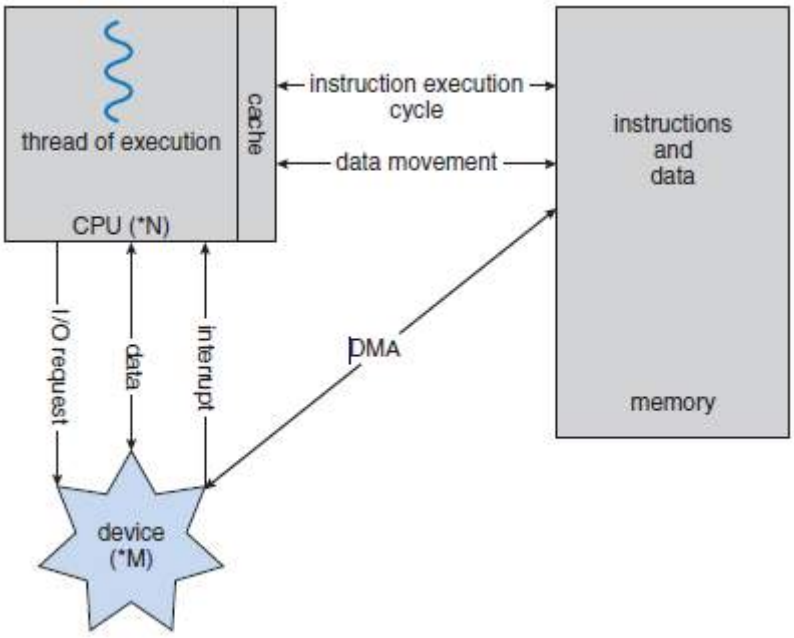


Figure 1.5 How a modern computer system works

Computer System Architecture

1. Single-Processor Systems
2. Multiprocessor Systems
3. Clustered Systems

Single Processor Systems

- The system has only one general-purpose CPU.
- The CPU is capable of executing a general-purpose instruction-set.
- These systems range from PDAs through mainframes.
- Almost all systems have following processors:
  - ***1. Special Purpose Processors***
    - Include disk, keyboard, and graphics controllers.
  - ***2. General Purpose Processors***
    - Include I/O processors.
- Special-purpose processors run a limited instruction set and do not run user-processes.

Multiprocessor Systems

- These systems have two or more processors which can share:
  → bus → clock → memory/peripheral devices
- Advantages:
  - ***1. Increased Throughput***
    - By increasing no. of processors, we expect to get more work done in less time.
  - ***2. Economy of Scale***
    - These systems are cheaper because they can share
      → peripherals → mass-storage → power-supply.
    - If many programs operate on same data, they will be stored on one disk & all processors can share them.
  - ***3. Increased Reliability***
    - The failure of one processor will not halt the system.
- Two types of multiple-processor systems:
  1) Asymmetric multiprocessing(AMP) and 2) Symmetric multiprocessing(SMP)

Asymmetric Multiprocessing

- This uses master-slave relationship (Figure 1.6).
- Each processor is assigned a specific task.
- A master-processor controls the system.
  The other processors look to the master for instruction.
- The master-processor schedules and allocates work to the slave-processors.

Symmetric Multiprocessing

- Each processor runs an identical copy of OS.
- All processors are peers; no master-slave relationship exists between processors.
- Advantages:
  1. Many processes can run simultaneously.
  2. Processes and resources are shared dynamically among the various processors.
- Disadvantage:
  1. Since CPUs are separate, one CPU may be sitting idle while another CPU is overloaded. This results in inefficiencies.

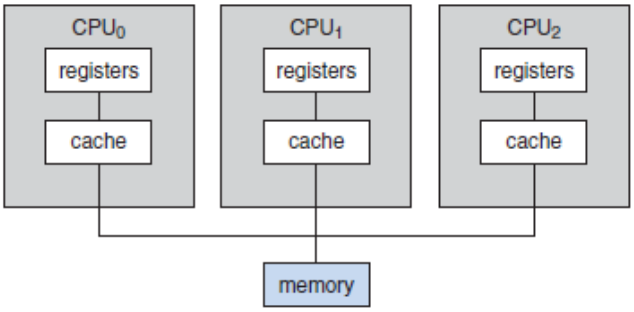


Figure 1.6 Symmetric multiprocessing architecture

Clustered Systems

- These systems consist of two or more systems coupled together (Figure 1.7).
- These systems share storage & closely linked via LAN.
- Advantage:
  1. Used to provide high-availability service.
- High-availability is obtained by adding a level of redundancy in the system.
- Working procedure:
  - A cluster-software runs on the cluster-nodes.
  - Each node can monitor one or more other nodes (over the LAN).
  - If the monitored-node fails, the monitoring-node can
    → take ownership of failed-node's storage and
    → restart the applications running on the failed-node.
  - The users and clients of the applications see only a brief interruption of service.
- Two types are:
  1) Asymmetric and 2) Symmetric

Asymmetric Clustering

- One node is in hot-standby mode while the other nodes are running the applications.
- The hot-standby node does nothing but monitor the active-server.
- If the server fails, the hot-standby node becomes the active server.

Symmetric Clustering

- Two or more nodes are running applications, and are monitoring each other.
- Advantage:
  1. This mode is more efficient, as it uses all of the available hardware.
- It does require that more than one application be available to run.

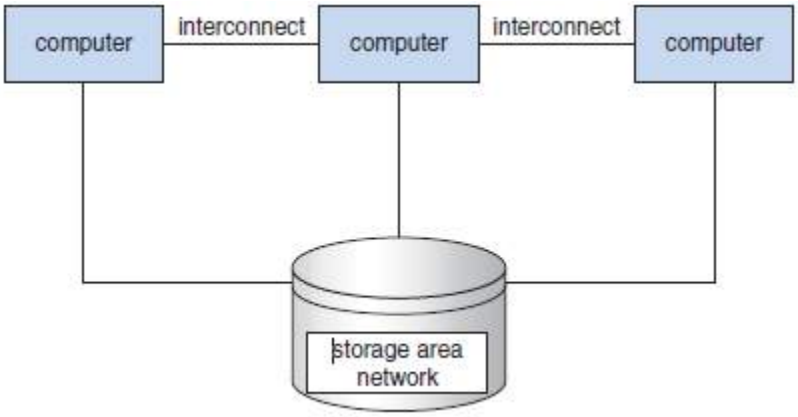


Figure 1.7 General structure of a clustered system

Operating System Structure

1. Batch systems
2. Multiprogrammed systems
3. Time-Sharing systems

Batch Systems

• Early computers were physically enormous machines run from a console.
• The common input devices were card readers and tape drives.
• The common output devices were line printers, tape drives, and card punches.
• The user
→ prepared a job which consisted of the program, the data, and control information
→ submitted the job to the computer-operator.
• The job was usually in the form of punch cards.
• At some later time (after minutes, hours, or days), the output appeared.
• To speed up processing, operators batched together jobs with similar needs and ran them through the computer as a group.
• Disadvantage:

1. The CPU is often idle, because the speeds of the mechanical I/O devices.

Multiprogrammed Systems

• Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.
• The idea is as follows:

1. OS keeps several jobs in memory simultaneously (Figure 1.8).
2. OS picks and begins to execute one of the jobs in the memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete.
3. OS simply switches to, and executes, another job.
4. When that job needs to wait, the CPU is switched to another job, and so on.
5. As long as at least one job needs to execute, the CPU is never idle.

• If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is *job scheduling***.**
• If several jobs are ready to run at the same time, the system must choose among them. Making this decision is *CPU scheduling.*

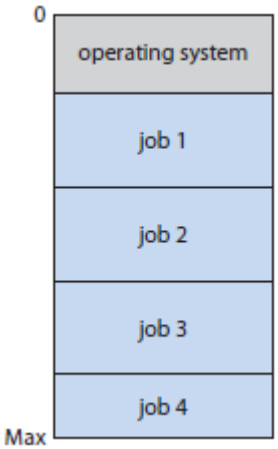


Figure 1.8 Memory layout for a multiprogramming system

Time Sharing Systems

• Time sharing (or multitasking) is a logical extension of multiprogramming.
• The CPU executes multiple jobs by switching between them.
• Switching between jobs occur so frequently that the users can interact with each program while it is running.
• Many users are allowed to share the computer simultaneously.
• CPU scheduling and multiprogramming are used to provide each user with a small portion of a time-shared computer.
• To obtain a good response time, jobs may have to be swapped in and out of main memory to the disk (called as backing store).
• Virtual memory is a technique that allows the execution of a job that may not be completely in memory.
• Advantage of virtual-memory:
    1. Programs can be larger than physical memory.
• Main requirements:
    - The system must provide a file-system.
    - The system must provide disk-management.
    - The system must provide CPU-scheduling to support concurrent execution.
    - The system must provide job-synchronization to ensure orderly execution.

Operating System Operations

• Modern OS is *interrupt driven.*
• Events are always signaled by the occurrence of an *interrupt or a trap*.
• A trap is a software generated interrupt caused either by
→ error (for example division by zero) or
→ request from a user-program that an OS service be performed.
• For each type of interrupt, separate segments of code in the OS determine what action should be taken.
• ISR (interrupt service routine) is provided that is responsible for dealing with the interrupt.

Dual Mode Operation

• Problem: We must be able to differentiate between the execution of
→ OS code and
→ user-defined code.
Solution: Most computers provide hardware-support.
• We have two modes of operation (Figure 1.9):
1. User mode and
2. Kernel mode
• A *mode bit* is a bit added to the hardware of the computer to indicate the current mode:
i.e. kernel (0) or user (1)

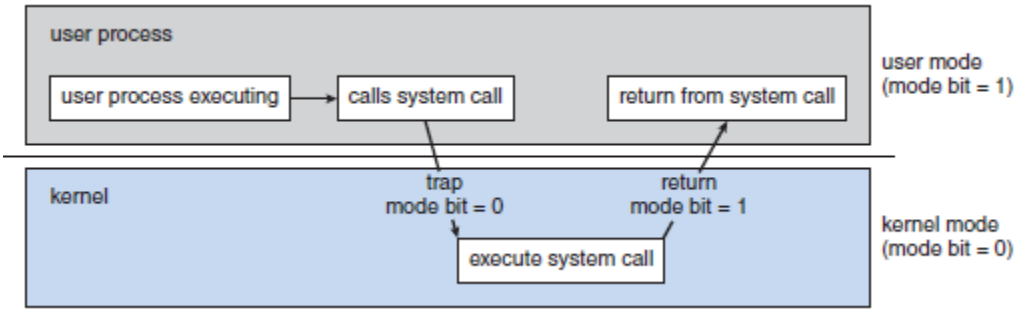


Figure 1.9 Transition from user to kernel mode

• Working principle:
1. At system boot time, the hardware starts in kernel-mode.
2. The OS is then loaded and starts user applications in user-mode.
3. Whenever a trap or interrupt occurs, the hardware switches from user-mode to kernel-mode (that is, changes the state of the mode bit to 0).
4. The system always switches to user-mode (by setting the mode bit to 1) before passing control to a user-program.
• Dual mode protects
→ OS from errant users and
→ errant users from one another.
• *Privileged instruction* is executed only in kernel-mode.
• If an attempt is made to execute a privileged instruction in user-mode, the hardware treats it as illegal and traps it to the OS.
• A *system calls* are called by user-program to ask the OS to perform the tasks on behalf of the user program.

Timer

• Problem: We cannot allow a user-program to get stuck in an infinite loop and never return control to the OS.

Solution: We can use a timer.

• A timer can be set to interrupt the computer after a specific period.

• The period may be

→ fixed (for ex: 1/60 second) or

→ variable (for ex: from 1ns to 1ms).

• A variable timer is implemented by

→ a fixed-rate clock and

→ a counter.

• Working procedure:

1. The OS sets the counter.
2. Every time the clock ticks, the counter is decremented.
3. When the counter reaches 0, an interrupt occurs.

• The instructions that modify the content of the timer are privileged instructions.

Process Management

- The OS is responsible for the following activities:
    1. Creating and deleting both user and system processes
    2. Suspending and resuming processes
    3. Providing mechanisms for process synchronization
    4. Providing mechanisms for process communication
    5. Providing mechanisms for deadlock handling
- A process needs following resources to do a task:
    → CPU
    → memory and
    → files.
- The resources are allocated to process
    → when the process is created or
    → while the process is running.
- When the process terminates, the OS reclaims all the reusable resources.
- A program by itself is not a process;
    1. A program is a passive entity (such as the contents of a file stored on disk).
    2. A process is an active entity.
- Two types of process:
    1. ***Single-threaded process*** has one PC(program counter) which specifies location of the next instruction to be executed.
    2. ***Multi-threaded process*** has one PC per thread which specifies location of next instruction to execute in each thread

Memory Management

- The OS is responsible for the following activities:
    1. Keeping track of which parts of memory are currently being used and by whom
    2. Deciding which processes are to be loaded into memory when memory space becomes available
    3. Allocating and de-allocating memory space as needed.
- Main memory is the array of bytes ranging from hundreds to billions.
- Each byte has its own address.
- The CPU
    → reads instructions from main memory during the instruction-fetch cycle.
    → reads/writes data from/to main-memory during the data-fetch cycle.
- To execute a program:
    1. The program will be
        → loaded into memory and
        → mapped to absolute addresses.
    2. Then, program accesses instructions & data from memory by generating absolute addresses.
    3. Finally, when program terminates, its memory-space is freed.
- To improve CPU utilization, keep several programs will be kept in memory
- Selection of a memory-management scheme depends on hardware-design of the system.

Storage Management

1. File-System Management
2. Mass-Storage Management
3. Caching

File System Management

- The OS is responsible for following activities:
  1. Creating and deleting files.
  2. Creating and deleting directories.
  3. Supporting primitives for manipulating files & directories.
  4. Mapping files onto secondary storage.
  5. Backing up files on stable (non-volatile) storage media.
- Computer stores information on different types of physical media.
  For ex: magnetic disk, optical disk.
- Each medium is controlled by a device (e.g. disk drive).
- The OS
  → maps files onto physical media and
  → accesses the files via the storage devices
- File is a logical collection of related information.
- File consists of both program & data.
- Data files may be numeric, alphabets or binary.
- When multiple users have access to files, access control (read, write) must be specified.

Mass Storage Management

- The OS is responsible for following activities:
  1. Free-space management
  2. Storage allocation and
  3. Disk scheduling.
- Usually, disks used to store
  → data that does not fit in main memory or
  → data that must be kept for a "long" period of time.
- Most programs are stored on disk until loaded into memory.
- The programs include
  → compilers
  → word processors and
  → editors.
- The programs use the disk as both the source and destination of their processing.
- Entire speed of computer operation depends on disk and its algorithms.

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 called as the *cache* on a temporary basis.
- When we need a particular piece of information:
  1. We first check whether the information is in the cache.
  2. If information is in cache, we use the information directly from the cache.
  3. If information 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.
- In addition, internal programmable registers, such as index registers, provide high-speed cache for main memory.
- The compiler implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.
- Most systems have an instruction cache to hold the instructions expected to be executed next.
- Most systems have one or more high-speed data caches in the memory hierarchy
- Because caches have limited size, cache management is an important design problem
  Careful selection of cache size & of a replacement policy can result in greatly increased performance

I/O Systems

- The OS must hide peculiarities of hardware devices from users.
- In UNIX, the peculiarities of I/O devices are hidden from the bulk of the OS itself by the I/O subsystem.
- The I/O subsystem consists of
    1. A memory-management component that includes buffering, caching, and spooling.
    2. A general device-driver interface.
    3. Drivers for specific hardware devices.
- Only the device driver knows the peculiarities of the specific device to which it is assigned.

Protection and Security

- Protection is a mechanism for controlling access of processes or users to resources defined by OS.
- This mechanism must provide
    - → means for specification of the controls to be imposed and
    - → means for enforcement.
- Protection can improve reliability by detecting latent errors at the interfaces between subsystems.
- Security means defense of the system against internal and external attacks.
- The attacks include
    - → viruses and worms
    - → DOS(denial-of-service)
    - → identity theft.
- Protection and security require the system to be able to distinguish among all its users.
    1. User identities (user IDs) include name and associated number, one per user.
       User IDs are associated with all files (or processes) of that user to determine access control.
    2. Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file.

Distributed System

- This is a collection of physically separate, possibly heterogeneous computer-systems.
- The computer-systems are networked to provide the users with access to the various resources.
- 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
    - → distances between nodes and
    - → transport media.
- Common network protocol are
    - → TCP/IP
    - → ATM.
- Networks are characterized based on the distances between their nodes.
    - → A local-area network (LAN) connects computers within a building.
    - → A wide-area network (WAN) usually links buildings, cities, or countries.
    - → A metropolitan-area network (MAN) could link buildings within a city.
- The media to carry networks are equally varied. They include
    - → copper wires,
    - → fiber strands, and
    - → wireless transmissions.

Special Purpose Systems

1. Real-Time Embedded Systems
2. Multimedia Systems
3. 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.
• The systems they run on are usually primitive, and so the operating systems provide limited features.
• Usually, they prefer to spend their time monitoring & managing hardware devices such as
→ automobile engines and
→ robotic arms.
• Embedded systems almost always run real-time operating systems.
• A real-time system is used when rigid time requirements have been placed on the operation of a processor.

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 must be delivered(streamed) according to certain time restrictions.
• Multimedia describes a wide range of applications. These include
→ audio files such as MP3
→ DVD movies
→ video conferencing
→ live webcasts of speeches

Handheld Systems

• Handheld systems include
→ PDAs and
→ cellular telephones.
• Main challenge faced by developers of handheld systems: Limited size of devices.
• Because of small size, most handheld devices have a
→ small amount of memory,
→ slow processors, and
→ small display screens.

Computing Environments

1. Traditional Computing
2. Client-Server Computing
3. Peer-to-Peer Computing
4. Web-Based Computing

Traditional Computing

- Used in office environment:
  - PCs connected to a network, with servers providing file and print services.
- Used in home networks:
  - At home, most users had a single computer with a slow modem.
  - Some homes have *firewalls* to protect their networks from security breaches.
- Web technologies are stretching the boundaries of traditional computing.
  - Companies establish *portals,* which provide web accessibility to their internal servers.
  - Network computers are terminals that understand web computing.
  - Handheld PDAs can connect to *wireless networks* to use company's web portal.
- Systems were either batch or interactive.
  1. Batch system processed jobs in bulk, with predetermined input.
  2. Interactive systems waited for input from users.

Client-Server Computing

- Servers can be broadly categorized as (Figure 1.10):
  1. Compute servers and 2. File servers

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.

File-server system provides a file-system interface where clients can create, read, and delete files.

- For example: web server that delivers files to clients running web browsers.

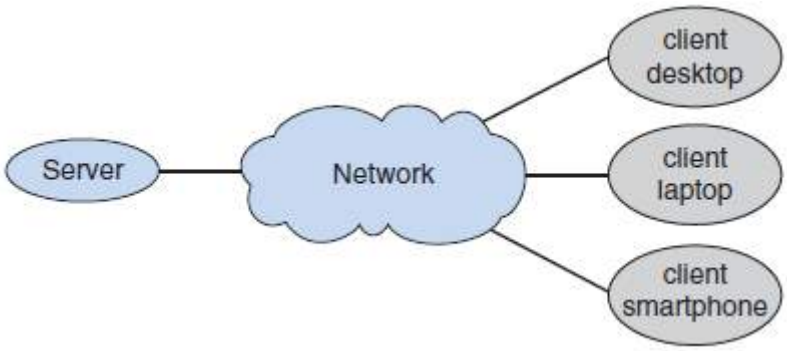


Figure 1.10 General structure of a client–server system.

Peer-to-Peer Computing

- All nodes are considered peers, and each may act as either a client or a server(Figure 1.11).
- Advantage:
    1. In a client-server system, the server is a bottleneck;
       but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.
- A node must first join the network of peers.
- Determining what services are available is done in one of two general ways:
    1. 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.
    2. A peer broadcasts a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer.

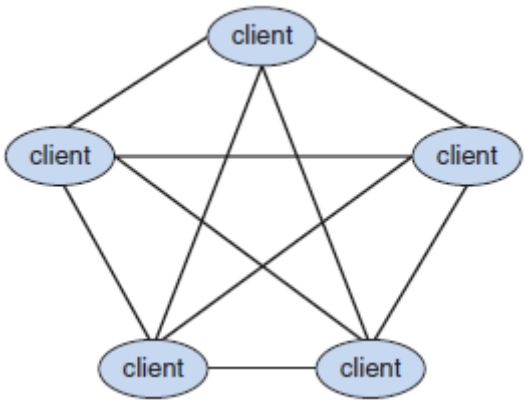


Figure 1.11 Peer-to-peer system with no centralized service.

Web Based Computing

- This includes
    → PC
    → handheld PDA &
    → cell phones
- Load balancer is a new category of devices to manage web traffic among similar servers.
- In load balancing, network connection is distributed among a pool of similar servers.
- More devices becoming networked to allow web access
- Use of operating systems like Windows 95, client-side, have evolved into Linux and Windows XP, which can be clients and servers

UNIT 1(CONT.): SYSTEM STRUCTURES

Operating System Services

• An OS provides an environment for the execution of programs.
• It provides services to
→ programs and
→ users.

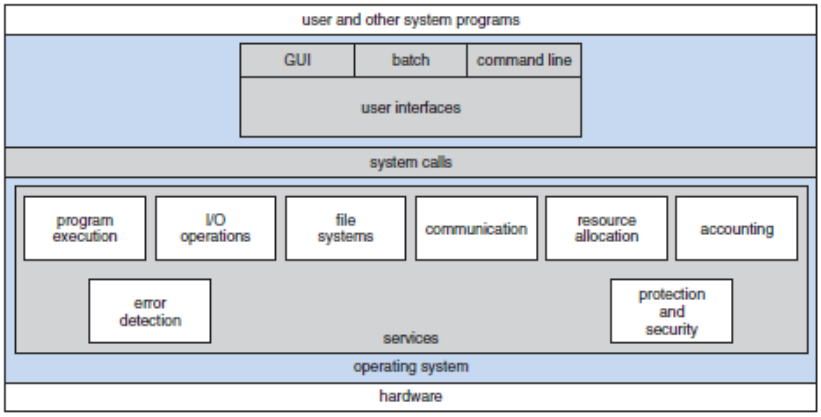


Figure 1.12 A view of OS services

• Common functions helpful to the user are (Figure 1.12):

1. User Interface

➢ Almost all OS have a user-interface (UI).
➢ Different interfaces are:

i) CLI (Command line Interface)

➢ This uses
→ text commands and
→ method for entering the text commands.

ii) Batch interface

➢ Commands & directives to control those commands are entered into files, and those files are executed.

iii) GUI (Graphical User Interface)

➢ The interface is a window-system with a pointing-device to
→ direct I/0
→ choose from menus and
→ make selections.

2. Program Execution

➢ The system must be able to
→ load a program into memory and
→ run the program.
➢ The program must be able to end its execution, either normally or abnormally.

3. I/O Operations

➢ The OS must provide a means to do I/O operations because users cannot control I/O devices directly.
➢ For specific devices, special functions may be desired (ex: to blank CRT screen).

4. File-system Manipulation

➢ Programs need to
→ read & write files (or directories)
→ create & delete files
→ search for a given file and
→ allow or deny access to files.

5. Communications

- In some situations, one process needs to communicate with another process.
- Communications may be implemented via
  1. Shared memory or
  2. Message passing
- In *message passing*, packets of information are moved between processes by OS.

6. Error Detection

- Errors may occur in
  → CPU & memory-hardware (ex: power failure)
  → I/O devices (ex: lack of paper in the printer) and
  → user program (ex: arithmetic overflow)
- For each type of error, OS should take appropriate action to ensure correct & consistent computing.

• Common functions for efficient operation of the system are:

1. Resource Allocation

- When multiple users are logged on the system at the same time, resources must be allocated to each of them.
- The OS manages different types of resources.
- Some resources (say CPU cycles) may have special allocation code.
  Other resources (say I/O devices) may have general request & release code.

2. Accounting

- We want to keep track of
  → which users use how many resources and
  → which kinds of resources.
- This record keeping may be used for
  → accounting (so that users can be billed) or
  → gathering usage-statistics.

3. Protection

- When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the OS itself.
- Protection involves ensuring that all access to resources is controlled.
- Security starts with each user having authenticated to the system by means of a password.

User Operating System Interface

• Two ways that users interface with the OS:
1. Command Interpreter (Command-line interface)
2. Graphical User Interface (GUI)

Command Interpreter

• Main function:
To get and execute the next user-specified command (Figure 1.13).
• The commands are used to manipulate files i.e. create, copy, print, execute, etc.
• Two general ways to implement:
1. Command interpreter itself contains code to execute command.
2. Commands are implemented through system programs. This is used by UNIX.

Graphical User Interfaces

• No entering of commands but the use of a mouse-based window and menu system (Figure 1.14).
• The mouse is used to move a pointer to the position of an icon that represents
→ file
→ program or
→ folder
• By clicking on the icon, the program is invoked.

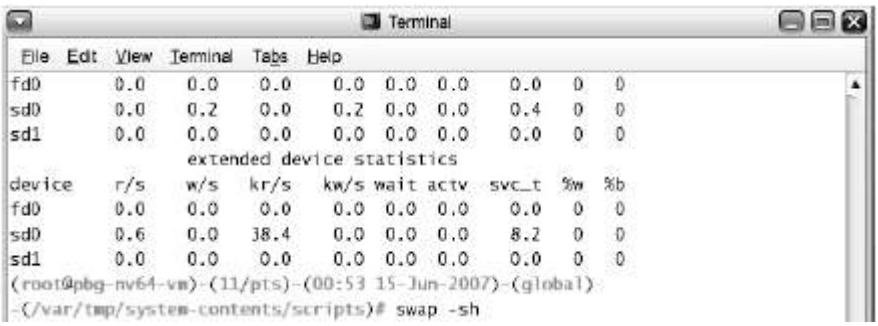


Figure 1.13 The Bourne shell command interpreter in Solaris 10.


Figure 1.14 The iPad touchscreen

System Calls

- These provide an interface to the OS services.
- These are available as routines written in C and C++.
- The programmers design programs according to an API.
  (API=application programming interface).
- The API
  → defines a set of functions that are available to the programmer (Figure 1.15).
  → includes the parameters passed to functions and the return values.
- The functions that make up an API invoke the actual system-calls on behalf of the programmer.
- Benefits of API:
  1. Program portability.
  2. Actual system-calls are more detailed (and difficult) to work with than the API available to the programmer.
- Three general methods are used to pass parameters to the OS:
  1, via registers.
  2. Using a table in memory & the address is passed as a parameter in a register (Figure 1.16).
  3. The use of a stack is also possible where parameters are pushed onto a stack and popped off the stack by the OS.
  3. Using a stack where parameters are
  → pushed onto a stack and
  → popped off the stack.

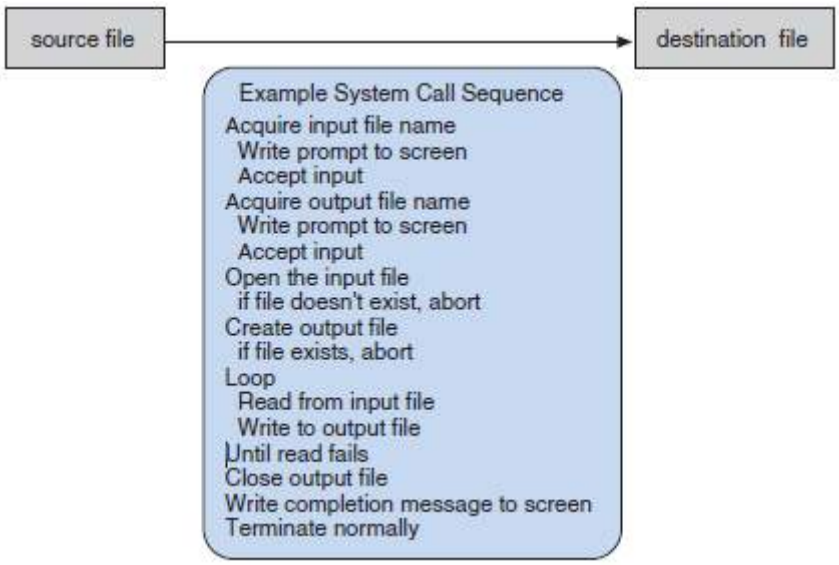


Figure 1.15 Example of how system calls are used.

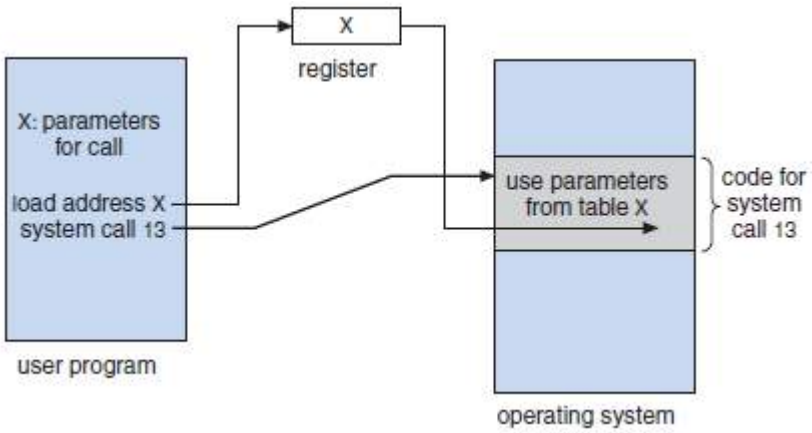


Figure 1.16 Passing of parameters as a table.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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

Types of System Calls

1. Process control
2. File management
3. Device management
4. Information maintenance
5. Communications

Process Control

- System calls used:
  - 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
- A running program needs to be able to halt its execution either normally (**end**) or abnormally (**abort**).
- If program runs into a problem, error message may be generated and dumped into a file.
  This file can be examined by a debugger to determine the cause of the problem.
- The OS must transfer control to the next invoking command interpreter.
  - Command interpreter then reads next command.
  - In interactive system, the command interpreter simply continues with next command.
  - In GUI system, a pop-up window will request action from user.

How to deal with new process?

- A process executing one program can **load** and **execute** another program.
- Where to return control when the loaded program terminates?
  The answer depends on the existing program:
  1) If control returns to the existing program when the new program terminates, we must save the memory image of the existing program. (Thus, we have effectively created a mechanism for one program to call another program).
  2) If both programs continue concurrently, we **create**d a new process to be multiprogrammed.
- We should be able to control the execution of a process. i.e. we should be able to determine and reset the attributes of a process such as:
  → job's priority or
  → maximum execution time
- We may also want to **terminate** process that we created if we find that it
  → is incorrect or
  → is no longer needed.
- We may need to **wait** for processes to finish their execution.
  We may want to wait for a specific event to occur.
- The processes should then signal when that event has occurred.

File Management

- System calls used:
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Working procedure:
  1. We need to **create** and **delete** files.
  2. Once the file is created,
     → we need to **open** it and to use it.
     → we may also **read** *or* **write**.
  3. Finally, we need to **close** the file.
- We need to be able to
  → determine the values of file-attributes and
  → reset the file-attributes if necessary.
- File attributes include
  → file name
  → file type
  → protection codes and
  → accounting information.

Device Management

- System calls used:
  - request device, release device;
  - read, write, reposition;
  - get device attributes, set device attributes;
  - logically attach or detach devices.
- A program may need additional resources to execute.
- Additional resources may be
  → memory
  → tape drives or
  → files.
- If the resources are available, they can be granted, and control can be returned to the user program; If the resources are unavailable, the program may have to wait until sufficient resources are available.
- Files can be thought of as virtual devices. Thus, many of the system calls used for files are also used for devices.
- In multi-user environment,
  1. We must first **request** the device, to ensure exclusive use of it.
  2. After we are finished with the device, we must **release** it.
- Once the device has been requested (and allocated), we can **read** and **write** the device.
- Due to lot of similarity between I/O devices and files, OS (like UNIX) merges the two into a combined *file-device structure***.**
- UNIX merges I/O devices and files into a combined *file-device structure***.**

Information Maintenance

• System calls used:
- 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

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

For ex,

1. Most systems have a system call to return
   → current time and
   → current date.
2. Other system calls may return information about the system, such as
   → number of current users
   → version number of the OS
   → amount of free memory or disk space.
3. The OS keeps information about all its processes, and there are system calls to access this information.

Communication

• System calls used:
- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

• Two models of communication.
1) Message-passing model and 2) Shared Memory Model

Message Passing Model

• Information is exchanged through an IPC provided by OS.
(IPC=inter process communication).

• Steps for communication:
1. Firstly, a connection must be opened using **open connection** system-call.
2. Each computer has a host-name, such as an IP name.
Similarly, each process has a process-name, which is translated into an equivalent identifier.
The **get hostid** & **get processid** system-calls do this translation.
3. Then, identifiers are passed to the **open** and **close** system-calls.
4. The recipient-process must give its permission for communication to take place with an **accept connection** system-call.
(The processes that will be receiving connections are called *daemons processes).*
5. Daemon processes
→ execute a **wait for connection** system-call and
→ are awakened when a connection is made.
6. Then, client & server exchange messages by **read message** and **write message** system calls.
7. Finally, the **close connection** system-call terminates the communication.

• Advantages:
1. Useful when smaller numbers of data need to be exchanged.
2. It is also easier to implement than is shared memory.

Shared Memory Model

• Processes use *map memory* system-calls to gain access to regions of memory owned by other processes.

• Several processes exchange information by reading and writing data in the shared memory.

• The shared memory
→ is determined by the processes and
→ are not under the control of OS.

• The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

• Advantage:
1. Shared memory allows maximum speed and convenience of communication,

• Disadvantage:
1. Problems exist in the areas of protection and synchronization.

System Programs

- They provide a convenient environment for program development and execution.
  (System programs also known as *system utilities*).
- They can be divided into these categories:
- Six categories of system-programs:

1. File Management

➢ These programs manipulate files i.e. create, delete, copy, and rename files.

2. Status Information

➢ Some programs ask the system for
→ date (or time)
→ amount of memory(or disk space) or
→ no. of users.

➢ These information is then printed to the terminal (or output-device or file).

3. File Modification

➢ Text editors can be used to create and modify the content of files stored on disk.

4. Programming Language Support

➢ Compilers, assemblers, and interpreters for common programming-languages (such as C, C++) are provided to the user.

5. Program Loading & Execution

➢The system may provide
→ absolute loaders
→ relocatable loaders
→ linkage editors and
→ overlay loaders.

➢ Debugging-systems are also needed.

6. Communications

➢ These programs are used for creating virtual connections between
→ processes
→ users and
→ computer-systems.

➢ They allow users to
→ browse web-pages
→ send email or
→ log-in remotely.

- Most OSs are supplied with programs that
  → solve common problems or
  → perform common operations. Such programs include
  → web-browsers
  → word-processors
  → spreadsheets and
  → games.

  These programs are known as *application programs***.**

Operating System Design & Implementation

Design Goals

- The first problem in designing a system is to
  - → define goals and
  - → define specifications.
- The design of the system will be affected by
  - → choice of hardware and
  - → type of system such as
    - i) batch or time shared
    - ii) single user or multiuser
- Two basic groups of requirements:
  1. User goals and 2. System goals

User Goals

- The system should be
  - → convenient to use
  - → easy to learn and to use
  - → reliable, safe, and fast.

System Goals

- The system should be
  - → easy to design
  - → implement, and maintain
  - → flexible, reliable, error free, and efficient.

Mechanisms & Policies

- Mechanisms determine how to do something.
- Policies determine what will be done.
- Separating policy and mechanism is important for flexibility.
- Policies change over time; mechanisms should be general.

Implementation

- OS's are nowadays written in higher-level languages like C/C++
- Advantages of higher-level languages:
  1. Faster development and
  2. OS is easier to port.
- Disadvantages of higher-level languages:
  1. Reduced speed and
  2. Increased storage requirements.

Operating System Structure

1. Simple Structure
2. Layered Approach
3. Microkernels
4. Modules

Simple Structure

- These OSs are small, simple, and limited system.
- For example: MS-DOS and UNIX.

1. MS-DOS was written to provide the most functionality in the least space.

➢ Disadvantages:

i) It was not divided into modules carefully (Figure 1.17).

ii) The interfaces and levels of functionality are not well separated.

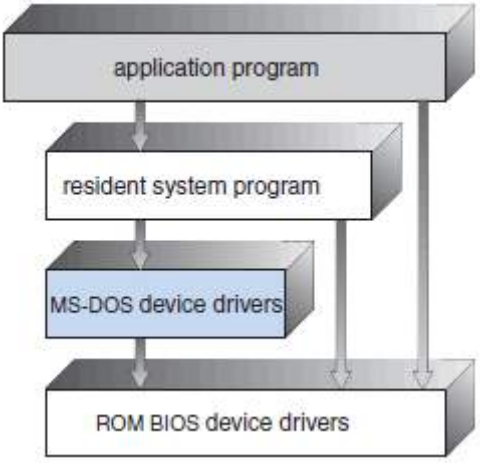


Figure 1.17 MS-DOS layer structure

2. UNIX was initially limited by hardware functionality.

➢ Two parts of UNIX (Figure 1.18):

1. Kernel and
2. System programs.

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

➢ Everything below the system-call interface and above the physical hardware is the kernel.

➢ The kernel provides following functions through system calls:

→ file system

→ CPU scheduling and

→ memory management.

➢ Disadvantage:

1. Difficult to enhance, as changes in one section badly affects other areas.

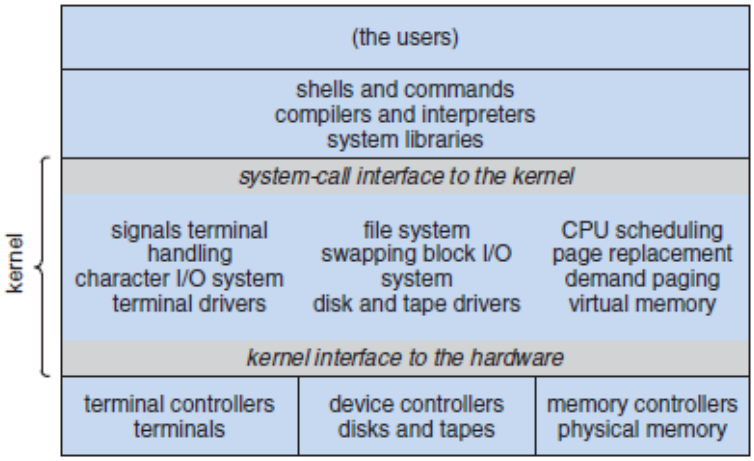


Figure 1.18 Traditional UNIX system structure

Layered Approach

- The OS is divided into a number of layers.
- Each layer is built on the top of another layer.
- The bottom layer is the hardware.
  The highest is the user interface (Figure 1.19).
- A layer is an implementation of an abstract-object.
  i.e. The object is made up of
  → data and
  → operations that can manipulate the data.
- The layer consists of a set of routines that can be invoked by higher-layers.
- Higher-layer
  → does not need to know how lower-layer operations are implemented
  → needs to know only what lower-layer operations do.
- Advantage:
  1. Simplicity of construction and debugging.
- Disadvantages:
  1. Less efficient than other types.
  2. Appropriately defining the various layers.(‘.’ a layer can use only lower-layers, careful planning is necessary).

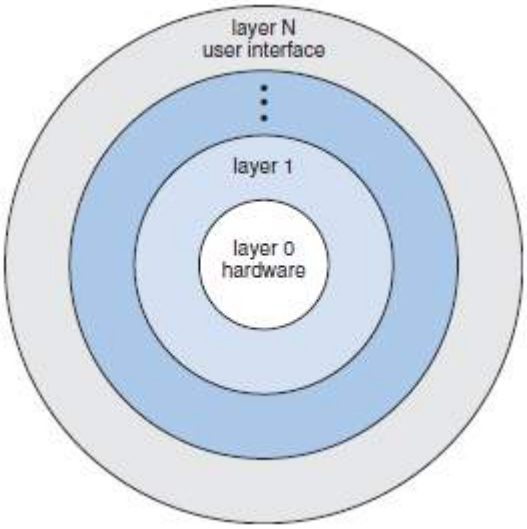


Figure 1.19 A layered OS

MicroKernels

- Main function:
  To provide a communication facility between
  → client program and
  → various services running in user-space.
- Communication is provided by *message passing* (Figure 1.20).
- All non-essential components are
  → removed from the kernel and
  → implemented as system- & user-programs.
- Advantages:
  1. Ease of extending the OS. (New services are added to user space w/o modification of kernel).
  2. Easier to port from one hardware design to another.
  3. Provides more security & reliability.(If a service fails, rest of the OS remains untouched.).
  4. Provides minimal process and memory management.
- Disadvantage:
  1. Performance decreases due to increased system function overhead.

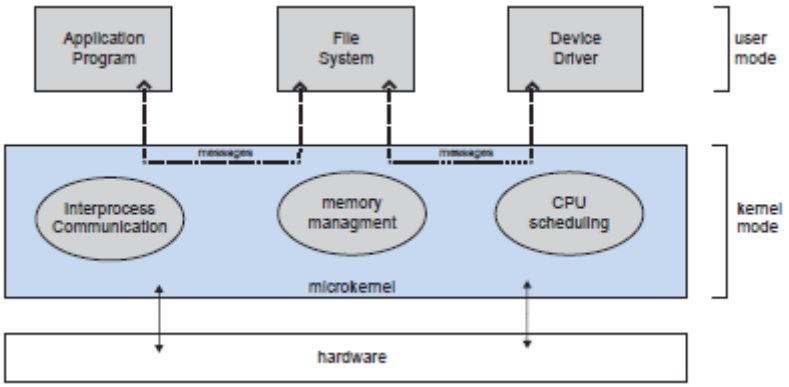


Figure 1.20 Architecture of a typical microkernel

Modules

- The kernel has
  - → set of core components and
  - → dynamic links in additional services during boot time(or run time).
- Seven types of modules in the kernel (Figure 1.21):
  1. Scheduling classes
  2. File systems
  3. Loadable system calls
  4. Executable formats
  5. STREAMS modules
  6. Miscellaneous
  7. Device and bus drivers
- The top layers include
  - → application environments and
  - → set of services providing a graphical interface to applications.
- Kernel environment consists primarily of
  - → Mach microkernel and
  - → BSD kernel.
- Mach provides
  - → memory management;
  - → support for RPCs & IPC and
  - → thread scheduling.
- BSD component provides
  - → BSD command line interface
  - → support for networking and file systems and
  - → implementation of POSIX APIs
- The kernel environment provides an I/O kit for development of
  - → device drivers and
  - → dynamic loadable modules (which Mac OS X refers to as *kernel extensions*).

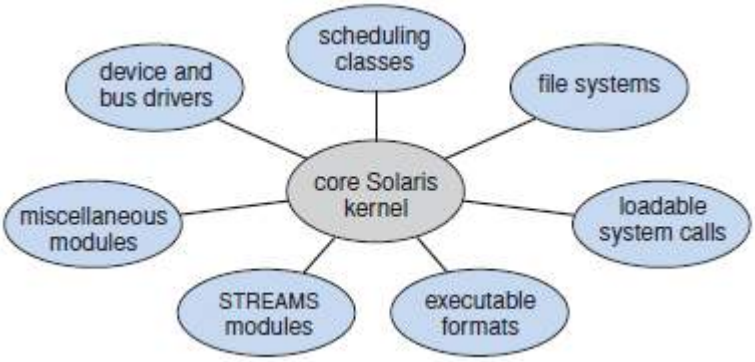


Figure 1.21 Solaris loadable modules

Virtual Machines

• Main idea:
To abstract hardware of a single computer into several different execution environments.
• An OS creates the illusion that a process has
→ own processor &
→ own (virtual) memory.
• The virtual-machine provides
→ an interface that is identical to the underlying hardware (Figure 1.22).
→ a (virtual) copy of the underlying computer to each process.

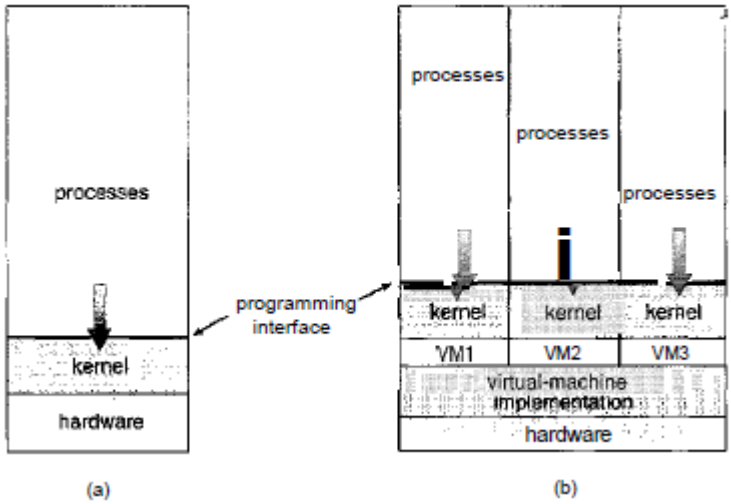


Figure 1.22 System models, (a) Nonvirtual machine, (b) Virtual machine.

• Problem: Virtual-machine software itself will need substantial disk space to provide virtual memory. Solution: provide virtual disks that are identical in all respects except size.
• Advantages:
1. Complete protection of the various system resources.
2. It is a perfect vehicle for OS's R&D.
• Disadvantage:
1. Difficult to implement due to effort required to provide an exact duplicate to underlying machine.

Operating System Generation

• OS is designed to run on any of a class of machines.
However, the system must be configured for each specific computer site
• SYSGEN is used for configuring a system for each specific computer site
• SYSGEN program must determine:
1. What CPU will be used?
2. How will boot disk be formatted?
3. How much memory is available?
4. What devices are available?
5. What OS options are desired?
• A system-administrator can use the above information to modify a copy of the source code of the OS

System Boot

• Booting means starting a computer by loading the kernel.
• Bootstrap program is a code stored in ROM.
• The bootstrap program
→ locates the kernel
→ loads the kernel into main memory and
→ starts execution of kernel.
• OS must be made available to hardware so hardware can start it.

UNIT 2: PROCESS CONCEPTS

Process Concept

- A process is the unit-of-work.
- A system consists of a collection of processes:
    1. **OS process** can execute system-code and
    2. **User process** can execute user-code.

The Process

- A process is a program in execution.
- It also includes (Figure 2.1):
    1. **Program counter** to indicate the current activity.
    2. **Registers content** of the processor.
    3. **Process stack** contains temporary data.
    4. **Data section** contains global variables.
    5. **Heap** is memory that is dynamically allocated during process run time.
- A program by itself is not a process.
    1. A process is an active-entity.
    2. A program is a passive-entity such as an executable-file stored on disk.
- A program becomes a process when an executable-file is loaded into memory.
- If you run many copies of a program, each is a separate process.
    The text-sections are equivalent, but the data-sections vary.

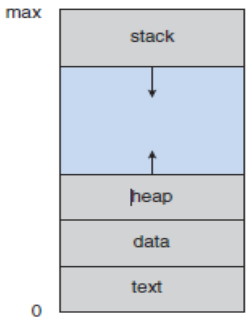


Figure 2.1 Process in memory

Process State

- As a process executes, it changes state.
- Each process may be in one of the following states (Figure 2.2):
    1. New: The process is being created.
    2. Running: Instructions are being executed.
    3. Waiting: The process is waiting for some event to occur (such as I/0 completions).
    4. Ready: The process is waiting to be assigned to a processor.
    5. Terminated: The process has finished execution.
- Only one process can be *running* on any processor at any instant.

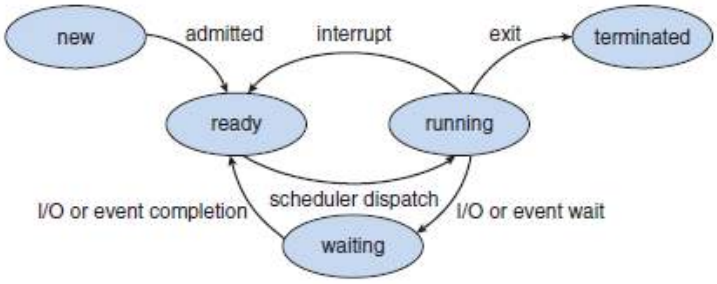


Figure 2.2 Diagram of process state

Process Control Block

• In OS, each process is represented by a PCB (Process Control Block).

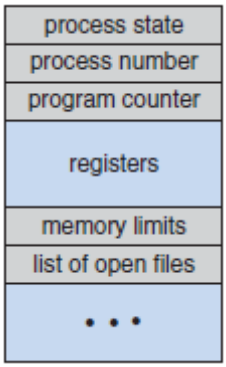


Figure 2.3 Process control block (PCB)

• PCB contains following information about the process (Figure 2.3):

1. Process State

➢ The current state of process may be
→ new
→ ready
→ running
→ waiting or
→ halted.

2. Program Counter

➢ This indicates the address of the next instruction to be executed for the process.

3. CPU Registers

➢ These include
→ accumulators (AX)
→ index registers (SI, DI)
→ stack pointers (SP) and
→ general-purpose registers (BX, CX, DX).

4. CPU Scheduling Information

➢ This includes
→ priority of process
→ pointers to scheduling-queues and
→ scheduling-parameters.

5. Memory Management Information

➢ This includes
→ value of base- & limit-registers and
→ value of page-tables(or segment-tables).

6. Accounting Information

➢ This includes
→ amount of CPU time
→ time-limit and
→ process-number.

7. I/O Status Information

➢ This includes
→ list of I/O devices
→ list of open files.

Process Scheduling

• Objective of multiprogramming:
To have some process running at all times to maximize CPU utilization.
Objective of time-sharing:
To switch the CPU between processes so frequently that users can interact with each program while it is running.

• To meet above 2 objectives: **Process scheduler** is used to select an available process for program-execution on the CPU.

Scheduling Queues

• Three types of scheduling-queues:

1. Job Queue
- This consists of all processes in the system.
- As processes enter the system, they are put into a job-queue.

2. Ready Queue
- This consists of the processes that are
  → residing in main-memory and
  → ready & waiting to execute (Figure 2.4).
- This queue is generally stored as a **linked list**.
- A ready-queue header contains pointers to the first and final PCBs in the list.
- Each PCB has a pointer to the next PCB in the ready-queue.

3. Device Queue
- This consists of the processes that are waiting for an I/O device.
- Each device has its own device-queue.

• When the process is executing, one of following events could occur (Figure 2.5):
1. The process could issue an I/0 request and then be placed in an I/0 queue.
2. The process could create a new subprocess and wait for the subprocess's termination.
3. The process could be interrupted and put back in the ready-queue.

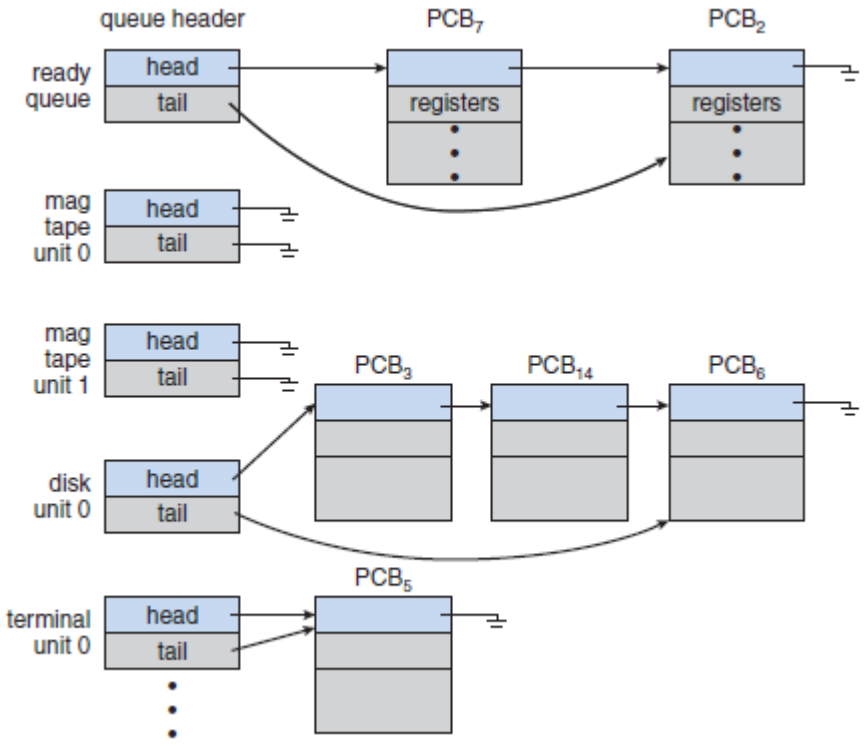


Figure 2.4 The ready-queue and various I/O device-queues

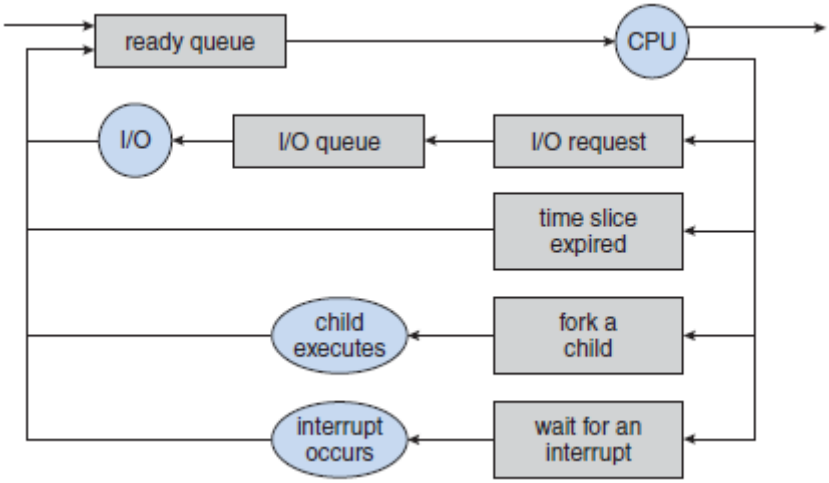


Figure 2.5 Queueing-diagram representation of process scheduling

Schedulers

• Three types of schedulers:
1. Long-term scheduler
2. Short-term scheduler and
3. Medium-term schedulers

Long-term Scheduler	***Short-term Scheduler***
Also called job scheduler.	Also called CPU scheduler.
Selects which processes should be brought into the ready-queue.	Selects which process should be executed next and allocates CPU.
Need to be invoked only when a process leaves the system and therefore executes much less frequently.	Need to be invoked to select a new process for the CPU and therefore executes much more frequently.
May be slow ‘,’ minutes may separate the creation of one new process and the next.	Must be fast ‘,’ a process may execute for only a few milliseconds.
Controls the degree of multiprogramming.	

• Processes can be described as either:

1. I/O-bound Process
- ➢ Spends more time doing I/O operation than doing computations.
- ➢ Many short CPU bursts.

2. CPU-bound Process
- ➢ Spends more time doing computations than doing I/O operation.
- ➢ Few very long CPU bursts.

• Why long-term scheduler should select a good process mix of I/O-bound and CPU-bound processes ?

Ans: 1) If all processes are I/0 bound, then
- i) Ready-queue will almost always be empty, and
- ii) Short-term scheduler will have little to do.

2) If all processes are CPU bound, then
- i) I/0 waiting queue will almost always be empty (devices will go unused) and
- ii) System will be unbalanced.

• Some time-sharing systems have **medium-term scheduler** (Figure 2.6).
- ➢ The scheduler removes processes from memory and thus reduces the degree of multiprogramming.
- ➢ Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- ➢ The process is swapped out, and is later swapped in, by the scheduler.

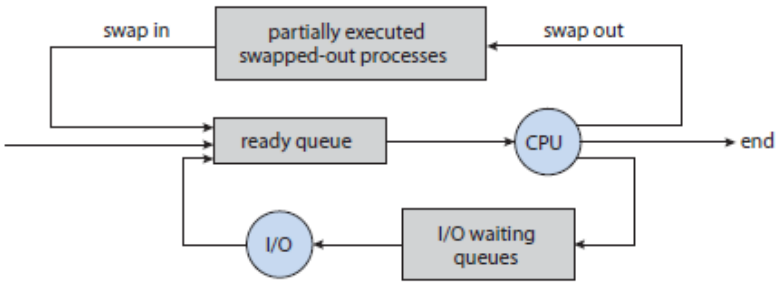


Figure 2.6 Addition of medium-term scheduling to the queueing diagram

Context Switch

• Context-switch means saving the state of the old process and switching the CPU to another process**.**

• The context of a process is represented in the PCB of the process; it includes
- → value of CPU registers
- → process-state and
- → memory-management information.

• Disadvantages:
1. Context-switch time is pure overhead, because the system does no useful work while switching.
2. Context-switch times are highly dependent on hardware support.

Operations on Processes

1. Process Creation and
2. Process Termination

Process Creation

- A process may create a new process via a **create-process** system-call.
- The creating process is called a *parent-process*.
  The new process created by the parent is called the *child-process* (Sub-process).
- OS identifies processes by pid (process identifier), which is typically an integer-number.
- A process needs following resources to accomplish the task:
  → CPU time
  → memory and
  → I/0 devices.
- Child-process may
  → get resources directly from the OS or
  → get resources of parent-process. This prevents any process from overloading the system.
- Two options exist when a process creates a new process:
  1. The parent & the children execute concurrently.
  2. The parent waits until all the children have terminated.
- Two options exist in terms of the address-space of the new process:
  1. The child-process is a duplicate of the parent-process (it has the same program and data as the parent).
  2. The child-process has a new program loaded into it.

Process creation in UNIX

- In UNIX, each process is identified by its process identifier (pid), which is a unique integer.
- A new process is created by the **fork()** system-call (Figure 2.7 & 2.8).
- The new process consists of a copy of the address-space of the original process.
- Both the parent and the child continue execution with one difference:
  1. The return value for the fork() is
     zero for the new (child) process.
  2. The return value for the fork() is
     nonzero pid of the child for the parent-process.
- Typically, the **exec()** system-call is used after a fork() system-call by one of the two processes to replace the process's memory-space with a new program.
- The parent can issue **wait()** system-call to move itself off the ready-queue.

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

int main()
{
pid_t pid;

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

   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

Figure 2.7 Creating a separate process using the UNIX fork() system-call

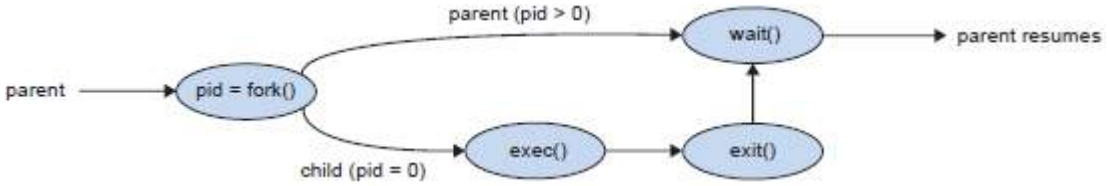


Figure 2.8 Process creation using the fork() system-call

Process Termination

- A process terminates when it executes the last statement (in the program).
- Then, the OS deletes the process by using **exit()** system-call.
- Then, the OS de-allocates all the resources of the process. The resources include
  - → memory
  - → open files and
  - → I/0 buffers.
- Process termination can occur in following cases:
  - → A process can cause the termination of another process via **TerminateProcess()** system-call.
  - → Users could arbitrarily **kill** the processes.
- A parent terminates the execution of children for following reasons:
  1. The child has exceeded its usage of some resources.
  2. The task assigned to the child is no longer required.
  3. The parent is exiting, and the OS does not allow a child to continue.
- In some systems, if a process terminates, then all its children must also be terminated. This phenomenon is referred to as **cascading termination**.

Inter Process Communication (IPC)

• Processes executing concurrently in the OS may be either
1. Independent processes or 2. Co-operating processes.

1. A process is ***independent*** if
   i) The process cannot affect or be affected by the other processes.
   ii) The process does not share data with other processes.
2. A process is ***co-operating*** if
   i) The process can affect or be affected by the other processes.
   ii) The process shares data with other processes.

• Advantages of process co-operation:

1. Information Sharing
- Since many users may be interested in same piece of information (ex: shared file).

2. Computation Speedup
- We must break the task into subtasks.
- Each subtask should be executed in parallel with the other subtasks.
- The speed can be improved only if computer has multiple processing elements such as
  → CPUs or
  → I/O channels.

3. Modularity
- Divide the system-functions into separate processes or threads.

4. Convenience
- An individual user may work on many tasks at the same time.
- For ex, a user may be editing, printing, and compiling in parallel.

• Two basic models of IPC (Figure 2.9):
1) Shared-memory and
2) Message passing.

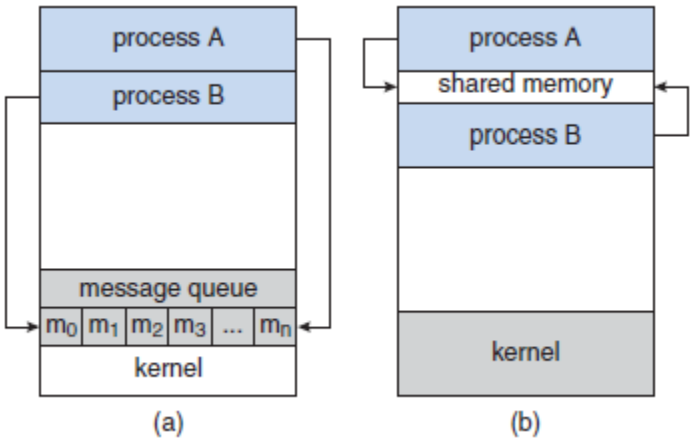


Figure 2.9 Communications models. (a) Message passing. (b) Shared-memory

Shared-Memory Systems

• Communicating-processes must establish a region of shared-memory.
• A shared-memory resides in address-space of the process creating the shared-memory.
Other processes must attach their address-space to the shared-memory.
• The processes can then exchange information by reading and writing data in the shared-memory.
• The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
• For ex, **producer-consumer problem**:
Producer-process produces information that is consumed by a consumer-process
• Two types of buffers can be used:
1. ***Unbounded-buffer*** places no practical limit on the size of the buffer.
2. ***Bounded-buffer*** assumes that there is a fixed buffer-size.
• Advantages:
1. Allows maximum speed and convenience of communication.
2. Faster.

Message-Passing Systems

• These allow processes to communicate and to synchronize their actions without sharing the same address-space.
• For example, a chat program used on the WWW.
• Messages can be of either
    1) Fixed size or 2) Variable size.
        1. If ***fixed-sized messages*** are used, the system-level implementation is simple.
        ➢ However, the programming task becomes more difficult.
        2. If ***variable-sized messages*** are used, the system-level implementation is complex.
        ➢ However, the programming task becomes simpler.
• A communication-link must exist between processes to communicate
• Three methods for implementing a link:
    1. Direct or indirect communication.
    2. Symmetric or asymmetric communication.
    3. Automatic or explicit buffering.
• Two operations:
    1. send(P,message): Send a message to process P.
    2. receive(Q,message): Receive a message from process Q.
• Advantages:
    1. Useful for exchanging smaller amounts of data ('.' No conflicts need be avoided).
    2. Easier to implement.
    3. Useful in a distributed environment.

Naming

• Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Direct Communication	***Indirect Communication***
Each process must explicitly name the recipient/sender.	Messages are sent to/received from mailboxes (or ports).
<u>Properties of a communication link:</u> ➢ A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate. ➢ A link is associated with exactly two processes. ➢ Exactly one link exists between each pair of processes.	<u>Properties of a communication link:</u> ➢ A link is established between a pair of processes only if both members have a shared mailbox. ➢ A link may be associated with more than two processes. ➢ A number of different links may exist between each pair of communicating processes.
<u>Symmetric addressing:</u> ➢ Both sender and receiver processes must name the other to communicate.	<u>Mailbox owned by a process:</u> ➢ The owner can only receive, and the user can only send. ➢ The mailbox disappears when its owner process terminates.
<u>Asymmetric addressing:</u> ➢ Only the sender names the recipient; the recipient needn't name the sender.	<u>Mailbox owned by the OS:</u> ➢ The OS allows a process to: 1. Create a new mailbox 2. Send & receive messages via it 3. Delete a mailbox.

Synchronization

• Message passing may be either blocking or non-blocking (also known as synchronous and asynchronous).

Synchronous Message Passing	***Asynchronous Message Passing***
Blocking send: ➢The sending process is blocked until the message is received by the receiving process or by the mailbox.	Non-blocking send: ➢ The sending process sends the message and resumes operation.
Blocking receive: ➢The receiver blocks until a message is available.	Non-blocking receive: ➢ The receiver retrieves either a valid message or a null.

Buffering

• Messages exchanged by processes reside in a temporary queue.
• Three ways to implement a queue:

1. Zero Capacity
- ➢ The queue-length is zero.
- ➢ The link can't have any messages waiting in it.
- ➢ The sender must block until the recipient receives the message.

2. Bounded Capacity
- ➢ The queue-length is finite.
- ➢ If the queue is not full, the new message is placed in the queue.
- ➢ The link capacity is finite.
- ➢ If the link is full, the sender must block until space is available in the queue.

3. Unbounded Capacity
- ➢ The queue-length is potentially infinite.
- ➢ Any number of messages can wait in the queue.
- ➢ The sender never blocks.

UNIT 2(CONT.): MULTITHREADED PROGRAMMING

Multithreaded Programming

- A thread is a basic unit of CPU utilization.
- It consists of
  - → thread ID
  - → PC
  - → register-set and
  - → stack.
- It shares with other threads belonging to the same process its code-section & data-section.
- A traditional (or heavy weight) process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time. Such a process is called **multithreaded process** (Figure 2.10).

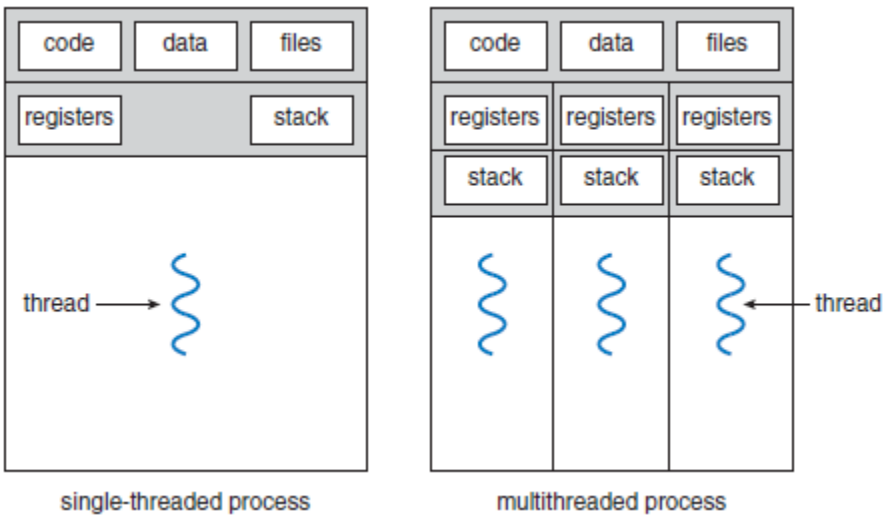


Figure 2.10 Single-threaded and multithreaded processes

Motivation for Multithreaded Programming

1. The software-packages that run on modern PCs are multithreaded.
   An application is implemented as a separate process with several threads of control.
   For ex: A word processor may have
   → first thread for displaying graphics
   → second thread for responding to keystrokes and
   → third thread for performing grammar checking.
2. In some situations, a single application may be required to perform several similar tasks.
   For ex: A web-server may create a separate thread for each client request.
   This allows the server to service several concurrent requests.
3. RPC servers are multithreaded.
   When a server receives a message, it services the message using a separate thread.
   This allows the server to service several concurrent requests.
4. Most OS kernels are multithreaded;
   Several threads operate in kernel, and each thread performs a specific task, such as
   → managing devices or
   → interrupt handling.

Benefits of Multithreaded Programming

1. Responsiveness

• A program may be allowed to continue running even if part of it is blocked.
Thus, increasing responsiveness to the user.

2. Resource Sharing

• By default, threads share the memory (and resources) of the process to which they belong.
Thus, an application is allowed to have several different threads of activity within the same address-space.

3. Economy

• Allocating memory and resources for process-creation is costly.
Thus, it is more economical to create and context-switch threads.

4. Utilization of Multiprocessor Architectures

• In a multiprocessor architecture, threads may be running in parallel on different processors.
Thus, parallelism will be increased.

Multithreading Models

- Support for threads may be provided at either
    1. The user level, for **user threads** or
    2. By the kernel, for **kernel threads**.
- User-threads are supported above the kernel and are managed without kernel support.
  Kernel-threads are supported and managed directly by the OS.
- Three ways of establishing relationship between user-threads & kernel-threads:
    1. Many-to-one model
    2. One-to-one model and
    3. Many-to-many model.

Many-to-One Model

- Many user-level threads are mapped to one kernel thread (Figure 2.11).
- Advantage:
    1. Thread management is done by the thread library in user space, so it is efficient.
- Disadvantages:
    1. The entire process will block if a thread makes a blocking system-call.
    2. Multiple threads are unable to run in parallel on multiprocessors.
- For example:
    → Solaris green threads
    → GNU portable threads.

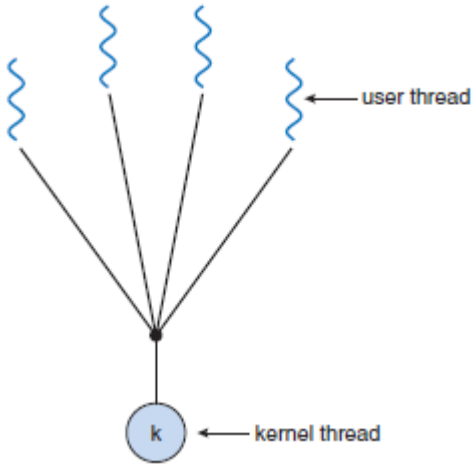


Figure 2.11 Many-to-one model

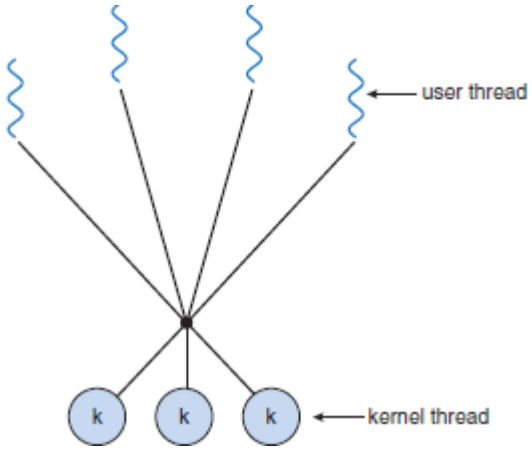


Figure 2.12 One-to-one model

One-to-One Model

- Each user thread is mapped to a kernel thread (Figure 2.12).
- Advantages:
    1. It provides more concurrency by allowing another thread to run when a thread makes a blocking system-call.
    2. Multiple threads can run in parallel on multiprocessors.
- Disadvantage:
    1. Creating a user thread requires creating the corresponding kernel thread.
- For example:
    → Windows NT/XP/2000
    → Linux

Many-to-Many Model

- Many user-level threads are multiplexed to a smaller number of kernel threads (Figure 2.13).
- Advantages:
  1. Developers can create as many user threads as necessary
  2. The kernel threads can run in parallel on a multiprocessor.
  3. When a thread performs a blocking system-call, kernel can schedule another thread for execution.

Two Level Model

- A variation on the many-to-many model is the two level-model (Figure 2.14).
- Similar to M:M, except that it allows a user thread to be bound to kernel thread.
- For example:
  - → HP-UX
  - → Tru64 UNIX

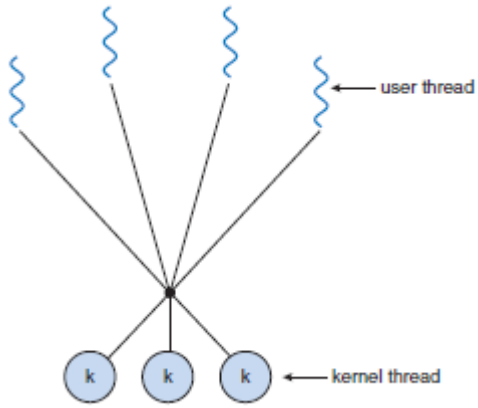


Figure 2.13 Many-to-many model

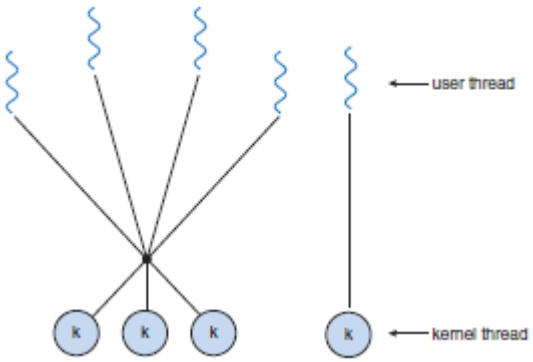


Figure 2.14 Two-level model

Thread Libraries

- It provides the programmer with an API for the creation and management of threads.
- Two ways of implemention:
  - ***1. First Approach***
    - Provides a library entirely in user space with no kernel support.
    - All code and data structures for the library exist in the user space.
  - ***2. Second Approach***
    - Implements a kernel-level library supported directly by the OS.
    - Code and data structures for the library exist in kernel space.
- Three main thread libraries:
  1. POSIX Pthreads
  2. Win32 and
  3. Java.

Pthreads

- This is a POSIX standard API for thread creation and synchronization.
- This is a specification for thread-behavior, not an implementation.
- OS designers may implement the specification in any way they wish.
- Commonly used in: UNIX and Solaris.

Java Threads

- Threads are the basic model of program-execution in
  - → Java program and
  - → Java language.
- The API provides a rich set of features for the creation and management of threads.
- All Java programs comprise at least a single thread of control.
- Two techniques for creating threads:
  1. Create a new class that is derived from the Thread class and override its run() method.
  2. Define a class that implements the Runnable interface. The Runnable interface is defined as follows:

```
public interface Runnable
{
    public abstract void run();
}
```

Threading Issues

fork() and exec() System-calls

• fork() is used to create a separate, duplicate process.
• If one thread in a program calls fork(), then
    1. Some systems duplicates all threads and
    2. Other systems duplicate only the thread that invoked the forkO.
• If a thread invokes the exec(), the program specified in the parameter to exec() will replace the entire process including all threads.

Thread Cancellation

• This is the task of terminating a thread before it has completed.
• Target thread is the thread that is to be canceled
• Thread cancellation occurs in two different cases:
    1. ***Asynchronous cancellation***: One thread immediately terminates the target thread.
    2. ***Deferred cancellation***: The target thread periodically checks whether it should be terminated.

Signal Handling

• In UNIX, a signal is used to notify a process that a particular event has occurred.
• All signals follow this pattern:
    1. A signal is generated by the occurrence of a certain event.
    2. A generated signal is delivered to a process.
    3. Once delivered, the signal must be handled.
• A signal handler is used to process signals.
• A signal may be received either synchronously or asynchronously, depending on the source.
    1. Synchronous signals
    ➢ Delivered to the same process that performed the operation causing the signal.
    ➢ E.g. illegal memory access and division by 0.
    2. Asynchronous signals
    ➢ Generated by an event external to a running process.
    ➢ E.g. user terminating a process with specific keystrokes <ctrl><c>.
• Every signal can be handled by one of two possible handlers:
    1. A Default Signal Handler
    ➢ Run by the kernel when handling the signal.
    2. A User-defined Signal Handler
    ➢ Overrides the default signal handler.
• In ***single-threaded programs***, delivering signals is simple (since signals are always delivered to a process).
In ***multithreaded programs***, delivering signals is more complex. Then, the following options exist:
    1. Deliver the signal to the thread to which the signal applies.
    2. Deliver the signal to every thread in the process.
    3. Deliver the signal to certain threads in the process.
    4. Assign a specific thread to receive all signals for the process.

Thread Pools

• The basic idea is to
    → create a no. of threads at process-startup and
    → place the threads into a pool (where they sit and wait for work).
• Procedure:
    1. When a server receives a request, it awakens a thread from the pool.
    2. If any thread is available, the request is passed to it for service.
    3. Once the service is completed, the thread returns to the pool.
• Advantages:
    1. Servicing a request with an existing thread is usually faster than waiting to create a thread.
    2. The pool limits the no. of threads that exist at any one point.
• No. of threads in the pool can be based on factors such as
    → no. of CPUs
    → amount of memory and
    → expected no. of concurrent client-requests.

UNIT 2(CONT.): PROCESS SCHEDULING

Basic Concepts

- In a single-processor system,
  → only one process may run at a time.
  → other processes must wait until the CPU is rescheduled.
- Objective of multiprogramming:
  → to have some process running at all times, in order to maximize CPU utilization.

CPU-I/0 Burst Cycle

- Process execution consists of a cycle of
  → CPU execution and
  → I/O wait (Figure 2.15 & 2.16).
- Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc…
- Finally, a CPU burst ends with a request to terminate execution.
- An I/O-bound program typically has many short CPU bursts.
  A CPU-bound program might have a few long CPU bursts.

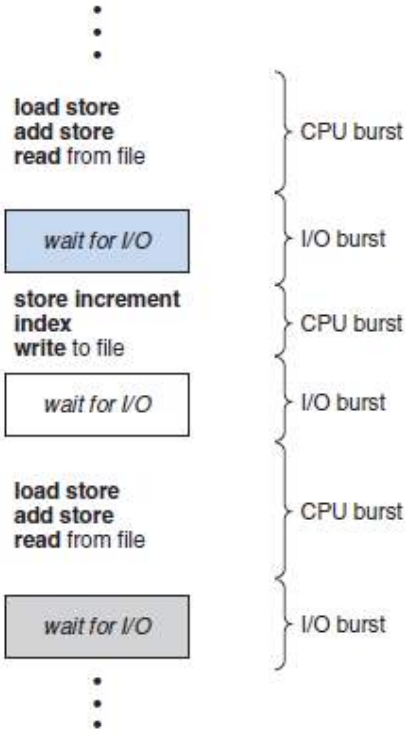


Figure 2.15 Alternating sequence of CPU and I/O bursts

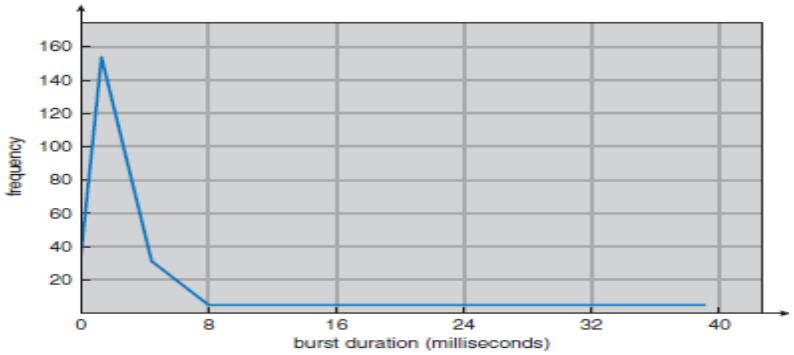


Figure 2.16 Histogram of CPU-burst durations

CPU Scheduler

- This scheduler
  - → selects a waiting-process from the ready-queue and
  - → allocates CPU to the waiting-process.
- The ready-queue could be a FIFO, priority queue, tree and list.
- The records in the queues are generally process control blocks (PCBs) of the processes.

CPU Scheduling

- Four situations under which CPU scheduling decisions take place:
  1. When a process switches from the running state to the waiting state.
     For ex; I/O request.
  2. When a process switches from the running state to the ready state.
     For ex: when an interrupt occurs.
  3. When a process switches from the waiting state to the ready state.
     For ex: completion of I/O.
  4. When a process terminates.
- Scheduling under 1 and 4 is non-preemptive.
  Scheduling under 2 and 3 is preemptive.

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

- This is driven by the idea of prioritized computation.
- Processes that are runnable may be temporarily suspended
- Disadvantages:
  1. Incurs a cost associated with access to shared-data.
  2. Affects the design of the OS kernel.

Dispatcher

- It gives control of the CPU to the process selected by the short-term scheduler.
- The function involves:
  1. Switching context
  2. Switching to user mode &
  3. Jumping to the proper location in the user program to restart that program.
- It should be as fast as possible, since it is invoked during every process switch.
- ***Dispatch latency*** means the time taken by the dispatcher to
  - → stop one process and
  - → start another running.

Scheduling Criteria

- Different CPU-scheduling algorithms
  → have different properties and
  → may favor one class of processes over another.
- Criteria to compare CPU-scheduling algorithms:
  1. CPU Utilization
  - We must keep the CPU as busy as possible.
  - In a real system, it ranges from 40% to 90%.

  2. Throughput
  - Number of processes completed per time unit.
  - For long processes, throughput may be 1 process per hour;
    For short transactions, throughput might be 10 processes per second.

  3. Turnaround Time
  - The interval from the time of submission of a process to the time of completion.
  - Turnaround time is the sum of the periods spent
    → waiting to get into memory
    → waiting in the ready-queue
    → executing on the CPU and
    → doing I/O.

  4. Waiting Time
  - The amount of time that a process spends waiting in the ready-queue.

  5. Response Time
  - The time from the submission of a request until the first response is produced.
  - The time is generally limited by the speed of the output device.
- We want
  → 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.

• Following are some scheduling algorithms:

1. FCFS scheduling (First Come First Served)
2. Round Robin scheduling
3. SJF scheduling (Shortest Job First)
4. SRT scheduling
5. Priority scheduling
6. Multilevel Queue scheduling and
7. Multilevel Feedback Queue scheduling

FCFS Scheduling

• The process that requests the CPU first is allocated the CPU first.

• The implementation is easily done using a FIFO queue.

• Procedure:

1. When a process enters the ready-queue, its PCB is linked onto the tail of the queue.
2. When the CPU is free, the CPU is allocated to the process at the queue's head.
3. The running process is then removed from the queue.

• Advantage:

1. Code is simple to write & understand.

• Disadvantages:

1. **Convoy effect:** All other processes wait for one big process to get off the CPU.
2. Non-preemptive (a process keeps the CPU until it releases it).
3. Not good for time-sharing systems.
4. The average waiting time is generally not minimal.

• Example: Suppose that the processes arrive in the order P1, P2, P3.

Process	Burst Time
P_1	24
P_2	3
P_3	3

• The Gantt Chart for the schedule is as follows:

• 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 as follows:

• Waiting time for P1 = 6;P2 = 0; P3 = 3

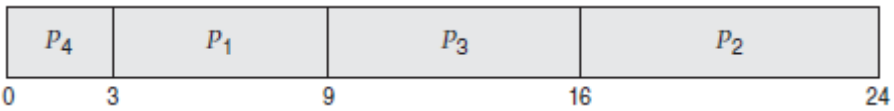
Average waiting time: (6 + 0 + 3)/3 = 3

SJF Scheduling

• The CPU is assigned to the process that has the smallest next CPU burst.
• If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.
• For long-term scheduling in a batch system, we can use the process time limit specified by the user, as the 'length'
• SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst
• Advantage:
    1. The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.
• Disadvantage:
    1. Determining the length of the next CPU burst.
• SJF algorithm may be either 1) non-preemptive or 2) preemptive.
    1. Non preemptive SJF
    ➢ The current process is allowed to finish its CPU burst.
    2. Preemptive SJF
    ➢ If the new process has a shorter next CPU burst than what is left of the executing process, that process is preempted.
    ➢ It is also known as **SRTF** scheduling (Shortest-Remaining-Time-First).
• Example (for non-preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

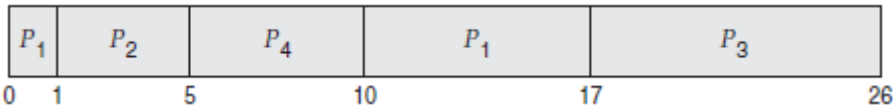
• For non-preemptive SJF, the Gantt Chart is as follows:



• Waiting time for P1 = 3; P2 = 16; P3 = 9; P4=0
    Average waiting time: (3 + 16 + 9 + 0)/4 = 7
• Example (preemptive SJF): Consider the following set of processes, with the length of the CPU-burst time given in milliseconds.

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

• For preemptive SJF, the Gantt Chart is as follows:



• The average waiting time is ((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6.5.

Priority Scheduling

- A priority is associated with each process.
- The CPU is allocated to the process with the highest priority.
- Equal-priority processes are scheduled in FCFS order.
- Priorities can be defined either internally or externally.
    1. ***Internally-defined*** priorities.
    - Use some measurable quantity to compute the priority of a process.
    - For example: time limits, memory requirements, no. of open files.

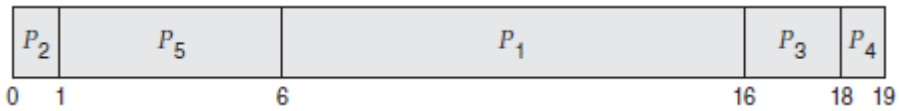
    2. ***Externally-defined*** priorities.
    - Set by criteria that are external to the OS
    - For example:
        → importance of the process
        → political factors
- Priority scheduling can be either preemptive or nonpreemptive.
    1. ***Preemptive***
    - The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process.

    2. ***Non Preemptive***
    - The new process is put at the head of the ready-queue
- Advantage:
    1. Higher priority processes can be executed first.
- Disadvantage:
    1. Indefinite blocking, where low-priority processes are left waiting indefinitely for CPU.

  Solution: ***Aging*** is a technique of increasing priority of processes that wait in system for a long time.
- Example: Consider the following set of processes, assumed to have arrived at time 0, in the order PI, P2, ..., P5, with the length of the CPU-burst time given in milliseconds.

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

- The Gantt chart for the schedule is as follows:



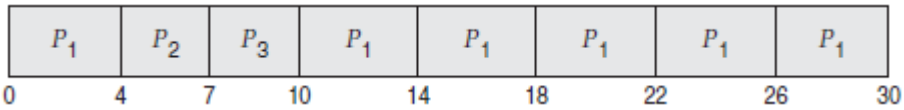
- The average waiting time is 8.2 milliseconds.

Round Robin Scheduling

- Designed especially for timesharing systems.
- It is similar to FCFS scheduling, but with preemption.
- A small unit of time is called a **time quantum** *(or time slice).*
- Time quantum is ranges from 10 to 100 ms.
- The ready-queue is treated as a **circular queue**.
- The CPU scheduler
  → goes around the ready-queue and
  → allocates the CPU to each process for a time interval of up to 1 time quantum.
- To implement:
  The ready-queue is kept as a FIFO queue of processes
- CPU scheduler
  1. Picks the first process from the ready-queue.
  2. Sets a timer to interrupt after 1 time quantum and
  3. Dispatches the process.
- One of two things will then happen.
  1. The process may have a CPU burst of less than 1 time quantum.
     In this case, the process itself will release the CPU voluntarily.
  2. 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 OS.
     The process will be put at the tail of the ready-queue.
- Advantage:
  1. Higher average turnaround than SJF.
- Disadvantage:
  1. Better response time than SJF.
- Example: Consider the following set of processes that arrive at time 0, with the length of the CPU-burst time given in milliseconds.

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart for the schedule is as follows:



- The average waiting time is 17/3 = 5.66 milliseconds.
- *The RR scheduling algorithm is preemptive.*
  No process is allocated the CPU for more than *1* time quantum in a row. If a process' CPU burst exceeds *1* time quantum, that process is preempted and is put back in the ready-queue..
- The performance of algorithm depends heavily on the size of the time quantum (Figure 2.17 & 2.18).
  1. If time quantum=very large, RR policy is the same as the FCFS policy.
  2. If time quantum=very small, RR approach appears to the users as though each of n processes has its own processor running at l/n the speed of the real processor.
- In software, we need to consider the effect of context switching on the performance of RR scheduling
  1. Larger the time quantum for a specific process time, less time is spend on context switching.
  2. The smaller the time quantum, more overhead is added for the purpose of context-switching.

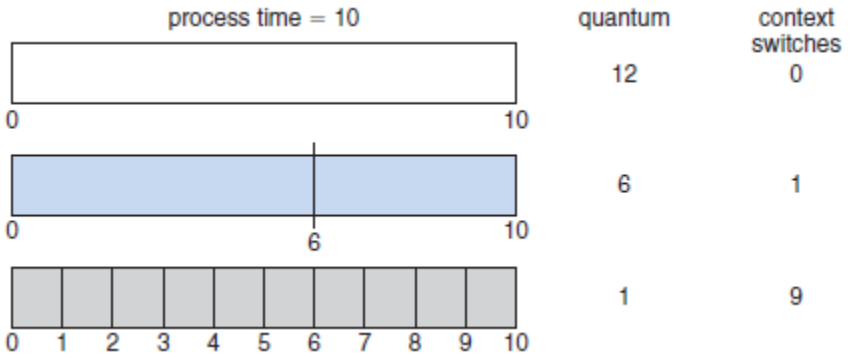


Figure 2.17 How a smaller time quantum increases context switches

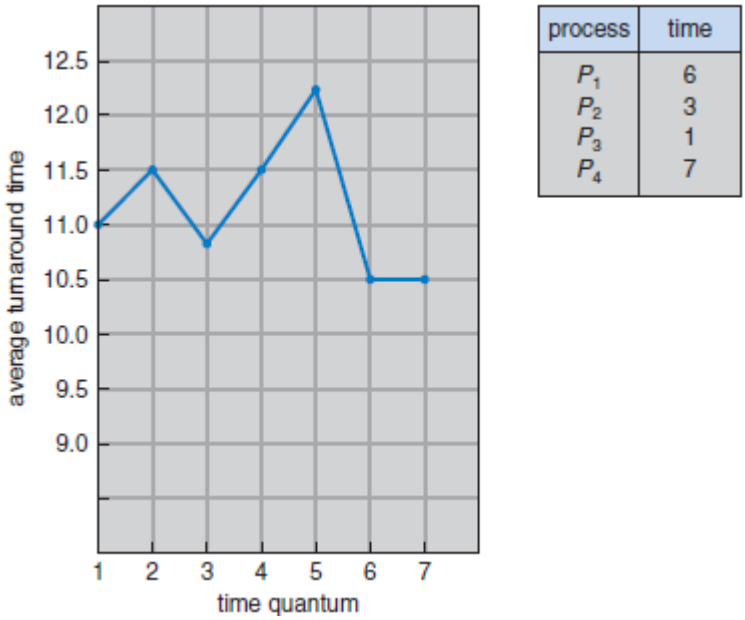


Figure 2.18 How turnaround time varies with the time quantum

Multilevel Queue Scheduling

- Useful for situations in which processes are easily classified into different groups.
- For example, a common division is made between
  → foreground (or interactive) processes and
  → background (or batch) processes.
- The ready-queue is partitioned into several separate queues (Figure 2.19).
- The processes are permanently assigned to one queue based on some property like
  → 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.

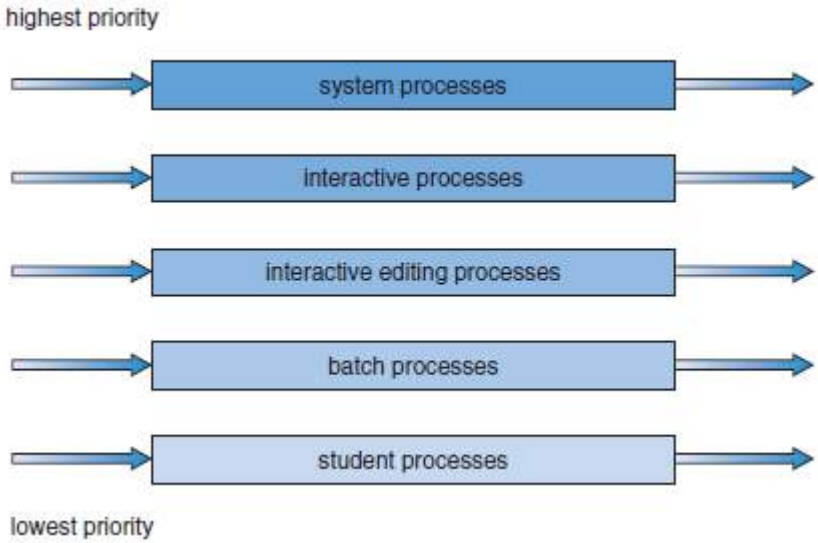


Figure 2.19 Multilevel queue scheduling

- 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.
- **Time slice**: each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  20% to background in FCFS

Multilevel Feedback Queue Scheduling

- A process may move between queues (Figure 2.20).
- The basic idea:

Separate processes according to the features of their CPU bursts. For example

1. 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.
2. If a process waits too long in a lower-priority queue, it may be moved to a higher-priority queue
   - This form of aging prevents starvation.

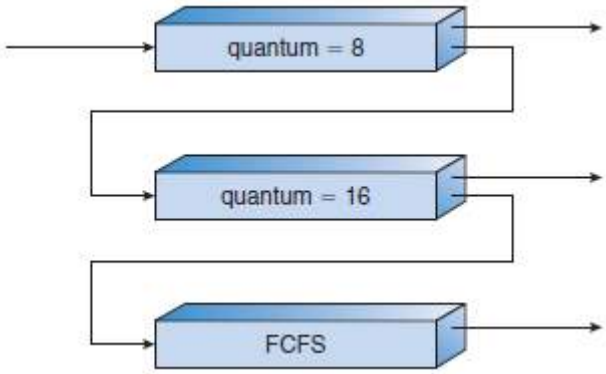


Figure 2.20 Multilevel feedback queues.

- In general, a multilevel feedback queue scheduler is defined by the following parameters:
  1. The number of queues.
  2. The scheduling algorithm for each queue.
  3. The method used to determine when to upgrade a process to a higher priority queue.
  4. The method used to determine when to demote a process to a lower priority queue.
  5. The method used to determine which queue a process will enter when that process needs service.

Multiple Processor Scheduling

- If multiple CPUs are available, the scheduling problem becomes more complex.
- Two approaches:

1. Asymmetric Multiprocessing

- The basic idea is:
  1. A master server is a single processor responsible for all scheduling decisions, I/O processing and other system activities.
  2. The other processors execute only user code.
- Advantage:
  1. This is simple because only one processor accesses the system data structures, reducing the need for data sharing.

2. Symmetric Multiprocessing

- The basic idea is:
  1. Each processor is self-scheduling.
  2. To do scheduling, the scheduler for each processor
     i. Examines the ready-queue and
     ii. Selects a process to execute.
- Restriction: We must ensure that two processors do not choose the same process and that processes are not lost from the queue.

Processor Affinity

- In SMP systems,
  1. Migration of processes from one processor to another are avoided and
  2. Instead processes are kept running on same processor. This is known as processor affinity.
- Two forms:

1. Soft Affinity

- When an OS try to keep a process on one processor because of policy, but cannot guarantee it will happen.
- It is possible for a process to migrate between processors.

2. Hard Affinity

- When an OS have the ability to allow a process to specify that it is not to migrate to other processors. Eg: Solaris OS

Load Balancing

- This attempts to keep the workload evenly distributed across all processors in an SMP system.
- Two approaches:

1. Push Migration

- A specific task periodically checks the load on each processor and if it finds an imbalance, it evenly distributes the load to idle processors.

2. Pull Migration

- An idle processor pulls a waiting task from a busy processor.

Symmetric Multithreading

- The basic idea:
  1. Create multiple logical processors on the same physical processor.
  2. Present a view of several logical processors to the OS.
- Each logical processor has its own architecture state, which includes general-purpose and machine-state registers.
- Each logical processor is responsible for its own interrupt handling.
- SMT is a feature provided in hardware, not software.

Thread Scheduling

- On OSs, it is kernel-level threads but not processes that are being scheduled by the OS.
- 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 be mapped to an associated kernel-level thread.

Contention Scope

- Two approaches:

1. Process-Contention scope

- 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.
- Competition for the CPU takes place among threads belonging to the same process.

2. System-Contention scope

- The process of deciding which kernel thread to schedule on the CPU.
- Competition for the CPU takes place among all threads in the system.
- Systems using the one-to-one model schedule threads using only SCS.

Pthread Scheduling

- Pthread API that allows specifying either PCS or SCS during thread creation.
- Pthreads identifies the following contention scope values:
  1. PTHREAD_SCOPEJPROCESS schedules threads using PCS scheduling.
  2. PTHREAD-SCOPE_SYSTEM schedules threads using SCS scheduling.
- Pthread IPC provides following two functions for getting and setting the contention scope policy:
  1. pthread_attr_setscope(pthread_attr_t *attr, int scope)
  2. pthread_attr_getscope(pthread_attr_t *attr, int *scope)

UNIT 3: PROCESS SYNCHRONIZATION

Synchronization

- Co-operating process is one that can affect or be affected by other processes.
- Co-operating processes may either
  - → share a logical address-space (i.e. code & data) or
  - → share data through files or
  - → messages through threads.
- Concurrent-access to shared-data may result in data-inconsistency.
- To maintain data-consistency:
  The orderly execution of co-operating processes is necessary.
- Suppose that we wanted to provide a solution to **producer-consumer problem** that fills all buffers.
  We can do so by having an variable *counter* that keeps track of the no. of full buffers.
  Initially, *counter=0*.
  - *counter* is incremented by the producer after it produces a new buffer.
  - *counter* is decremented by the consumer after it consumes a buffer.
- **Shared-data:**

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer Process:

```
while (true) {
    /* produce an item in next_produced */

    while (counter == BUFFER_SIZE)
       ; /* do nothing */

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer Process:

```
while (true) {
    while (counter == 0)
       ; /* do nothing */

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;

    /* consume the item in next_consumed */
}
```

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

counter++ could be implemented as:

$register_1 = counter$
$register_1 = register_1 + 1$
$counter = register_1$

counter— may be implemented as:

$register_2 = counter$
$register_2 = register_2 - 1$
$counter = register_2$

- Consider this execution interleaving with *counter* = *5* initially:

T_0:	*producer*	execute	$register_1 = counter$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = counter$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$counter = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$counter = register_2$	$\{counter = 4\}$

- The value of *counter* may be either 4 or 6, where the correct result should be 5. This is an example for race condition.
- To prevent race conditions, concurrent-processes must be synchronized.

Critical-section Problem

- **Critical-section** is a segment-of-code in which a process may be
  → changing common variables
  → updating a table or
  → writing a file.
- Each process has a critical-section in which the shared-data is accessed.
- General structure of a typical process has following (Figure 3.1):

1. Entry-section

➢ Requests permission to enter the critical-section.

2. Critical-section

➢ Mutually exclusive in time i.e. no other process can execute in its critical-section.

3. Exit-section

➢ Follows the critical-section.

4. Remainder-section

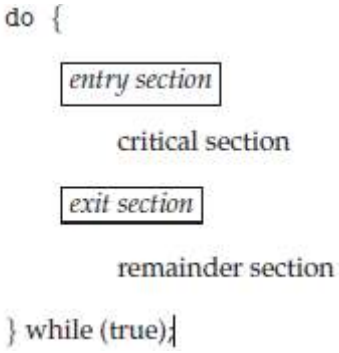


Figure 3.1 General structure of a typical process

- Problem statement:
  "Ensure that when one process is executing in its critical-section, no other process is to be allowed to execute in its critical-section".
- A solution to the problem must satisfy the following 3 requirements:

1. Mutual Exclusion

➢ Only one process can be in its critical-section.

2. Progress

➢ Only processes that are not in their remainder-section can enter their critical-section, and the selection of a process cannot be postponed indefinitely.

3. Bounded Waiting

➢ There must be a bound 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 the request is granted.

- Two approaches used to handle critical-sections:

1. Preemptive Kernels

➢ Allows a process to be preempted while it is running in kernel-mode.

2. Non-preemptive Kernels

➢ Does not allow a process running in kernel-mode to be preempted.

Peterson's Solution

• This is a classic **software-based solution** to the critical-section problem.
• This is limited to 2 processes.
• The 2 processes alternate execution between
→ critical-sections and
→ remainder-sections.
• The 2 processes share 2 variables (Figure 3.2):

```
int turn;
boolean flag[2];
```

where *turn* = indicates whose turn it is to enter its critical-section.
(i.e., if turn==i, then process Pi is allowed to execute in its critical-section).
flag = used to indicate if a process is ready to enter its critical-section.
(i.e. if flag[i]=true, then Pi is ready to enter its critical-section).

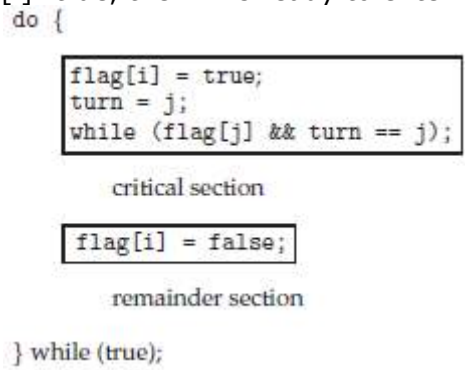


Figure 3.2 The structure of process Pi in Peterson's solution

• To enter the critical-section,
→ firstly process *Pi* sets *flag[i]* to be true and
→ then sets *turn* to the value j.
• If both processes try to enter at the same time, *turn* will be set to both i and j at roughly the same time.
• The final value of *turn* determines which of the 2 processes is allowed to enter its critical-section first.
• To prove that this solution is correct, we show that:
1. Mutual-exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

Synchronization Hardware

Hardware based Solution for Critical-section Problem

• A *lock is* a simple tool used to solve the critical-section problem.

• Race conditions are prevented by following restriction (Figure 3.3).

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

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

Figure 3.3: Solution to the critical-section problem using locks

Hardware instructions for solving critical-section problem

• Modern systems provide special hardware instructions
→ to test & modify the content of a word *atomically* or
→ to swap the contents of 2 words *atomically*.

• Atomic-operation means an operation that completes in its entirety without interruption.

TestAndSet()

• This instruction is executed atomically (Figure 3.4).

• If two TestAndSet() are executed simultaneously (on different CPU), they will be executed sequentially in some arbitrary order.

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

   return rv;
}
```

Figure 3.4 The definition of the test and set() instruction

TestAndSet with Mutual Exclusion

• If the machine supports the TestAndSet(), we can implement mutual-exclusion by declaring a boolean variable *lock*, initialized to false (Figure 3.5).

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

      /* critical section */

   lock = false;

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

Figure 3.5 Mutual-exclusion implementation with test and set()

Swap()

- This instruction is executed atomically (Figure 3.6).
- If the machine supports the Swap(), then mutual-exclusion can be provided as follows:
    1. A global boolean variable *lock* is declared and is initialized to false.
    2. In addition, each process has a local Boolean variable *key* (Figure 3.7).

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

Figure 3.6 The definition of swap() instruction

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

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 3.7 Mutual-exclusion implementation with the swap() instruction

Bounded waiting Mutual Exclusion with TestAndSet()

- Common data structures are

```
boolean waiting[n];
boolean lock;
```

- These data structures are initialized to false (Figure 3.8).

```
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);
```

Figure 3.8 Bounded-waiting mutual-exclusion with TestandSet()

Semaphores

• A *semaphore* is a synchronization-tool.

• It used to control access to shared-variables so that only one process may at any point in time change the value of the shared-variable.

• A semaphore(S) is an integer-variable that is accessed only through 2 atomic-operations:

1. wait() and
2. signal().

• wait() is termed P ("to test").

signal() is termed V ("to increment").

• **Definition of wait():**

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

Definition of signal():

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

• When one process modifies the semaphore-value, no other process can simultaneously modify that same semaphore-value.

• Also, in the case of wait(S), following 2 operations must be executed without interruption:

1. Testing of S(S<=0) and
2. Modification of S (S--)

Semaphore Usage

Counting Semaphore

• The value of a semaphore can range over an unrestricted domain

Binary Semaphore

• The value of a semaphore can range only between 0 and 1.

• On some systems, binary semaphores are known as **mutex locks**, as they are locks that provide mutual-exclusion.

1) Solution for Critical-section Problem using Binary Semaphores

• Binary semaphores can be used to solve the critical-section problem for multiple processes.

• The 'n' processes share a semaphore *mutex* initialized to 1 (Figure 3.9).

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

      // remainder section
} while (TRUE);
```

Figure 3.9 Mutual-exclusion implementation with semaphores

2) Use of counting semaphores

• Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.

• The semaphore is initialized to the number of resources available.

• Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).

• When a process releases a resource, it performs a signal() operation (incrementing the count).

• When the count for the semaphore goes to 0, all resources are being used.

• After that, processes that wish to use a resource will block until the count becomes greater than 0.

3) Solving synchronization problems

• Semaphores can also be used to solve synchronization problems.

• For example, consider 2 concurrently running-processes:

P1 with a statement S1 and
P2 with a statement S2.

• Suppose we require that S2 be executed only after S1 has completed.

• We can implement this scheme readily

→ by letting P1 and P2 share a common semaphore *synch* initialized to 0, and

→ by inserting the following statements in process P1

```
S1;
signal(synch);
```

and the following statements in process P2

```
wait(synch);
S2;
```

• Because *synch* is initialized to 0, P2 will execute S2 only after P1 has invoked signal (*synch*), which is after statement S1 has been executed.

Semaphore Implementation

• Main disadvantage of semaphore:
1. Busy waiting.

• **Busy waiting**: While a process is in its critical-section, any other process that tries to enter its critical-section must loop continuously in the 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 **spinlock** (because the process "spins" while waiting for the lock).

• To overcome busy waiting, we can modify the definition of the wait() and signal() as follows:
1. When a process executes the wait() and finds that the semaphore-value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *block* itself.
2. A process that is blocked (waiting on a semaphore S) should be restarted when some other process executes a signal(). The process is restarted by a wakeup().

• We assume 2 simple operations:
1. **block()** suspends the process that invokes it.
2. **wakeup(P)** resumes the execution of a blocked process P.

• We define a semaphore as follows:

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

• **Definition of wait():**

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

Definition of signal():

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

• This (critical-section) problem can be solved in two ways:
1. In a **uni-processor** environment
   - ➢ Inhibit interrupts when the wait and signal operations execute.
   - ➢ Only current process executes, until interrupts are re-enabled & the scheduler regains control.
2. In a **multiprocessor** environment
   - ➢ Inhibiting interrupts doesn't work.
   - ➢ Use the hardware / software solutions described above.

Deadlocks & Starvation

• Deadlock occurs when 2 or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

• The event in question is the execution of a signal() operation.

• To illustrate this, consider 2 processes, *Po* and *P1*, each accessing 2 semaphores, S and Q. Let S and Q be initialized to 1.

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

• Suppose that *Po* executes wait(S) and then P1 executes wait(Q).

When *Po* executes wait(Q), it must wait until P1 executes signal(Q).

Similarly, when P1 executes wait(S), it must wait until *Po* executes signal(S).

Since these signal() operations cannot be executed, *Po* & P1 are deadlocked.

• Starvation (indefinite blocking) is another problem related to deadlocks.

• ***Starvation*** is a situation in which processes wait indefinitely within the semaphore.

• Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classic Problems of Synchronization

1. Bounded-Buffer Problem
2. Readers and Writers Problem
3. Dining-Philosophers Problem

The Bounded-Buffer Problem

- The bounded-buffer problem is related to the producer consumer problem.
- There is a pool of n buffers, each capable of holding one item.
- **Shared-data**

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

where,

- *mutex* provides mutual-exclusion for accesses to the buffer-pool.
- *empty* counts the number of empty buffers.
- *full* counts the number of full buffers.

- The symmetry between the producer and the consumer.
  - The producer produces full buffers for the consumer.
  - The consumer produces empty buffers for the producer.

- **Producer Process:**

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

Consumer Process:

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

The Readers-Writers Problem

- A data set is shared among a number of concurrent processes.
- **Readers** are processes which want to only read the database (DB).
  Writers are processes which want to update (i.e. to read & write) the DB.
- Problem:
  - Obviously, if 2 readers can access the shared-DB simultaneously without any problems.
  - However, if a writer & other process (either a reader or a writer) access the shared-DB simultaneously, problems may arise.

Solution:

- The writers must have exclusive access to the shared-DB while writing to the DB.

- **Shared-data**

```
semaphore mutex, wrt;
int readcount;
```

where,

- *mutex* is used to ensure mutual-exclusion when the variable *readcount* is updated.
- *wrt* is common to both reader and writer processes.
  wrt is used as a mutual-exclusion semaphore for the writers.
  wrt is also used by the first/last reader that enters/exits the critical-section.
- *readcount* counts no. of processes currently reading the object.

Initialization

mutex = 1, wrt = 1, readcount = 0

Writer Process:

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

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);
```

- The readers-writers problem and its solutions are used to provide **reader-writer locks** on some systems.
- The mode of lock needs to be specified:
  1. **read mode**
     - When a process wishes to read shared-data, it requests the lock in *read mode*.
  2. **write mode**
     - When a process wishes to modify shared-data, it requests the lock in *write mode*.
- Multiple processes are permitted to concurrently acquire a lock in read mode,
  but only one process may acquire the lock for writing.
- These locks are most useful in the following situations:
  1. In applications where it is easy to identify
     → which processes only read shared-data and
     → which threads only write shared-data.
  2. In applications that have more readers than writers.

The Dining-Philosophers Problem

- Problem statement:
  - There are 5 philosophers with 5 chopsticks (semaphores).
  - A philosopher is either eating (with two chopsticks) or thinking.
  - The philosophers share a circular table (Figure 3.10).
  - The table has
    - → a bowl of rice in the center and
    - → 5 single chopsticks.
  - From time to time, a philosopher gets hungry and tries to pick up the 2 chopsticks that are closest to her.
  - A philosopher may pick up only one chopstick at a time.
  - Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
  - When hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
  - When she is finished eating, she puts down both of her chopsticks and starts thinking again.
- Problem objective:
  To allocate several resources among several processes in a deadlock-free & starvation-free manner.
- Solution:
  - Represent each chopstick with a semaphore (Figure 3.11).
  - A philosopher tries to grab a chopstick by executing a wait() on the semaphore.
  - The philosopher releases her chopsticks by executing the signal() on the semaphores.
  - This solution guarantees that no two neighbors are eating simultaneously.
  - **Shared-data**
    semaphore chopstick[5];
    Initialization
    chopstick[5]={1,1,1,1,1}.

Figure 3.10 Situation of dining philosophers

```
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);
```

Figure 3.11 The structure of philosopher

- Disadvantage:
  1. Deadlock may occur if all 5 philosophers become hungry simultaneously and grab their left chopstick. When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Three possible remedies to the deadlock problem:
  1. Allow **at most 4** philosophers to be sitting simultaneously at the table.
  2. Allow a philosopher to pick up her chopsticks **only if both chopsticks are available**.
  3. Use an **asymmetric solution**; i.e. an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Monitors

• **Monitor** is a high-level synchronization construct.

• It provides a convenient and effective mechanism for process synchronization.

Need for Monitors

• When programmers use semaphores incorrectly, following types of errors may occur:

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

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

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

2. 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.

3. 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.

Monitors Usage

• A ***monitor type*** presents a set of programmer-defined operations that are provided to ensure mutual-exclusion within the monitor.

• It also contains (Figure 3.12):

→ declaration of variables

→ bodies of procedures(or functions).

• A procedure defined within a monitor can access only those variables declared locally within the monitor and its formal-parameters.

Similarly, the local-variables of a monitor can be accessed by only the local-procedures.

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

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

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

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

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

Figure 3.12 Syntax of a monitor

• Only one process at a time is active within the monitor (Figure 3.13).

• To allow a process to wait within the monitor, a condition variable must be declared, as

```
condition x, y;
```

• Condition variable can only be used with the following 2 operations (Figure 3.14):

1. **x.signal()**

➢ This operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

2. **x.wait()**

➢ The process invoking this operation is suspended until another process invokes x.signal().

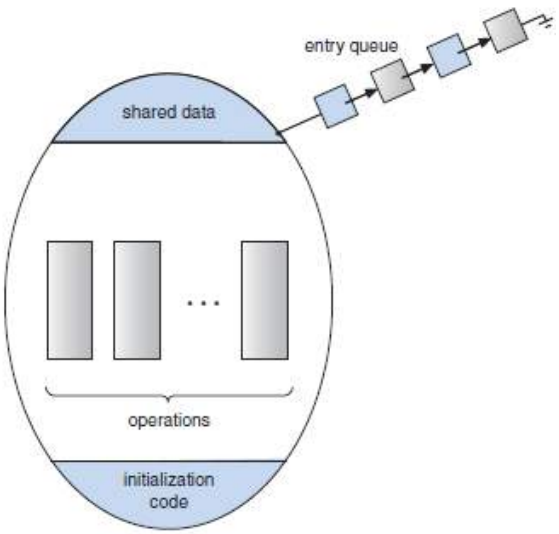


Figure 3.13 Schematic view of a monitor

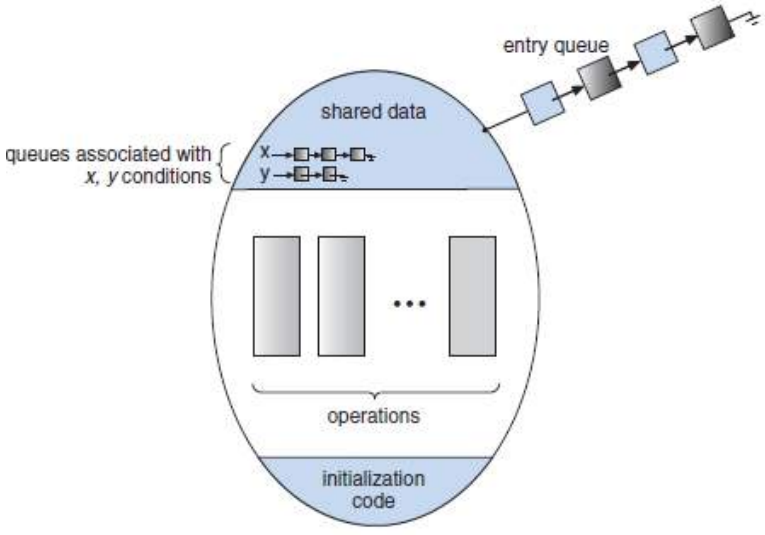


Figure 3.14 Monitor with condition variables

• Suppose when the x.signal() operation is invoked by a process P, there exists a suspended process Q associated with condition x.
• Both processes can conceptually continue with their execution. Two possibilities exist:

1. Signal and wait

➢ P either waits until *Q* leaves the monitor or waits for another condition.

2. Signal and continue

➢ Q either waits until P leaves the monitor or waits for another condition.

Dining-Philosophers Solution Using Monitors

• The restriction is

➢ A philosopher may pick up her chopsticks only if both of them are available.

• Description of the solution:

1. The distribution of the chopsticks is controlled by the monitor *dp* (Figure 3.15).
2. Each philosopher, before starting to eat, must invoke the operation *pickup()*. This act may result in the suspension of the philosopher process.
3. After the successful completion of the operation, the philosopher may eat.
4. Following this, the philosopher invokes the *putdown()* operation.
5. Thus, philosopher *i* must invoke the operations *pickup()* and *putdown()* in the following sequence:

```
dp.pickup(i);
   ...
   eat
   ...
dp.putdown(i);
```

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

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

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

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

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

Figure 3.15 A monitor solution to the dining-philosopher problem

Implementing a Monitor using Semaphores

- A process
  → must execute wait(mutex) before entering the monitor and
  → must execute signal(mutex) after leaving the monitor.
- Variables used:

```
semaphore mutex;       // (initially = 1)
semaphore next;        // (initially = 0)
int next-count = 0;
```

  where
  - mutex is provided for each monitor.
  - next is used a signaling process to wait until the resumed process either leaves or waits
  - next-count is used to count the number of processes suspended
- Each external procedure 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.
- How condition variables are implemented ?
  For each condition variable x, we have:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

- **Definition of x.wait()**

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

Definition of x.signal()

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

Resuming Processes within a Monitor

• Problem:

If several processes are suspended, then how to determine which of the suspended processes should be resumed next?

Solution-1: Use an FCFS ordering i.e. the process that has been waiting the longest is resumed first.

Solution-2: Use *conditional–wait* construct i.e. x.wait(c)

- c is a integer expression evaluated when the wait operation is executed (Figure 3.16).
- Value of c (a priority number) is then stored with the name of the process that is suspended.
- When x.signal is executed, process with smallest associated 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;
   }
}
```

Figure 3.16 A monitor to allocate a single resource

• *ResourceAllocator monitor* controls the allocation of a single resource among competing processes.
• Each process, when requesting an allocation of the resource, specifies the maximum time it plans to use the resource.
• The monitor allocates the resource to the process that has the shortest time-allocation request.
• 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.

• 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.

UNIT 5: MEMORY MANAGEMENT

Basic Hardware

- Program must be
  - → brought (from disk) into memory and
  - → placed within a process for it to be run.
- Main-memory and registers are only storage CPU can access directly.
- Register access in one CPU clock.
- Main-memory can take many cycles.
- Cache sits between main-memory and CPU registers.
- Protection of memory required to ensure correct operation.
- A pair of base- and limit-registers define the logical (virtual) address space (Figure 5.1 & 5.2).

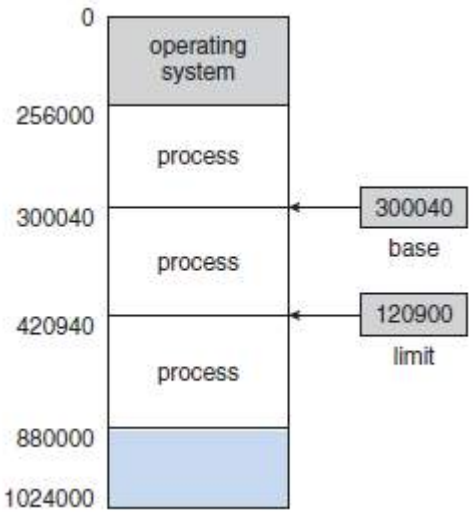


Figure 5.1 A base and a limit-register define a logical-address space

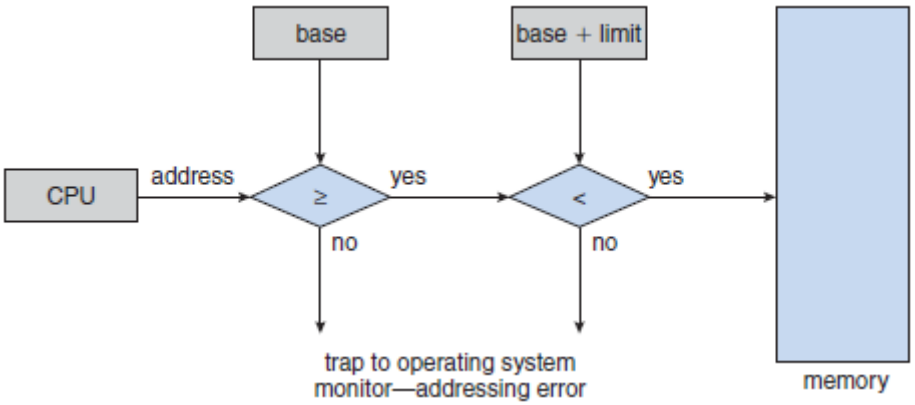


Figure 5.2 Hardware address protection with base and limit-registers

Address Binding

• Address binding of instructions to memory-addresses can happen at 3 different stages (Figure 5.3):

1. Compile Time

➢ If memory-location known a priori, absolute code can be generated.

➢ Must recompile code if starting location changes.

2. Load Time

➢ Must generate relocatable code if memory-location is not known at compile time.

3. Execution Time

➢ Binding delayed until run-time if the process can be moved during its execution from one memory-segment to another.

➢ Need hardware support for address maps (e.g. base and limit-registers).

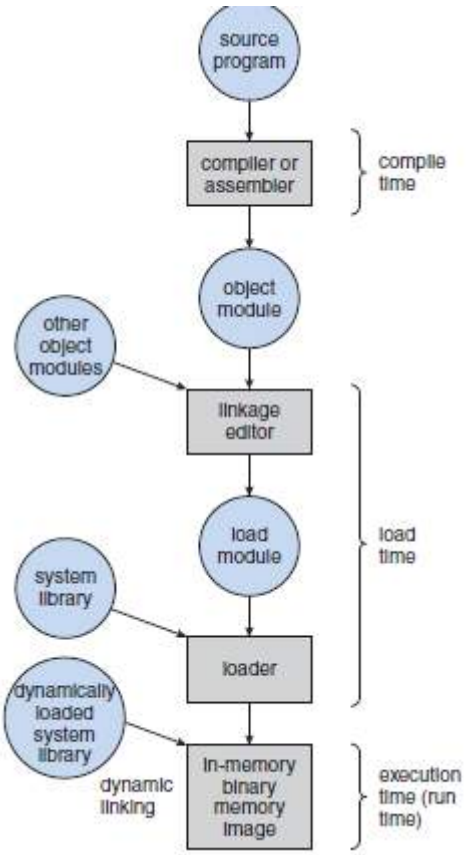


Figure 5.3 Multistep processing of a user-program

Logical versus Physical Address Space

• ***Logical-address*** is generated by the CPU (also referred to as virtual-address).
Physical-address is the address seen by the memory-unit.

• Logical & physical-addresses are the same in compile-time & load-time address-binding methods.
Logical and physical-addresses differ in execution-time address-binding method.

• MMU (Memory-Management Unit)
- Hardware device that maps virtual-address to physical-address (Figure 5.4).
- The value in the relocation-register is added to every address generated by a user-process at the time it is sent to memory.
- The user-program deals with logical-addresses; it never sees the real physical-addresses.

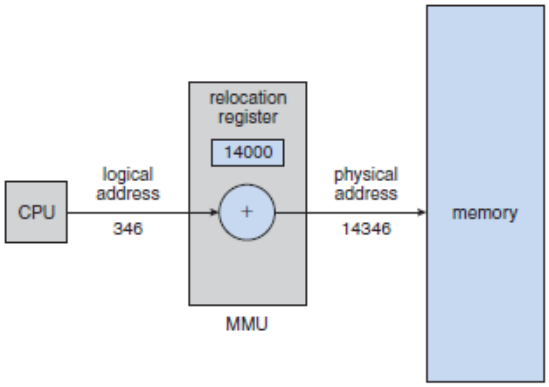


Figure 5.4 Dynamic relocation using a relocation-register

Dynamic Loading

• This can be used to obtain better memory-space utilization.
• A routine is not loaded until it is called.
• This works as follows:
1. Initially, all routines are kept on disk in a relocatable-load format.
2. Firstly, the main-program is loaded into memory and is executed.
3. When a main-program calls the routine, the main-program first checks to see whether the routine has been loaded.
4. If routine has been not yet loaded, the loader is called to load desired routine into memory.
5. Finally, control is passed to the newly loaded-routine.

• Advantages:
1. An unused routine is never loaded.
2. Useful when large amounts of code are needed to handle infrequently occurring cases.
3. Although the total program-size may be large, the portion that is used (and hence loaded) may be much smaller.
4. Does not require special support from the OS.

Dynamic Linking and Shared Libraries

• Linking postponed until execution-time.
• This feature is usually used with system libraries, such as language subroutine libraries.
• A stub is included in the image for each library-routine reference.
• The **stub** is a small piece of code used to locate the appropriate memory-resident library-routine.
• When the stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.
• Stub replaces itself with the address of the routine, and executes the routine.
• Thus, the next time that particular code-segment is reached, the library-routine is executed directly, incurring no cost for dynamic-linking.
• All processes that use a language library execute only one copy of the library code.

Shared libraries

• A library may be replaced by a new version, and all programs that reference the library will automatically use the new one.
• Version info. is included in both program & library so that programs won't accidentally execute incompatible versions.

Swapping

• A process must be in memory to be executed.
• A process can be
→ swapped temporarily out-of-memory to a backing-store and
→ then brought into memory for continued execution.
• ***Backing-store*** is a fast disk which is large enough to accommodate copies of all memory-images for all users.
• ***Roll out/Roll in*** is a swapping variant used for priority-based scheduling algorithms.
- Lower-priority process is swapped out so that higher-priority process can be loaded and executed.
- Once the higher-priority process finishes, the lower-priority process can be swapped back in and continued (Figure 5.5).

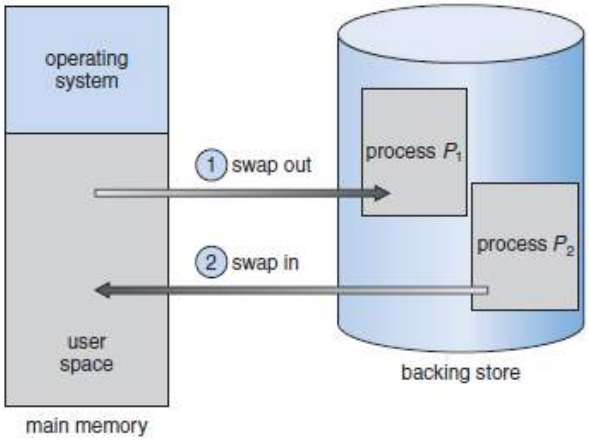


Figure 5.5 Swapping of two processes using a disk as a backing-store

• Swapping depends upon address-binding:
1. If binding is done at load-time, then process cannot be easily moved to a different location.
2. If binding is done at execution-time, then a process can be swapped into a different memory-space, because the physical-addresses are computed during execution-time.
• Major part of swap-time is transfer-time; i.e. total transfer-time is directly proportional to the amount of memory swapped.
• Disadvantages:
1. Context-switch time is fairly high.
2. If we want to swap a process, we must be sure that it is completely idle.
Two solutions:
i) Never swap a process with pending I/O.
ii) Execute I/O operations only into OS buffers.

Contiguous Memory Allocation

• Memory is usually divided into 2 partitions:
→ One for the resident OS.
→ One for the user-processes.
• Each process is contained in a single contiguous section of memory.

Memory Mapping & Protection

• Memory-protection means
→ protecting OS from user-process and
→ protecting user-processes from one another.
• Memory-protection is done using
→ ***Relocation-register***: contains the value of the smallest *physical-address.*
→ ***Limit-register***: contains the range of *logical-addresses*.
• Each logical-address must be less than the limit-register.
• The MMU maps the logical-address *dynamically* by adding the value in the relocation-register. This mapped-address is sent to memory (Figure 5.6).
• When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit-registers with the correct values.
• Because every address generated by the CPU is checked against these registers, we can protect the OS from the running-process.
• The relocation-register scheme provides an effective way to allow the OS size to change dynamically.
• ***Transient OS code***: Code that comes & goes as needed to save memory-space and overhead for unnecessary swapping.

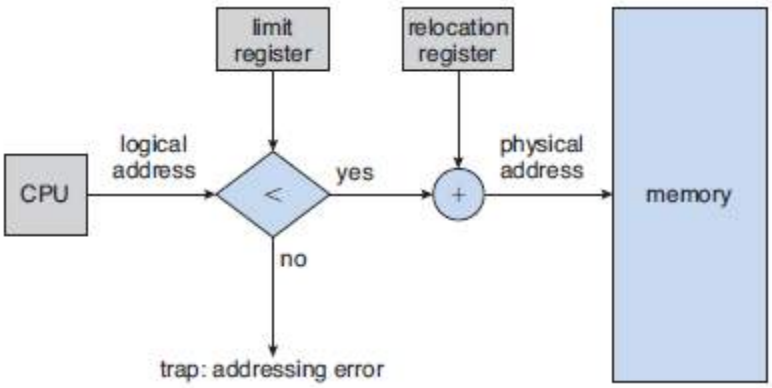


Figure 5.6 Hardware support for relocation and limit-registers

Memory Allocation

- Two types of memory partitioning are:
  1) Fixed-sized partitioning and 2) Variable-sized partitioning

Fixed-sized Partitioning

- The memory is divided into fixed-sized partitions.
- Each partition may contain exactly one process.
- The degree of multiprogramming is bound by the number of partitions.
- When a partition is free, a process is
  → selected from the input queue and
  → loaded into the free partition.
- When the process terminates, the partition becomes available for another process.

Variable-sized Partitioning

- The OS keeps a table indicating
  → which parts of memory are available and
  → which parts are occupied.
- A **hole** is a block of available memory.
- Normally, memory contains a set of holes of various sizes.
- Initially, all memory is
  → available for user-processes and
  → considered one large hole.
- When a process arrives, the process is allocated memory from a large hole.
- If we find the hole, we
  → allocate only as much memory as is needed and
  → keep the remaining memory available to satisfy future requests.

- Three strategies used to select a free hole from the set of available holes.

1. First Fit

- Allocate the first hole that is big enough.
- Searching can start either
  → at the beginning of the set of holes or
  → at the location where the previous first-fit search ended.

2. Best Fit

- Allocate the *smallest* hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.

3. Worst Fit

- Allocate the *largest* hole.
- Again, we must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole.

- First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

Fragmentation

• Two types of memory fragmentation:
1) Internal fragmentation and 2) External fragmentation

Internal Fragmentation

• The general approach is to
→ break the physical-memory into fixed-sized blocks and
→ allocate memory in units based on block size.
• The allocated-memory to a process may be slightly larger than the requested-memory.
• The difference between requested-memory and allocated-memory is called *internal fragmentation* i.e. Unused memory that is internal to a partition.

External Fragmentation

• *External fragmentation* occurs when there is enough total memory-space to satisfy a request but the available-spaces are not contiguous. (i.e. storage is fragmented into a large number of small holes).
• Both the first-fit and best-fit strategies for memory-allocation suffer from external fragmentation.
• Statistical analysis of first-fit reveals that
→ given *N* allocated blocks, another 0.5 *N* blocks will be lost to fragmentation.
This property is known as the ***50-percent rule***.
• Two solutions to external fragmentation:

1. Compaction

➢ The goal is to shuffle the memory-contents to place all free memory together in one large hole.
➢ Compaction is possible *only* if relocation is
→ dynamic and
→ done at execution-time.

2. Permit the logical-address space of the processes to be non-contiguous.

➢ This allows a process to be allocated physical-memory wherever such memory is available.
➢ Two techniques achieve this solution:
1) Paging and
2) Segmentation.

Paging

• Paging is a memory-management scheme.
• This permits the physical-address space of a process to be non-contiguous.
• This also solves the considerable problem of fitting memory-chunks of varying sizes onto the backing-store.
• Traditionally: Support for paging has been handled by hardware.
Recent designs: The hardware & OS are closely integrated.

Basic Method of Paging

• Physical-memory is broken into fixed-sized blocks called **frames**(Figure 5.7).
Logical-memory is broken into same-sized blocks called **pages.**
• When a process is to be executed, its pages are loaded into any available memory-frames from the backing-store.
• The backing-store is divided into fixed-sized blocks that are of the same size as the memory-frames.

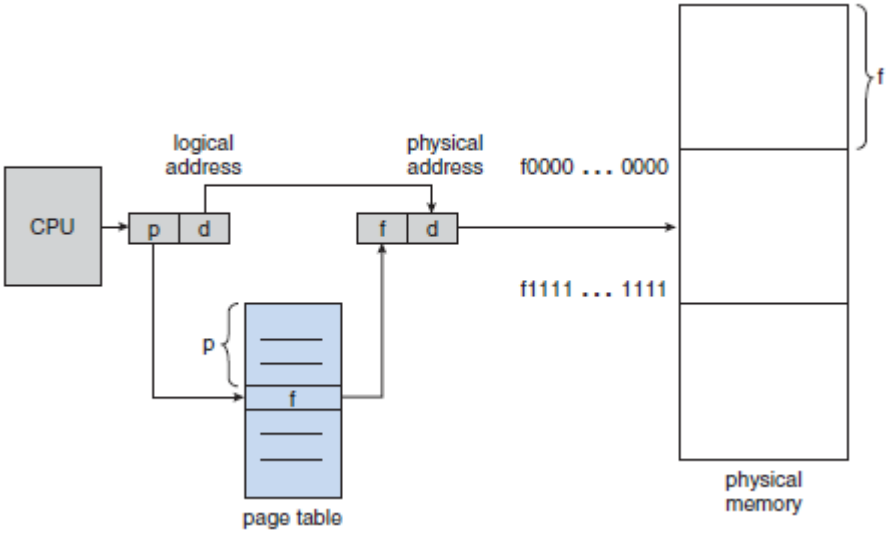


Figure 5.7 Paging hardware

• The page-table contains the base-address of each page in physical-memory.
• Address generated by CPU is divided into 2 parts (Figure 5.8):
1) **Page-number**(p) is used as an index to the page-table and
2) **Offset**(d) is combined with the base-address to define the physical-address.
This physical-address is sent to the memory-unit.

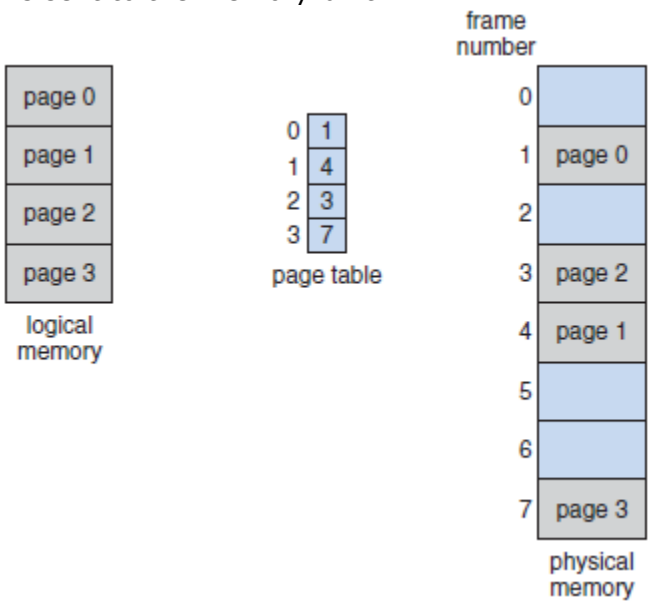


Figure 5.8 Paging model of logical and physical-memory

• The page-size (like the frame size) is defined by the hardware (Figure 5.9).
• If the size of the logical-address space is 2^m, and a page-size is 2^n addressing-units (bytes or words) then the high-order *m-n* bits of a logical-address designate the page-number, and the *n* low-order bits designate the page-offset.

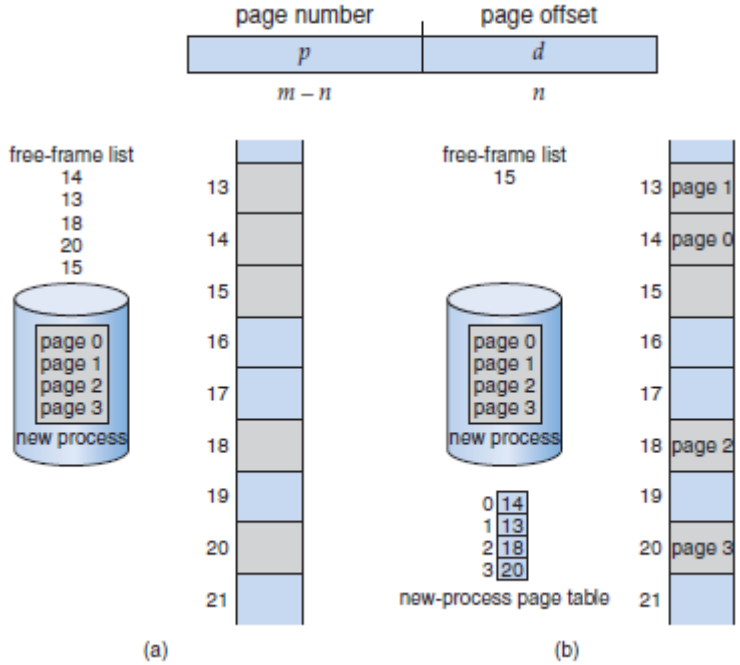


Figure 5.9 Free frames (a) before allocation and (b) after allocation

Hardware Support for Paging

- Most OS's store a page-table for each process.
- A pointer to the page-table is stored in the PCB.

Translation Lookaside Buffer

- The TLB is associative, high-speed memory.
- The TLB contains only a few of the page-table entries.
- Working:
  - When a logical-address is generated by the CPU, its page-number is presented to the TLB.
  - If the page-number is found (**TLB hit**), its frame-number is
    → immediately available and
    → used to access memory.
  - If page-number is not in TLB (**TLB miss**), a memory-reference to page table must be made.
  - The obtained frame-number can be used to access memory (Figure 5.10).
  - In addition, we add the page-number and frame-number to the TLB, so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the OS must select one for replacement.
- Percentage of times that a particular page-number is found in the TLB is called **hit ratio**.
- Advantage: Search operation is fast.
  Disadvantage: Hardware is expensive.
- Some TLBs have wired down entries that can't be removed.
- Some TLBs store ASID (address-space identifier) in each entry of the TLB that uniquely
  → identify each process and
  → provide address space protection for that process.

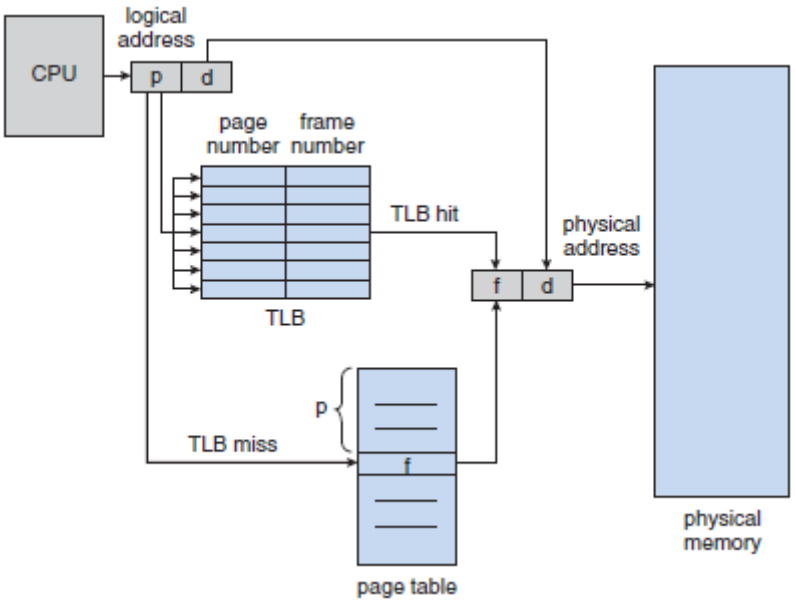


Figure 5.10 Paging hardware with TLB

Protection

• Memory-protection is achieved by **protection-bits** for each frame.
• The protection-bits are kept in the page-table.
• One protection-bit can define a page to be read-write or read-only.
• Every reference to memory goes through the page-table to find the correct frame-number.
• Firstly, the physical-address is computed. At the same time, the protection-bit is checked to verify that no writes are being made to a read-only page.
• An attempt to write to a read-only page causes a hardware-trap to the OS (or memory-protection violation).

Valid Invalid Bit

• This bit is attached to each entry in the page-table (Figure 5.11).
Valid bit: The page is in the process' logical-address space.
Invalid bit: The page is not in the process' logical-address space.
• Illegal addresses are trapped by use of valid-invalid bit.
• The OS sets this bit for each page to allow or disallow access to the page.

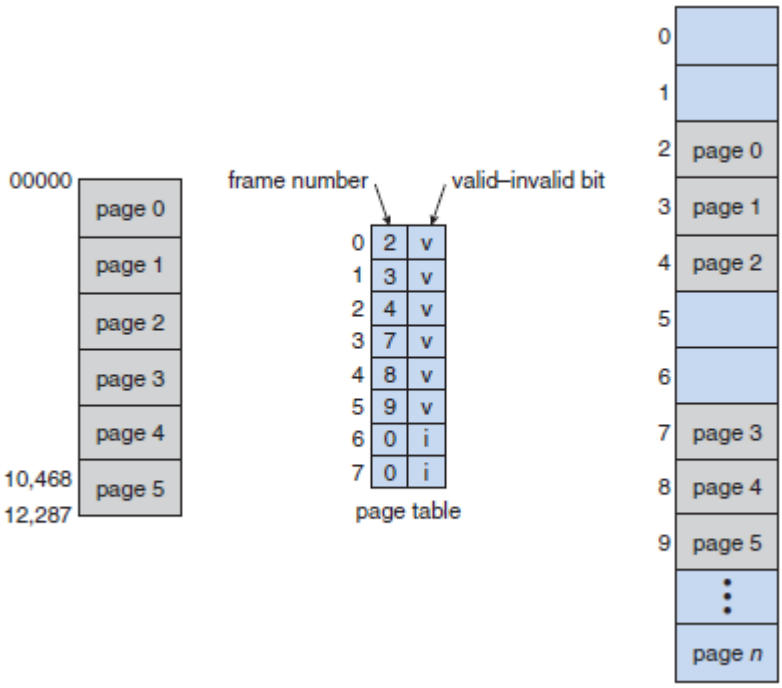


Figure 5.11 Valid (v) or invalid (i) bit in a page-table

Shared Pages

- Advantage of paging:
  1. Possible to share common code.
- Re-entrant code is non-self-modifying code, it never changes during execution.
- Two or more processes can execute the same code at the same time.
- Each process has its own copy of registers and data-storage to hold the data for the process's execution.
- The data for 2 different processes will be different.
- Only one copy of the editor need be kept in physical-memory (Figure 5.12).
- Each user's page-table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
- Disadvantage:
  1. Systems that use inverted page-tables have difficulty implementing shared-memory.

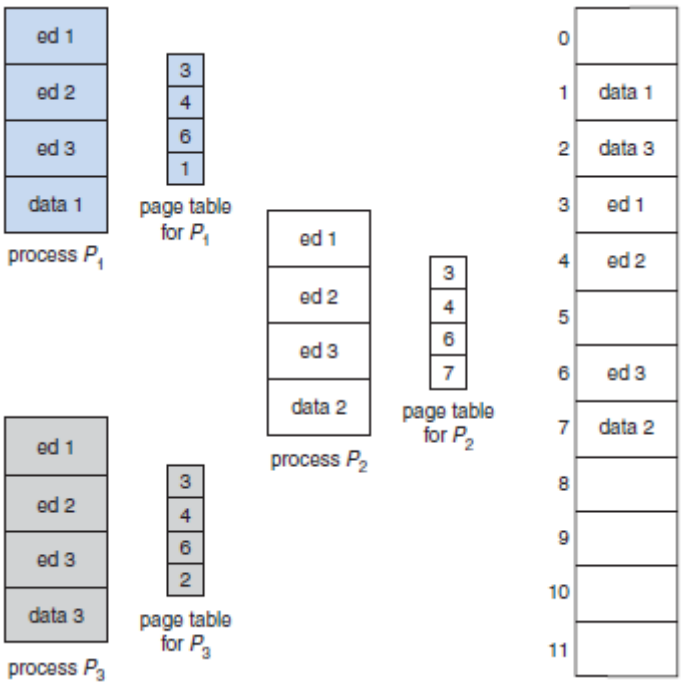


Figure 5.12 Sharing of code in a paging environment

Structure of the Page Table

1. Hierarchical Paging
2. Hashed Page-tables
3. Inverted Page-tables

Hierarchical Paging

• Problem: Most computers support a large logical-address space (2^{32} to 2^{64}). In these systems, the page-table itself becomes excessively large.

Solution: Divide the page-table into smaller pieces.

Two Level Paging Algorithm

• The page-table itself is also paged (Figure 5.13).

• This is also known as a forward-mapped page-table because address translation works from the outer page-table inwards.

• For example (Figure 5.14):

- Consider the system with a 32-bit logical-address space and a page-size of 4 KB.
- A logical-address is divided into
  - → 20-bit page-number and
  - → 12-bit page-offset.
- Since the page-table is paged, the page-number is further divided into
  - → 10-bit page-number and
  - → 10-bit page-offset.
- Thus, a logical-address is as follows:

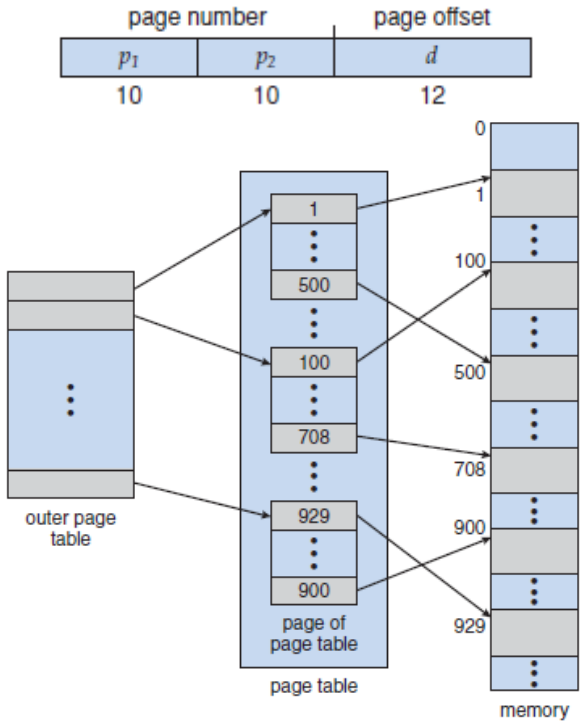


Figure 5.13 A two-level page-table scheme

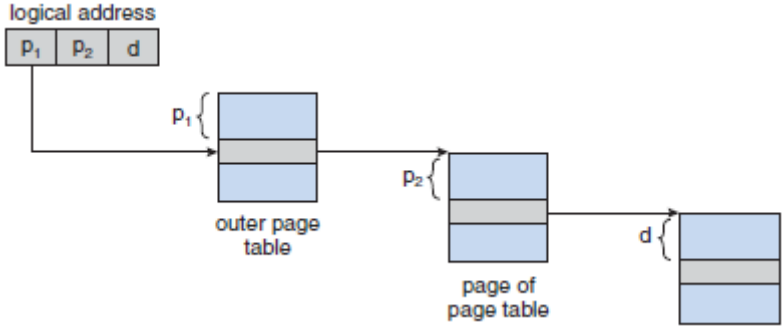


Figure 5.14 Address translation for a two-level 32-bit paging architecture

Hashed Page Tables

• This approach is used for handling address spaces larger than 32 bits.
• The hash-value is the virtual page-number.
• Each entry in the hash-table contains a linked-list of elements that hash to the same location (to handle collisions).
• Each element consists of 3 fields:
    1. Virtual page-number
    2. Value of the mapped page-frame and
    3. Pointer to the next element in the linked-list.
• The algorithm works as follows (Figure 5.15):
    1. The virtual page-number is hashed into the hash-table.
    2. The virtual page-number is compared with the first element in the linked-list.
    3. If there is a match, the corresponding page-frame (field 2) is used to form the desired physical-address.
    4. If there is no match, subsequent entries in the linked-list are searched for a matching virtual page-number.

Clustered Page Tables

• These are similar to hashed page-tables except that each entry in the hash-table refers to several pages rather than a single page.
• Advantages:
    1. Favorable for 64-bit address spaces.
    2. Useful for address spaces, where memory-references are noncontiguous and scattered throughout the address space.

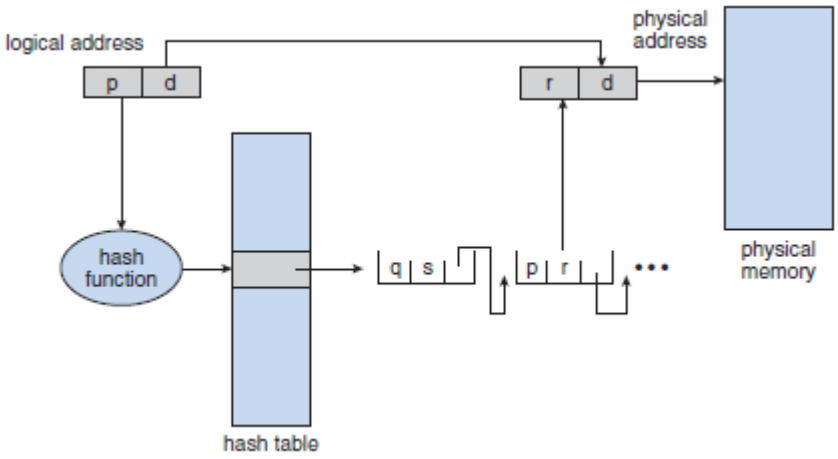


Figure 5.15 Hashed page-table

Inverted Page Tables

- Has one entry for each real page of memory.
- Each entry consists of
  → virtual-address of the page stored in that real memory-location and
  → information about the process that owns the page.

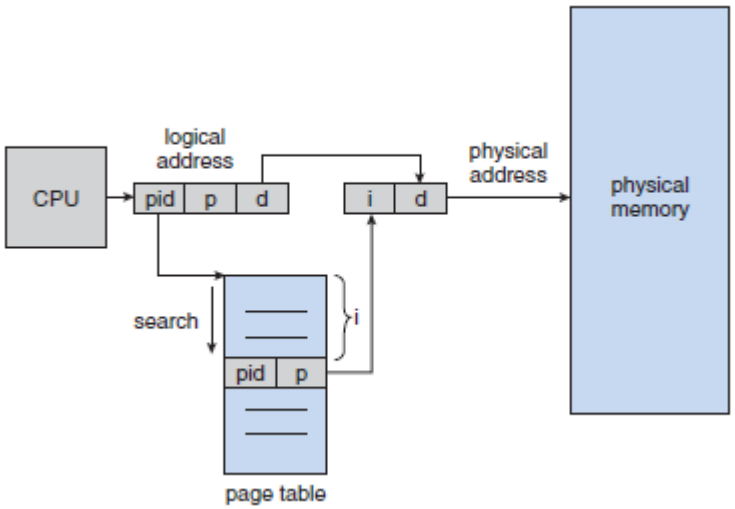


Figure 5.16 Inverted page-table

- Each virtual-address consists of a triplet (Figure 5.16):
  <process-id, page-number, offset>.
- Each inverted page-table entry is a pair <process-id, page-number>
- The algorithm works as follows:
  1. When a memory-reference occurs, part of the virtual-address, consisting of <process-id, page-number>, is presented to the memory subsystem.
  2. The inverted page-table is then searched for a match.
  3. If a match is found, at entry i-then the physical-address <i, offset> is generated.
  4. If no match is found, then an illegal address access has been attempted.
- Advantage:
  1. Decreases memory needed to store each page-table
- Disadvantages:
  1. Increases amount of time needed to search table when a page reference occurs.
  2. Difficulty implementing shared-memory.

Segmentation

Basic Method of Segmentation

- This is a memory-management scheme that supports user-view of memory(Figure 5.17).
- A logical-address space is a collection of segments.
- Each segment has a name and a length.
- The addresses specify both
  - → segment-name and
  - → offset within the segment.
- Normally, the user-program is compiled, and the compiler automatically constructs segments reflecting the input program.

For ex:

→ The code
→ The heap, from which memory is allocated
→ The standard C library
→ Global variables
→ The stacks used by each thread

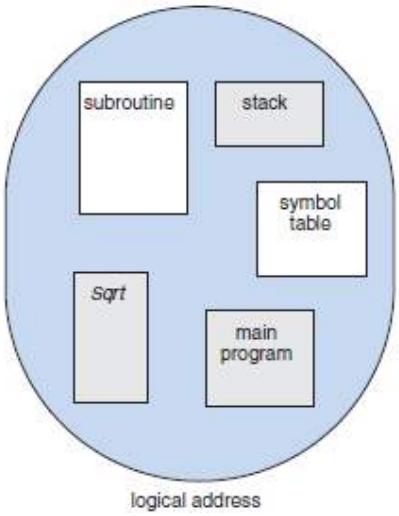


Figure 5.17 Programmer's view of a program

Hardware support for Segmentation

- Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical-addresses.
- In the segment-table, each entry has following 2 fields:
  1. **Segment-base** contains starting physical-address where the segment resides in memory.
  2. **Segment-limit** specifies the length of the segment (Figure 5.18).
- A logical-address consists of 2 parts:
  1. **Segment-number**(s) is used as an index to the segment-table .
  2. **Offset**(d) must be between 0 and the segment-limit.
- If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).
- If offset is legal, then it is added to the segment-base to produce the physical-memory address.

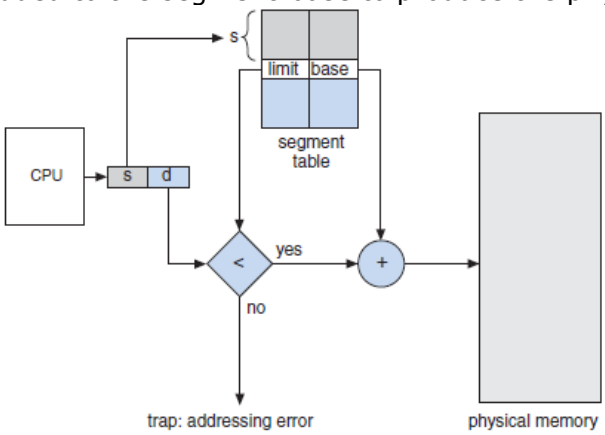


Figure 5.18 Segmentation hardware

UNIT 5(CONT.): VIRTUAL MEMORY

Virtual Memory

• In many cases, the entire program is not needed.
For example:
→ Unusual error-conditions are almost never executed.
→ Arrays & lists are often allocated more memory than needed.
→ Certain options & features of a program may be used rarely.

• Benefits of executing a program that is only partially in memory.
- More programs could be run at the same time.
- Programmers could write for a large virtual-address space and need no longer use overlays.
- Less I/O would be needed to load/swap programs into memory, so each user program would run faster.

• *Virtual Memory* is a technique that allows the execution of processes that are not completely in memory (Figure 5.19).
- VM involves the separation of logical-memory as perceived by users from physical-memory.
- VM allows files and memory to be shared by several processes through page-sharing.
- Logical-address space can .'. be much larger than physical-address space.

• Virtual-memory can be implemented by:
1. Demand paging and
2. Demand segmentation.

• The virtual (or logical) address-space of a process refers to the logical (or virtual) view of how a process is stored in memory.

• Physical-memory may be organized in page-frames and that the physical page-frames assigned to a process may not be contiguous.

• It is up to the MMU to map logical-pages to physical page-frames in memory.

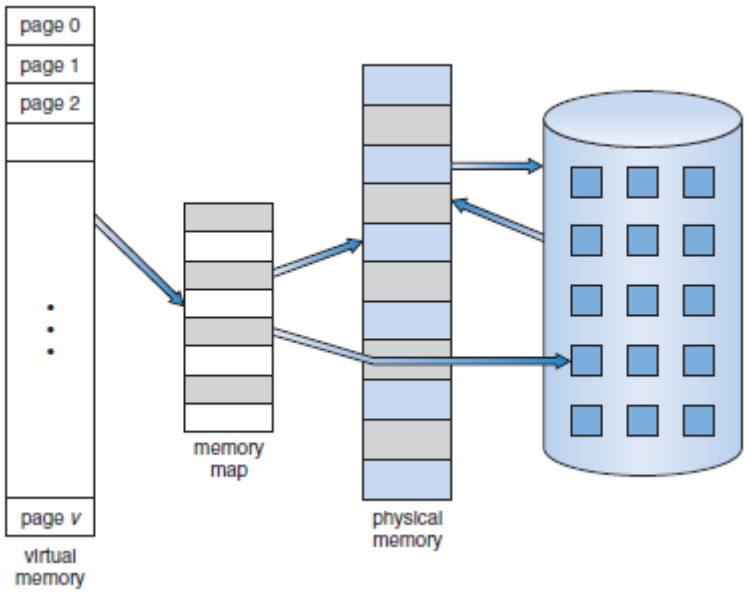


Figure 5.19 Diagram showing virtual memory that is larger than physical-memory

Demand Paging

• A demand-paging system is similar to a paging-system with swapping (Figure 5.20).
• Processes reside in secondary-memory (usually a disk).
• When we want to execute a process, we swap it into memory.
• Instead of swapping in a whole process, **lazy swapper** brings only those necessary pages into memory.

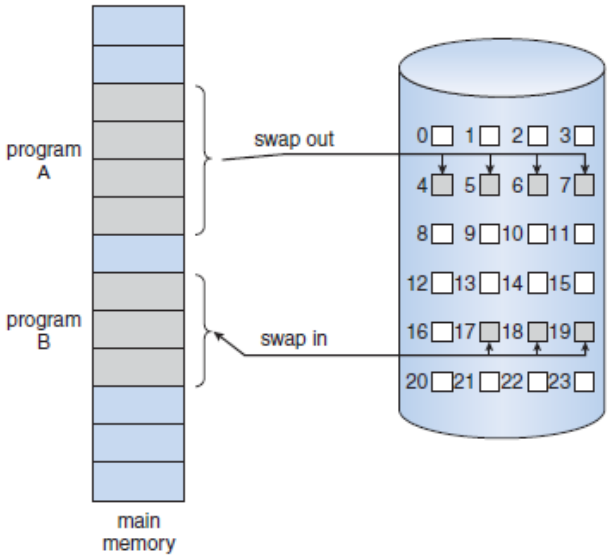


Figure 5.20 Transfer of a paged memory to contiguous disk space

Basic Concepts of Demand Paging

• Instead of swapping in a whole process, the pager brings only those necessary pages into memory.
• Advantages:
    1. Avoids reading into memory-pages that will not be used,
    2. Decreases the swap-time and
    3. Decreases the amount of physical-memory needed.
• The *valid-invalid bit scheme* can be used to distinguish between
    → pages that are in memory and
    → pages that are on the disk.
• If the bit is set to **valid**, the associated page is both legal and in memory.
• If the bit is set to **invalid**, the page either
    → is not valid (i.e. not in the logical-address space of the process) or
    → is valid but is currently on the disk (Figure 5.21).

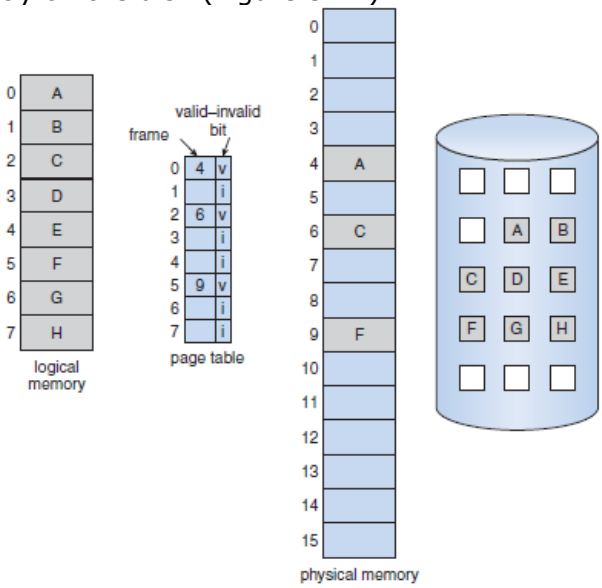


Figure 5.21 Page-table when some pages are not in main-memory

• A **page-fault** occurs when the process tries to access a page that was not brought into memory**.**
• Procedure for handling the page-fault (Figure 5.22):

1. Check an internal-table to determine whether the reference was a valid or an invalid memory access.
2. If the reference is invalid, we terminate the process.
   If reference is valid, but we have not yet brought in that page, we now page it in.
3. Find a free-frame (by taking one from the free-frame list, for example).
4. Read the desired page into the newly allocated frame.
5. Modify the internal-table and the page-table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted by the trap.

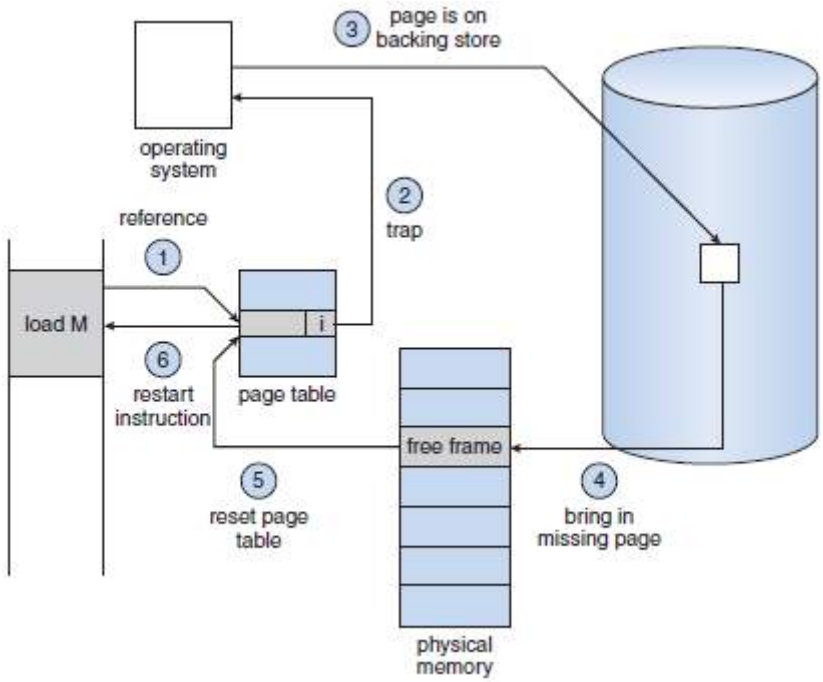


Figure 5.22 Steps in handling a page-fault

• **Pure demand paging:** Never bring pages into memory until required.
• Some programs may access several new pages of memory with each instruction, causing multiple page-faults and poor performance.
• Programs tend to have locality of reference, so this results in reasonable performance from demand paging.
• Hardware support:

1. Page-table

• Mark an entry invalid through a valid-invalid bit.

2. Secondary memory

• It holds pages that are not present in main-memory.
• It is usually a high-speed disk.
• It is known as the **swap device** (and the section of disk used for this purpose is known as swap space).

Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer-system.
- Let p be the probability of a page-fault ($0 \leq p \leq 1$).
  - if p = 0, no page-faults.
  - if p = 1, every reference is a fault.
- *effective access time(EAT)=[(1 - p) *memory access]+ [p *page-fault time]*
- A page-fault causes the following events to occur:
  1. Trap to the OS.
  2. Save the user-registers and process-state.
  3. Determine that the interrupt was a page-fault. '
  4. Check that the page-reference was legal and determine the location of the page on the disk.
  5. Issue a read from the disk to a free frame:
     a. Wait in a queue for this device until the read request is serviced.
     b. Wait for the device seek time.
     c. Begin the transfer of the page to a free frame.
  6. While waiting, allocate the CPU to some other user.
  7. Receive an interrupt from the disk I/O subsystem (I/O completed).
  8. Save the registers and process-state for the other user (if step 6 is executed).
  9. Determine that the interrupt was from the disk.
  10. Correct the page-table and other tables to show that the desired page is now in memory.
  11. Wait for the CPU to be allocated to this process again.
  12. Restore the user-registers, process-state, and new page-table, and then resume the interrupted instruction.

Copy-on-Write

- This technique allows the parent and child processes initially to share the same pages.
- If either process writes to a shared-page, a copy of the shared-page is created.
- For example:
  - Assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
  - OS will then create a copy of this page, mapping it to the address space of the child process.
  - Child process will then modify its copied page & not the page belonging to the parent process.

Page Replacement

1. FIFO page replacement
2. Optimal page replacement
3. LRU page replacement (Least Recently Used)
4. LFU page replacement (Least Frequently Used)

Need for Page Replacement

• If we increase our degree of multiprogramming, we are over-allocating memory.
• While a user-process is executing, a page-fault occurs.
• The OS determines where the desired page is residing on the disk but then finds that there are *no* free frames on the free-frame list (Figure 5.23).
• The OS then could:

→ Terminate the user-process (Not a good idea).
→ Swap out a process, freeing all its frames, and reducing the level of multiprogramming.
→ Perform page replacement.

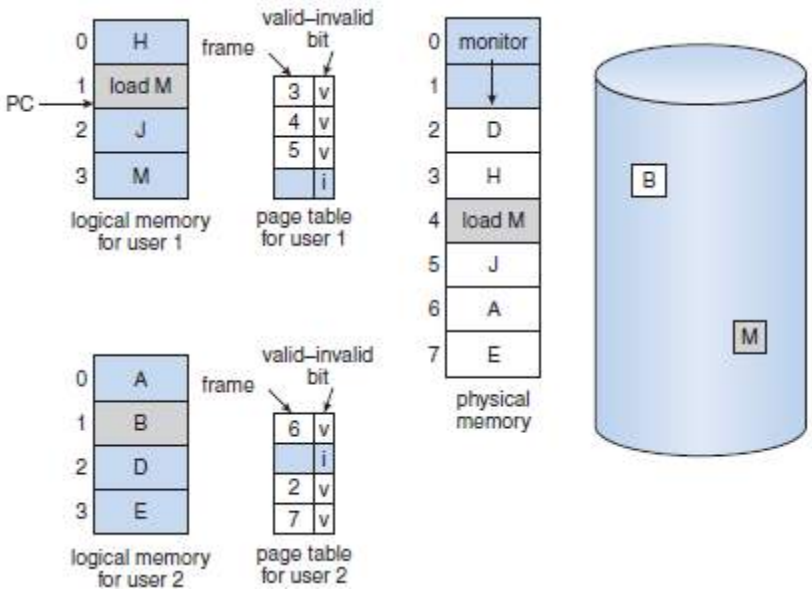


Figure 5.23 Need for page replacement

Basic Page Replacement

- Basic page replacement approach:
  → If no frame is free, we find one that is not currently being used and free it (Figure 5.24).
- Page replacement takes the following steps:
  1. Find the location of the desired page on the disk.
  2. Find a free frame:
     - If there is a free frame, use it.
     - If there is no free frame, use a page-replacement algorithm to select a *victim-frame.*
     - Write the victim-frame to the disk; change the page and frame-tables accordingly.
  3. Read the desired page into the newly freed frame; change the page and frame-tables.
  4. Restart the user-process.

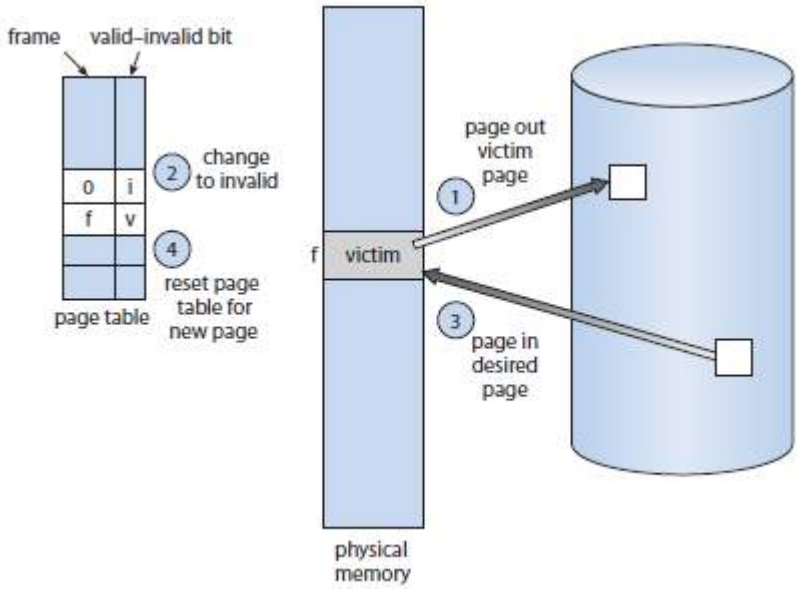


Figure 5.24 Page replacement

- Problem: If no frames are free, *2* page transfers (1 out & 1 in) are required. This situation
  → doubles the page-fault service-time and
  → increases the EAT accordingly.

  Solution: Use a *modify-bit (or dirty bit).*
- Each page or frame has a modify-bit associated with the hardware.
- The **modify-bit** for a page is set by the hardware whenever any word is written into the page (indicating that the page has been modified).
- Working:
  1. When we select a page for replacement, we examine it's modify-bit.
  2. *If the modify-bit =1*, the page has been modified. So, we must write the page to the disk.
  3. *If the modify-bit=0,* the page has *not* been modified. So, we need not write the page to the disk, it is already there.
- Advantage:
  1. Can reduce the time required to service a page-fault.
- We must solve 2 major problems to implement demand paging:
  1. Develop a ***Frame-allocation algorithm:***
     - If we have multiple processes in memory, we must decide how many frames to allocate to each process.
  2. Develop a ***Page-replacement algorithm:***
     - We must select the frames that are to be replaced.

FIFO Page Replacement

- Each page is associated with the time when that page was brought into memory.
- When a page must be replaced, the oldest page is chosen.
- We use a **FIFO queue** to hold all pages in memory (Figure 5.25).

  When a page must be replaced, we replace the page at the *head* of the queue

  When a page is brought into memory, we insert it at the *tail* of the queue.
- Example: Consider the following references string with frames initially empty.

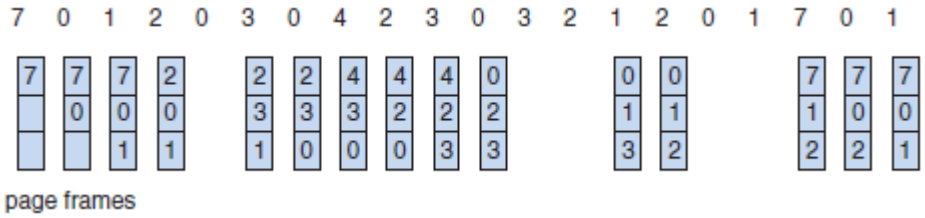


Figure 5.25 FIFO page-replacement algorithm

- The first three references(7, 0, 1) cause page-faults and are brought into these empty frames.
- The next reference(2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line.
- This process continues till the end of string.
- There are fifteen faults altogether.

- Advantage:

  1. Easy to understand & program.
- Disadvantages:

  1. Performance is not always good (Figure 5.26).
  2. A bad replacement choice increases the page-fault rate (Belady's anomaly).
- For some algorithms, the page-fault rate may increase as the number of allocated frames increases. This is known as **Belady's anomaly**.
- Example: Consider the following reference string:

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

  For this example, the number of faults for four frames (ten) is *greater* than the number of faults for three frames (nine)!

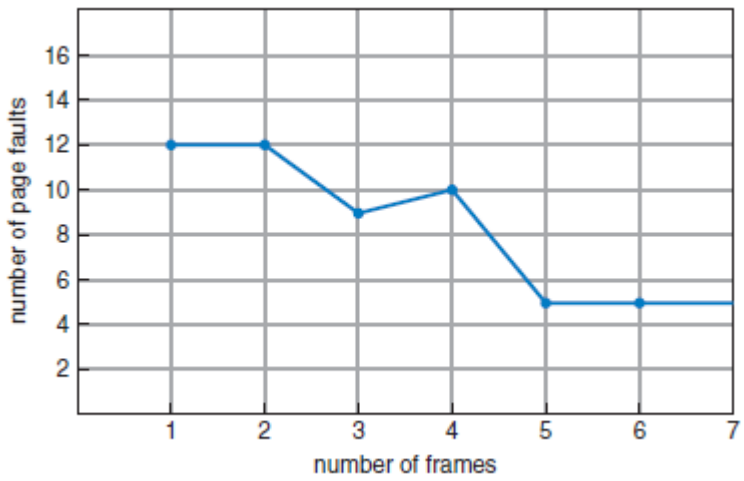


Figure 5.26 Page-fault curve for FIFO replacement on a reference string

Optimal Page Replacement

- Working principle: Replace the page that will not be used for the longest period of time (Figure 5.27).
- This is used mainly to solve the problem of Belady's Anamoly.
- This has the lowest page-fault rate of all algorithms.
- Consider the following reference string:

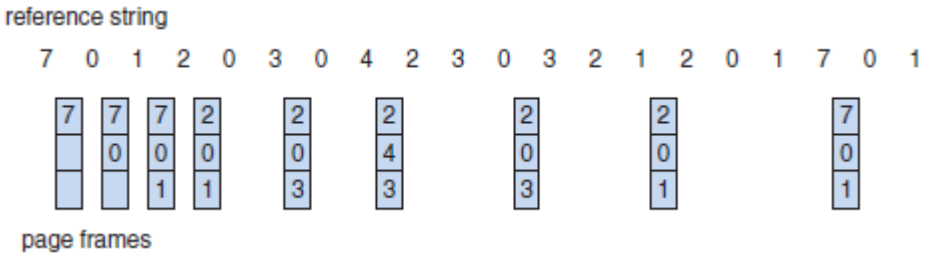


Figure 5.27 Optimal page-replacement algorithm

- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because page 7 will not be used until reference 18.
- The page 0 will be used at 5, and page 1 at 14.
- With only nine page-faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults.

- Advantage:
  1. Guarantees the lowest possible page-fault rate for a fixed number of frames.
- Disadvantage:
  1. Difficult to implement, because it requires future knowledge of the reference string.

LRU Page Replacement

- The key difference between FIFO and OPT:
  - → FIFO uses the time when a page was brought into memory.
  - → OPT uses the time when a page is to be used.
- Working principle: Replace the page that has not been used for the longest period of time.
- Each page is associated with the time of that page's last use (Figure 5.28).
- Example: Consider the following reference string:

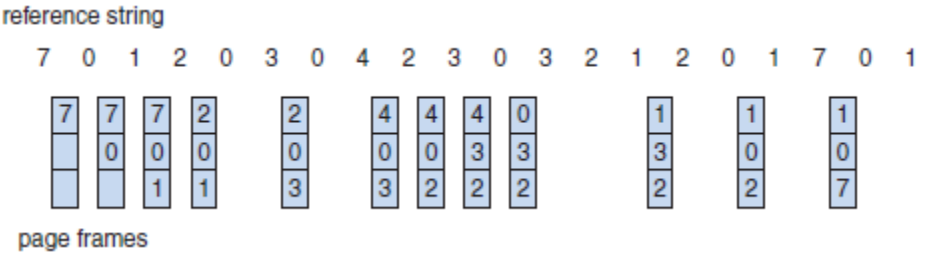


Figure 5.28 LRU page-replacement algorithm

- ➢ The first five faults are the same as those for optimal replacement.
- ➢ When the reference to page 4 occurs, LRU sees that of the three frames, page 2 was used least recently. Thus, the LRU replaces page 2.
- ➢ The LRU algorithm produces twelve faults.

- Two methods of implementing LRU:

1. Counters

- ➢ Each page-table entry is associated with a **time-of-use** field.
- ➢ A **counter(or logical clock)** is added to the CPU.
- ➢ The clock is incremented for every memory-reference.
- ➢ Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
- ➢ We replace the page with the smallest time value.

2. Stack

- ➢ Keep a stack of page-numbers (Figure 5.29).
- ➢ Whenever a page is referenced, the page is removed from the stack and put on the top.
- ➢ The most recently used page is always at the top of the stack.
  The least recently used page is always at the bottom.
- ➢ Stack is best implement by a doubly linked-list.
- ➢ Advantage:
  1. Does not suffer from Belady's anomaly.
- ➢ Disadvantage:
  1. Few computer systems provide sufficient h/w support for true LRU page replacement.
- ➢ Both LRU & OPT are called stack algorithms.

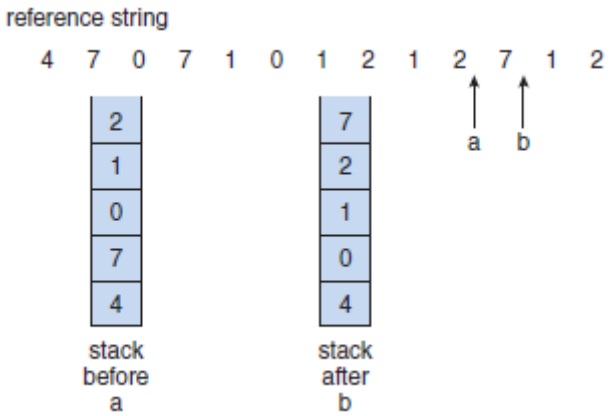


Figure 5.29 Use of a stack to record the most recent page references

LRU-Approximation Page Replacement

• Some systems provide a **reference bit** for each page.
• Initially, all bits are cleared(to 0) by the OS.
• As a user-process executes, the bit associated with each page referenced is set (to 1) by the hardware.
• By examining the reference bits, we can determine
→ which pages have been used and
→ which have not been used.
• This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Additional-Reference-Bits Algorithm

• We can gain additional ***ordering information*** by recording the reference bits at regular intervals.
• A 8-bit byte is used for each page in a table in memory.
• At regular intervals, a timer-interrupt transfers control to the OS.
• The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte.
• These 8-bit shift registers contain the history of page use, for the last eight time periods.
• Examples:
00000000 - This page has not been used in the last 8 time units (800 ms).
11111111 - Page has been used every time unit in the past 8 time units.
11000100 has been used more recently than 01110111.
• The page with the lowest number is the LRU page, and it can be replaced.
• If numbers are equal, FCFS is used

Second-Chance Algorithm

• The number of bits of history included in the shift register can be varied to make the updating as fast as possible.
• In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the **second-chance algorithm.**
• Basic algorithm is a FIFO replacement algorithm.
• Procedure:
- When a page has been selected, we inspect its reference bit.
- If reference bit=0, we proceed to replace this page.
- If reference bit=1, we give the page a second chance & move on to select next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset.

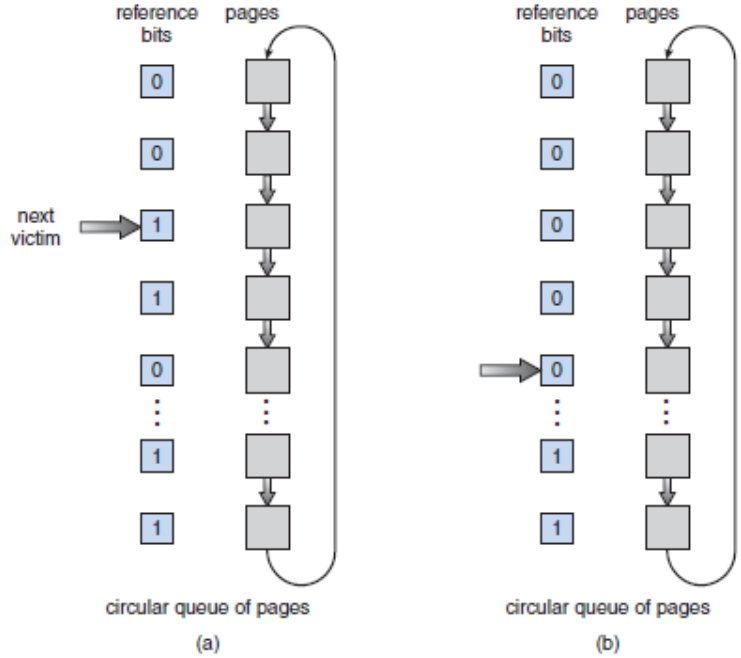


Figure 5.30 Second-chance (clock) page-replacement algorithm

- A circular queue can be used to implement the second-chance algorithm (Figure 5.30).
  - *A* pointer (that is, a hand on the clock) 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.

Enhanced Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering
  1) Reference bit and 2) modify-bit.
- We have following 4 possible classes:
  1. **(0, 0)** neither recently used nor modified -best page to replace.
  2. **(0, 1)** not recently used hut modified-not quite as good, because the page will need to be written out before replacement.
  3. **(1, 0)** recently used but clean-probably will be used again soon.
  4. **(1, 1)** recently used and modified -probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.
- Each page is in one of these four classes.
- When page replacement is called for, we examine the class to which that page belongs.
- We replace the first page encountered in the lowest nonempty class.

Counting-Based Page Replacement

1. LFU page-replacement algorithm

- Working principle: The page with the smallest count will be replaced.
- The reason for this selection is that an actively used page should have a large reference count.
- Problem:
  When a page is used heavily during initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.

  Solution:
  Shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

2. MFU (Most Frequently Used) page-replacement algorithm

- Working principle: The page with the smallest count was probably just brought in and has yet to be used.

Allocation of Frames

Minimum Number of Frames

• We must also allocate at least a minimum number of frames. One reason for this is performance.
• As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
• In addition, when a page-fault occurs before an executing instruction is complete, the instruction must be restarted.
• The minimum number of frames is defined by the computer architecture.

Allocation Algorithms

1. Equal Allocation

- We split *m* frames among *n* processes is to give everyone an equal share, *m/n* frames. (For ex: if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool).

2. Proportional Allocation

- We can allocate available memory to each process according to its size.

• In both 1 & 2, the allocation may vary according to the multiprogramming level.
• If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process.
• Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

Global versus Local Allocation

Global Replacement	***Local Replacement***
Allows a process to a replacement frame from the set of all frames.	Each process selects from only its own set of allocated frames.
A process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.	Number of frames allocated to a process does not change.
Disadvantage: A process cannot control its own page-fault rate.	Disadvantage: Might prevent a process by not making available to it other less used pages of memory.
Advantage: Results in greater system throughput.	

Thrashing

• If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
→ low CPU utilization
→ operating system thinks that it needs to increase the degree of multiprogramming
→ another process added to the system.

• If the number of frames allocated to a low-priority process falls below the minimum number required, it must be suspended.

• A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

• Thrashing results in severe performance-problems (Figure 5.31).

• The thrashing phenomenon:

➢ As processes keep faulting, they queue up for the paging device, so CPU utilization decreases

➢ The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.

➢ The new process causes even more page-faults and a longer queue!

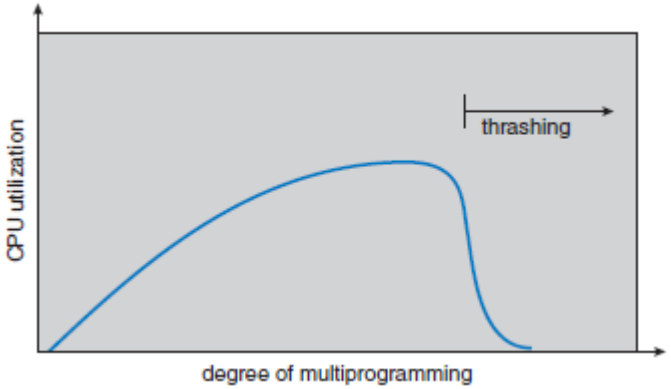


Figure 5.31 Thrashing

• Methods to avoid thrashing:

1. **Use Local Replacement**

➢ If one process starts thrashing, it cannot
→ steal frames from another process and
→ cause the latter to thrash as well.

2. We must provide a process with as many frames as it needs. This approach defines the **locality model** of process execution.

➢ Locality Model states that
As a process executes, it moves from locality to locality.

➢ A locality is a set of pages that are actively used together.

➢ A program may consist of several different localities, which may overlap.

UNIT 6: FILE SYSTEMS

File Concepts

- A ***file*** is a named collection of related info. on secondary-storage.
- Commonly, file represents
  - → program and
  - → data.
- Data in file may be
  - → numeric
  - → alphabetic or
  - → binary.
- Four types of file:
  1. **Text file**: sequence of characters organized into lines.
  2. **Source file**: sequence of subroutines & functions.
  3. **Object file**: sequence of bytes organized into blocks.
  4. **Executable file**: series of code sections.

File Attributes

1. Name

- The only information kept in human-readable form.

2. Identifier

- It is a unique number which identifies the file within file-system.
- It is in non-human-readable form.

3. Type

- It is used to identify different types of files.

4. Location

- It is a pointer to
  - → device and
  - → location of file.

5. Size

- Current-size of file in terms of bytes, words, or blocks.
- It also includes maximum allowed size.

6. Protection

- Access-control info. determines who can do
  - → reading
  - → writing and
  - → executing.

7. Time, date, & user identification

- These info. can be kept for
  - → creation
  - → last modification and
  - → last use.
- These data can be useful for
  - → protection
  - → security and
  - → usage monitoring.

- Information about files are kept in the ***directory-structure***, which is maintained on the disk.

File Operations

1. Creating a file

- Two steps are:
  - i) Find the space in the file-system for the file.
  - ii) An entry for the new file is made in the directory.

2. Writing a file

- Make a system-call specifying both
  - → file-name and
  - → info. to be written to the file.
- The system searches the directory to find the file's location. (The system keeps a *write-pointer(wp)* to the location in the file where the next write is to take place).
- The write-pointer must be updated whenever a write-operation occurs.

3. Reading a file

- Make a system-call specifying both
  - → file-name and
  - → location of the next block of the file in the memory.
- The system searches the directory to find the file's location. (The system keeps a *read-pointer(rp)* to the location in the file where the next read is to take place).
- The read-pointer must be updated whenever a read-operation occurs.
- Same pointer (rp & wp) is used for both read- & write-operations. This results in
  - → saving space and
  - → reducing system-complexity.

4. Repositioning within a file

- Two steps are:
  - i) Search the directory for the appropriate entry.
  - ii) Set the current-file-position to a given value.
- This file-operation is also known as ***file seek***.

5. Deleting a file

- Two steps are:
  - i) Search the directory for the named-file.
  - ii) Release all file-space and erase the directory-entry.

6. Truncating a file

- The contents of a file are erased but its attributes remain unchanged.
- Only file-length attribute is set to zero.

(Most of the above file-operations involve searching the directory for the entry associated with the file. To avoid this constant searching, many systems require that an 'open' system-call be used before that file is first used).

- The OS keeps a small table which contains info. about all open files (called **open-file table**).
- If a file-operation is requested, then
  - → file is specified via an index into open-file table
  - → so no searching is required.
- If the file is no longer actively used, then
  - → process closes the file and
  - → OS removes its entry in the open-file table.
- Two levels of internal tables:

1. Per-process Table

- Tracks all files that a process had opened.
- Includes access-rights to
  - → file and
  - → accounting info.
- Each entry in the table in turn points to a system-wide table

2. System-wide Table

- Contains process-independent info. such as
  - → file-location on the disk
  - → file-size and
  - → access-dates.

- Information associated with an open file:
  - ***1. File-pointer***
    - ➢ Used by the system to keep track of last read-write location.
  - ***2. File-open Count***
    - ➢ The counter
      - → tracks the no. of opens & closes and
      - → reaches zero on the last close.
  - ***3. Disk Location of the File***
    - ➢ Location-info is kept in memory to avoid having to read it from disk for each operation.
  - ***4. Access Rights***
    - ➢ Each process opens a file in an access-mode (read, write or execute).
- **File locks** allow one process to
  - → lock a file and
  - → prevent other processes from gaining access to locked-file**.**

Shared Lock	***Exclusive Lock***
Similar to a reader lock.	Behaves like a writer lock.
Several processes can acquire the lock concurrently.	Only one process at a time can acquire the lock.

Mandatory	***Advisory***
OS will prevent any other process from accessing the locked-file.	OS will not prevent other process from accessing the locked-file.
OS ensures locking integrity.	It is up to software-developers to ensure that locks are appropriately acquired and released.
Used by windows OS.	Used by UNIX systems.

File Types

- Common technique for implementing file-types: Include the type as part of the file-name.
- Two parts of file-name (Figure 6.1):
  - 1. Name and 2. Extension
- The system uses the extension to indicate
  - → type of file and
  - → type of operations (read or write).
- Example:
  - → Only a file with a .com, .exe, or .bat extension can be executed.
  - → .com and .exe are two forms of binary executable files.
  - → .bat file is a batch file containing, in ASCII format, commands to the OS.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure 6.1 Common file types

File Structure

- File types can be used to indicate the internal structure of the file.
- Disadvantage of supporting multiple file structures: Large size.
- All OSs must support at least one structure: an executable file
- In Mac OS, file contains 2 parts:
    1. **Resource fork**: contains info. of interest to the user.
    2. **Data fork**: contains program-code or data.
- Too few structures make programming inconvenient.
- Too many structures make programmer confusion.

Internal File Structure

- Locating an offset within a file can be complicated for the OS.
- Disk-systems typically have a well-defined block-size.
- All disk I/0 is performed in units of one block (physical record), and all blocks are the same size.
- Problem: It is unlikely that physical-record size will exactly match length of desired logical-record.
  Solution: **Packing** a number of logical-records into physical-blocks.
- Following things determine how many logical-records are in each physical-block:
    → logical-record size
    → physical-block size and
    → packing technique.
- The packing can be done either by
    → user's application program or
    → OS.
- Disadvantage of packing:
    → All file-systems suffer from internal fragmentation (the larger the block size, the greater the internal fragmentation).

Access Methods

Sequential Access

- This is based on a tape model of a file.
- This works both on
  - → sequential-access devices and
  - → random-access devices.
- Info. in the file is *processed in order* (Figure 6.2).
  For ex: editors and compilers
- Reading and writing are the 2 main operations on the file.
- File-operations:
  - ***1. read next***
  - ➢ This is used to
    - → read the next portion of the file and
    - → advance a file-pointer, which tracks the I/O location.
  - ***2. write next***
  - ➢ This is used to
    - → append to the end of the file and
    - → advance to the new end of file.

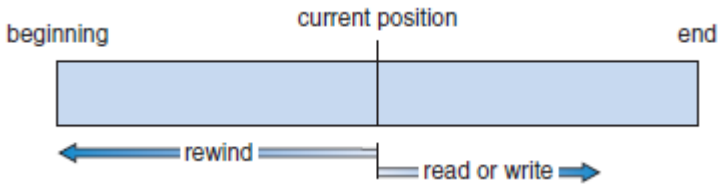


Figure 6.2 Sequential-access file

Direct Access (Relative Access)

- This is based on a disk model of a file (since disks allow random access to any file-block).
- A file is made up of fixed length *logical records***.**
- Programs can read and write records rapidly in no particular order.
- Disadvantages:
  1. Useful for immediate access to large amounts of info.
  2. Databases are often of this type.
- File-operations include a relative block-number as parameter.
- The **relative block-number** is an index relative to the beginning of the file.
- File-operations (Figure 6.3):
  - ***1. read n***
  - ***2. write n***
    where n is the block-number
- Use of relative block-numbers:
  - → allows OS to decide where the file should be placed and
  - → helps to prevent user from accessing portions of file-system that may not be part of his file.

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

Figure 6.3 Simulation of sequential access on a direct-access file

Other Access Methods

- These methods generally involve constructing a **file-index**.
- The index contains pointers to the various blocks (like an index in the back of a book).
- To find a record in the file(Figure 6.4):
    1. First, search the index and
    2. Then, use the pointer to
        → access the file directly and
        → find the desired record.
- Problem: With large files, the index-file itself may become too large to be kept in memory.

Solution: Create an index for the index-file. (The primary index-file may contain pointers to secondary index-files, which would point to the actual data-items).

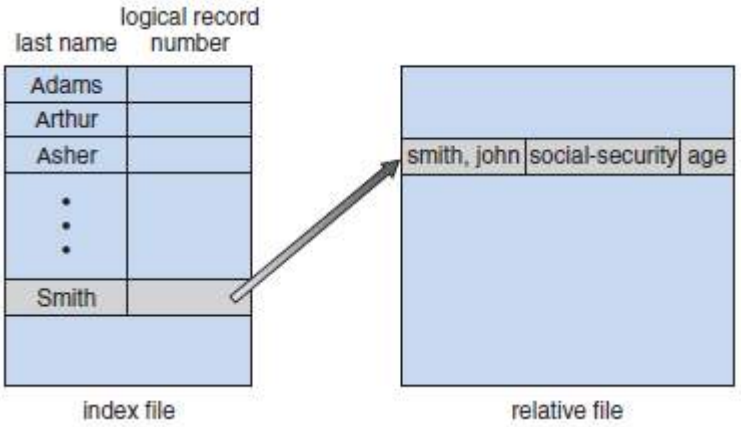


Figure 6.4 Example of index and relative files

Directory Structure

1. Single level directory
2. Two level directory
3. Tree structured directories
4. Acyclic-graph directories
5. General graph directory

Storage Structure

• A storage-device can be used in its entirety for a file-system.
• The storage-device can be split into 1 or more partitions (known as **slices** or **minidisk**).
• Any entity containing a file-system is known as a **volume**.
• The volume may be
→ a subset of a device or
→ a whole device.
• Each volume must also contain info. about the files in the system. This info. is kept in entries in a **device directory(** or volume table of contents).
• Device directory (or directory) records following info. for all files on that volume (Figure 6.5):
→ name
→ location
→ size and
→ type.

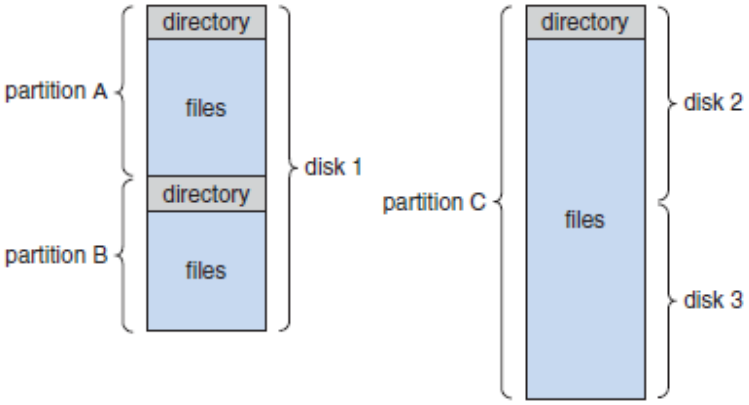


Figure 6.5: A typical file-system organization

Directory Overview

• Operations performed on a directory:

1.Search for a File

➢ We need to be able to search a directory-structure to find the entry for a particular file.

2. Create a File

➢ We need to be able to create and add new files to the directory.

3. Delete a File

➢ When a file is no longer needed, we want to be able to remove it from the directory.

4. List a Directory

➢ We need to be able to
→ list the files in a directory and
→ list the contents of the directory-entry for each file.

5. Rename a File

➢ Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.

6. Traverse the File-system

➢ We may wish to access
→ every directory and
→ every file within a directory-structure.
➢ For reliability, it is a good idea to save the contents and structure of the entire file-system at regular intervals.

Single Level Directory

- All files are contained in the same directory (Figure 6.6).
- Disadvantages (Limitations):
    1. Naming problem: All files must have unique names.
    2. Grouping problem: Difficult to remember names of all files, as number of files increases.

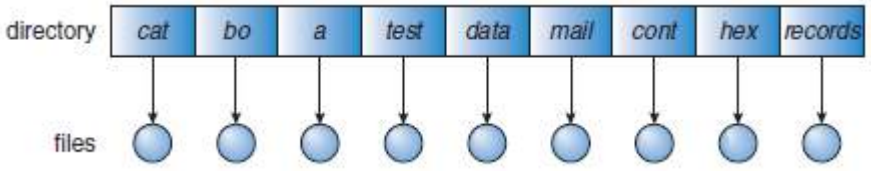


Figure 6.6 Single-level directory

Two Level Directory

- A separate directory for each user.
- Each user has his own UFD (user file directory).
- The UFDs have similar structures.
- Each UFD lists only the files of a single user.
- When a user job starts, the system's MFD is searched (MFD=master file directory).
- The MFD is indexed by user-name.
- Each entry in MFD points to the UFD for that user (Figure 6.7).

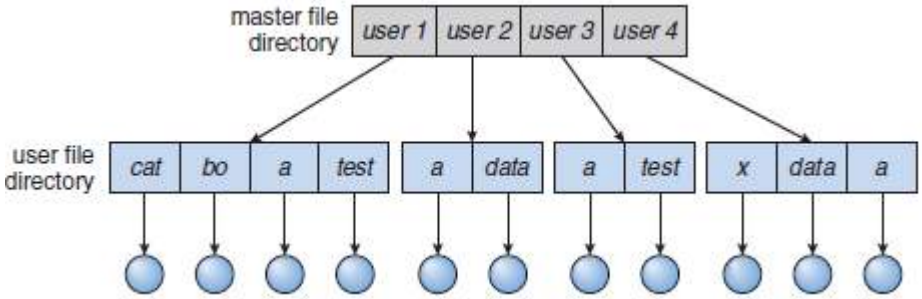


Figure 6.7 Two-level directory-structure

- To create a file for a user,
  the OS searches only that user's UFD to determine whether another file of that name exists.
- To delete a file,
  the OS limits its search to the local UFD. (Thus, it cannot accidentally delete another user's file that has the same name).
- Advantage:
    1. No filename-collision among different users.
    2. Efficient searching.
- Disadvantage
    1. Users are isolated from one another and can’t cooperate on the same task.

Tree Structured Directories

- Users can create their own subdirectories and organize files (Figure 6.8).
- A tree is the most common directory-structure.
- The tree has a root directory.
- Every file in the system has a unique path-name.

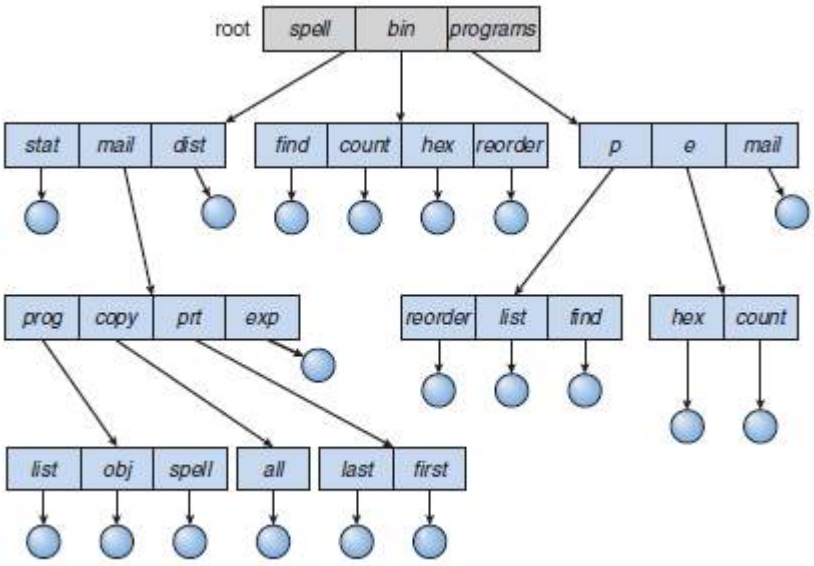


Figure 6.8 Tree-structured directory-structure

- A directory contains a set of files (or subdirectories).
- A directory is simply another file, but it is treated in a special way.
- In each directory-entry, one bit defines as
  file (0) or
  subdirectory (1).
- Path-names can be of 2 types:
- Two types of path-names:
  1. ***Absolute path-name*** begins at the root.
  2. ***Relative path-name*** defines a path from the current directory.
- How to delete directory?
  1. To delete an ***empty directory***:
     → Just delete the directory.
  2. To delete a ***non-empty directory***:
     → First, delete all files in the directory.
     → If any subdirectories exist, this procedure must be applied recursively to them.
- Advantage:
  1. Users can be allowed to access the files of other users.
- Disadvantages:
  1. A path to a file can be longer than a path in a two-level directory.
  2. Prohibits the sharing of files (or directories).

Acyclic Graph Directories

- The directories can share subdirectories and files (Figure 6.9).
  (An **acyclic graph** means a graph with no cycles).
- The same file (or subdirectory) may be in 2 different directories.
- Only *one* shared-file exists, so any changes made by one person are immediately visible to the other.

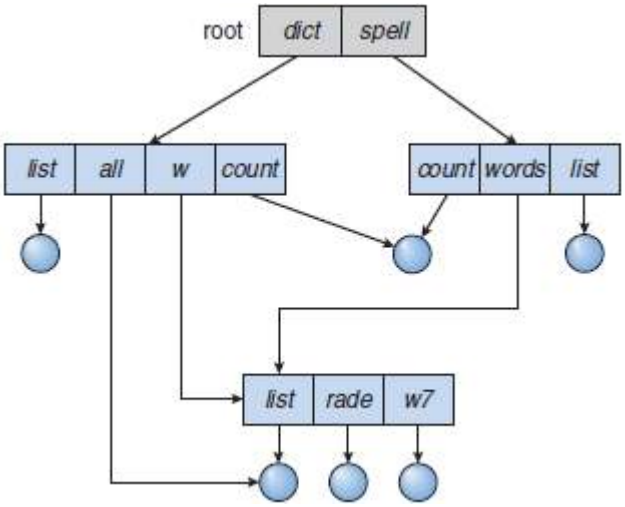


Figure 6.9 Acyclic-graph directory-structure

- Two methods to implement shared-files(or subdirectories):
  1. Create a new directory-entry called a link.
     A link is a pointer to another file (or subdirectory).
  2. Duplicate all info. about shared-files in both sharing directories.
- Two problems:
  1. A file may have multiple absolute path-names.
  2. Deletion may leave dangling-pointers to the non-existent file.

  Solution to deletion problem:
  1. Use backpointers: Preserve the file until all references to it are deleted.
  2. With symbolic links, remove only the link, not the file. If the file itself is deleted, the link can be removed.

General Graph Directory

• Problem: If there are cycles, we want to avoid searching components twice (Figure 6.10).
Solution: Limit the no. of directories accessed in a search.

• Problem: With cycles, the reference-count may be non-zero even when it is no longer possible to refer to a directory (or file). (A value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted).

Solution: Garbage-collection scheme can be used to determine when the last reference has been deleted.

• Garbage collection involves

1. First pass
   → traverses the entire file-system and
   → marks everything that can be accessed.
2. A second pass collects everything that is not marked onto a list of free-space.

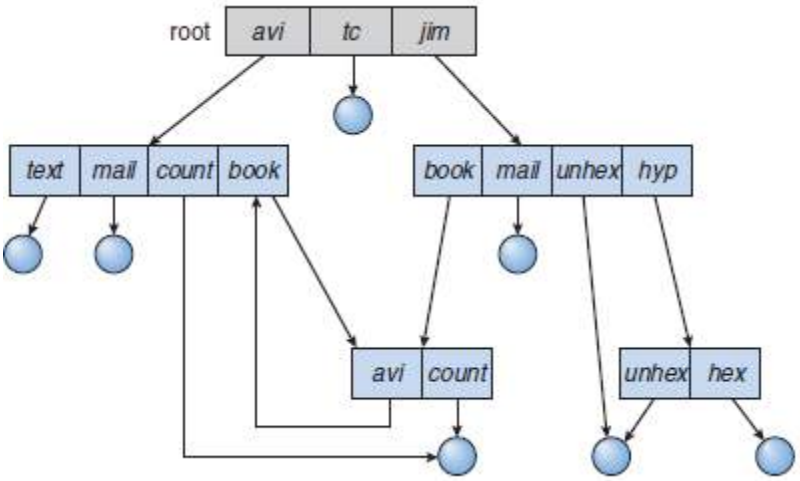


Figure 6.10 General graph directory

File System Mounting

- A file-system must be mounted before it can be available to processes on the system (Figure 6.11).
- **Mount-point** is the location in the file-structure where the file-system is to be attached.
- Procedure:
    1. OS is given
        → name of the device and
        → mount-point (Figure 6.12).
    2. OS verifies that the device contains a valid file-system.
    3. OS notes in its directory-structure that a file-system is mounted at specified mount-point.

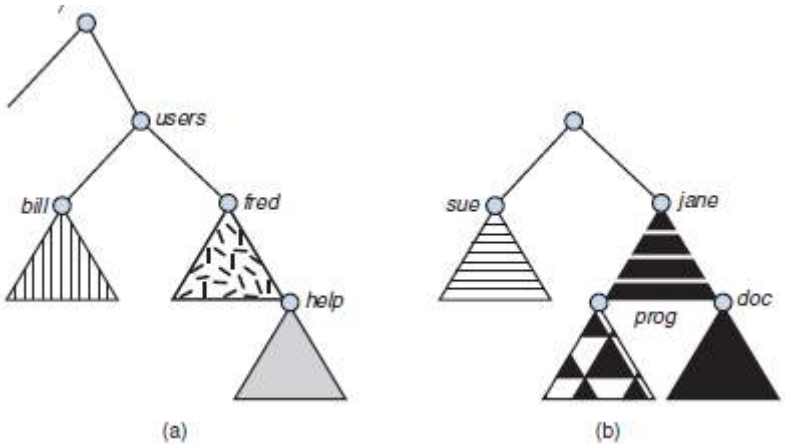


Figure 6.11 File system. (a) Existing system. (b) Unmounted volume

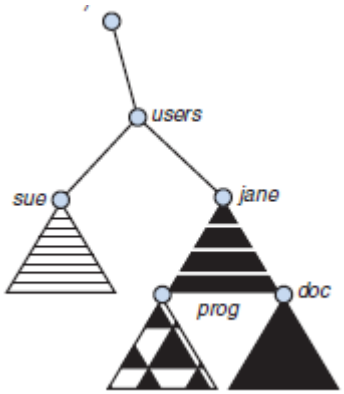


Figure 6.12 Mount point

File Sharing
• Sharing of files on multi-user systems is desirable.
• Sharing may be done through a protection scheme.
• On distributed systems, files may be shared across a network.
• Network File-system (NFS) is a common distributed file-sharing method.

Multiple Users
• File-sharing can be done in 2 ways:
1. The system can allow a user to access the files of other users by default or
2. The system may require that a user specifically grant access.
• To implement file-sharing, the system must maintain more file- & directory-attributes than on a single-user system.
• Most systems use concepts of file owner and group.
1. Owner
➢ The user who
→☐may change attributes & grant access and
→ has the most control over the file (or directory).
➢ Most systems implement owner attributes by managing a list of user-names and user IDs
2. Group
➢ The group attribute defines a subset of users who can share access to the file.
➢ Group functionality can be implemented as a system-wide list of group-names and group IDs.
• Exactly which operations can be executed by group-members and other users is definable by the file's owner.
• The owner and group IDs of a file
→ are stored with the other file-attributes.
→ can be used to allow/deny requested operations.

Remote File Systems
• Allows a computer to mount 1 or more file-systems from 1 or more remote-machines.
• Three methods:
1. **Manually via programs like FTP**.
2. **Automatically DFS** (Distributed file-system): remote directories are visible from a local machine.
3. **Semi-automatically via www** (World Wide Web): A browser is needed to gain access to the remote files, and separate operations (a wrapper for ftp) are used to transfer files.
• ftp is used for both anonymous and authenticated access.
• Anonymous access allows a user to transfer files without having an account on the remote system.

Client Server Model
• Allows clients to mount remote file-systems from servers.
• The machine containing the files is called the **server**.
The machine seeking access to the files is called the **client**.
• A server can serve multiple clients, and
A client can use multiple servers.
• The server specifies which resources (files) are available to which clients.
• A client can be specified by a network-name such as an IP address.
• Disadvantage:
1. Client identification is more difficult.
• In UNIX and its NFS (network file-system), authentication takes place via the client networking info., by default.
• Once the remote file-system is mounted, file-operation requests are sent to the server via the DFS protocol.

Distributed Information Systems
• Provides unified access to the info. needed for remote computing.
• The DNS (domain name system) provides hostname-to-networkaddress translations.
• Other distributed info. systems provide username/password space for a distributed facility.

Failure Modes

• Local file-systems can fail for a variety of reasons such as
→ failure of disk (containing the file-system)
→ corruption of directory-structure &
→ cable failure.
• Remote file-systems have more failure modes because of the complexity of network-systems.
• The network can be interrupted between 2 hosts. Such interruptions can result from
→ hardware failure
→ poor hardware configuration or
→ networking implementation issues.
• DFS protocols allow delaying of file-system operations to remote-hosts, with the hope that the remote-host will become available again.
• To implement failure-recovery, some kind of state info. may be maintained on both the client and the server.

Consistency Semantics

• These represent an important criterion of evaluating file-systems that supports file-sharing.
• These specify how multiple users of a system are to access a shared-file simultaneously.
• In particular, they specify when modifications of data by one user will be observed by other users.
• These semantics are typically implemented as code with the file-system.
• These are directly related to the process-synchronization algorithms.
• A successful implementation of complex sharing semantics can be found in the Andrew file-system (AFS).

UNIX Semantics

• UNIX file-system (UFS) uses the following consistency semantics:
1. Writes to an open-file by a user are visible immediately to other users who have this file opened.
2. One mode of sharing allows users to share the pointer of current location into a file. Thus, the advancing of the pointer by one user affects all sharing users.
• A file is associated with a single physical image that is accessed as an exclusive resource.
• Contention for the single image causes delays in user processes.

Session Semantics

• The AFS uses the following consistency semantics:
1. Writes to an open file by a user are not visible immediately to other users that have the same file open.
2. Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.
• A file may be associated temporarily with several (possibly different) images at the same time.
• Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, without delay.
• Almost no constraints are enforced on scheduling accesses.

Immutable Shared Files Semantics

• Once a file is declared as shared by its creator, it cannot be modified.
• An immutable file has 2 key properties:
1. File-name may not be reused and
2. File-contents may not be altered.
• Thus, the name of an immutable file signifies that the contents of the file are fixed.
• The implementation of these semantics in a distributed system is simple, because the sharing is disciplined

Protection

• When info. is stored in a computer system, we want to keep it safe from physical damage (reliability) and improper access (protection).
• Reliability is generally provided by duplicate copies of files.
• For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer.
• File owner/creator should be able to control:
→ what can be done
→ by whom.

Types of Access

• Systems that do not permit access to the files of other users do not need protection. This is too extreme, so controlled-access is needed.
• Following operations may be controlled:

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

Access Control

• Common approach to protection problem: make access dependent on identity of user.
• Files can be associated with an ACL (access-control list) which specifies
→ username and
→ types of access for each user.
• Problems:
1. Constructing a list can be tedious.
2. Directory-entry now needs to be of variable-size, resulting in more complicated space management.

Solution: These problems can be resolved by combining ACLs with an 'owner, group, universe' access-control scheme

• To reduce the length of the ACL, many systems recognize 3 classifications of users:

1. Owner

The user who created the file is the owner.

2. Group

A set of users who are sharing the file and need similar access is a group.

3. Universe

All other users in the system constitute the universe.

• Samples:

			RWX
a) owner access	7	⇒	1 1 1
b) group access	6	⇒	RWX 1 1 0
c) public access	1	⇒	RWX 0 0 1

E.g. rwx bits indicate which users have permission to read/write/execute

A Sample UNIX directory listing:

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 jwg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2012	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2012	program
drwx--x--x	4 tag	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Other Protection Approaches

• A password can be associated with each file.
• Disadvantages:
1. The no. of passwords you need to remember may become large.
2. If only one password is used for all the files, then all files are accessible if it is discovered.
3. Commonly, only one password is associated with all of the user's files, so protection is all-or-nothing.
• In a multilevel directory-structure, we need to provide a mechanism for directory protection.
• The directory operations that must be protected are different from the File-operations:
1. Control creation & deletion of files in a directory.
2. Control whether a user can determine the existence of a file in a directory.

UNIT 6(CONT.): IMPLEMENTING FILE SYSTEM

File System Structure

• Disks provide the bulk of secondary-storage on which a file-system is maintained.
• The disk is a suitable medium for storing multiple files. This is because

1. A disk can be rewritten in place.

➢ It is possible to
→ read a block from the disk
→ modify the block and
→ write the block into the disk.

2. A disk can access directly any block of information.

➢ It is possible to access any file either sequentially or randomly.
➢ 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.
➢ Each block has one or more sectors.
➢ Depending on the disk drive, sector-size varies from 32 bytes to 4096 bytes.
➢ The usual size is 512 bytes.
• File-systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily
• Design problems of file-systems:
1. Defining how the file-system should look to the user.
2. Creating algorithms & data-structures to map the logical file-system onto the physical secondary-storage devices.
• The file-system itself is generally composed of many different levels.
➢ Every level in design uses features of lower levels to create new features for use by higher levels.

Layered File System

• Levels of the file-system(Figure 6.13):

1. I/O Control (Lowest level)

➢ Consists of device-drivers & interrupt handlers to transfer info. between main-memory & disk.

➢ A device-driver can be thought of as a translator.

Its input consists of high-level commands.

Its output consists of low-level instructions.

2. Basic File-system

➢ Needed only to issue basic commands to the appropriate device-driver to read & write blocks on the disk.

3. File-organization Module

➢ Knows about files and their logical & physical blocks.

➢ Translates logical-block address to physical-block address.

4. Logical File-system

➢ Manages metadata information. i.e. **Metadata** includes all of the file-system structure except the actual data.

➢ Manages the directory-structure.

➢ Maintains file-structure via FCB (File Control Blocks). i.e. **FCB** contains info. about the file, including

→ ownership

→ permissions and

→ location of the file.

• Advantages of layered structure:

1. Duplication of code is minimized.
2. I/O control can be used by multiple file-systems.

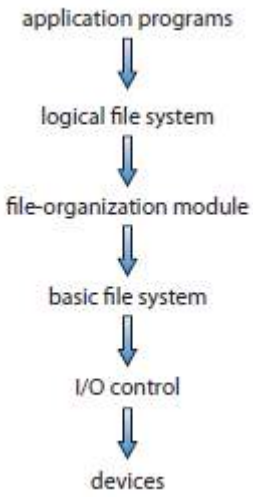


Figure 6.13 Layered file system

File System Implementation

Overview

• On-disk & in-memory structures are used to implement a file-system.

• On-disk structures include (Figure 6.14):

1. Boot Control Block

➢ Contains info. needed to boot an OS from the partition.

➢ 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.

2. Partition Control Block

➢ Contains partition-details like

→ no. of blocks

→ size of blocks and

→ free-block count.

➢ In UFS, this is called a superblock.

In NTFS, it is stored in the master file table.

3. Directory-structure

➢ Used to organize the files.

➢ In UFS, this includes file-names and associated inode-numbers.

In NTFS, it is stored in the master file table.

4. FCB (file control block)

➢ Contains file-details including

→ file-permissions

→ ownership

→ file-size and

→ location of data-blocks.

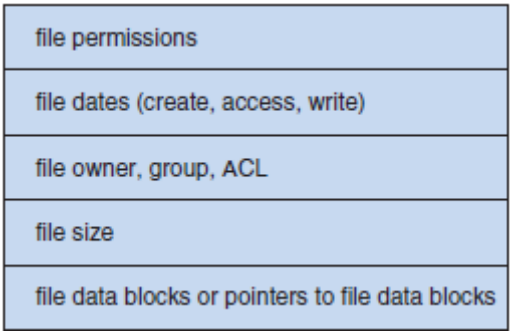


Figure 6.14 A typical file-control block

• In-memory structures are used for both file-system management and performance improvement via caching: The structures may include:

1. In-memory Mount Table

➢ Contains info. about each mounted partition.

2. An in-memory Directory-structure

➢ Holds directory info. of recently accessed directories.

3. System-wide Open-file Table

➢ Contains a copy of the FCB of each open file

4. Per-process Open-file Table

➢ Contains a pointer to the appropriate entry in the system-wide open-file table.

• **Buffers** hold file-system blocks when they are being read from disk or written to disk.

• To create a new file, a program calls the LFS (logical file-system).

The 'LFS' knows the format of the directory-structures.

• To create a new file, the LFS

1. Allocates a new FCB.
2. Reads the appropriate directory into memory.
3. Updates LFS with the new file-name and FCB.
4. Writes LFS back to the disk (Figure 6.15).

- After a file has been created, it can be used for I/O.
    1. First the file must be opened.
    2. FCB is copied to a system-wide open-file table in memory.
    3. An entry is made in the **per-process** open-file table, with a pointer to the entry in the **system-wide** open-file table.
    4. The open call returns a pointer to the appropriate entry in the per-process file-system table.
    5. All file operations are then performed via this pointer.
    6. When a process closes the file
        i) The per-process table entry is removed.
        ii) The system-wide entry's open count is decremented.

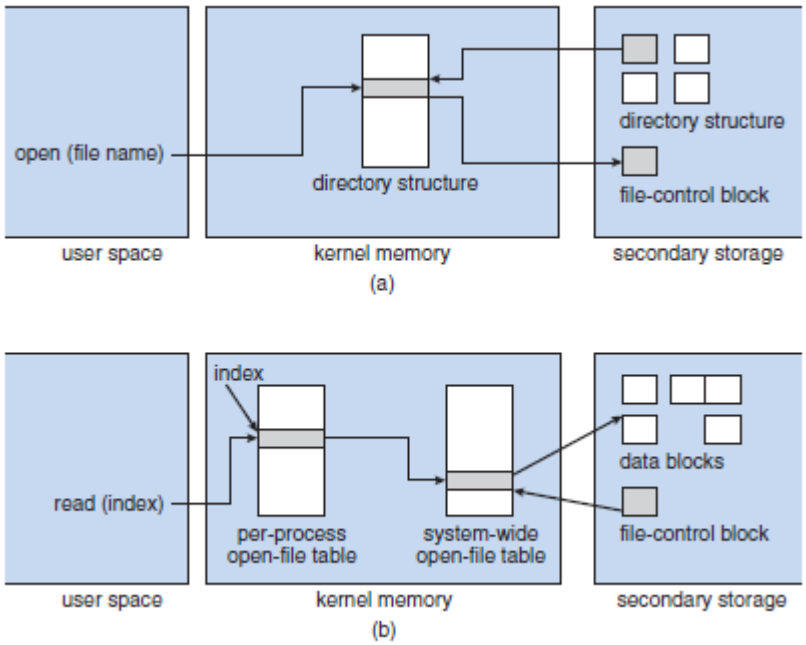


Figure 6.15 In-memory file-system structures. (a) File open. (b) File read

Partitions & Mounting

- Disk layouts can be:
    1. A disk can be divided into multiple partitions or
    2. A partition can span multiple disks (RAID).
- Each partition can either be:
    1. **Raw** i.e. containing no file-system or
    2. **Cooked** i.e. containing a file-system.
- ***Boot info.*** is a sequential series of blocks, loaded as an image into memory.
    - ➢ Execution of the image starts at a predefined location, such as the first byte.
- The boot info. has its own format, because
    → at boot time the system does not have device-drivers loaded and
    → . `. the system cannot interpret the file-system format.
- Steps for mounting:
    1. The root partition containing the kernel is mounted at boot time.
    2. Then, the OS verifies that the device contains a valid file-system.
    3. Finally, the OS notes in its in-memory mount table structure that
        i) A file-system is mounted and
        ii) Type of the file-system.

Virtual File Systems

- The OS allows multiple types of file-systems to be integrated into a directory-structure.
- Three layers (Figure 6.16):

1. File-system Interface

➢ This is based on the open(), read(), writeO and closeO calls on file descriptors.

2. File-system (VFS) Interface

➢ This serves 2 functions:

1. Separates file-system basic operations from their implementation by defining a clean VFS interface.
2. The VFS is based on a file-representation structure called a **vnode**.
   vnode contains a numerical designator for a network-wide unique file.

3. Local File-system

➢ Local files are distinguished according to their file-system types.

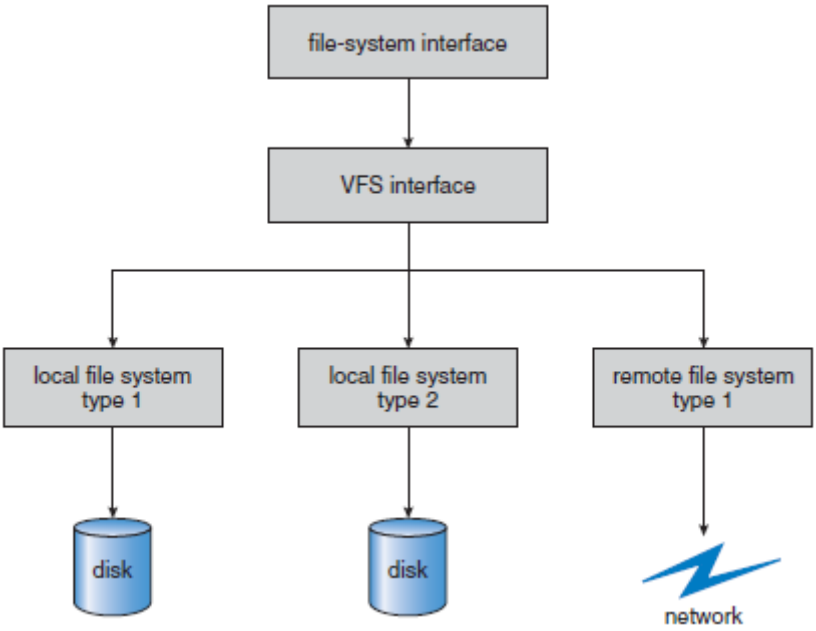


Figure 6.16 Schematic view of a virtual file system

Directory Implementation

1. Linear-list
2. Hash-table

Linear List

- A linear-list of file-names has pointers to the data-blocks.
- To create a new file:
    1. First search the directory to be sure that no existing file has the same name.
    2. Then, add a new entry at the end of the directory.
- To delete a file:
    1. Search the directory for the named-file and
    2. Then release the space allocated to the file.
- To reuse the directory-entry, there are 3 solutions:
    1. Mark the entry as unused (by assigning it a special name).
    2. Attach the entry to a list of free directory entries.
    3. Copy the last entry in the directory into the freed location & to decrease length of directory.
- Problem: Finding a file requires a linear-search which is slow to execute.
  Solutions:
    1. A cache can be used to store the most recently used directory information.
    2. A sorted list allows a binary search and decreases search time.
- Advantage:
    1. Simple to program.
- Disadvantage:
    1. Time-consuming to execute.

Hash Table

- A linear-list stores the directory-entries. In addition, 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.
- Advantages:
    1. Decrease the directory search-time.
    2. Insertion & deletion are easy.
- Disadvantages:
    1. Some provision must be made for collisions i.e. a situation in which 2 file-names hash to the same location.
    2. Fixed size of hash-table and the dependence of the hash function on that size.

Allocation Methods

- The direct-access nature of disks allows us flexibility in the implementation of files.
- In almost every case, many files are stored on the same disk.
- Main problem:
  How to allocate space to the files so that
  → disk-space is utilized effectively and
  → files can be accessed quickly.
- Three methods of allocating disk-space:
  1. Contiguous
  2. Linked and
  3. Indexed
- Each method has advantages and disadvantages.
- Some systems support all three (Data General's RDOS for its Nova line of computers).

Contiguous Allocation

- Each file occupies a set of contiguous-blocks on the disk (Figure 6.17).
- Disk addresses define a linear ordering on the disk.
- The number of disk seeks required for accessing contiguously allocated files is minimal.
- Both sequential and direct access can be supported.
- Problems:
  1. Finding space for a new file
  ➢ External fragmentation can occur.
  2. Determining how much space is needed for a file.
  ➢ If you allocate too little space, it can't be extended.
  Two solutions:
  i) The user-program can be terminated with an appropriate error-message. The user must then allocate more space and run the program again.
  ii) Find a larger hole,
  copy the contents of the file to the new space and
  release the previous space.
- To minimize these drawbacks:
  1. A contiguous chunk of space can be allocated initially and
  2. Then when that amount is not large enough, another chunk of contiguous space (known as an 'extent') is added.

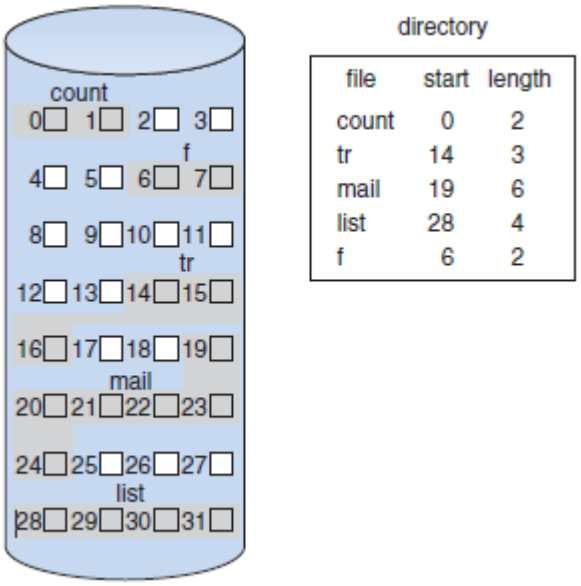


Figure 6.17 Contiguous allocation of disk-space

Linked Allocation

- Each file is a linked-list of disk-blocks.
- The disk-blocks may be scattered anywhere on the disk.
- The directory contains a pointer to the first and last blocks of the file (Figure 6.18).
- To create a new file, just create a new entry in the directory (each directory-entry has a pointer to the disk-block of the file).
    1. A **write** to the file causes a free block to be found. This new block is then written to and linked to the eof (end of file).
    2. A **read** to the file causes moving the pointers from block to block.
- Advantages:
    1. No external fragmentation, and any free block on the free-space list can be used to satisfy a request.
    2. The size of the file doesn't need to be declared on creation.
    3. Not necessary to compact disk-space.
- Disadvantages:
    1. Can be used effectively only for sequential-access files.
    2. Space required for the pointers.
       Solution: Collect blocks into multiples (called 'clusters') & allocate clusters rather than blocks.
    3. Reliability: Problem occurs if a pointer is lost(or damaged).
       Partial solutions: i) Use doubly linked-lists.
       ii) Store file name and relative block-number in each block.

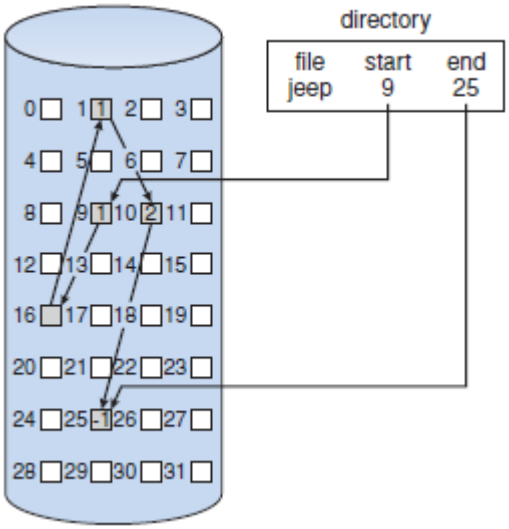


Figure 6.18 Linked allocation of disk-space

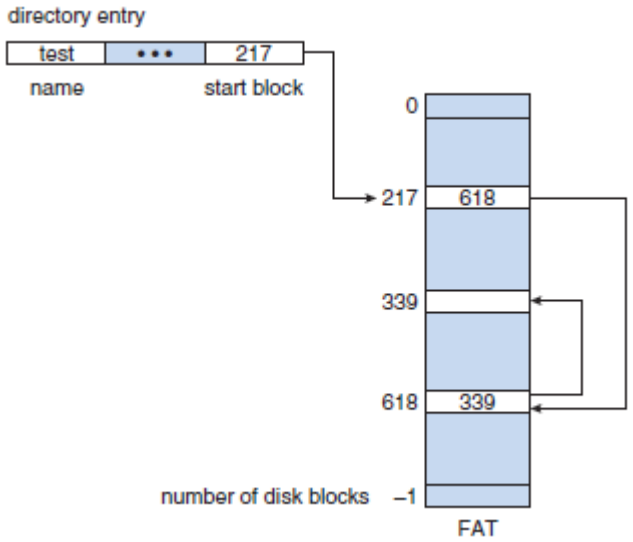


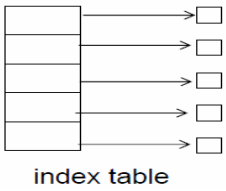
Figure 6.19 File-allocation table

- FAT is a variation on linked allocation (FAT=File Allocation Table).
- A section of disk at the beginning of each partition is set aside to contain the table (Figure 6.19).
- The table
    → has one entry for each disk-block and
    → is indexed by block-number.
- The directory-entry contains the block-number of the first block in the file.
- The table entry indexed by that block-number then contains the block-number of the next block in the file.
- This chain continues until the last block, which has a special end-of-file value as the table entry.
- Advantages:
    1. Cache can be used to reduce the no. of disk head seeks.
    2. Improved access time, since the disk head can find the location of any block by reading the info in the FAT.

Indexed Allocation

• Solves the problems of linked allocation (without a FAT) by bringing all the pointers together into an index block.

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



Logical view of the Index Table

• The ith entry in the index block points to the ith file block (Figure 6.20).

• The directory contains the address of the index block.

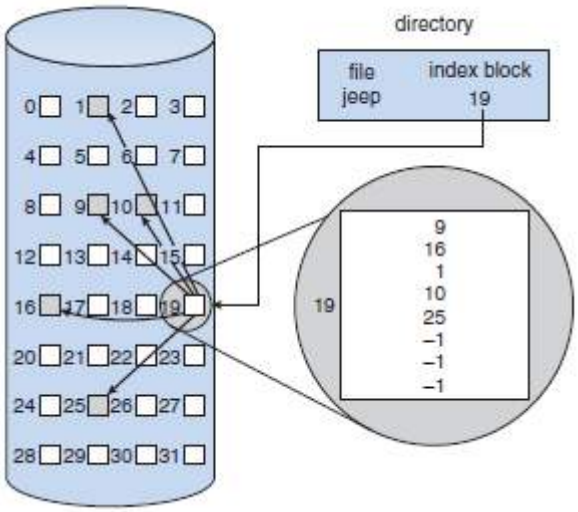


Figure 6.20 Indexed allocation of disk space

• When the file is created, all pointers in the index-block are set to nil.

• When writing the ith block, a block is obtained from the free-space manager, and its address put in the ith index-block entry,

• Problem: If the index block is too small, it will not be able to hold enough pointers for a large file,
Solution: Three mechanisms to deal with this problem:

1. Linked Scheme

➢ To allow for large files, link several index blocks,

2. Multilevel Index

➢ A first-level index block points to second-level ones, which in turn point to the file blocks,

3. Combined Scheme

➢The first few pointers point to direct blocks (i.e. they contain addresses of blocks that contain data of the file).

➢ The next few pointers point to indirect blocks.

• Advantage:

1. Supports direct access, without external fragmentation,

• Disadvantages:

1. Suffer from wasted space,
2. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation,
3. Suffer from performance problems,

Performance

Contiguous Allocation

1Adv. Requires only one access to get a disk-block
2Adv. We can calculate immediately the disk address of the next block and read it directly
3Adv. Good for direct access

Linked Allocation

1Adv. Good for sequential access
1Dis. Not be used for an application requiring direct access

Indexed Allocation

1Adv. If the index block is already in memory, then the access can be made directly
1Dis. keeping the index block in memory requires considerable space

Free Space Management

• A ***free-space list*** keeps track of free disk-space (i.e. those not allocated to some file or directory).
• To create a file,
1. We search the free-space list for the required amount of space.
2. Allocate that space to the new file.
3. This space is then removed from the free-space list.
• To delete a file, its disk-space is added to the free-space list.

Bit Vector

• The free-space list is implemented as a bit map/bit vector.
• Each block is represented by a bit.
1. If the block is free, the bit is 1.
2. If the block is allocated, the bit is 0.
• For example, consider a disk where blocks 2, 3, 4, 5 and 7 are free and the rest of the blocks are allocated. The free-space bit map will be
00111101
• Advantage:
1. Relative simplicity & efficiency in finding the first free block, or 'n' consecutive free blocks.
• Disadvantages:
1. Inefficient unless the entire vector is kept in main memory.
2. The entire vector is written to disc occasionally for recovery.

Linked List

• The basic idea:
1. Link together all the free disk-blocks (Figure 6.21).
2. Keep a pointer to the first free block in a special location on the disk.
3. Cache the block in memory.
• The first block contains a pointer to the next free one, etc.
• Disadvantage:
1. Not efficient, because to traverse the list, each block is read.
• Usually the OS simply needs a free block, and uses the first one.

Grouping

• The addresses of n free blocks are stored in the 1st free block.
• The first n-1 of these blocks are actually free.
• The last block contains addresses of another n free blocks, etc.
• Advantage:
1. Addresses of a large no of free blocks can be found quickly.

Counting

• Takes advantage of the fact that, generally, several contiguous blocks may be allocated/freed simultaneously.
• Keep the address of the first free block and the number 'n' of free contiguous blocks that follow the first block.
• Each entry in the free-space list then consists of a disk address and a count.

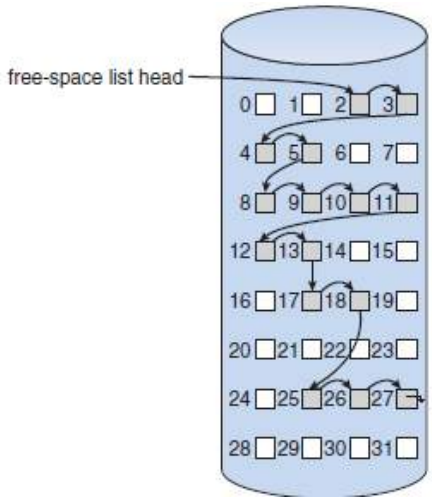


Figure 6.21 Linked free-space list on disk