SYLLABUS

Level	:	Bachelor	Full Marks: 60+20+20
Course	:	B.Sc. CSIT	Pass Marks: 24+8+8
Subject	:	**Operating System** (CSC-203)	
Year	:	II	
Credit Hour	:	3 CH	Semester: I

1 INRTODUCTION 5 hrs

1.1 History of Operating System:

- The First Generation of Computer
- The Second Generation of Computer
- The Third Generation of Computer
- The Fourth Generation of Computer

1.2 Operating System Concept:

- Real-Time & Time Sharing
- Mainframe Operating System
- Personal Computer (PC) Operating System
- Introduction To System Calls
- The Shell

1.3 Operating System Structure:

- Monolithic Systems
- Layered Systems
- Virtual Machines
- Client-Server Model

2 PROCESS MANAGEMENT 14 hrs

2.1 Introduction to Processes:

- The Process Model
- Implementation of Processes
- Threads
- Thread Model
- Thread Usage
- Implementing Thread In User Space

2.2 Interprocess Communication & Synchronization:

- Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep & Wakeup
- Semaphores
- Introduction To Message Passing
- The Dining Philosophers Problem

2.3 Process Scheduling:

- Round Robin Scheduling
- Priority Scheduling

- Multiple Queues

3 DEADLOCK 5 hrs

3.1 Deadlocks:

- Conditions for Deadlock
- Deadlock Modeling

3.2 Deadlock Detection, Recovery and Prevention:

- Deadlock Detection with One Resource of Each Type
- Deadlock Detection with Multiple Resource of Each Type
- Deadlock Prevention

4 MEMORY MANAGEMENT 7 hrs

4.1 Memory Management without Swapping or Paging:

- Monoprogramming without Swapping & Paging
- Multiprogramming and Memory Usage
- Multiprogramming and Fixed Partition

4.2 Swapping:

- Memory Management with Bit Maps
- Memory Management with Linked Lists
- Memory Management with Buddy System
- Allocation of Swap Space
- Analysis of Swapping Systems

4.3 Virtual Memory:

- Paging
- Page Tables
- Example of Paging Hardware
- Associative Memory

4.4 Page Replacement Algorithms:

- The optimal Page Replacement Algorithms
- The First-in, First-out
- The Second Chance Page Replacement Algorithms
- The Least Recently Used
- Modeling Paging Algorithms (Stack Algo.)

4.5 Segmentation:

- Implementation of Pure Segmentation
- Segmentation with Paging: MULTIC
- Segmentation with Paging: The Intel

5 DEVICE MANAGEMENT 8 hrs

5.1 Principle of I/O Hardware:

- I/O Device
- Device Controller
- Direct Memory Access

5.2 Principle of I/O Software:

- Goals of I/O Software
- Interrupt Handlers

- Device Drivers

5.3 Disk Management:

- Disk Structure
- Disk Scheduling Algorithm
- Error Handling and Formatting
- Stable Storage Management

5.4 Terminals:

- Terminal Hardware
- Memory-Mapped Terminals
- Input / Output Software

6 FILE SYSTEM 6 hrs

6.1 Files:

- File Naming
- File Structure
- File Types
- File Access
- File Attributes
- File Operations
- Memory Mapped Files

6.2 Directories:

- Hierarchical Directory System
- Path Names
- Directory Operations

6.3 File System Implementation:

- Implementing Files
- Implementing Directories
- Shared Files
- Disk Space Management
- File System Reliability
- File System Performance

Text Books:

1. Modern Operating System – Andrew S. Tanenbaum, 2nd Edition
2. An Introduction to Operating System Concepts and Practice – Pramod Chandra P. Bhatt, 2nd Edition
3. Operating System Concept - Silberschatz, Galvin adn Gagne, 6th Edition

Laboratories Works:

Small type of programming (using C programming) of:

- Process Creation
- Process Termination
- Process Deletion
- Process Communication
- Classical Interprocess Communication Problems
- Filing System

- I/O Handling

Assignments:

- 10 Assignments

Tests:

- Internal Tests

Teaching Techniques:

- Lectures
- Demonstration
- Assignment (after completion of a unit)
- Oral/Viva

Working Environment:

- Linux/Windows Based

Case Study:

- Any One Operating System

TRIBHUVAN UNIVERSITY
Central Department of Computer Science
&
Information Technology

Level	:	Bachelor (B.Sc. CSIT)	Code :	CSC-203
Subject	:	Operating System		
Year	:	II	Semester:	I

SECTION "A"

(2Q x 10 = 20 Marks)

Any Two Questions:

1. What is Files? Discuss the must common system calls relating to files.

   OR

   What is System Calls? Explain the system call flow with the help of a block diagram.
2. Explain the four basic modes of Input/Output operations.
3. a) How is the Direct Memory Access (DMA) set up?

   b) Explain the concept of Buffering.

   c) How interrupt is enabling and detected?

SECTION "B"

(8Q x 5 = 40 Marks)

Any Eight Questions:

4. What are the main motivations and issues in primary memory management?
5. List some differences between Personal Computer Operating Systems and Mainframe Operating Systems.
6. Explain the difference between Busy Waiting and Blocking.

7. Explain why two-level and scheduling is commonly used.
8. Explain the Hierarchical Directory Systems with diagrammatic examples.
9. What is the difference between Program and Process?
10. Give briefly at least three different ways of establishing interprocess communication?
11. A system has four processes P1 through P4 and two resource types R1 and R2. It has 2 units of R1 and 3 units of R2. Given that:

    P1 request 2 units of R2 and 1 unit of R1
    P2 holds 2 units of R1 and 1 unit of R2
    P3 holds 1 unit of R1
    P4 requests 1 unit of R1

    Show the resource graph for the state of the system. Is the system in deadlock, and if so, which processes are involved.
12. Write short notes on:
    a) File Structure
    b) The First-In, First-Out (FIFO) Page Replacement Algorithms.

Modern computers come with one or more processors, some main memory, disks, printers, keyboards, displays, networks interface and other input devices. Over all modern computers are complex systems. For well-functioning of these components as well as keeping track of these components an operating system is required.

An operating system is an interface between users and hardware. The operating system hides the complexities of hardware from users. Or in easy words an operating system performs the logical operations in machine language and displays it to the user in a non-complex way as in fig 1.1.

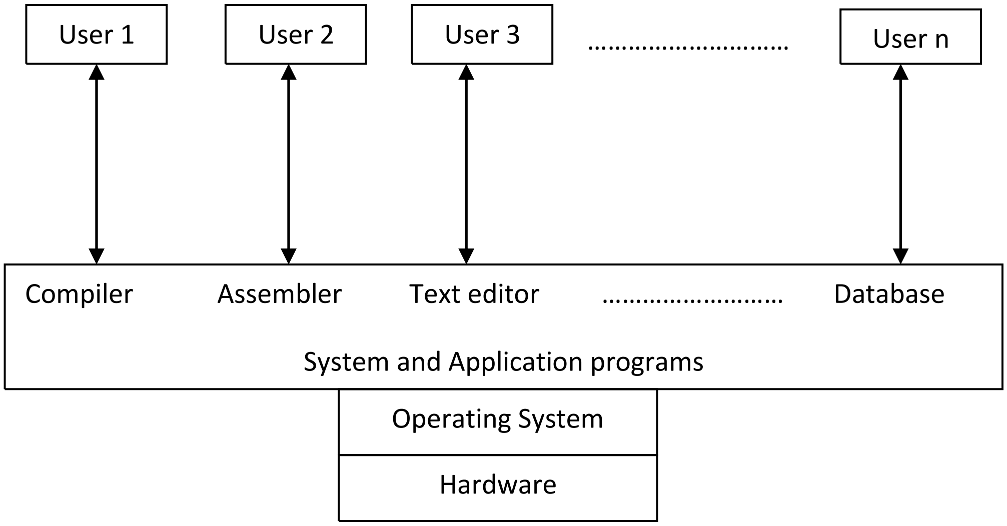


Fig 1.1: An overview of complex system

Hardware: Provides basic computing resources (CPU, Memory, I/O devices etc)

Operating System: Controls and co-ordinates the use of the hardware among the various users. It is usually that portion of software that runs in kernel mode or supervisor mode.

Application Programs: Define the way in which the system resources are used to solve the computing problems.

Users: People, machine, other computers.

History of operating System:

Operating systems have been evolving through the years. The first true digital computer was designed by the English mathematician Charles Babbage (1792-1871). Although Babbage spent most of his life and fortune trying to build his “analytical engine.” he never got it working properly because it was purely mechanical, and the technology of his day could not produce the required wheels, gears, and cogs to the high precision that he needed. Needless to say, the analytical engine did not have an operating system.

Babbage realized that he would need software for his analytical engine, so he hired a young woman named Ada Lovelace. The programming language Ada® is named after her.

The First Generation (1945-55) Vacuum Tubes and Plug boards

After Babbage's unsuccessful efforts, little progress was made in constructing digital computers until World War II. Around the mid-1940s, Howard Aiken at Harvard, John von Neumann at the Institute for Advanced Study in Princeton, J. Presper Eckert and William Mauchley at the University of Pennsylvania, and Konrad Zuse in Germany, among others, all succeeded in building calculating engines. These machines were enormous, filling up entire rooms with tens of thousands of vacuum tubes, but they were still millions of times slower than even the cheapest personal computers available today.

In these early days, a single group of people designed, built, programmed, operated, and maintained each machine. All programming was done in absolute machine language, often by wiring up plug boards to control the machine's basic functions. The usual made of operation was for the programmer to sign up for a block of time on the signup sheet on the wall, then come down to the machine room, insert his or her plug board into the computer, and spend the next few hours hoping that none of the 20,000 or so vacuum tubes would burn out during the run. Virtually all the problems were straightforward numerical calculations, such as grinding out tables of sines, cosines, and logarithms.

By the early 1950s, the routine had improved somewhat with the introduction of punched cards. It was now possible to write programs on cards and read them in instead of using plug boards; otherwise, the procedure was the same.

The Second Generation (1955-65) Transistors and Batch Systems

The transistor introduced in the mid-1950s changed the picture radically. For the first time, there was a clear separation between designers, builders, operators, programmers, and maintenance personnel.

These machines, now called **mainframes,** were locked away in specially air conditioned computer rooms, with staffs of professional operators to run them. To run a **job** (i.e., a program or set of programs), a programmer would first write the program on paper (in FORTRAN or assembler), then punch it on cards. When the computer finished whatever job it was currently running, an operator would go over to the printer and tear off the output and carry it over to the output room, so that the programmer could collect it later. Then he would take one of the card decks that had been brought from the input room and read it in. If the FORTRAN compiler was needed, the operator would have to get it from a file cabinet and read it in. Much computer time was wasted while operators were walking around the machine room.

Given the high cost of the equipment, it is not surprising that people quickly looked for ways to reduce the wasted time. The solution generally adopted was the **batch system.** The idea behind it was to collect a tray full of jobs in the input room and then read them onto a magnetic tape using a small (relatively) inexpensive computer, such as the IBM 1401, which was very good at reading cards, copying tapes, and printing output, but not at all good at numerical calculations. Other, much more expensive machines, such as the IBM 7094, were used for the real computing. This situation is shown in Fig. 1.2

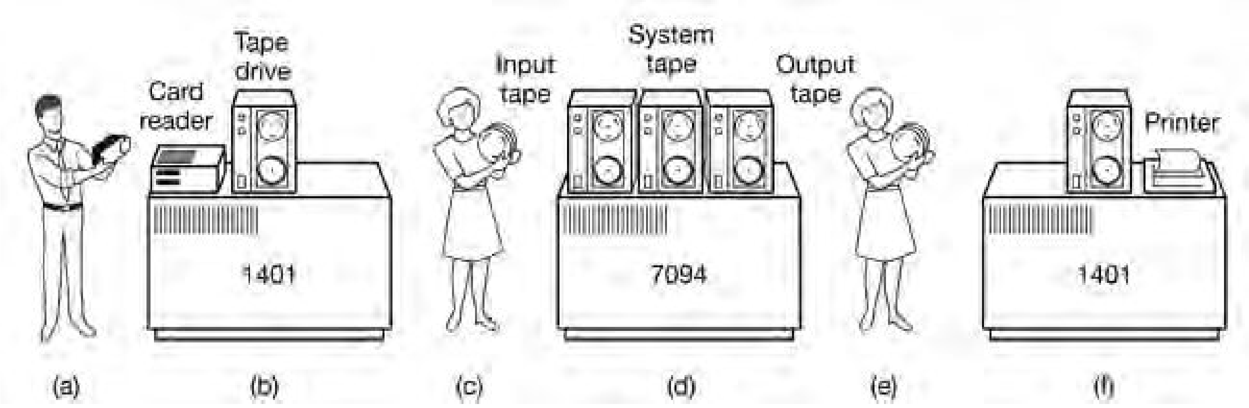


Fig 1.2

The structure of a typical input job is shown in Fig. 1-3. It started out with a $JOB card, specifying the maximum run time in minutes, the account number to be charged, and the programmer's name. Then came a $FORTRAN card, telling the operating system to load the FORTRAN compiler from the system tape. It was followed by the program to be compiled, and then a $LOAD card, directing the operating system to load the object program just compiled. (Compiled programs were often written on scratch tapes and had to be loaded explicitly.) Next came the $RUN card, telling the operating system to run the program with the data following it. Finally, the $END card marked the end of the job. These primitive control cards were the forerunners of modern job control languages and command interpreters.

Large second-generation computers were used mostly for scientific and engineering calculations, such as solving the partial differential equations that often occur in physics and engineering. They were largely programmed in FORTRAN and assembly language. Typical operating systems were FMS (the Fortran Monitor System) and IBSYS, IBM's operating system for the 7094.

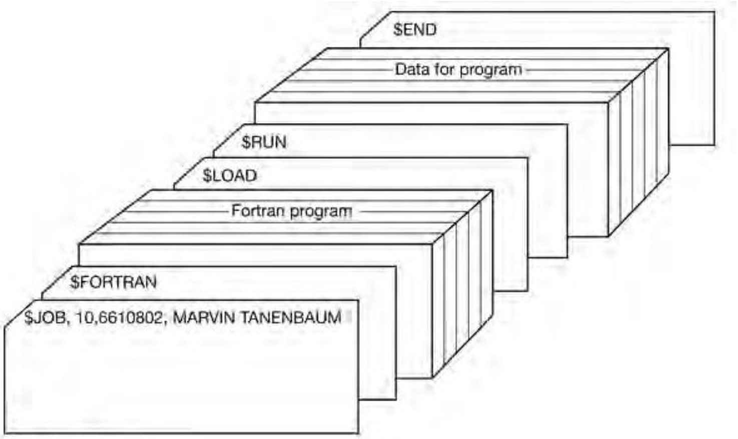


Figure 1-3. Structure of a typical FMS job.

The Third Generation (1965-1980) ICs and Multiprogramming

By the early 1960s, On the one hand there were the word-oriented, large-scale scientific computers, which were used for numerical calculations in science and engineering. On the other hand, there were the character-oriented, commercial computers, which were widely used for tape sorting and printing by banks and insurance companies.

IBM attempted to solve both of these problems at a single stroke by introducing the System/360. The 360 was a series of software-compatible machines ranging from 1401-sized to much more powerful than the 7094. Since all the machines had the same architecture and instruction set, programs written for one machine could run on all the others, at least in theory. Furthermore, the 360 was designed to handle both scientific (i.e., numerical) and commercial computing. Thus a single family of machines could satisfy the needs of all customers. In subsequent years, IBM has come out with compatible successors to the 360 line, using more modern technology, known as the 370, 4300, 3080, and 3090 series.

The 360 was the first major computer line to use (small-scale) Integrated Circuits (ICs), which were built up from individual transistors. It was an immediate success, and the idea of a family of compatible computers was soon adopted by all the other major manufacturers. The descendants of these machines are still in use at computer centers today.

The most important feature developed then was **multiprogramming.** When the current job paused to wait for a tape or other I/O operation to complete, the CPU simply sat idle until the I/O finished. With commercial data processing, the I/O wait time can often be 80 or 90 percent of the total time, so something had to be done to avoid having the CPU be idle so much.

The solution was to partition memory into several pieces with a different job in each partition, as shown in Fig. 1-4. While one job was waiting for I/O to complete, another job could be using the CPU. Having multiple jobs safely in memory at once requires special hardware to protect each job against snooping and mischief by the other ones, but the 360 and other third-generation systems were equipped with this hardware.

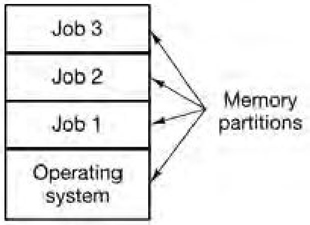


Figure 1-4. A multiprogramming system with three jobs in memory.

Another major feature present in third-generation operating systems was the ability to read jobs from cards onto the disk. Then, whenever a running job finished, the operating system could load a new job from the disk into the now-empty partition and run it. This technique is called **spooling** (from Simultaneous Peripheral Operation On Line) and was also used for output. With spooling, the 1401s were no longer needed, and much carrying of tapes disappeared.

Although third-generation operating systems were well suited for big scientific calculations and massive commercial data processing runs, they were still basically batch systems. With third-generation systems, the time between submitting a job and getting back the output was often several hours, so a single misplaced comma could cause a compilation to fail, and the programmer to waste half a day.

If 20 users are logged in and 17 of them are thinking or talking or drinking coffee, the CPU can be allocated in turn to the three jobs that want service, this concept is called time sharing. Since people debugging programs usually issue short commands (e.g., compile a five-page procedure) rather than long ones (e.g., sort a million-record file), the computer can provide fast, interactive service to a number of users when the CPU is otherwise idle. The first serious timesharing system, **CTSS (Compatible Time**

Sharing System), was developed at **M.I.T.** However, timesharing did not really become popular until the necessary protection hardware became widespread during the third generation.

After the success of the CTSS system, MIT, Bell Labs, and General Electric (then a major computer manufacturer) decided to embark on the development of a “computer utility,” a machine that would support hundreds of simultaneous timesharing users. Their model was the electricity distribution system. The designers of this system, known as **MULTICS** (**MULTiplexed Information and Computing Service),** envisioned one huge machine providing computing power for everyone in the Boston area. MULTICS was a mixed success. It was designed to support hundreds of users on a machine only slightly more powerful than an Intel 386-based PC, although it had much more I/O capacity.

One of the computer scientists at Bell Labs who had worked on the MULTICS project. Ken Thompson, subsequently found a small PDP-7 minicomputer that no one was using and set out to write a stripped-down, one-user version of MULTICS. This work later developed into the **UNIX®** operating system, which became popular in the academic world, with government agencies, and with many companies. Two major versions developed, **System V**, from AT&T, and **BSD,** (Berkeley Software Distribution) from the University of California at Berkeley. These had minor variants as well. To make it possible to write programs that could run on any UNIX system. IEEE developed a standard for UNIX, called **POSIX** that most versions of UNIX now support. In 1987, the author released a small clone of UNIX, called **MINIX,** for educational purposes. Functionally, MINIX is very similar to UNIX, including POSIX support.

The desire for a free production (as opposed to educational) version of MINIX led a Finnish student, Linus Torvalds, to write **Linux.** This system was developed on MINIX and originally supported various MINIX features (e.g., the MINIX file system).

The Fourth Generation (1980-Present) Personal Computers

With the development of LSI (Large Scale Integration) circuits, chips containing thousands of transistors on a square centimeter of silicon, the age of the personal computer dawned. In 1974, when Intel came out with the 8080, the first general-purpose 8-bit CPU, it wanted an operating system for the 8080, in part to be able to test it. Kildall and a friend first built a controller for the newly-released Shugart Associates 8-inch floppy disk and hooked the floppy disk up to the 8080, thus producing the first microcomputer with a disk. Kildall (owner of Digital Research) then wrote a disk-based operating system called **CP/M (Control Program for Microcomputers)** for it. Many application programs were written to run on CP/M, allowing it to completely dominate the world of micro-computing for about 5 years.

In the early 1980s, IBM designed the IBM PC and looked around for software to run on it. People from IBM contacted Bill Gates to license his BASIC interpreter. They also asked him if he knew of an operating system to run on the PC, Gates suggested that IBM contact Digital Research, then the world’s dominant operating systems company. Making what was surely the worst business decision in recorded history, Kildall refused to meet with IBM, sending a subordinate instead. To make matters worse, his lawyer even refused to sign IBM’s nondisclosure agreement covering the not-yet-announced PC. Consequently, IBM went back to Gates asking if he could provide them with an operating system.

When IBM came back, Gates realized that a local computer manufacturer, Seattle Computer Products, had a suitable operating system. **DOS (Disk Operating System).** He approached them and asked to buy it (allegedly for $50,000). Gates then offered IBM a DOS/BASIC package which IBM accepted. IBM wanted certain modifications, so Gates hired the person who wrote DOS, Tim Paterson, as an employee of Gates’ fledgling company, Microsoft, to make them. The revised system was renamed **MS-DOS (Microsoft Disk Operating System)** and quickly came to dominate the IBM PC market.

At Stanford Research Institute in the 1960s, Engelbart invented the **GUI (Graphical User Interface),** pronounced “gooey,” complete with windows, icons, menus, and mouse. These ideas were adopted by researchers at Xerox PARC and incorporated into machines they built.

One day, Steve Jobs, who co-invented the Apple computer in his garage, visited PARC, saw a GUI, and instantly realized its potential value; Jobs then embarked on building an Apple with a GUI. This project led to the Lisa, which was too expensive and failed commercially. Jobs’ second attempt, the Apple Macintosh, was a huge success, not only because it was much cheaper than the Lisa, but also because it was **user friendly,** meaning that it was intended for users who not only knew nothing about computers but furthermore had absolutely no intention whatsoever of learning.

When Microsoft decided to build a successor to MS-DOS, it was strongly influenced by the success of the Macintosh. It produced a GUI-based system called Windows, which originally ran on top of MS-DOS. However, starting in 1995 a freestanding version of Windows, Windows 95, was released that incorporated many operating system features into it, using the underlying MS-DOS system only for booting and running old MS-DOS programs, in 1998, a slightly modified version of this system, called Windows 98 was released. Nevertheless, both Windows 95 and Windows 98 still contain a large amount of 16-bit Intel assembly language.

Another Microsoft operating system is Windows NT (NT stands for New Technology), which is compatible with Windows 95 at a certain level, but a complete rewrite from scratch internally. It is a full 32-bit system. The lead designer for Windows NT was David Cutler, who was also one of the designers of the VAX VMS operating system, so some ideas from VMS are present in NT. Microsoft expected that the first version of NT would kill off MS-DOS and all other versions of Windows since it was a vastly superior system, but it fizzled. Only with Windows NT 4.0 did it finally catch on in a big way, especially on corporate networks. Version 5 of Windows NT was renamed Windows 2000 in early 1999. It was intended to be the successor to both Windows 98 and Windows NT 4.0. That did not quite work out either, so Microsoft came out with yet another version of Windows 98 called **Windows Me (Millennium edition).**

The other major contender in the personal computer world is UNIX. UNIX is strongest on workstations and other high-end computers, such as network servers. It is especially popular on machines powered by high-performance RISC chips. On Pentium-based computers, Linux is becoming a popular alternative to Windows for students and increasingly many corporate users.

Mainframe Operating Systems

These computers distinguish themselves from personal computers in terms of their I/O capacity. A mainframe with 1000 disks and thousands of gigabytes of data is not unusual: a personal computer with these specifications would be odd indeed. Mainframes are also making something of a comeback as high-end Web servers, servers for large-scale electronic commerce sites, and servers for business-to-business transactions.

The operating systems for mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amounts of I/O. They typically offer three kinds of services: batch, transaction processing, and timesharing. A batch system is one that processes routine jobs without any interactive user present. Claims processing in an insurance company or sales reporting for a chain of stores are typically done in batch mode. Transaction processing systems handle large numbers of small requests, for example, check processing at a bank or airline reservations. Each unit of work is small, but the system must handle hundreds or thousands per second. Timesharing systems allow multiple remote users to run jobs on the computer at once, such as querying a big database. These functions are closely related: mainframe operating systems often perform all of them. An example mainframe operating system is OS/390, a descendant of OS/360.

Server Operating Systems

They run on servers, which are very large personal computers, workstations, or even mainframes. They serve multiple users at once over a network and allow the users to share hardware and software resources. Servers can provide print service, file service, or Web service. Typical server operating systems are UNIX and Windows 2000. Linux is also gaining ground for servers.

Multiprocessor Operating Systems

An increasingly common way to get major-league computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multi-computers, or multiprocessors. They need special operating systems, but often these are variations on the server operating systems, with special features for communication and connectivity.

Personal Computer Operating Systems

Their job is to provide a good interface to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Windows 2000, Windows vista, windows 7, the Macintosh operating system, and Linux. Personal computer operating systems are so widely known that probably little introduction is needed.

Real-Time Operating Systems

These systems are characterized by having time as a key parameter. For example, in industrial process control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time, if a welding robot welds too early or too late, the car will be ruined, a **hard real-time system.**

Another kind of real-time system is a **soft real-time** system, in which missing an occasional deadline is acceptable. Digital audio or multimedia systems fall in this category. VxWorks and QNX are well-known real-time operating systems.

Embedded Operating Systems

A palmtop computer or **PDA (Personal Digital Assistant)** is a small computer that fits in a shirt pocket and performs a small number of functions such as an electronic address book and memo pad. Embedded systems run on the computers that control devices that are not generally thought of as computers, such as TV sets, microwave ovens, and mobile telephones. These often have some characteristics of real-time systems but also have size, memory, and power restrictions that make them special. Examples of such operating systems are PalmOS and Windows CE (Consumer Electronics).

Smart Card Operating Systems

The smallest operating systems run on smart cards, which are credit card-sized devices containing a CPU chip. They have very severe processing power and memory constraints. Some of them can handle only a single function, such as electronic payments, but others can handle multiple functions on the same smart card.

Some smart cards are Java oriented. What this means is that the ROM on the smart card holds an interpreter for the Java Virtual Machine (JVM). Java applets (small programs) are downloaded to the card and are interpreted by the JVM interpreter. Some of these cards can handle multiple Java applets at the same time, leading to multiprogramming and the need to schedule them. Resource management and protection also become an issue when two or more applets are present at the same time. These issues must be handled by the (usually extremely primitive) operating system present on the card.

Functions of operating System

- **O.S. as an extended machine.**
  OS creates higher-level abstraction for programmer (Floppy Disk I/O operation)
  1) Disk contains a collection of the named files
  2) Each file must be opened for READ/WRITE
  3) After READ/WRITE complete close that file
  4) No any detail to deal

  OS shields the programmer from the disk hardware and present a simple file oriented interface. Or it presents the user with the equivalent of an extended machine that is easier to program than the underlying hardware.

- **O.S. as a resource manager**
  What happens if three programs try to print their output on the same printer at the same time?
  What happens if two network users try to update a shared document at same time?
  OS primary function is to manage all pieces of a complex system.
  Its advantages are:
  i) Virtualizes resources so multiple users/applications can share
  ii) Protect applications from one another
  iii) Provide efficient and fair access to resources

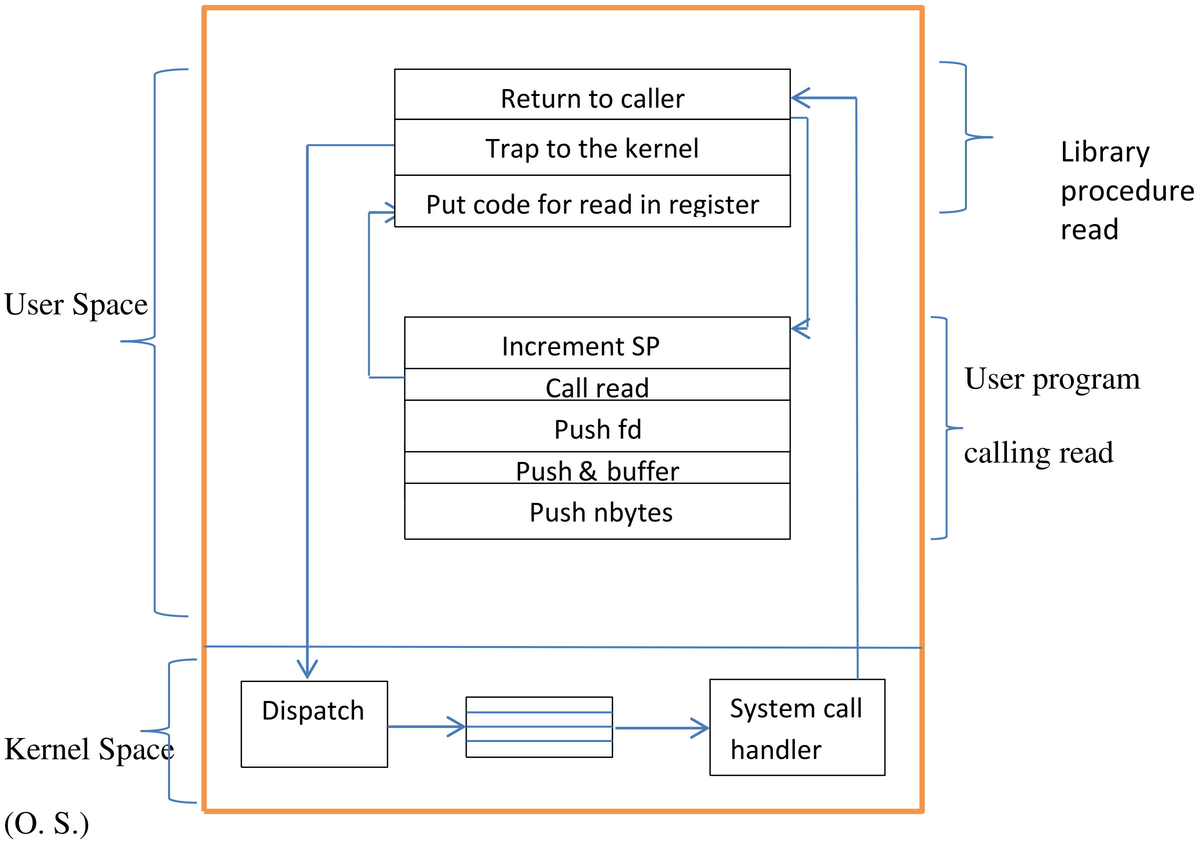
System Calls: The interface between the operating system and the user programs is defined by the set of system calls that the operating system provides. The system calls available in the interface vary from operating system to operating system. Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap or system call instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following the system call. In a sense, making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

To make the system call mechanism clearer, let us take a quick look at the read system call. Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: *read.* A call from a C program might look like this:

```
count = read(fd, buffer, nbytes);
```

Types of System Calls: There are five types of System calls

- Process control (end, abort, load, execute, etc.)
- File management (create file, delete file, close file)
- Device Manager (read, write)
- Information Maintenance (get system date, time)
- Communication (send, receive message)



The Shell

The operating system carries out the system calls. Editors, compilers, assemblers, linkers, and command interpreters definitely are not part of the operating system, even though they are important and useful, called the shell. Shell is not part of the operating system, but makes heavy use of many operating system features and thus serves as a good example of how the system calls can be used. It is also the primary interface between a user and the operating system, unless the user is using a graphical user interface. Examples of some shells are *sh, csh, ksh,* and *bash.*

When any user logs in, a shell is started up. The shell has the terminal as standard input and standard output. It starts out by typing the **prompt,** a character such as a dollar sign, which tells the user that the shell is waiting to accept a command. If the user types

```
date
```

The shell creates a child process and runs the *date* program as the child. When the child finishes, the shell types the prompt again and tries to read the next input line.

The user can specify that standard output be redirected to a file, for example,

```
date >file
```

Similarly, standard input can be redirected, as in

```
sort <file1 >file2
```

This invokes the sort program with input taken from file1 and output sent to file2.

Trap: A trap is a software generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user that an operating system service be provided.

OPERATING SYSTEM STRUCTURE

Many commercial operating systems do not have well defined structures. Such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such system.

In MS-DOS, the interface and the levels of functionality are not well separated. Application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such program leaves MS-DOS vulnerable to malicious programs, causing entire system crashes when user program fails.

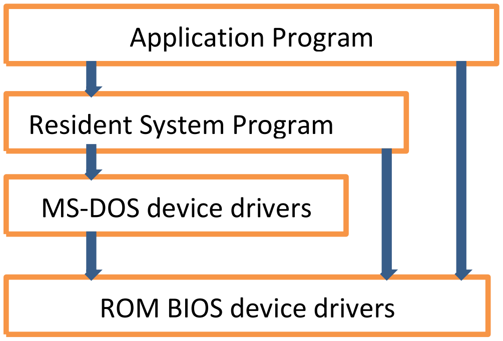


Fig: MS-DOS layer Structure

Monolithic Systems

Operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

Even in monolithic systems, however, it is possible to have *at* least a little structure. The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction. This instruction switches the machine from user mode to kernel mode and transfers control to the operating system. The operating system then fetches the parameters and determines which system call is to be carried out. After that, it indexes into a table that contains in slot *k* a pointer to the procedure that carries out system call *k*.

This organization suggests a basic structure for the operating system:

1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.

In this model, for each system call there is one service procedure that takes care of it. The utility procedures do things that are needed by several service procedures, such as fetching data from user programs. This division of the procedures into three layers is shown in Fig..

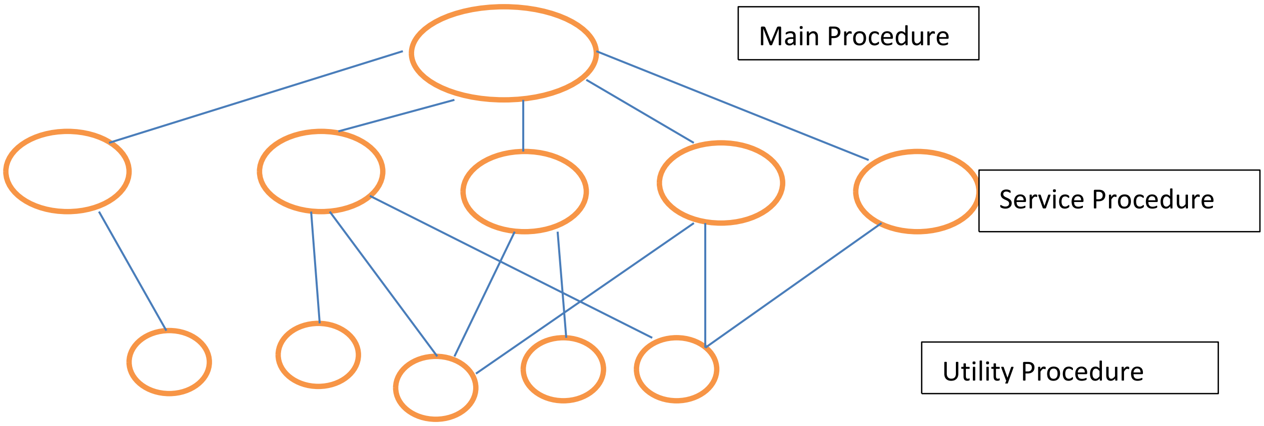


Fig: A simple structuring model for a monolithic system.

Layered Systems

The operating system is broken into a number of layers (levels). The bottom layer (0) is the hardware; the highest layer (n) is the user interface.

The system had 6 layers, as shown in Fig. Layer 0 dealt with allocation of the processor, Layer 1 did the memory management. Layer 2 handled communication between each process and the operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Layer 4 is where the user programs were found. The system operator process is located in layer 5.

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

Structure of the operating system.

The main advantage of layered approach is simplicity of construction and debugging. The first layer can be debugged without any concern for the rest of the system, because it uses only the basic hardware to implement its functions. If error is found in any layer, the error must be on that layer because layers below it are already debugged.

Each layer is implemented with only those operations provided by lower-level layers. Hence, each layer hides the existence of certain data structures, operations and hardware from higher-level layers.

The major difficulty with layered approach involves appropriately defining the various layers. Because a layer can use only lower- level layers. Finally the layered systems are less efficient as compared to non-layered, since some overheads are added at each layer for system calls.

Virtual Machines

The fundamental idea behind the virtual machine is to abstract the hardware of a single computer into several execution environments. i.e. creating an illusion that each separate execution environment is using its own private computer.

The virtual machine provides an interface that is identical to the underlying bare hardware. The heart of the system, known as virtual machine monitor, runs on the bare hardware and does the multiprogramming, not one but several machines to the next layer. However unlike all other operating systems, these virtual machines are not extended machines, with files and other features. Instead they are the exact copies of the bare hardware, including kernel/user mode, I/O interrupts, and everything else the real machines has.

Client server model: As assignment……

Assignment 1: (Deadline 18th January 2013)

1. Prepare a report on the client server Architecture.
2. Compare java virtual machine and VMware.
3. List the essential properties for the Batch-Oriented and Interactive operating system. For each of the following application which system (batch or Interactive) is more suitable? State the reason.

   (a) Word processing

   (b) Generating monthly bank statements

   (c) Computing pi to million decimal places

   (d) A flight simulator

   (e) Generating mark statement by university

4. Prepare a comparative report for windows operating system and Linux operating.
5. What is an operating system? Differentiate between time sharing and real time operating system.

PROCESSES

A process is sequential program in execution. A process defines the fundamental unit of computation for the computer. Components of process are:

1. Object Program
2. Data
3. Resources
4. Status of the process execution

Object program i.e. code to be executed. Data is used for executing the program. While executing the program, it may require some resources. Last component is used for verifying the status of the process execution. A process can run to completion only when all requested resources have been allocated to the process. Two or more processes could be executing the same program, each using their own data and resources.

Modern computers can do several things at the same time. For example while running a user program, a computer can also be playing music and outputting text to a screen or printer. In a multiprogramming system, the CPU switches from program to program, running each for tens or hundreds of milliseconds. Actually the CPU is running only one program, in the course of 1 second; it may work on several programs, thus giving the users the illusion of parallelism, termed as **pseudo parallelism.**

A batch system executes jobs, time shared systems has user programs, or tasks. Even on the systems running Microsoft Windows, a user can be able to run programs at once or a single program. The O.S. need to support its own internal programmed activities, such as memory management. In many aspects, all these activities are similar, so we call all of them *process*.

Processes and Programs

Process is a dynamic entity, which is a program in execution. A process is a sequence of information executions. Process exists in a limited span of time. Two or more processes could be executing the same program, each using their own data and resources.

Program is a static entity made up of program statement. Program contains the instructions. A program exists at single place in space and continues to exist. A program does not perform the action by itself.

The difference between a process and a program is subtle, but crucial.

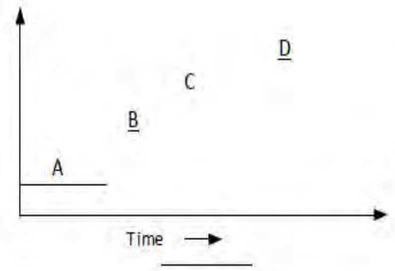
- A scientist’s daughter’s birthday (baking a cake)
- Birthday cake recipe (program) → i.e. an algorithm expressed in some suitable notation.
- Scientist (Processor / CPU)
- Cake indigrients (egg, sugar, flour etc. are inputs)

- Process is the activity consisting of reading the recipe, fetching the indegrients and baking the cake.
- Now imagine, scientist’s son came crying, saying he has been stung by bee.
- Scientist record where he was in the recipe (the stake of current process is recorded).
- The CPU is switched from one process (baking) to higher priority process (medical).
- After treatment, scientist goes back to his cake, continuing at the point where he had left.
- The key idea is that process is an activity of some kind; it has program, input and output.

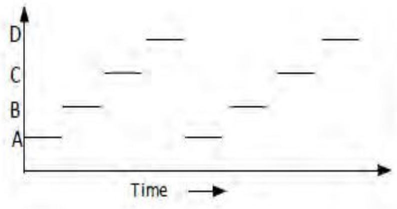
The Process Model

All the runnable software on the computer, sometimes including the operating system, is organized into a number of **sequential processes**, or just processes for short. A process is just an executing program, including the current values of the program counter, registers, and variables. Each process has its own virtual CPU. Basically there are three types of process models

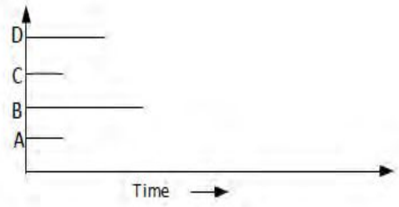
- Uniprogramming
- Multiprogramming
- Multiprocessing



Uniprogramming



Multiprogramming



Multi processing

Process Creation

There are four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

When an operating system is booted, typically several processes are created. Some of these are foreground processes andothers are background processes. For example, one background process may be designed to accept incoming email, sleeping most of the day but suddenly springing to life when email arrives. Processes that stay in the background to handle some activity such as email, Web pages, news, printing, and so on are called **daemons.**

Often a running process will issue system calls to create one or more new processes to help it do its job. For example, if a large amount of data is being fetched over a network for subsequent processing, it may be convenient to create one process to fetch the data and put them in a shared buffer while a second process removes the data items and processes them. On a multiprocessor, allowing each process to run on a different CPU may also make the job go faster.

In interactive systems, users can start a program by typing a command or (double) clicking an icon. Taking either of these actions starts a new process and runs the selected program in it. Using the mouse, the user can select a window and interact with the process, for example, providing input when needed.

The last situation in which processes are created applies only to the batch systems found on large mainframes. Here users can submit batch jobs to the system (possibly remotely). When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

Process Termination

After a process has been created, it starts running and does whatever its job is. Sooner or later the new process will terminate, usually due to one of the following conditions:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Most processes terminate because they have done their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. Screen-oriented programs also support voluntary termination. Word processors, Internet browsers and similar programs always have an icon or menu item that the user can click to tell the process to remove any temporary files it has open and then terminate.

The second reason for termination is that the process discovers a fatal error. For example, if a user types the command

gcchello.c

To compile the program *hello.c*and no such file exists, the compiler simply exits. Screen-oriented interactive processes generally do not exit when given bad parameters. Instead they pop up a dialog box and ask the user to try again.

The third reason for termination is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero.

The fourth reason a process might terminate is that a process executes a system call telling the operating system to kill some other process. The killer must have the necessary authorization to do in the killee. In some systems, when a process terminates, either voluntarily or otherwise, all processes it created are immediately killed as well.

Process States

Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input. Thus a process may be in any of the following states.

1. **New State:** The process being created.
2. **Running State:** A process is said to be running if it has the CPU, i.e. a running process possesses all the resources needed for its execution, including the processor.
3. **Blocked (or waiting) State:** A process is said to be blocked if it is waiting for someevent to happen such that as an I/O completion before it can proceed. Note that a process is unable to run until some external event happens.
4. **Ready State:** A process is said to be ready if it is waiting to be assigned to a processor.
5. **Terminated state:** The process has finished execution.

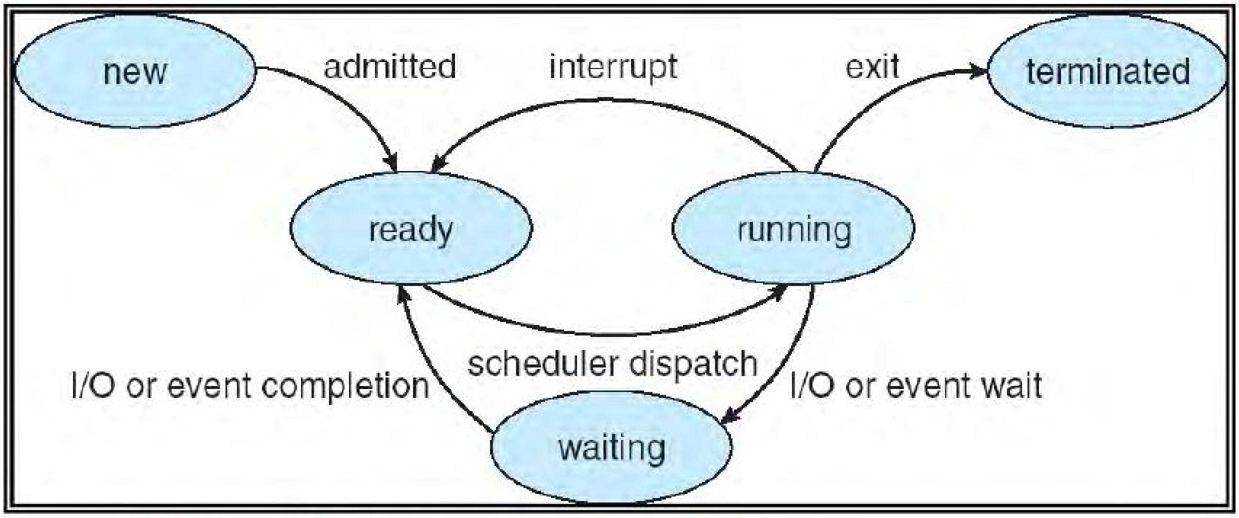


Diagram for Process State

This view gives rise to the model shown below. Here the lowest level of the operating system is the scheduler, with a variety of processes on top of it.

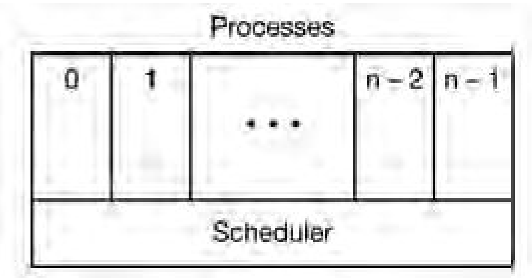


Figure: *The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.*

Suspended Processes

Characteristics of suspend process

1. Suspended process is not immediately available for execution.
2. The process may or may not be waiting on an event.
3. For preventing the execution, process is suspended by OS, parent process, process itself and an agent.
4. Process may not be removed from the suspended state until the agent orders the removal.

Note: Swapping is used to move all of a process from main memory to disk. When all the process by putting it in the suspended state and transferring it to disk.

Reasons for process suspension

1. Swapping
2. Timing
3. Interactive user request
4. Parent process request

Swapping: OS needs to release required main memory to bring in a process that is ready to execute.
Timing: Process may be suspended while waiting for the next time interval.
Interactive user request: Process may be suspended for debugging purpose by user.
Parent process request: To modify the suspended process or to coordinate the activity of various descendants.

Implementation of Processes

To implement the process model, the operating system maintains a table (an array of structures), called the **process table (Process Control Block)**, and also called a **task control block**, withone entry per process. This entry contains information about the process state, its program counter, stack pointer, memory allocation, the status of its open files, its accounting and scheduling information, and everything else about the process that must be saved when the process is switched from *running* to *ready* or *blocked* state so that it can be restarted later as if it had never been stopped.

The PCB is the data structure containing certain important information about the process.

- Pointer: Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
- Process State: Process may be New, Running, Ready, Blocked and so on.
- Program Counter: It indicates the address of next instruction to be for the process.

- Registers: It indicates general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.
- Event information: For a process in the blocked state this field contains information concerning the event for which the process is waiting.
- Scheduling Information: Process priority, pointer to scheduling queue etc.
- Memory Allocation: Value of base and limit register, page table, segment table etc.
- Accounting Information: Time limit, Process number etc.
- Status Information: List of I/O devices, list of open files etc.

Pointer	Process States
Process Number	
Program Counter	
Registers	
Event Information	
Memory limits	
List of Open Files	
. . .	

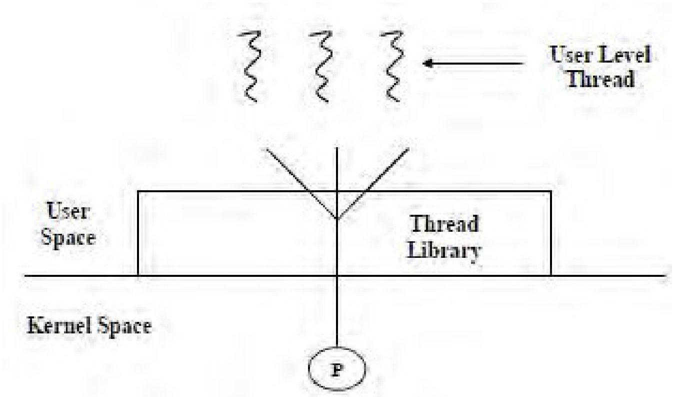
THREADS

Threads like process are a mechanism to allow a program to do more than one thing at a time. Conceptually, a thread (also called lightweight process) exists within a process (heavyweight process). Threads are a finger-grained unit of execution than processes.

- Traditional threads have their own address space and a single thread of control.
- The term multithreading is used to describe the situation of allowing the multiple threads in same process.
- When multithreaded process is run on a single-CPU system, the threads take turns running as in the multiple processes.
- All threads share the same address space, global variables, set of open file, child processes, alarms and signals etc.

Threads are implemented in two ways:

User Level Thread: In a user thread, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application begins with a single thread and begins running in that thread.



User level threads are generally fast to create and manage.

Kernel Level Threads: In Kernel level thread, thread management done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process. The Kernel maintains context information for the process as a whole and for individual threads within the process.
Scheduling by the Kernel is done on a thread basis. The Kernel performs thread creation, scheduling and management in Kernel space. Kernel threads are generally slower to create and manage than the user threads. All contemporary operating systems including window Xp, Linux, etc. support kernel threads.

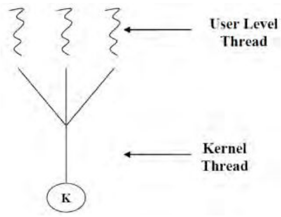
Benefits of threads

1. Responsiveness: Allows program to execute, even if it part of it is blocked or performing a lengthy operation. E.g a multithreaded browser could allow user interaction in one thread while an image is loaded in another thread.
2. Resource Sharing: Threads share resources and memory of the process to which they belong to by default.
3. Economy: Allocating memory and resources to processes creation is costly. Because threads share the resources of the process, it is more economical to create and context switch threads.
4. Scalability: Benefits of multithreading is greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processor. Multithreading in multiprocessor machine increases parallelism.

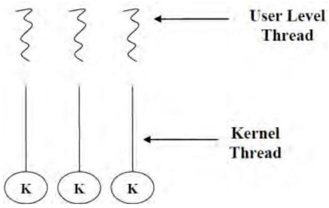
The Thread Model

Some operating systems provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and some relationship must exist between them.

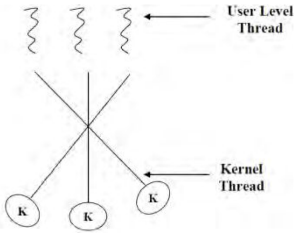
Multithreading models are of three types:



Many to One model: This model maps many user level threads to one kernel level thread. Thread management is done by thread library in user space, so it is efficient but the entire process is blocked if a thread makesblocking system call.



One to One model: One user thread is mapped to one kernel thread. It provides more concurrency than previous model by allowing another thread to run during blocking. It also allows parallelism. The only overhead is for each threads corresponding kernel thread should be created.



Many to Many Model: In this model, many user level threads multiplex to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine.

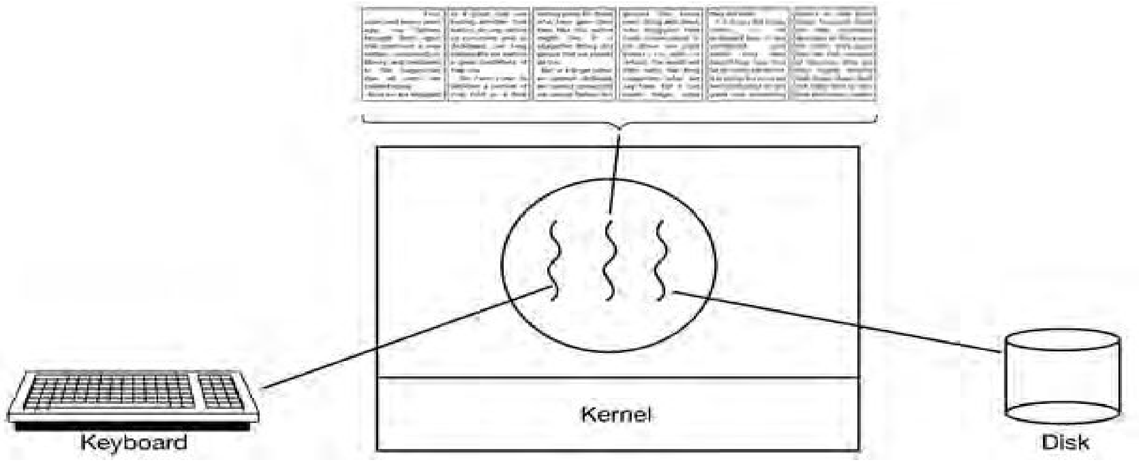
Thread Usage

For the systems running multiple activities at once, some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

Since threads do not have any resources attached to them, they are easier to create and destroy than processes. Threads yield no performance gain when all of them are CPU bound, but when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application. Threads are useful on systems with multiple CPUs, where real parallelism is possible.

It is probably easiest to see why threads are useful by giving some concrete examples. As a first example, consider a word processor. Most word processors display the document being created on the screen formatted exactly as it will appear on the printed page. In particular, all the line breaks and page breaks are in their correct and final position so the user can inspect them and change the document if need be (e.g., to eliminate widows and orphans—incomplete top and bottom lines on a page, which are considered esthetically unpleasing).

Suppose that the word processor is written as a two-threaded program. One thread interacts with the user and the other handles reformatting in the background. As soon as the sentence is deleted from page 1 the interactive thread tells the reformatting thread to reformat the whole book. Meanwhile, the interactive thread continues to listen to the keyboard and mouse and responds to simple commands like scrolling page 1 while the other thread is computing madly in the background. Many word processors have a feature of automatically saving the entire file to disk every few minutes to protect the user against losing a day's work in the event of a program crash, system crash, or power failure. The third thread can handle the disk backups without interfering with the other two.



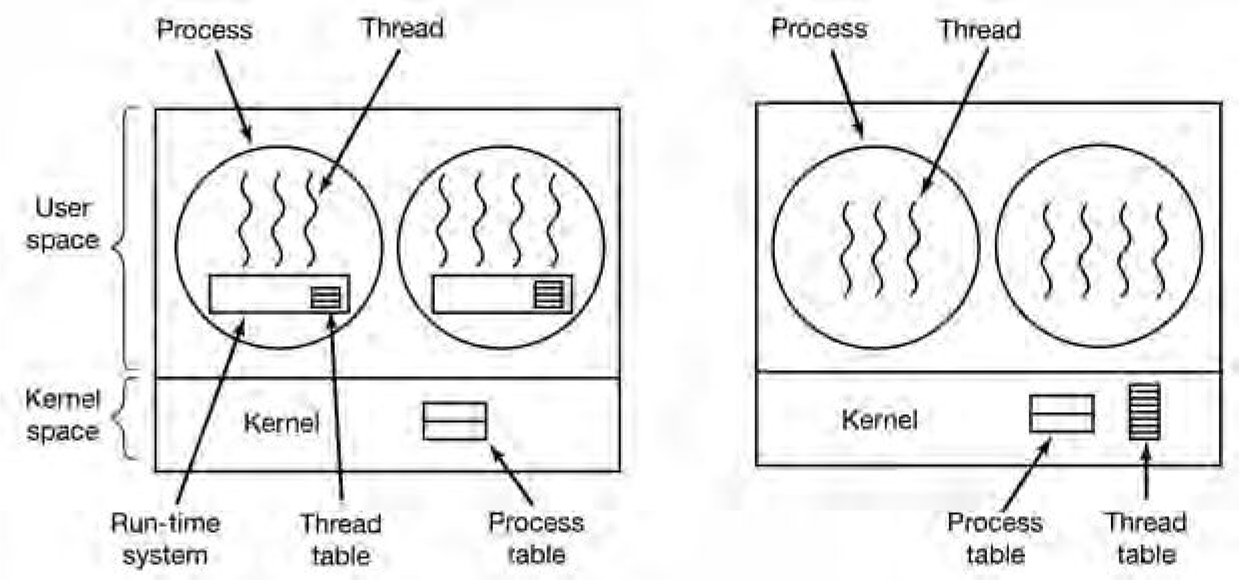
A word processor with three threads.

If the program were single-threaded, then whenever a disk backup started, commands from the keyboard and mouse would be ignored until the backup was finished.

Implementing Threads in User Space

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. All operating systems used to fall into this category, and even now some still do.

The threads run on top of a run-time system, which is a collection of procedures that manage threads. When threads are managed in user space, each process needs its own private **thread table**. It is analogous to the kernel's process table, except that it keeps track only of the per-thread properties such the each thread's program counter stack pointer, registers, state, etc. The thread table is managed by the runtime system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as the kernel stores information about processes in the process table.

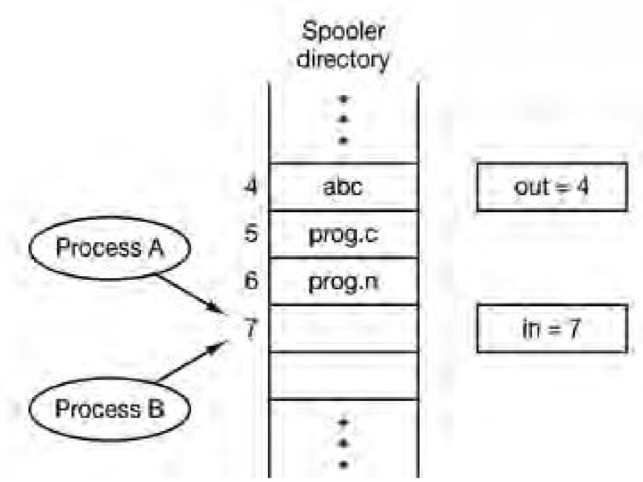


(a) A user-level threads package. (b) A threads package managed by the kernel.

Thread switching like this is at least an order of magnitude faster than trapping to the kernel and is a strong argument in favor of user-level threads packages.When a thread is finished running for the moment, for example, when it calls *thread_yield,* the code of *thread_yield*can save the thread's information in the thread table itself. Furthermore, it can then call the thread scheduler to pick another thread to run. It is all done by local procedures, so invoking them is much more efficient than making a kernel call. Among other issues, no trap is needed, no context switch is needed, and the memory cache need not be flushed, and so on. This makes thread scheduling very fast.

INTERPROCESS COMMUNICATION

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus there is a need for communication between processes, preferably in a well-structured way not using interrupts. Very briefly, there are three issues here.



1. How one process can pass information to another?
2. Making sure two or more processes do not get into each other's way when engaging in critical activities.
3. Proper sequencing when dependencies are present.

Race Conditions

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.
Suppose that two processes, A and B, share the global variable '*a*'. At some point in its execution, A updates '*a*'to the value 1, and at some point in its execution, B updates '*a*' to the value 2. Thus, the two tasks are in a race to write variable '*a*'. In this example the "loser" of the race (the process that updates last) determines the final value of '*a*'.
Therefore Operating System Concerns of following things

1. The operating system must be able to keep track of the various processes
2. The operating system must allocate and de-allocate various resources for each active process.
3. The operating system must protect the data and physical resources of each process against unintended interference by other processes.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

Process Interaction can be defined as

- Processes unaware of each other
- Processes indirectly aware of each other
- Processes directly aware of each other

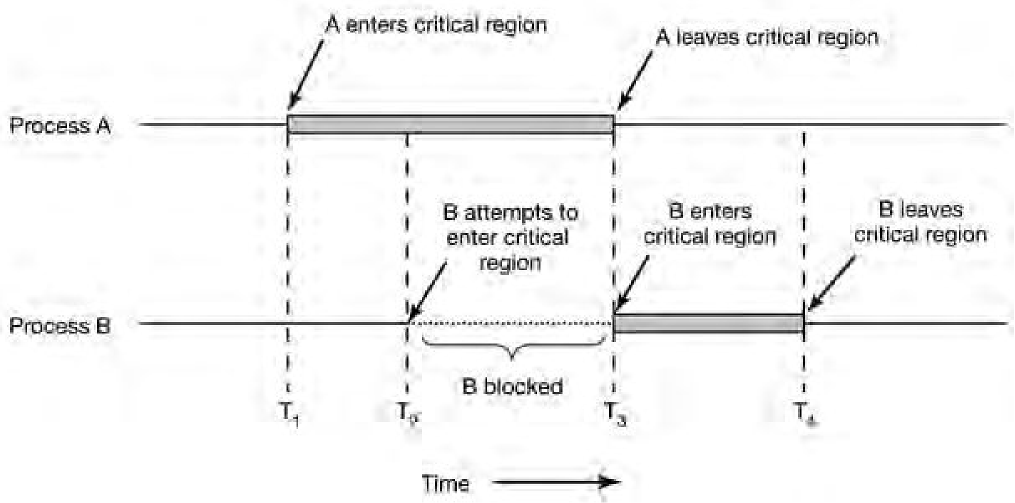
Concurrent processes come into conflict with each other when they are competing for the use of the same resource.
Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of the other processes. There is no exchange of information between the competing processes.

Critical Regions

How do we avoid race conditions? One way to prevent two or more process, using the shared data is mutual exclusion. i.e. some way of making sure that if one process is using a shared variable or file, the process will be excluded from doing same thing. Sometimes a process has to access shared memory or files that can lead to races. That part of the program where the shared memory is accessed is called the **critical region** or **critical section**. If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races.

Four conditions to have a good solution:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.



Mutual exclusion using Critical Region

Process*A* enters its critical region at time T_1, A little later, at time T_2 process *B* attempts to enter its critical region but fails because another process is already in its critical region and we allow only one at a time. Consequently, *B* is temporarily suspended until time T_3 when *A* leaves its critical region, allowing *B* to enter immediately. Eventually *B* leaves (at T_4) and we are back to the original situation with no processes in their critical regions.

Mutual Exclusion with Busy Waiting

It can be achieved by any of the following methods:

Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it and never turned them on again? That could be the end of the system.

Lock Variables

It is a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock.

1. If lock = 0, the process sets it to 1 and enters the critical region.
2. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region.

Problem: Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

Strict Alternation

In this method, process shares the common integer variable turn. If turn=0, then process P_0 is allowed to execute in its CR. If turn=1, then process P_1 is allowed to execute in its CR.

Initially P_0inspects turn, find it to be zero and enters its CR. P1 also finds it zero and therefore sits in a tight loop continuously testing the turn to become 1.

```
while (TRUE) {                       while (TRUE) {
    while (turn != 0)   /* loop */ ;     while (turn != 1);  /* loop */ ;
    critical_region();                   critical_region();
    turn = 1;                            turn = 0;
    noncritical_region();                noncritical_region();
}                                    }
```

(a) (b)

A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**.

Peterson's Solution

By combining the idea of taking turns with the idea of lock variables and warning variables, a Dutch mathematician, T. Dekker, was the first one to devise a software solution to the mutual exclusion problem that does not require strict alternation.

```
#define FALSE 0
#define TRUE  1
#define N     2      /* number of processes */

int turn;           /* whose turn is it? */
int interested[N];  /* all values initially 0 (FALSE) */

voidenter_region(int process)      /* process is 0 or 1 */
{
int other;                      /* number of the other process */

other = 1 - process;            /* the opposite of process */
interested[process] = TRUE;     /* show that you are interested */
turn = process;                 /* set flag */
while (turn == process && interested[other] == TRUE) /* null statement */;
}
```

```
voidleave_region (int process)     /* process, who is leaving */
{
interested[process] = FALSE; // indicate departure from critical region
}
```

Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls *enter_region.* It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region*returns immediately. If process 1 now calls *enter_region,* it will hang there until *interested*[0] goes to *FALSE*, an event that only happens when process 0 calls *leave_region*to exit the critical region.

Problem: Difficult to program for n processes, may lead to starvation.

The TSL Instruction

Now let us look at a proposal that requires a little help from the hardware. Many computers, especially those designed with multiple processors in mind, have an instruction

TSL RX,LOCK

(Test and Set Lock) that works as follows. It reads the contents of the memory word *lock* into register RX and then stores a nonzero value at the memory address *lock.* The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

To use the TSL instruction, we will use a shared variable, *lock*, to coordinate access to shared memory. When *lock* is 0, any process may set it to 1 using the TSL instruction and then read or write the shared memory. When it is done, the process sets *lock* back to 0 using an ordinary move instruction.

```
enter_region:
    TSL REGISTER,LOCK    | copy lock to register and set lock to 1
    CMP REGISTER,#0      | was lock zero?
    JNE enter_region     | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0         | store a 0 in lock
    RET | return to caller
```

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls *enter_region,* which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls *leave_region*, which stores a 0 in *lock*. As with all solutions based on critical regions, the processes must call *enter_region*and *leave_region*at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.

Alternative to busy waiting

Busy wait causes wastage of CPU time.

Sleep and Wakeup

Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring busy waiting.

Sleep is the system call that causes the caller to block, that is suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

Consider a computer with two processes, *H*, with high priority and *L*, with low priority. The scheduling rules are such that *H* is run whenever it is in ready state. At a certain moment, with *L* in its critical region, *H* becomes ready to run (e.g., an I/O operation completes). *H* now begins busy waiting, but since *L* is never scheduled while *H* is running, *L* never gets the chance to leave its critical region, so *H* loops forever. This situation is sometimes referred to as the **priority inversion problem.**

The Producer-Consumer Problem

As an example of sleep and wakeup, the **producer-consumer** problem (also known as the **bounded-buffer** problem). Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.

```
#define N 100       /* number of slots in the buffer */
int count = 0;      /* number of items in the buffer */

void producer (void)
{
int item;

while (TRUE) {                 /* repeat forever */
item = produce_item();     /* generate next item */
if (count == N) sleep();   /* if buffer is full, go to sleep */
insert_item(item);         /* put item in buffer */
count = count + 1;         /* increment count of items in buffer */
if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}

void consumer(void)
{
int item;

while (TRUE) {                 /* repeat forever */
if (count == 0) sleep();   /* if buffer is empty, got to sleep */
item = remove_item();      /* take item out of buffer */
count = count - 1;          /* decrement count of items in buffer */
if (count == N - 1) wakeup(producer); /* was buffer full? */
consume_item(item);         /* print item */
    }
}
```

The producer-consumer problem with a fatal race condition.

Problem: Leads to race condition as is spooler directory. Possibility of loss of wake up signals.

Semaphores

This was the situation in 1965, when E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use.

A semaphore S is an integer variable that, a part from initialization, is an accessed two standard atomic operations: wait and signal. These operations were originally termed P (for wait; from the Dutch proberen, to test) and V (for signal; from verhogen, to increment).
The Classical definition of wait and signal are

```
Wait (S)
{
if (S>0)
S =S - -;
}
Signal(S)
{
S = S + +;
}
```

Semaphores are not provided by hardware. But they have several attractive properties:

1. Semaphores are machine independent.
2. Semaphores are simple to implement.
3. Correctness is easy to determine.
4. Can have many different critical sections with different semaphores.
5. Semaphore acquires many resources simultaneously.

Use of Semaphores

1. For achieving mutual exclusion
2. To solve the synchronization problems

For example two concurrently running process, P1 with statement S1 and P2 with statement S2. Suppose it requires that S2 must be executed after S1 has completed. The problem can be implemented by using a common semaphore.

```
P1:S1;
Wait (synch);
Signal (synch);
S2;
```

Drawback of Semaphore
1. They are essentially shared global variables.
2. Access to semaphores can come from anywhere in a program.
3. There is no control or guarantee of proper usage.
4. There is no linguistic connection between the semaphore and the data to which the semaphore controls access.
5. They serve two purposes, mutual exclusion and scheduling constraints.

```
#define N 100              /* number of slots in the buffer */

typedefint semaphore;     /* semaphores are a special kind of int */
semaphoremutex = 1;       /* controls access to critical region */
semaphore empty = N;       /* counts empty buffer slots */
semaphore full = 0;        /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {             /* TRUE is the constant 1 */
        item = produce_item(); /* generate something to put in buffer */
        down(&empty);          /* decrement empty count */
        down(&mutex);          /* enter critical region */
        insert_item(item);     /* put new item in buffer */
        up(&mutex);            /* leave critical region */
        up(&full);             /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {             /* infinite loop */
        down(&full);           /* decrement full count */
        down(&mutex);          /* enter critical region */
        item a= remove_item(); /* take item from buffer */
        up(&mutex);            /* leave critical region */
        up(&empty);            /* increment count of empty slots */
        consume_item(item);    /* do something with the item */
    }
}
```

The producer-consumer problem using semaphores.

Monitors

A monitor is a collection of procedures, variables and data structures that are all grouped together in a special kind of module or package. Process may call the monitors whenever they need, but they cannot access the internal structure of monitor directly.

Only one process can be active in a monitor at an instant. When a process calls a monitor’s procedure, first few instruction of procedure will check to see if any process is concurrently active within or not, if yes the calling process is suspended for a certain time else get access. It use wait and signal operations.

```
Monitor Monitor_name
{
Shared variable declerations;
Procedure P1 ( );
Procedure P2 ( );
      -
      -
Procedure Pn ( );
}
```

Bounded Buffer Using a Monitor

```
item buffer[N]; /* buffer with capacity N */
int count; /* initially 0 */
cv *notfull, *notempty; /* Condition variables */
Produce(item)
{
while (count == N)
{
wait(notfull);
}
add item to buffer
count = count + 1;
signal(notempty);
} Produce  is implicitly executed atomically, because it is a monitor method.
Consume(item) {
while (count == 0) {
wait(notempty);
}
remove item from buffer
count = count - 1;
signal(notfull);

}
```

Message Passing

It is the mechanism for processes to communicate and to synchronize their actions. Message system – processes communicate with each other without resorting to shared variables. This method of interprocess communication uses two primitives, send and receive.

```
send(destination, &message);
and
receive(source, &message);
```

- If P and Q wish to communicate, they need to:
- establish a communication link between them

- exchange messages via send/receive
- Implementation of communication link
- physical (e.g., shared memory, hardware bus)
- logical (e.g., logical properties)

Direct Communication

With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the **send** and **receive** primitives are defined as:

- Send (P, message) – Send a message to process P.
- Receive (Q, message) – Receive a message from process Q.

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. This scheme exhibits symmetry in addressing; that is, both the sender and the receiver processes must name the other to communicate. A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the **send** and **receive** primitives are defined as follows:

- Send (P, message) – Send a message to process P.
- Receive (id, message) – Receive a message form any process; the variable id is set to the name of the process with which communication has taken place.

The disadvantage in both symmetric and asymmetric schemes is the limited modularity of the resulting process definitions. Changing the name of a process may necessitate examining all other process definitions. All references to the old name must be found, so that they can be modified to the new name. This situation is not desirable from the viewpoint of separate compilation.

Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. In this scheme, a process can communicate with some other process via a number of different mailboxes. Two processes can communicate only if they share a mailbox. The **send** and **receive** primitives are defined as follows:

- Send (A, message) - Send a message to mailbox A.
- Receive (A, message) - Receive message from mailbox A.

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

A link is established between a pair of processes only if both members of the pair have a shared mailbox. A link may be associated with more than two processes. A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Now suppose that processes PI, P2, and P3 all share mailbox ***A.*** Process P1 sends a message to A, while P2 and P3 each execute a **receive** from A. Which process will receive the message sent by *P1?* The answer depends on the scheme that we choose: Allow a link to be associated with at most two processes. Allow at most one process at a time to execute a **receive** operation. Allow the system to select arbitrarily which process will receive the message (that is, either P2 or P3, but not both, will receive the message). The system may identify the receiver to the sender. A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). Since each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists. On the other hand, a mailbox owned by the operating system is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following: Create a new mailbox. Send and receive messages through the mailbox. Delete a mailbox. The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receive privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

The Producer-Consumer Problem with Message Passing

Now let us see how the producer-consumer problem can be solved with message passing. Assume that all messages are the same size and that messages sent but not yet received are buffered automatically by the operating system. In this solution, a total of *N* messages is used, analogous to the *N* slots in a shared memory buffer. The consumer starts out by sending *N* empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer: the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up: the consumer will be blocked, waiting for a full message.

```
#define N 100     /* number of slots in the buffer */
void producer(void)
{
int item;
message m;    /* message buffer */

while (TRUE) {
item = produce_item( );        /* generate something to put in buffer */
receive(consumer, &m);         /* wait for an empty to arrive */
build_message (&m, item);      /* construct a message to send */
send(consumer, &m);            /* send item to consumer */
    }
}
```

```
void consumer(void) {
int item, i;
message m;

for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
while (TRUE) {
receive(producer, &m);          /* get message containing item */
item = extract_item(&m);        /* extract item from message */
send(producer, &m);             /* send back empty reply */
consume_item(tem);              /* do something with the item */
    }
}
```

The producer-consumer problem with N messages.

Synchronization:
Communication between processes takes place by calls to send and receive primitives. There are different design options for implementing each primitive. Message passing may be either blocking or nonblocking-also known as synchronous and asynchronous.

1. **Blocking send:** The sending process is blocked until the message is received by the receiving process or by the mailbox.
2. **Nonblocking send:** The sending process sends the message and resumes operation. Blocking receive: The receiver blocks until a message is available.
3. **Nonblocking receive:** The receiver retrieves either a valid message or a null.

Different combinations of send and receive are possible. When both the send and receive are blocking, we have a rendezvous between the sender and the receiver.

Buffering: Whether the communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such a queue can be implemented in three ways:

1. **Zero capacity**: The queue has maximum length 0; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
2. **Bounded capacity**: The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the latter is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link has a finite capacity, however. If the link is full, the sender must block until space is available in the queue.
3. **Unbounded capacity**: The queue has potentially infinite length; thus, any number of messages can wait in it. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering; the other cases are referred to as automatic buffering.

CLASSICAL IPC PROBLEMS

The operating systems literature is full of interesting problems that have been widely discussed and analyzed using a variety of synchronization methods. In the following sections we will examine three of the better-known problems.

The Dining Philosophers Problem

In 1965, Dijkstra posed and solved a synchronization problem he called the **dining philosophers problem**. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosophers' problem.

Scenario: consider the five philosophers are seated in a common round table for their lunch, each philosopher has a plate of spaghetti and there is a fork between two plates, a philosopher needs two forks to eat it. They alternate thinking and eating.

What is the solution (program) for each philosopher that does what is supposed to do and never got stuck?

Solution

Attempt 1: When the philosopher is hungry it picks up a fork and waits for another fork, when get, it eats for a while and put both forks back to the table.

Problem: What happen, if all five philosophers take their left fork simultaneously?

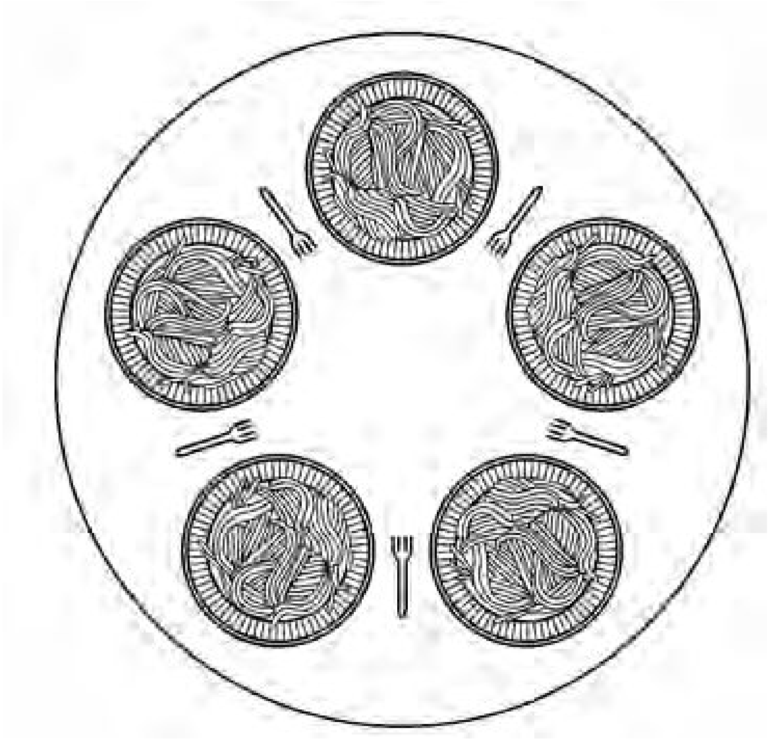


Fig: Lunch time in the Philosophy Department.

Attempt 2: After taking the fork, checks for right fork. If it is not available, the philosopher puts down the left one, waits for some time, and then repeats the whole process.

Problem: What happen, if all five philosophers take their left fork simultaneously?

Attempt 3: Using semaphore, before starting to acquire a fork he would do a down on mutex, after replacing the forks, he would do up on mutex.

Problem: adequate but not perfect: only one philosopher can be eating at any instant.

Attempt 4: Using semaphore for each philosopher, a philosopher move only in eating state if neither neighbor is eating. -perfect solution.

```
#define N          5   /* number of philosophers */

#define LEFT            (i+N-1)%N /* number of i's left neighbor */
#define RIGHT           (i+1)%N /* number of i's right neighbor */
#define THINKING        0   /* philosopher is thinking */
#define HUNGRY          1   /* philosopher is trying to get forks */
#define EATING          2   /* philosopher is eating */
Typedef int semaphore;      /* semaphores are a special kind of int */
int state[N];               /* array to keep track of everyone's state */
semaphore mutex = 1;        /* mutual exclusion for critical regions */
semaphore s[N];             /* one semaphore per philosopher */

void philosopher (int i)    /* i: philosopher number, from 0 to N-1 */
{
while (TRUE) {          /* repeat forever */
think();            /* philosopher is thinking */
take_forks(i);      /* acquire two forks or block */
eat();              /* yum-yum, spaghetti */
put_forks(i);       /* put both forks back on table */
    }
}

voidtake_forks(int i)      /* i: philosopher number, from 0 to N-1 */
{
down(&mutex);           /* enter critical region */
state[i] = HUNGRY;      /* record fact that philosopher i is hungry */
test(i);                /* try to acquire 2 forks */
up(&mutex);             /* exit critical region */
down(&s[i]);            /* block if forks were not acquired */
}

voidput_forks(i)           /* i: philosopher number, from 0 to N-1 */
{
down(&mutex);           /* enter critical region */
state[i] = THINKING;    /* philosopher has finished eating */
test(LEFT);             /* see if left neighbor can now eat */
test(RIGHT);            /* see if right neighbor can now eat */
up(&mutex);             /* exit critical region */
}

void test(i)                /* i: philosopher number, from 0 to N-1 */
{
  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
{
state[i] = EATING;
up(&s[i]);
  }
}
```

A solution to the dining philosophers' problem.

Self study: Readers and writer, sleeping barber and Cigarette-smokers.

SCHEDULING

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

Basic Concepts

The idea of multiprogramming is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then just sit idle. Scheduling is a fundamental operating-system function.
Almost all computer resources are scheduled before use.

CPU - I/O Burst Cycle

The success of CPU scheduling depends on the following observed property of processes: Process execution consists of a cycle of CPU execution and I/O wait. Processes alternate back and forth between these two states.

Context Switch

To give each process on a multiprogrammed machine a fair share of the CPU, a hardware clock generates interrupts periodically. This allows the operating system to schedule all processes in main memory (using scheduling algorithm) to run on the CPU at equal intervals. Each switch of the CPU from one process to another is called a context switch.

Non-preemptive scheduling: Picks a process to run until it releases the CPU.

- Once a process has been given the CPU, it runs until blocks for I/O or termination.
- Treatment of all processes is fair.
- Response times are more predictable.
- Useful in real-time system.
- Shorts jobs are made to wait by longer jobs -no priority

Preemptive Scheduling: Picks a process and let it run for a maximum of fixed time and releases the CPU after that quantum, whether it finishes or not.

- Processes are allowed to run for a maximum of some fixed time.
- Useful in systems in which high-priority processes requires rapid attention.
- In time sharing systems, preemptive scheduling is important in guaranteeing acceptable response times.
- High overhead.

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

1. When a process switches from the running state to the waiting state (for. example, I/O request, or invocation of wait for the termination of one of the child processes).
2. When a process switches from the running state to the ready state (for example, when an interrupt occurs).
3. When a process switches from the waiting state to the ready state (for example, completion of I/O).
4. When a process terminates.

Scheduling Criteria

- Different CPU scheduling algorithms have different properties and may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
- Many criteria have been suggested for comparing CPU scheduling algorithms.

Criteria that are used include the following:

1. CPU utilization – keep the CPU as busy as possible
2. Throughput – # of processes that complete their execution per time unit
3. Turnaround time – amount of time to execute a particular process
4. Waiting time – amount of time a process has been waiting in the ready queue
5. Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

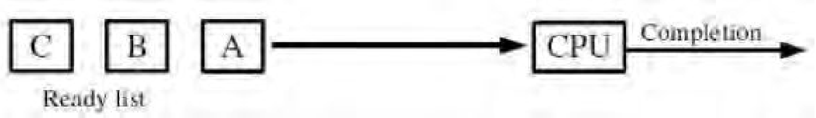
What we need? (Goals of scheduling)

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Scheduling in Batch Systems

First-Come First-Served (Non-preemptive)

- Processes are scheduled in the order they are received.
- Once the process has the CPU, it runs to completion
- Easily implemented, by managing a simple queue or by storing time the process was received.
- Fair to all processes.



Problems:

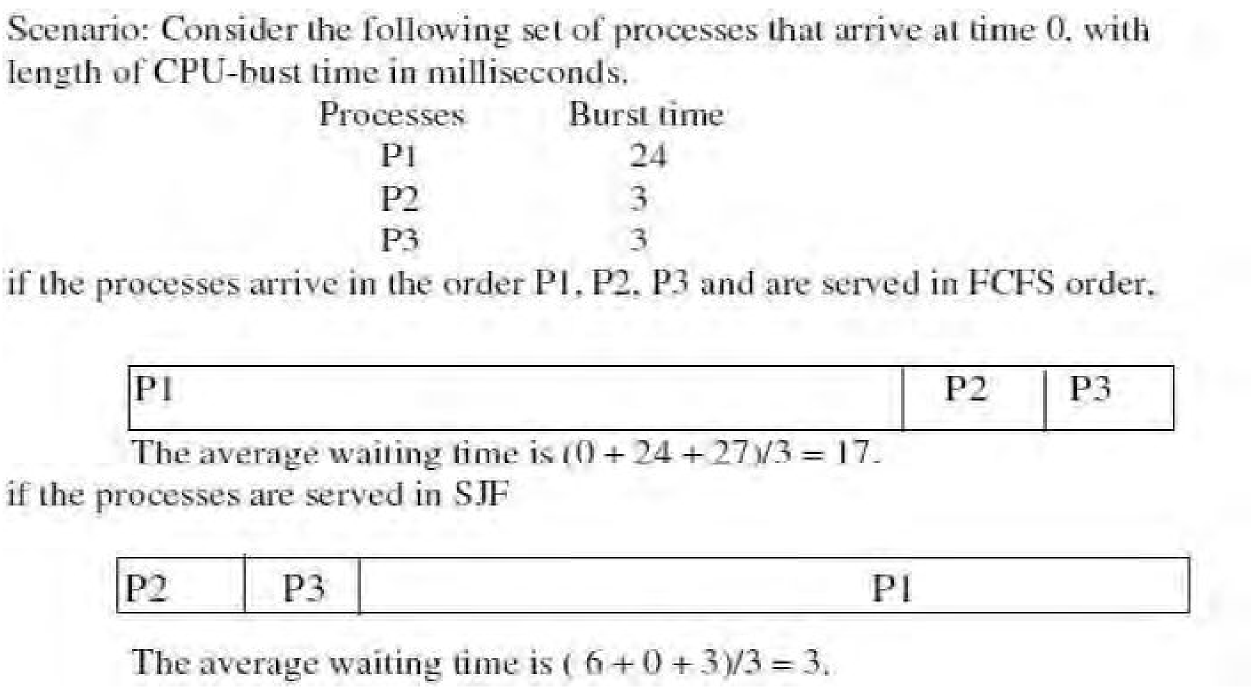
- No guarantee of good response time.
- Large average waiting time.

Shortest Job First (non preemptive)

- The processing times are known in advanced.
- SJF selects the process with shortest expected processing time. In case of the tie FCFS scheduling is used.
- The decision policies are based on the CPU burst time. Advantages:
- Reduces the average waiting time over FCFS.
- Favors shorts jobs at the cost of long jobs.

Problems:

- Estimation of run time to completion. Accuracy?
- Not applicable in timesharing system.



Scenario: Consider the following set of processes that arrive at time 0. with length of CPU-bust time in milliseconds.

Processes	Burst time
P1	24
P2	3
P3	3

if the processes arrive in the order P1, P2, P3 and are served in FCFS order.

P1	P2	P3

The average waiting time is (0 + 24 + 27)/3 = 17.

if the processes are served in SJF

P2	P3	P1

The average waiting time is (6 + 0 + 3)/3 = 3.

Shortest Remaining Time Next

Preemptive version of SJF.

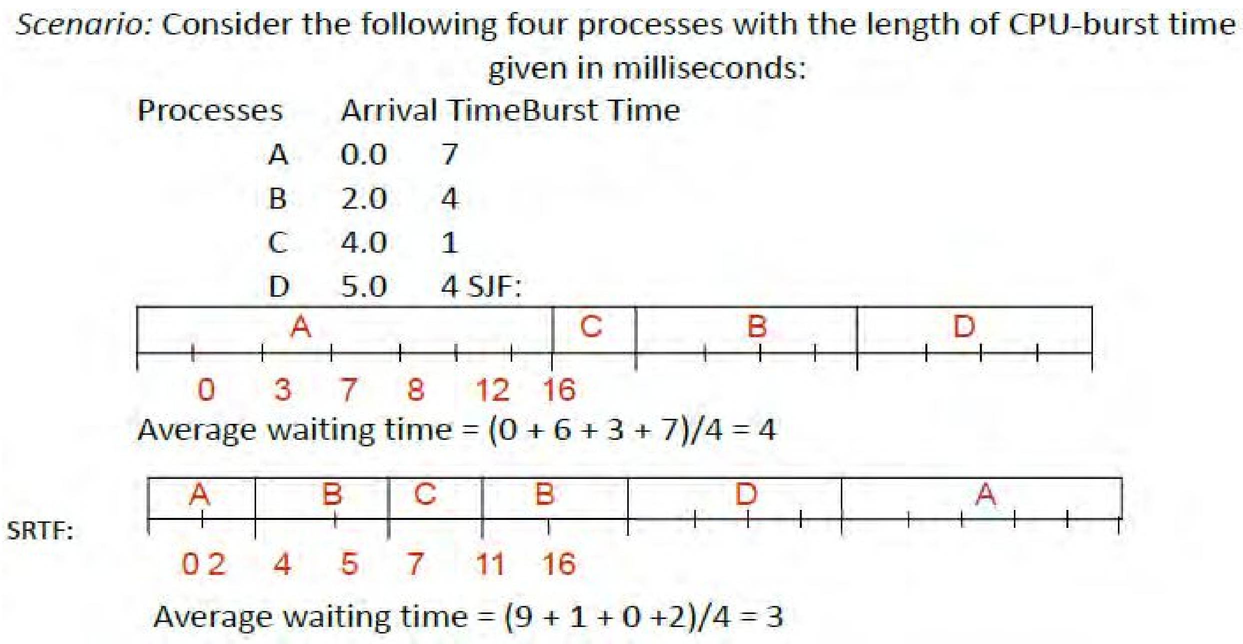
Any time a new process enters the pool of processes to be scheduled, the scheduler compares the expected value for its remaining processing time with that of the process currently scheduled. If the new process's time is less, the currently scheduled process is preempted.

Merits:

- Low average waiting time than SJF.
- Useful in timesharing.

Demerits:

- Very high overhead than SJF.
- Requires additional computation.
- Favors short jobs, longs jobs can be victims of starvation.



Scenario: Consider the following four processes with the length of CPU-burst time given in milliseconds:

Processes	Arrival Time	Burst Time
A	0.0	7
B	2.0	4
C	4.0	1
D	5.0	4 SJF:

Average waiting time = (0 + 6 + 3 + 7)/4 = 4

SRTF:

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

Scheduling in Interactive Systems

Round-Robin Scheduling

Each process is assigned a time interval (quantum), after the specified quantum, the running process is preempted and a new process is allowed to run.

Preempted process is placed at the back of the ready list.

Advantages:

- Fair allocation of CPU across the process.
- Used in timesharing system.
- Low average waiting time when process lengths vary widely.
- Poor average waiting time when process lengths are identical.

Imagine 10 processes each requiring 10 msec burst time and 1msec quantum is assigned.

RR: All complete after about 100 times.
FCFS is better! (About 20% time wastages in context-switching).
Performance depends on quantum size.

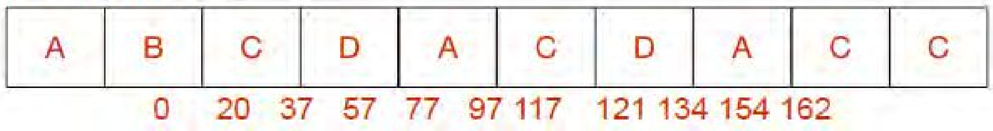
Quantum size: If the quantum is very large, each process is given as much time as needs for completion; RR degenerate to FCFS policy.
If quantum is very small, system busy at just switching from one process to another process, the overhead of context-switching causes the system efficiency degrading.

Optimal quantum size?

Key idea: 80% of the CPU bursts should be shorter than the quantum. 20-50 msec reasonable for many general processes.

Process	Burst Time
A	53
B	17
C	68
D	24

- The Gantt chart is:



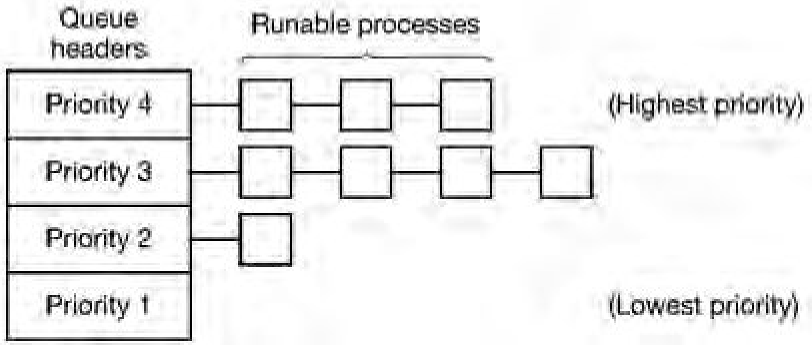
- Typically, higher average turnaround than SJF, but better *response*.

Example of round robin with quantum =20

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ° highest priority)
- SJF is a priority scheduling where priority is the predicted next CPU burst time

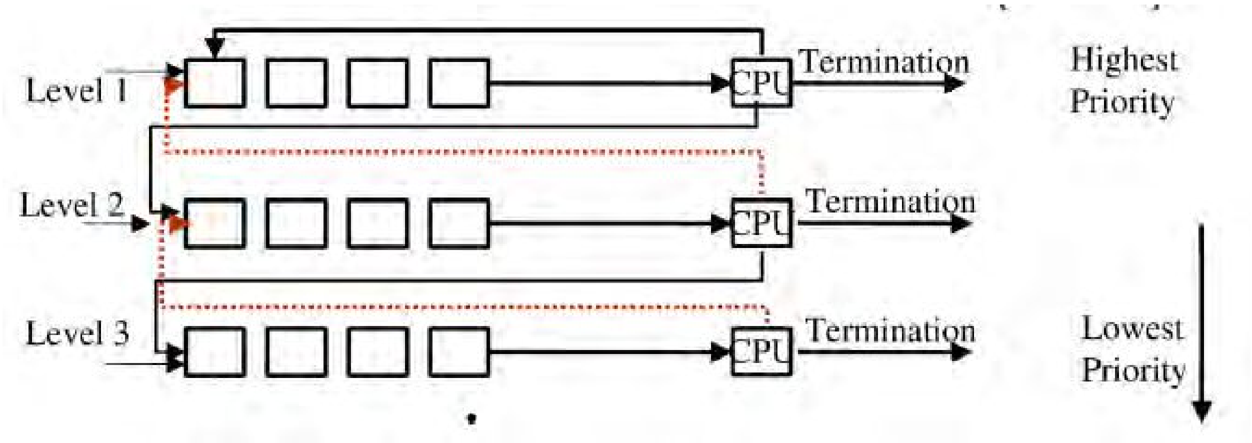
Problem: Starvation – low priority processes may never execute
Solution: Aging – as time progresses increase the priority of the process



A scheduling algorithm with four priority classes.

Multiple Queues

MFQ implements multilevel queues having different priority to each level (here lower level higher priority), and allows a process to move between the queues. If the process uses too much CPU time, it will be moved to a lower-priority queue. Each lower-priority queue larger the quantum size.
This leaves the I/O-bound and interactive processes in the high priority queue.



Consider a MFQ scheduler with three queues numbered 1 to 3, with quantum size 8, 16 and 32msec respectively.
The scheduler execute all process in queue 1, only when queue 1 is empty it execute process in queue 2 and process in queue 3 will execute only if queue 1 and queue 2 are empty.
A process first enters in queue 1 and execute for 8 msec. If it does not finish, it moves to the tail of queue 2.
If queue 1 is empty the processes of queue 2 start to execute in FCFS manner with 16msec quantum. If it still does not complete, it is preempted and move to the queue 3.
If the process blocks before using its entire quantum, it is moved to the next higher level queue.

Assignment 2: Deadline 17th Feb 2013

1. What are disadvantages of too much multiprogramming?
2. For each of the following transitions between the processes states, indicate whether the transition is possible. If it is possible, give an example of one thing that would cause it.
    a) Running -> Ready
    b) Running -> Blocked
    c) Blocked -> Running
3. Describe how multithreading improve performance over a singled-threaded solution.
4. What are the two differences between the kernel level threads and user level threads? Which one has a better performance?
5. List the differences between processes and threads.
6. What resources are used when a thread is created? How do they differ from those used when a process is created?
7. What is the meaning of busy waiting? What others kinds of waiting are in OS? Compare each types on their applicability and relative merits.
8. Compare the use of monitor and semaphore operations.
9. When a computer is being developed, it is usually first simulated by a program that runs one instruction at a time. Even multiprocessors are simulated strictly sequentially like this. Is it possible for a race condition to occur when there are no simultaneous events like this?

10. Does the busy waiting solution using the turn variable work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs sharing a common memory?
11. Does Peterson's solution to the mutual exclusion problem work when process scheduling is preemptive? How about when it is non-preemptive?
12. Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?
13. Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and X. In what order should they be run to minimize average response time? (Your answer will depend on X.)
14. Five batch jobs A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.
    (a) Round robin.
    (b) Priority scheduling.
    (c) First-come, first-served (run in order 10, 6, 2, 4, and 8).
    (d) Shortest job first.

    For (a), assume that the system is multiprogrammed, and that each job gets its fair share of the CPU. For (b) through (d) assume that only one job at a time runs, until it finishes. All jobs are completely CPU bound.

15. For the processes listed in following table, draw a Gantt chart illustrating their execution using:
    (a)First-Come-First-Serve.
    (b)Short-Job-First.
    (c)Shortest-Remaining-Time-Next.
    (d)Round-Robin (quantum = 2). (e) Round-Robin (quantum = 1).

Processes	Arrival Time	Burst Time
A	0.00	4
B	2.01	7
C	3.01	2
D	3.02	2

Deadlock

Example

- "It takes money to make money".
- You can't get a job without experience; you can't get experience without a job.

Computers are full of resources that can only be used by one process at a time if number of their instance is one. A process requests for resource and if the resource is not available at that time, it enters a waiting state. Sometimes a waiting process can never change its waiting state because the resource it has requested is held by other waiting process. This situation is called deadlock.

Under normal operation, resource allocations proceed like this:

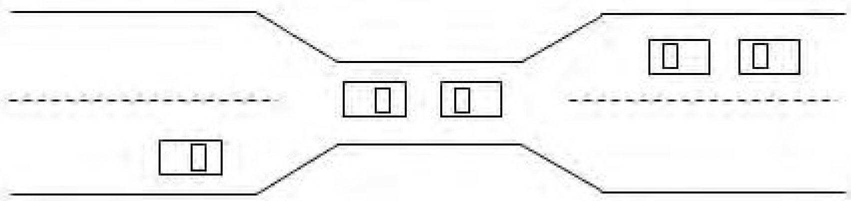
1. Request a resource (suspend until available if necessary).
2. Use the resource.
3. Release the resource.

Resources

There are two types of resources

i) Preemptable: One that can be taken away from the process owning it with no ill effect e.g. memory

ii) Non-Preemptble: one that cannot be taken away from its current owner without causing computation to fail e.g. CD burning



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

Conditions for Deadlock

ALL of these four **must** happen simultaneously for a deadlock to occur

- ✓ **Mutual exclusion:** One or more than one resource must be held by a process in a non-sharable (exclusive) mode.
- ✓ **Hold and Wait**: A process holds a resource while waiting for another resource.
- ✓ **No Preemption**: There is only voluntary release of a resource - nobody else can make a process give up a resource.

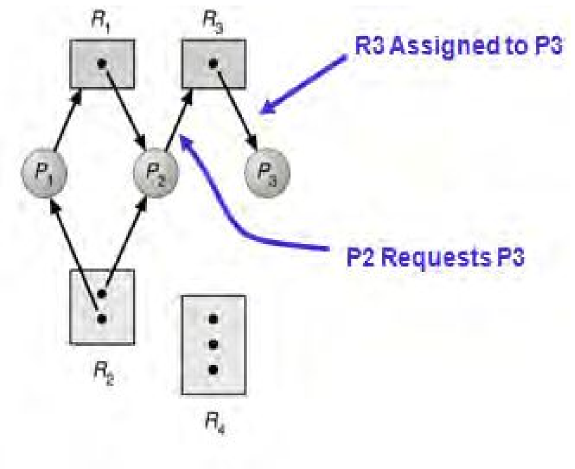
- ✓ **Circular Wait**: Process A waits for Process B waits for Process C .... Waits for Process A.

Resource Allocation Graph

Deadlocks can be defined more precisely in terms of directed graph called a system resource allocation graph:

- ✓ This graph consists of a set of vertices V and set of edges E
- ✓ Set of vertices V is partitioned into two different types of nodes
    - o P= {p1, p2, ……………… ,pn} → the set of all processes in the system
    - o R = {R1, R2, ……………., Rn} → the set consisting all resource types in the system
- ✓ A directed edge from process Pi to resource Rj is denoted by Pi→Rj (Request edge) means Pi has requested an instance of resource type Rj and currently waiting for that resource
- ✓ A directed edge from Rj to Pi denoted by Rj→Pi (assignment edge), means that an instance of resource type Rj has been allocated to Pi.

i) Set P, R, and E
P= {P1, P2, P3}
R= {R1, R2, R3, R4}
E= {P1→R1, R1→P2, P2→R3, R3→P3, R2→P1, P2→R2}

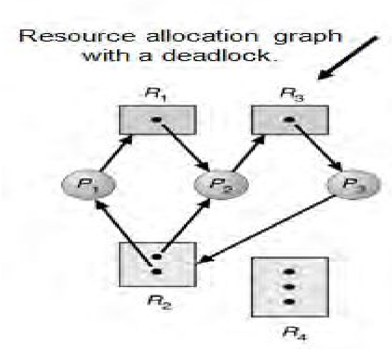


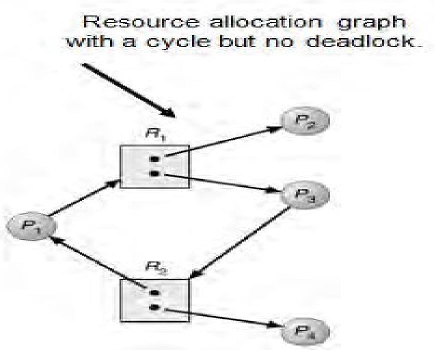
ii) One instance of resource type R1
One Instance of resource type R3
Two instances of resource type R2
Three instances of resource type R4

Process States

P1 is holding an instance of R2 and waiting for an instance of R1

Cycle 1 P1→R1→P2→R3→P3→R2→P1
Cycle 2 P2→R3→P3→R2→P2





Note: If each resource has only one instance, the cycle always cause the deadlock, if each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In the graph represented on right P4 may release its instance of resource type R2 and P3 is allowed breaking the cycle.

Strategies dealing with deadlocks

There are three methods:

- ✓ Ignore Deadlocks: (Ostrich Algorithm Most O.S. do this)
- ✓ Ensure deadlock **never** occurs using either
  - i. **Prevention** Prevent any one of the 4 conditions from happening.
  - ii. **Avoidance** Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations.
- ✓ **Allow** deadlock to happen. This requires using both:
  - i. **Detection** Know a deadlock has occurred.
  - ii. **Recovery** Regain the resources

(Deadlock Prevention) Do not allow one of the four conditions to occur.

1. **Mutual exclusion:**
   - i. Automatically holds for printers and other non-sharable.
   - ii. Shared entities (read only files) don't need mutual exclusion (and aren’t susceptible to deadlock.)
   - iii. Prevention not possible, since some devices are intrinsically non-sharable.
2. **Hold and wait:**
   - i. Collect all resources before execution.
   - ii. A particular resource can only be requested when no others are being held. A sequence of resources is always collected beginning with the same one.
   - iii. Utilization is low, starvation possible.
3. **No preemption:**
   - i. Release any resource already being held if the process can't get an additional resource.
   - ii. Allow preemption - if a needed resource is held by another process, which is also waiting on some resource, steal it. Otherwise wait.
4. **Circular wait:**
   - i. Number resources and only request in ascending order.
   - ii. EACH of these prevention techniques may cause a decrease in utilization and/or resources. For this reason, prevention isn't necessarily the best technique.
   - iii. Prevention is generally the easiest to implement.

(Deadlock Avoidance)

If we have prior knowledge of how resources will be requested, it's possible to determine if we are entering an "unsafe" state. Possible states are:

- ✓ **Deadlock** No forward progress can be made.
- ✓ **Unsafe state** A state that **may** allow deadlock.
- ✓ **Safe state** A state is safe if a sequence of processes exist such that there are enough resources for the first to finish, and as each finishes and releases its resources there are enough for the next to finish.

The rule is simple: If a request allocation would cause an unsafe state, do not honor that request.

NOTE: All deadlocks are unsafe, but all unsafes are NOT deadlocks.

Need an algorithm that can always avoid deadlock by making right choice all the time (bankers algorithm by Dijkstra)

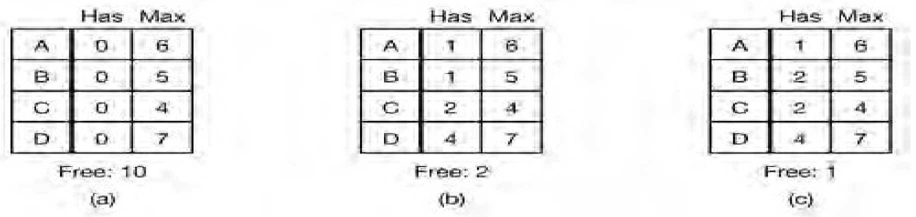
Safety algorithm

To find whether or not a system is in safe state, let work and finish be two vectors of length m and n respectively

```
Initialize
        Work = available
        Finish [i] = false i= 1, 2, ……, n
Find an I such that following both conditions hold
        Finish[i] = false
        Need i≤ Work
        If no such i exist goto step 4
Work = work + Allocation
        Finish[i]= true
        Goto step 2
If Finish[i] =true for all i then the system is in safe state.
```

Banker's Algorithm

Makes a "safe state" check to test for possible deadlock conditions. This algorithm is run by O.S., whenever a process request resources. The algorithm prevents deadlock by denying or postponding the request if it determines that accepting the request could put the system to unsafe state. When a process enters a system, it must declare the maximum number of instances of each resource type that may not exceed the total number of resource in the system. Also when a process gets all of its requested resources, it must return them in finite amount of time.



	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10
(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

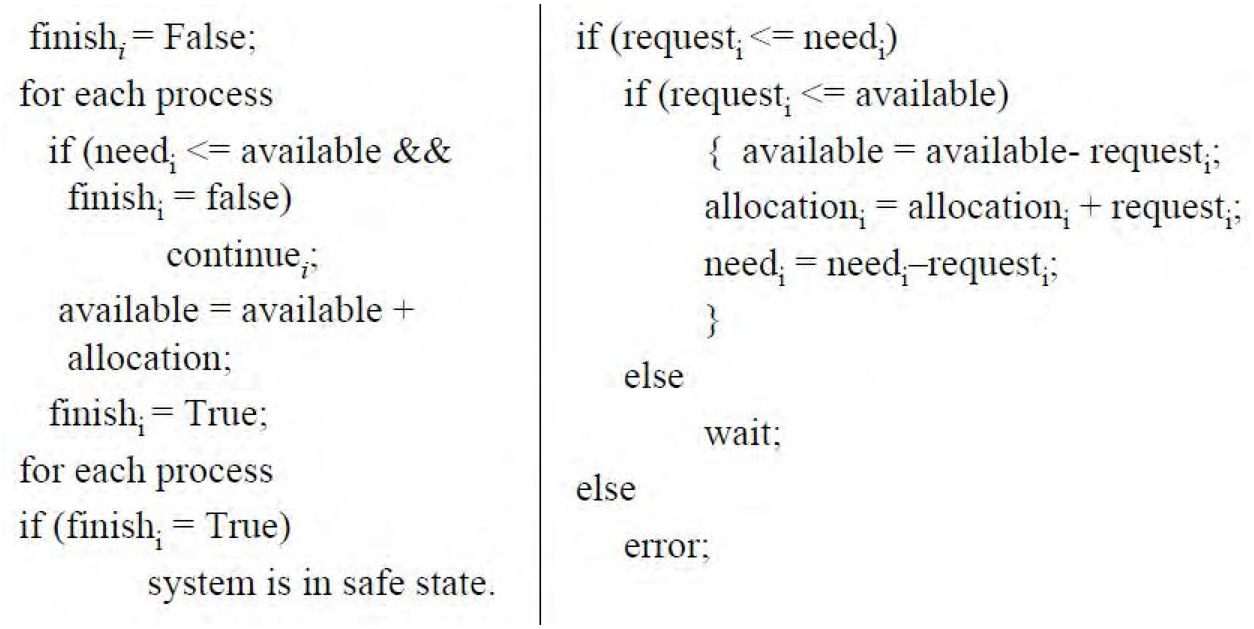
Free: 2
(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1
(c)

Figure: Three resource allocation states: (a) Safe. (b) Safe (c) Unsafe

- ✓ Four customers A, B, C, D
- ✓ Reserved only 10 units rather than 22 to service them
- ✓ After (b), the banker can delay all requests except C's
- ✓ After C finishes, it returns all of its 4 resources.
- ✓ If a request from B for one more unit from (b)→ (c) which is unsafe
- ✓ If customer suddenly asked for their maximum loans, the bankers should not satisfy any one of them lead to deadlock.



```
$finish_i$ = False;
for each process
    if ($need_i$ <= available &&
      $finish_i$ = false)
              $continue_i$;
     available = available +
      allocation;
    $finish_i$ = True;
for each process
if ($finish_i$ = True)
            system is in safe state.
```

```
if ($request_i$ <= $need_i$)
    if ($request_i$ <= available)
          { available = available- $request_i$;
          $allocation_i$ = $allocation_i$ + $request_i$;
          $need_i$ = $need_i$–$request_i$;
          }
    else
          wait;
else
    error;
```

Problem with Banker's Algorithm

- ✓ Algorithms requires fixed number of resources, some processes dynamically changes the number of resources.
- ✓ An algorithm requires the number of resources in advance; it is very difficult to predict the resources in advanced.
- ✓ Algorithms predict all process returns within finite time, but the system does not guarantee it.

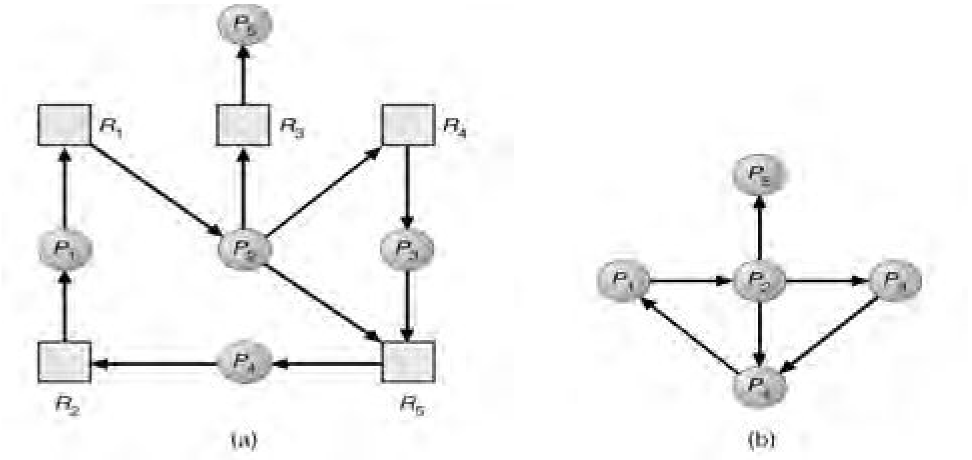
Deadlock Detection

The system does not attempt to prevent the deadlock from occurring, instead it gets them occur, try to detect them. When this happens, then takes action to recover them. In this environment, the system may provide

1. An algorithm that examines the state of the system to determine whether a deadlock has occurred
2. An algorithm to recover from deadlock

Single Instance of each resource type

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the nodes of type resource and collapsing the appropriate edges.



a) wait for graph b) resource allocation graph

Several Instances of resource type

Wait for graph is not applicable to a resource allocation system, when multiple instances of each resource type are available.

- ✓ Each process is initially said to be unmarked.
- ✓ As an algorithm process will be marked, indicating that they are able to complete and thus not deadlocked.
- ✓ When algorithm terminates, any unmarked process are known to be deadlocked.

Algorithm

i. Look for an unmarked process, Pi for the ith row of R is less than or equal to A

ii. If such process is found, add the ith row of C to A, mark the process and go back to step 1.

iii. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any are deadlocked.

Resources in existence E= {E1, E2, E3, ……………. En}
Resource available A= {A1, A2, A3, …………………… An}

Current Allocation matrix

$$C=\begin{pmatrix} C11 & \cdots & C1n \\ \vdots & \ddots & \vdots \\ Cn1 & \cdots & Cnn \end{pmatrix}$$

Resource matrix

$$R=\begin{pmatrix} R11 & \cdots & R1n \\ \vdots & \ddots & \vdots \\ Rn1 & \cdots & Rnn \end{pmatrix}$$

Questions

1. What is deadlock? State the conditions necessary for deadlock to exist. Give reason, why all conditions are necessary.
2. Define the term indefinite postponement. How does it differ from deadlock?
3. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.
4. Write a short note on deadlock?
5. Explain the characteristic of deadlock?
6. Describe various methods for deadlock prevention?
7. Explain the resource allocation graph?
8. Write a note on safe state'?
9. Explain how deadlocks are detected and corrected?
10. What are the difference between a deadlock prevention and deadlock Avoidance?
11. Students working at individual PCs in a computer laboratory send their files to be printed by a server which spools the files on its hard disk. Under what conditions may a deadlock occur if the disk space for the print spool is limited? How may the deadlock be avoided?
12. In the preceding question which resources are preemptable and which are nonpreemptable?
13. The discussion of the ostrich algorithm mentions the possibility of process table slots or other system tables filling up. Can you suggest a way to enable a system administrator to recover from such a situation?
14. In theory, resource trajectory graphs could be used to avoid deadlocks. By clever scheduling, the operating system could avoid unsafe regions. Suggest a practical problem with actually doing this. Can a system be in a state that is neither deadlocked nor safe? If so, give an example. If not, prove that all states are either deadlocked or safe.
15. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.
16. A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	Maximum	Available
Process A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Process B	2 0 1 1 0	2 2 2 1 0	
Process C	1 1 0 1 0	2 1 3 1 0	
Process D	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of *x* for which this is a safe state?

17. Two processes, *A* and *B*, each need three records, 1, 2. and 3, in a database. If *A* asks for them in the order 1, 2, 3, and *B* asks for them in the same order, deadlock is not possible. However, if *B* asks for them in the order 3, 2, 1, then deadlock is possible. With three resources, there are 3! or 6 possible combinations each process can request the resources. What fraction of all the combinations are guaranteed to be deadlock free?

MEMORY MANAGEMENT

Memory is an important resource that must be carefully managed. Most computers have a **memory hierarchy**, with a small amount of very fast, expensive, volatile cache memory, medium-speed, medium-price, volatile main memory (**RAM**), and slow, cheap, nonvolatile disk storage. Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes; each with its own address. It is the job of the operating system to coordinate how these memories are used. The part of the operating system that manages the memory hierarchy is called the **memory manager**. Its job is to keep track of which parts of memory are in use and which parts are not in use, to allocate memory to processes when they need it and de-allocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes. A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed depending on the memory management in use the process may be moved between disk and memory during its execution. The collection of processes on the disk that are waiting to be brought into memory for execution forms the input queue. i.e. selected one of the process in the input queue and to load that process into memory. (see figure 4.1)

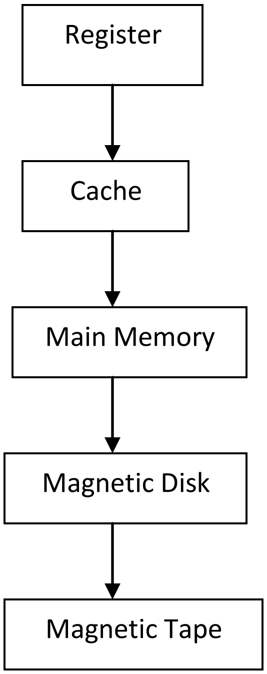


Figure: 4.1

Basic Memory Management

Continuous and non-continuous storage allocation

In Earliest Computer system, each program has to occupy a single continuous block of storage location. In non-continuous storage allocation a program is divided into several blocks or segments that may not necessarily adjacent to one another.

Memory management systems can be divided into two classes: Those that move processes back and forth between main memory and disk during execution (swapping and paging), and those that do not. The latter are simpler, so we will study them first. If main memory ever gets so large that there is truly enough of it, the arguments in favor of one kind of memory management scheme or another may become obsolete. On the other hand, as mentioned above, software seems to be growing even faster than memory, so efficient memory management may always be needed. The trend toward multimedia puts even more demands on memory, so good memory management is probably going to be needed for the next decade at least.

Mono-programming without Swapping or Paging

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. Three variations on this theme are shown in Fig 4.2. The operating system may be at the bottom of memory in RAM (Random

Access Memory), used by mainframes as shown in 4.2 (a), or it may be in ROM (Read-Only Memory) at the top of memory used by palmtop and embedded systems, as shown in 4.2 (b), or the device drivers may be at the top of memory in a ROM and the rest of the system in RAM down below early personal computers, as shown in 4.2 (c).

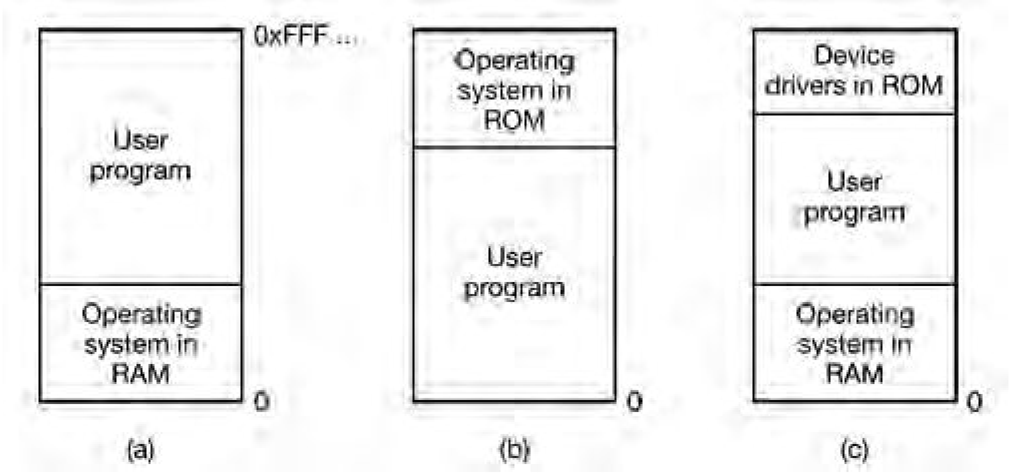


Figure 4.2: Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

When the system is organized in this way, only one process at a time can be running. As soon as the user types a command, the operating system copies the requested program from disk to memory and executes it. When the process finishes, the operating system displays a prompt character and waits for a new command. When it receives the command, it loads a new program into memory, overwriting the first one. ***e.g. Command Prompt***

Multiprogramming with Fixed Partitions

Most modern systems allow multiple processes to run at the same time. Having multiple processes running at once means that when one process is blocked waiting for I/O to finish, another one can use the CPU. Thus multiprogramming increases the CPU utilization.

The easiest way to achieve multiprogramming is simply to divide memory up into *n* (possibly unequal) partitions. This partitioning can, for example, be done manually when the system is started up. The multiprogramming can be achieved in two ways

1. Dedicated partition for each process (absolute transaction)
2. Maintaining a single queue (Relocatable Transaction)

Separate input queue is maintained for each partition, when a process arrives, it is put into the smallest partition large enough to hold it. As shown in figure (a) the processes of size about 200k are assigned to partition 1, processes of size about 200k are assigned to partition 2 and so on.

Problem: What happens when more than one process of size 200k arrives?

Answer: Since it is fixed partition multiprogramming none of the process can enter any other partition than 1. Thus every process has to wait until; the process previous to it has finished its execution. That means we are having more unused but not getting for the processes in queue.

Another strategy is to maintain a single queue for all partitions. Whenever a partition becomes free, the job closest to the front of queue that fits in it could be loaded into the empty partition and run.

Since it is undesirable to waste the large partition on a small job, a different strategy is to search the whole input queue, whenever a partition becomes free and pick the larger job that fits; but this algorithm discriminates against small jobs, because usually interactive jobs has to be serviced faster. As shown in figure 4.3 (b).

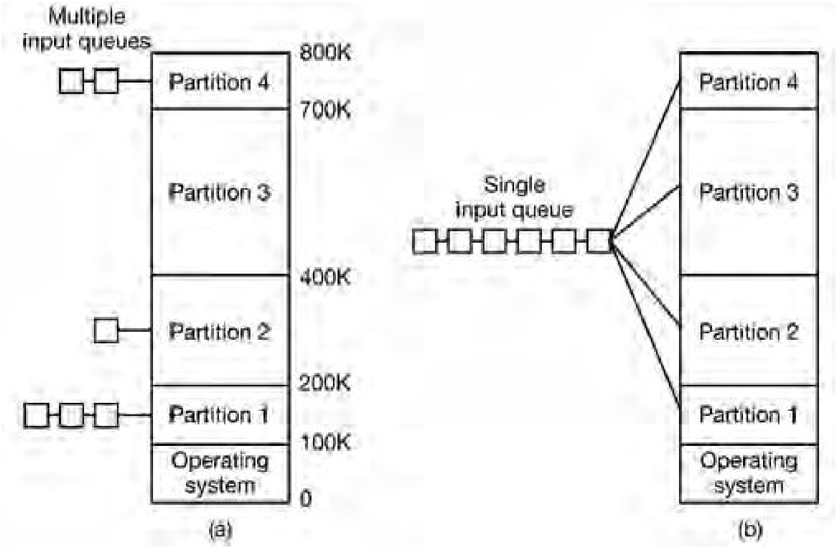


Figure 4.3: (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

Another approach is to have a rule stating that a job that is eligible to run may not be skipped over more than *k* times. Each time it is skipped over, it gets one point. When it has acquired *k* points, it may not be skipped again.

Variable partition multiprogramming

In this approach, no predefined partitions. The number of partitions varies with time. When a process arrives, they are given as much storage as they need. When process finish, they leave holes in the main memory as in figure 4.4.

Advantage

Process gets the exact size of their request for memory. i.e. no wastage of memory.

Problem: When holes are given to another process, they may again be partitioned; the remaining holes get even smaller eventually becoming too small to hold new process. Wastage of memory.

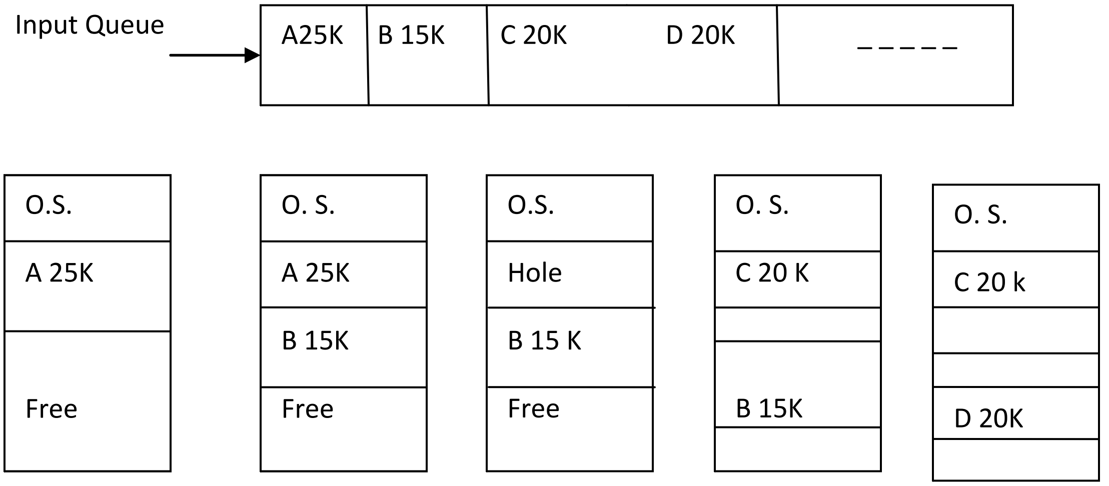


Figure 4.4: Memory Allocation and Holes

Memory Management with Bitmaps

With a bitmap, memory is divided up into allocation units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bitmap, which is 0 if the unit is free and 1 if it is occupied (or vice versa).

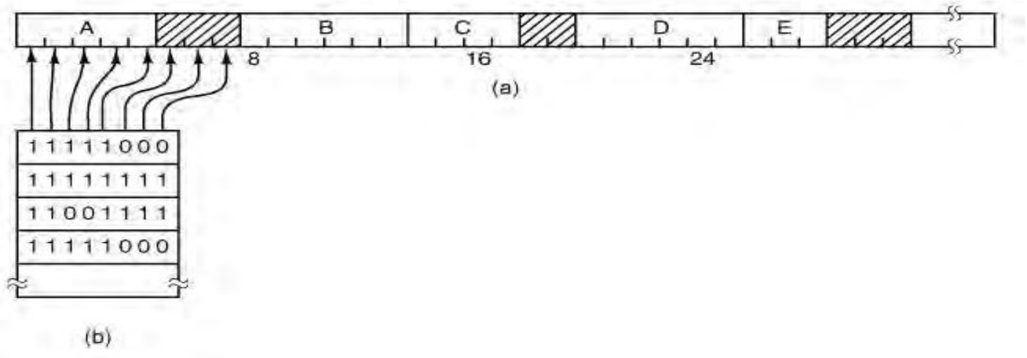


Figure 4.5: (a) A part of memory with five processes and three holes. (b) The corresponding bitmap

The size of the allocation unit is an important design issue. If the size of allocation unit is small, the bitmap will be large.

Disadvantage: Slow operation

Memory Management with Linked Lists

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a process or a hole between two processes. Each entry in the list specifies a hole (H) or process (P), the address at which it starts, the length, and a pointer to the next entry.

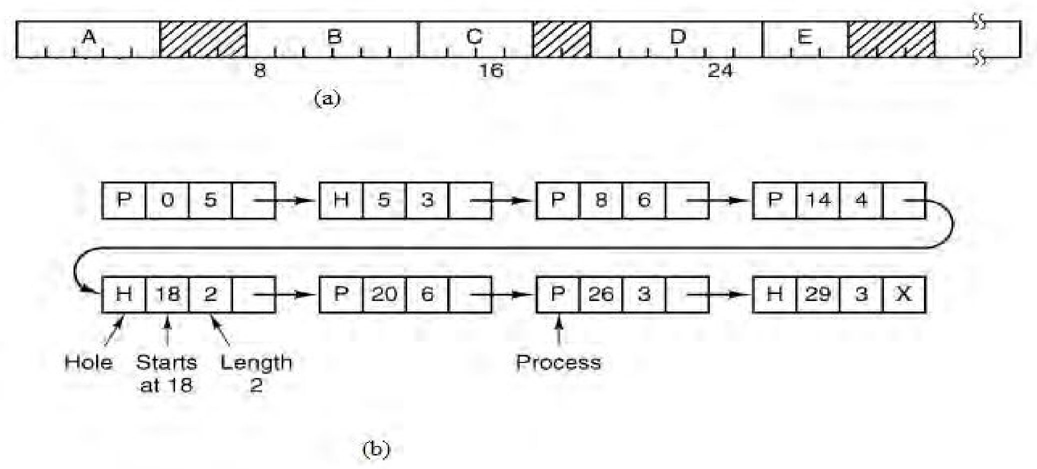


Figure 4.6: (a) A part of memory with three holes and five processes (b) Above information as list

When a process terminates, if any neighbor is already hole, merge the holes coalescing (compaction of neighboring holes).

Buddy System:

It is defined as power of two allocator.

1. Suppose, we have the entire block of size 2^u.
2. A request of size 'S', Such that $2^{u-1} < S \leq 2^{u-2}$is made, then entire block is allocated.
3. Otherwise, the block is split into two equal buddies of size u-1.
4. If the block $2^{u-2} < S \leq 2^{u-1}$, then the request is allocated to one of two buddies, otherwise one of the buddy is split into half again.
5. This process is continued until the smallest block greater than or equal to S is generated and allocated to the request.

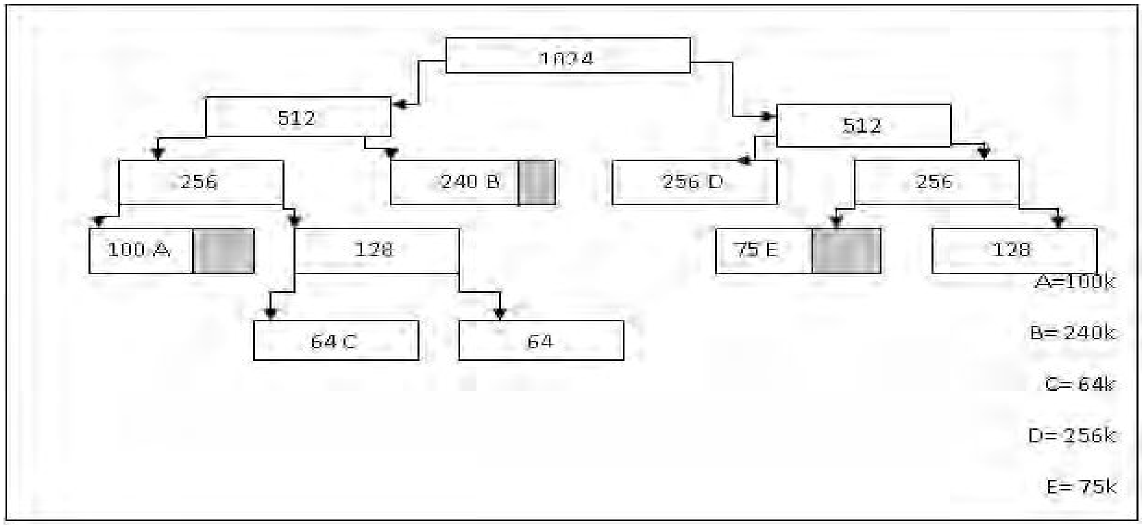


Figure 4.7: Buddy system

Advantage: The adjacent buddies can be combined quickly to form larger segment using a technique called coalescing.

Disadvantage: Fragmentation

Partition selection Algorithms

When a process arrives and needs memory, we search this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, we merge these adjacent holes to form one larger hole.
This procedure is a particular instance of the general dynamic storage-allocation problem, which is how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The set of holes is searched to determine which hole is best to allocate, first-fit, next-fit, best-fit, and worst-fit are the most common 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 where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
   Advantage: Fast
   Disadvantage: Not good in terms of storage utilization.
2. **Next-fit:** It work the same way as first-fit, except that it keeps track of where it finds a suitable hole. Next time if it is called to find a hole, it starts from that position.
3. **Best-fit:** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy-produces the smallest leftover hole.
   Advantage: More storage utilization than first fit.
   Disadvantage: Time consuming.

4. **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 which may be more useful than the smaller leftover hole from a best-t approach.
   Advantage: Sometimes it has more storage utilization than first fit.
   Disadvantage: Not good for performance.

Another algorithm is quick fit (Study yourself).

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces.

- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by compaction
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible only if relocation is dynamic, and is done at execution time.

In context of batch system, each job is loaded into a partition when it gets to the head of the queue. It stays in memory until it has finished. As long as enough jobs can be kept in memory to keep the CPU busy all the time, there is no reason to use anything more complicated.

With timesharing systems, the situation is different. Sometimes there is not enough main memory to hold all the currently active processes, so excess processes must he kept on disk and brought in to run dynamically.

Two general approaches to memory management can be used, depending (in part) on the available hardware.

- **Swapping:** Bringing in each process in its entirety, running it for a while, then putting it back on the disk.
- **Virtual memory:** Allows programs to run even when they are only partially in main memory.

Swapping

A process, can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution. Assume a multiprogramming environment with a round robin CPU-scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed as in figure 4.8. When each process finishes its quantum, it will be swapped with another process.

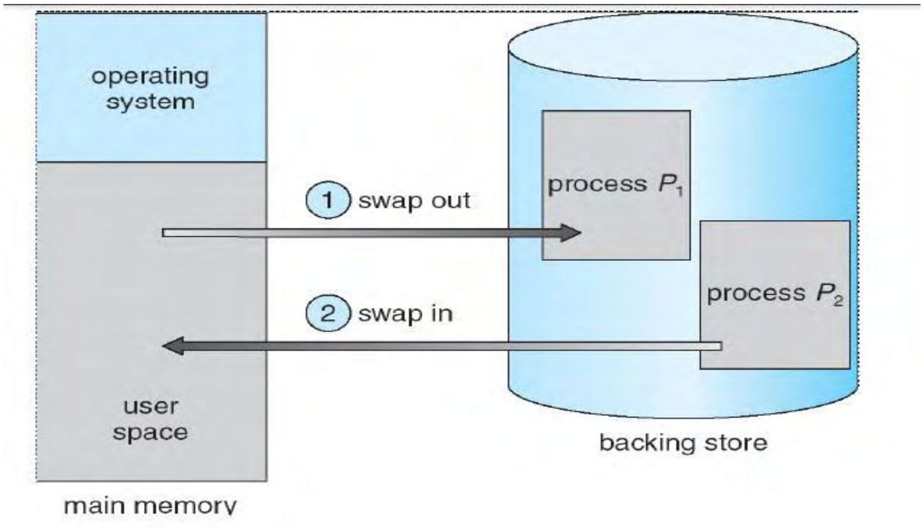


Figure 4.8: Swapping of two processes using disk as backing store

The operation of a swapping system is illustrated in Fig. 4.9. Initially only process *A* is in memory. Then processes *B* and *C* are created or swapped in from disk. In Fig. (d) *A* is swapped out to disk. Then *D* comes in and *B* goes out. Finally *A* comes in again. Since *A* is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution.

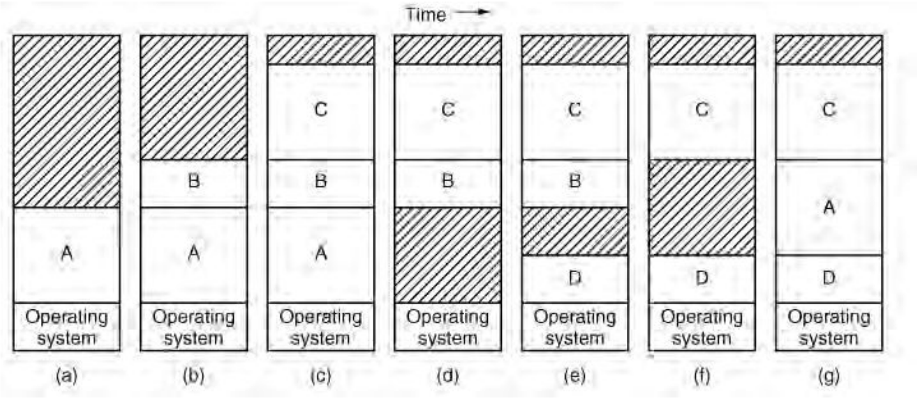


Figure 4.9: Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

A point that is worth making concern is how much memory should be allocated for a process when it is created or swapped in. If the process is created with fixed size that never change, the O.S. allocates exactly what is needed. Neither less nor more. If process data segment can grow, for example dynamically allocating memory from a heap, a problem occurs when ever a process tries to grow.

If a hole is adjacent to the process, it can be allocated but if process is adjacent to growing process; either it have to be moved to a hole in memory large enough to hold it or one or more have to be swapped out to create a large hole. If it is expected that most processes will grow as they run, it is probably a good idea to allocate a little extra memory whenever a process is moved or swapped in. But when swapping process to disk only actual memory in use should be swapped not its growing segment.

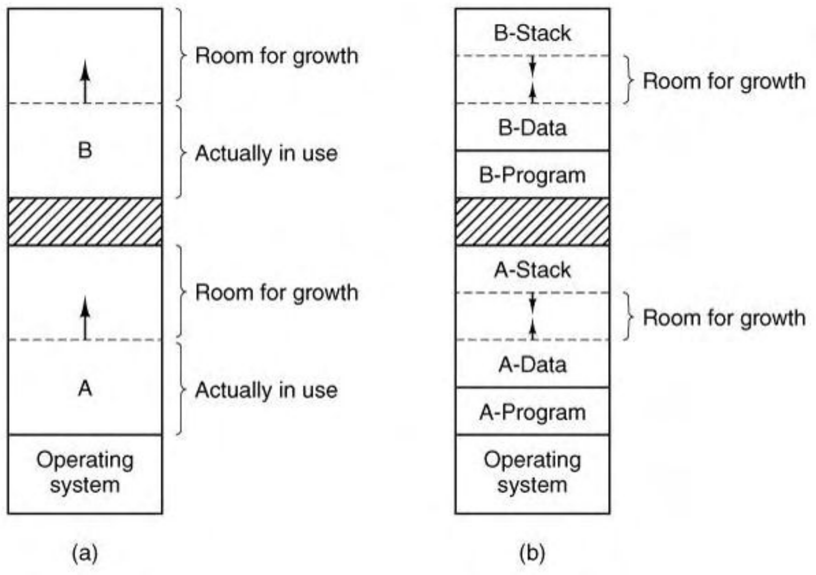


Figure 4.10: (a) allocating space for growing data segment (b) allocating space for growing data segment and stack

If processes can have two growing segments, for example, the data segment being used as a heap for variables that are dynamically allocated and released and a stack segment for the normal local variables and return addresses, an alternative arrangement suggests itself, namely that of figure 4.10(b). In this figure we see that each process illustrated has a stack at the top of its allocated memory that is growing downward, and a data segment just beyond the program text that is growing upward. The memory between them can be used for either segment.

What if a process cannot grow in memory and the swap area in the disk is full?

Process will have to wait or to be killed.

VIRTUAL MEMORY

How to run too big to fit in the available memory. The solution is to split the program into pieces, called **overlays**. Overlay 0 would start running first. When it is done, it would call another overlay. Some overlay systems were highly complex, allowing multiple overlays in memory at once. The overlays were kept on the disk and swapped in and out of memory by the operating system, dynamically, as needed.

Although the actual work of swapping overlays in and out was done by the system, the work of splitting the program into pieces had to be done by the programmer. Splitting up large programs into small, modular pieces is time consuming and boring.

Fotheringham, (1961) came with another solution known as **virtual memory** The basic idea behind virtual memory is that the combined size of the program, data, and stack may exceed the amount of physical memory available for it. The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk. For example, a 16-MB program can run on a 4-MB machine by carefully choosing which 4 MB to keep in memory at each instant, with pieces of the program being swapped between disk and memory as needed.

It can be implemented in two ways

i. Paging
ii. Segmentation

Memory Management Unit (MMU)

The run time mapping from virtual address to physical address is done by hardware device calle MMU.

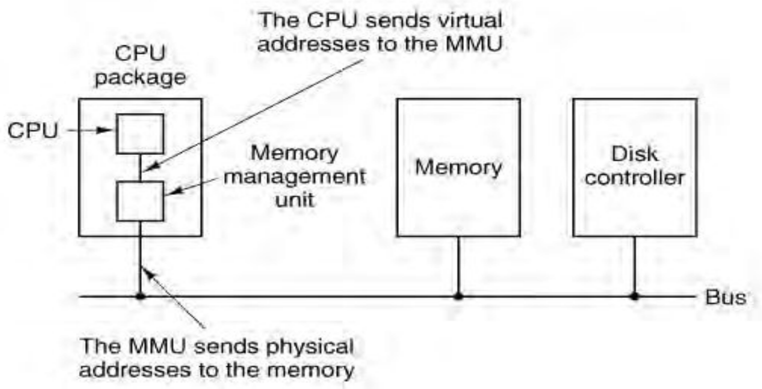

The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

Figure 4.11: MMU

Logical vs. Physical Address Space

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

Paging

Most virtual memory systems use a technique called **paging.** Paging is the technique that permits the physical address space of a process to be noncontiguous. Paging avoids external fragmentation and the need of compaction. It also solves the need of swapping.

To implement paging, the virtual address space is divided up into units called **pages**, and the corresponding units in the physical memory are called **page frames**. The pages and page frames are always the same size. When a process is to be executed, its pages are loaded into any available memory frame from their source.

Every address generated by CPU is divided into two parts: **page number (p)** and a **page offset (d)**. The page number is used as index to the page table. The **page table** contains the base address of each page in physical memory. Base address is combined with the page offset to define physical memory address that is sent to the memory unit.

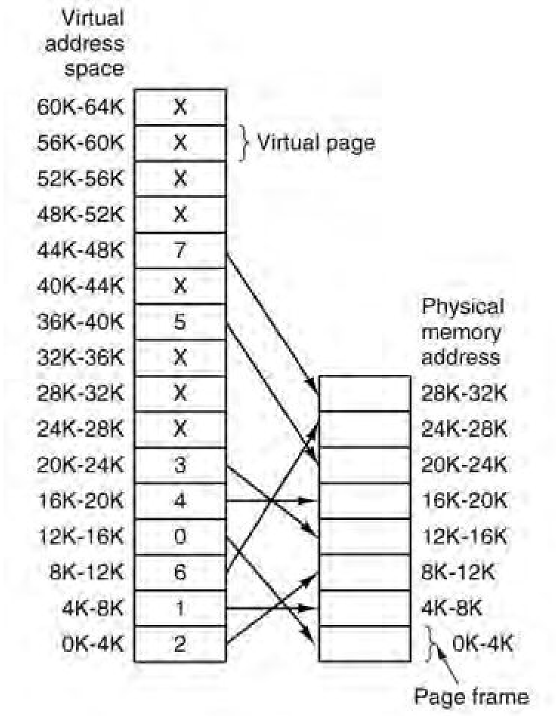


Figure 4.12: Relation between virtual address and physical address

The page size is defined by the hardware as like frame size. It is power to 2 varying from 512 bytes to 16MB per page depending upon the computer architecture. The power of two selections makes translation of logical address into page number and page offset easy. If 2^m is logical address and 2^n is page size, then *m-n* designates the page number and *n* designates the page offset.

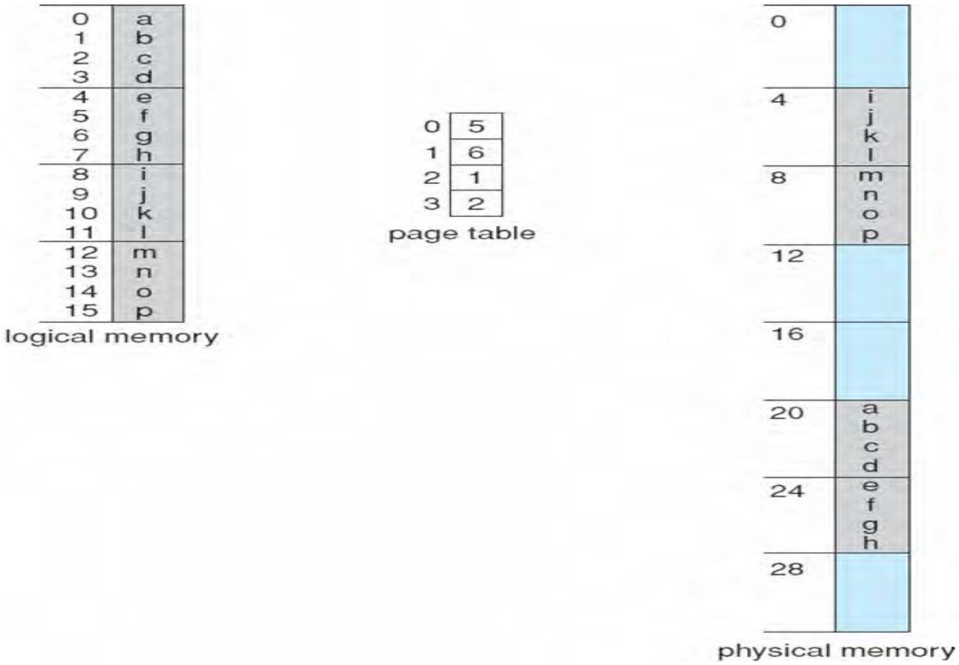


Figure 4.13: Paging example for a 32 byte memory with a 4 byte pages.

Here n=2 and m=4, using a page size of 4 bytes and a physical memory of 32 bytes (8 pages). Logical address 0 is page 0 and offset 0. Indexing into the page table, it is found that page 0 is in frame 5. Thus logical address 0 maps to physical address 20 = [(5*4) + 0] i.e. page frame * page size + offset (displacement within the page). Similarly logical address 3 maps to physical address 23 = [(5*4) + 3]. Logical address 13 maps to physical address 13.

What happens when program tries using an unmapped page?

Page Fault.

Hardware Support

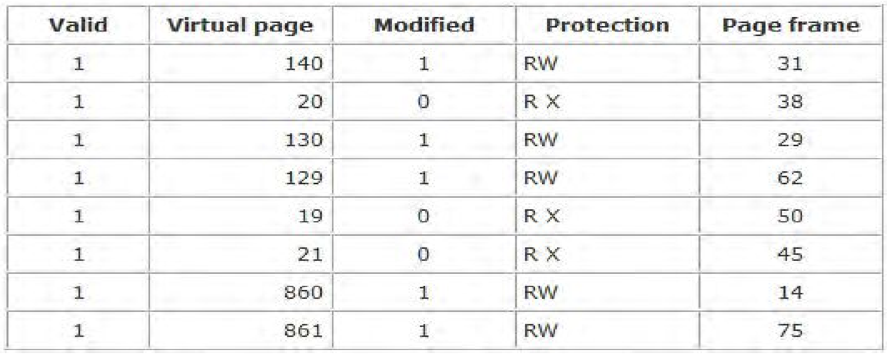
Each operating system has its own methods for storing page tables. Most allocate a page table for each process. A pointer to the page table is stored with the other register values (like the instruction counter) in the process control block.

The simplest way of hardware implementation is implementation of page table as a set of dedicated registers, built with high-speed logic to make paging-address translation efficient. Since every access to memory goes through paging map, so efficiency is major concern.

Use of registers as page table is satisfactory if the page table is small (256 entries), but not feasible for large page tables (1 million entries). Also while using page table two memory accesses are needed to access a byte (page# and frame#). Thus memory access is slowed by a factor of two.

The standard solution to this problem is to use a special kind of small, fast lookup hardware cache called a **Translation Look-aside Buffer (TLB)** in side MMU sometimes also called **Associative memory.** Each entry in the TLB consist of two parts a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys. If key is found, the corresponding value field is returned, it is called TLB hit. And if the page number is not found in the TLB (TLB miss), a memory reference to the page table must be made. When the frame number is obtained, it can be used to access the memory. The search is fast; hardware however is expensive. Some TLB allows certain entries to be **wired down,** meaning that they cannot be removed from the TLB.

Some TLBs store **address- space identifiers** (ASIDs) in each TLB entry. ASID is used to provide unique address space protection for process. When TLB tries to resolve virtual page numbers, it ensures that the ASID of currently running process and ASID associated with virtual page number match. If not treated as TLB miss.



Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Figure 4.17: A page table with 8 entries

How TLB works?

The percentage of times a particular page is found in the TLB is called **hit ratio.** A 60% hit ratio means, the desired page number is found 60% of the time in TLB. If it takes 20 nanoseconds to reach TLB and 100 nanoseconds to access memory then a mapped memory access takes 120 nanoseconds when the page number is in TLB.

If we fail to find the page number in TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), a total of 220 nanoseconds. To find total **effective memory access time**, we weight the case by its probability:

$$\text{Effective Access Time (EAT)} = 0.60 * 120 + 0.40 * 220$$

$$= 160 \text{ nanoseconds}$$

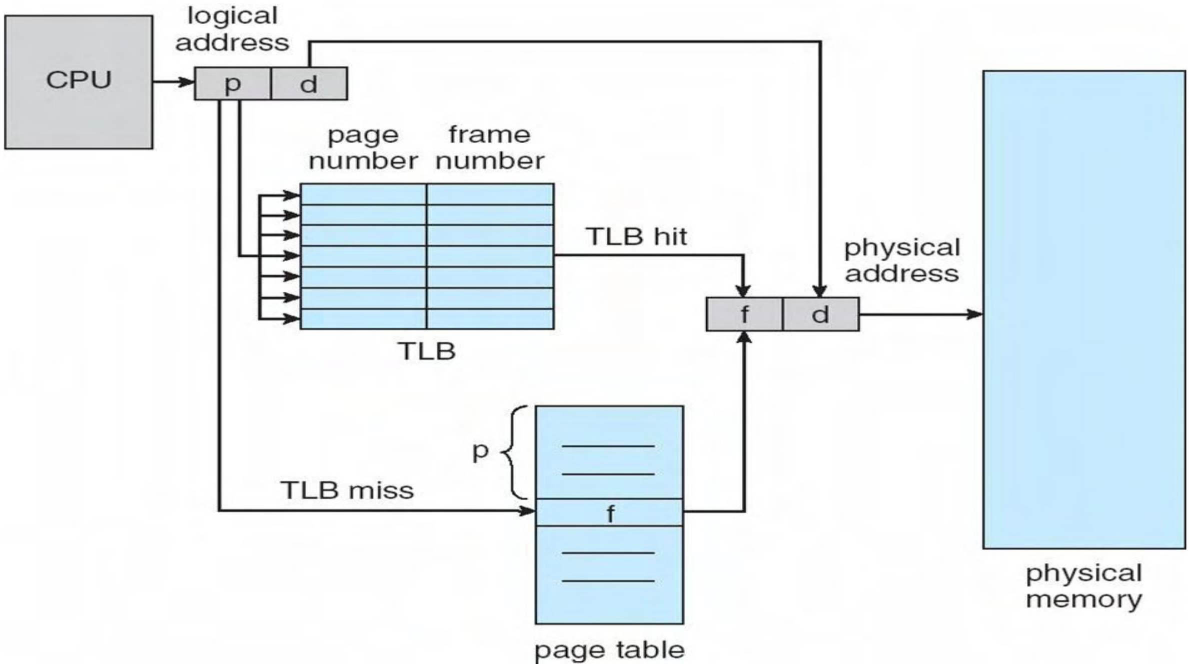


Figure 4.18: Paging hardware with TLB

Protection

Memory protection in a paged environment is accomplished by protection bits associated with each frame. These bits are kept in page table.

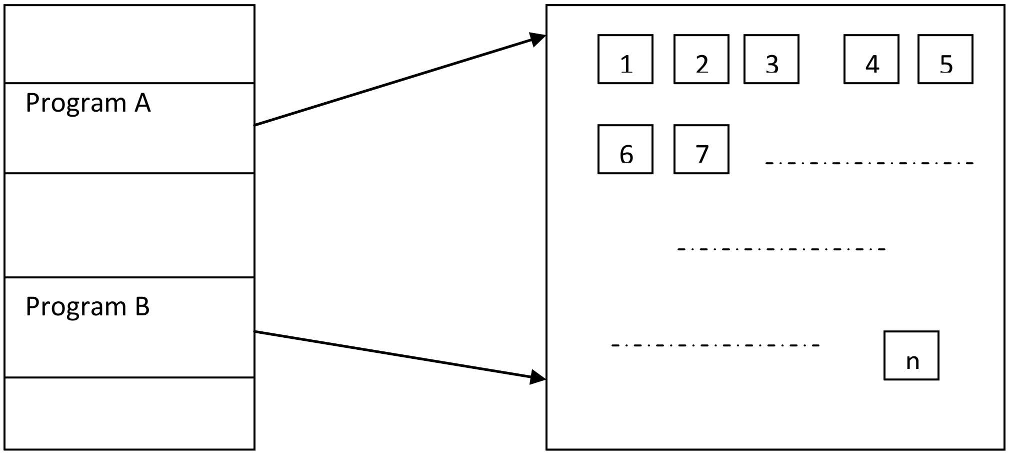
One bit can define a page to be read-write and read only. Since every reference to memory goes through page table to find correct frame number. At the time of physical address computation, protection bits can be checked to verify the permission associated with it.

An attempt to write to a read-only page causes a hardware trap to the operating system. It can be easily expanded further to provide a finer level of protection. We can create hardware to provide read-only, read-write or execute only protection by providing separate protection bits for each kind of access. We can allow any kind of combination of these accesses. Illegal attempts will be trapped to the operating system.

Demand Paging

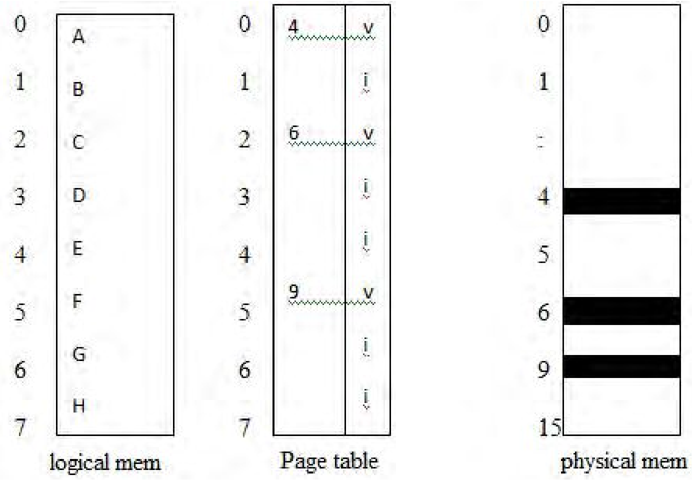
Suppose an executable program might be loaded from disk to the memory. One option is to load the entire program in physical memory at program execution time, but the problem with it that we may not need the whole program initially (i.e. wastage of memory).

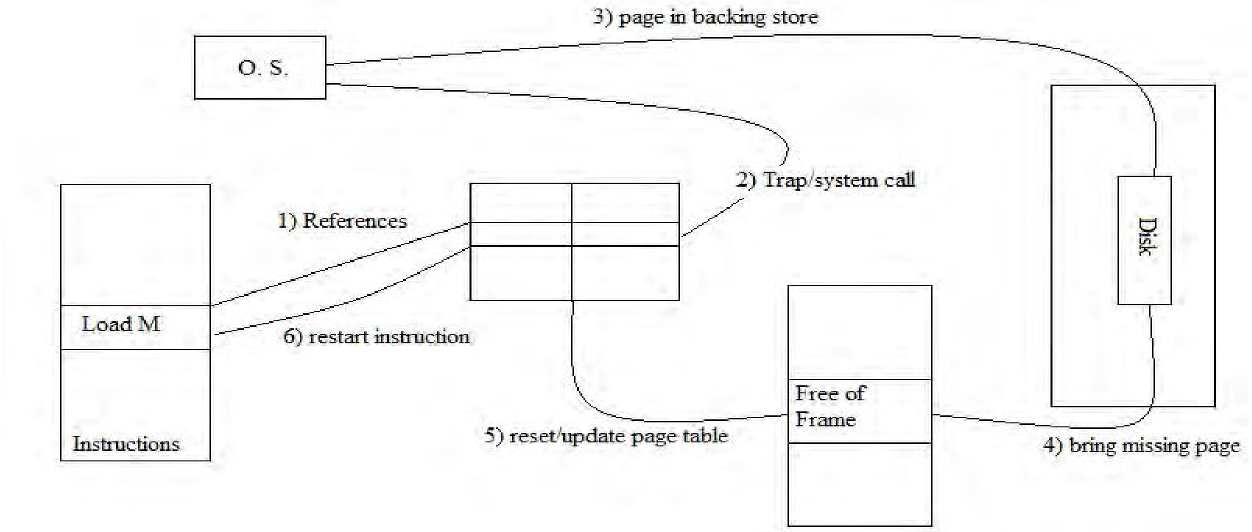
An alternative strategy is to break the program into pages and load the pages only as they are needed. This technique is known as demand paging. Here pages are loaded as their demand during execution. Pages that are never accessed are thus never loaded in memory. A lazy swapper is used (never swap a page in memory unless that page is needed).



With this schema, we need some format of hardware support to distinguish between those pages

- ✓ Valid – page is legal and in memory
- ✓ Invalid- page either is not valid or is valid but currently in disk.





Page Tables

The purpose of the page table is to map virtual pages onto page frames. Mathematically speaking, the page table is a function, with the virtual page number as argument and the physical frame number as result. Using the result of this function, the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

Structure of a Page Table

The exact layout of a page table is highly machine dependent, but the kind of information present is roughly the same from machine to machine. The size varies from computer to computer, but 32 bits is a common size. The most important field is the page frame number. After all, the goal of the page mapping is to locate this value. Next to it we have the present/absent bit. If this bit is 1, the entry is valid and can be used. If it is 0, the virtual page to which the entry belongs is not currently in memory. Accessing a page table entry with this bit set to 0 causes a page fault.

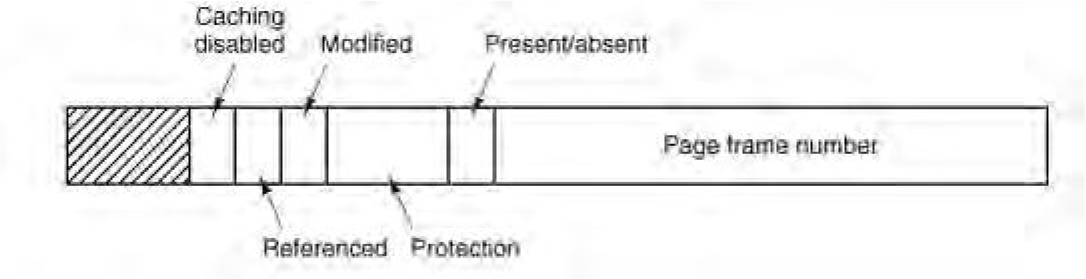


Figure 4.14: Structure of Page Table

The protection bits tell what kinds of access are permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only. A more sophisticated arrangement is having 3 independent bits, one bit each for individually enabling reading, writing, and executing the page.

Multilevel (Hierarchical) Page Tables

Most modern computers support a large logical address space (2^{32} to 2^{64}). In such environments, the page table itself becomes excessively large.

e.g. for a system of 32-bit logical address, if page size is of 4 KB (2^{12}), page table consists of 1 million entries i.e. $2^{32}/2^{12}$. Assume each entry consist of 4 bytes.

In such case, each page table may need up to 4 MB of physical address space. One way to solve this problem is, use of multilevel page table.

- ✓ For a 32 bit logical address and a page size of 4KB
- ✓ Logical address is divided into page number consisting of 20 bits
- ✓ Page offset consisting of 12 bits.
- ✓ Since the page table is paged, page number is further divided into 10 bit page number and 10 bit offset.

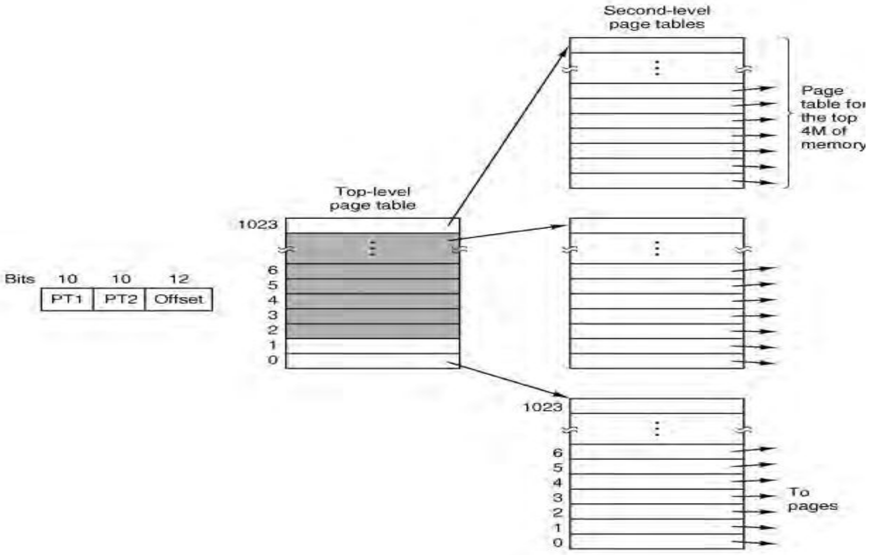


Figure 4.15: A two level page table.

For a system with 64-bit logical address space

Outer	Inner	Offset
P1	P2	d
42	10	12

Not efficient since P_1 is still of size 2^{52}

P1	P2	P3	d
32	10	10	12

It can be further divided.

Hashed Page Table

A common approach to handle address spaces larger than 32 bits is to use a hashed page table, with the hash value being virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collision). Each element consist of three fields (1) virtual page number (2) value of mapped page frame (3) pointer to the next element in the linked list.

Working: virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the link list. If there is a match, the corresponding page frame (2) is used to find the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

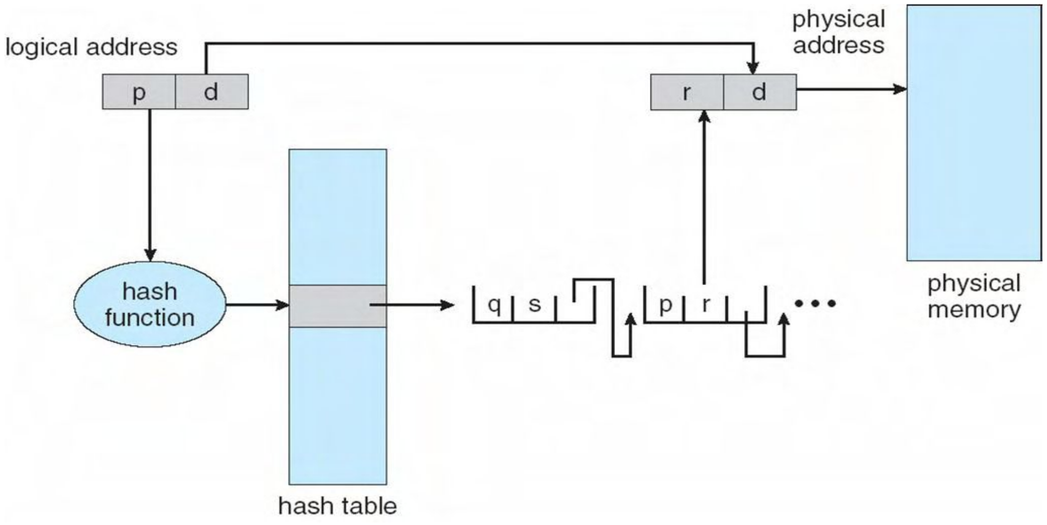


Figure 4.16: Structure of hashed page table

Inverted page table

- One entry for each page of memory
- Entry consists of the virtual address of the page stored in that memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs
- Use hash table to limit the search to one — or at most a few — page-table entries

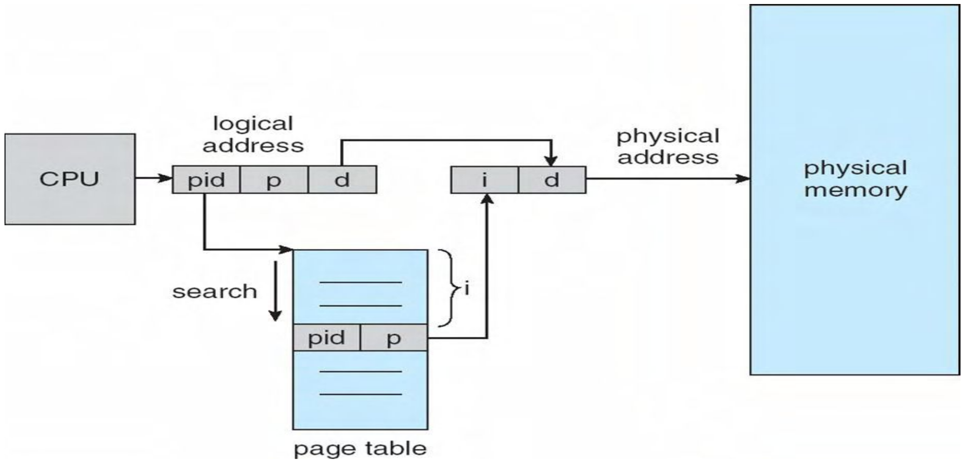


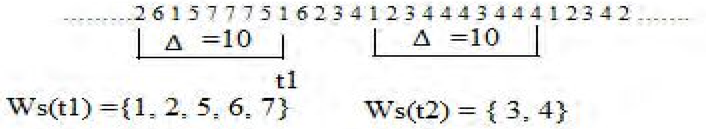
Figure 4.16: Inverted page table

Thrashing

If the process does not have any frame number, page fault occurs. At this time, it must replace some page, since all pages are in active use, it must replace a page that will be needed again. Consequently, it quickly faults again and again and again. This high paging activity is called thrashing. To prevent thrashing, we must provide a process as many frames as it needs. How one can know how many frames do we need? Several techniques have been developed, one of them is working set model.

Working Set Model

- ✓ This approach defines the locality model of process execution.
- ✓ During any phase of execution, the process references only a relatively small function of its pages called locality of reference.
- ✓ This model use, a parameter Δ to define the working set.
- ✓ The idea is to examine the most recent Δ page references.

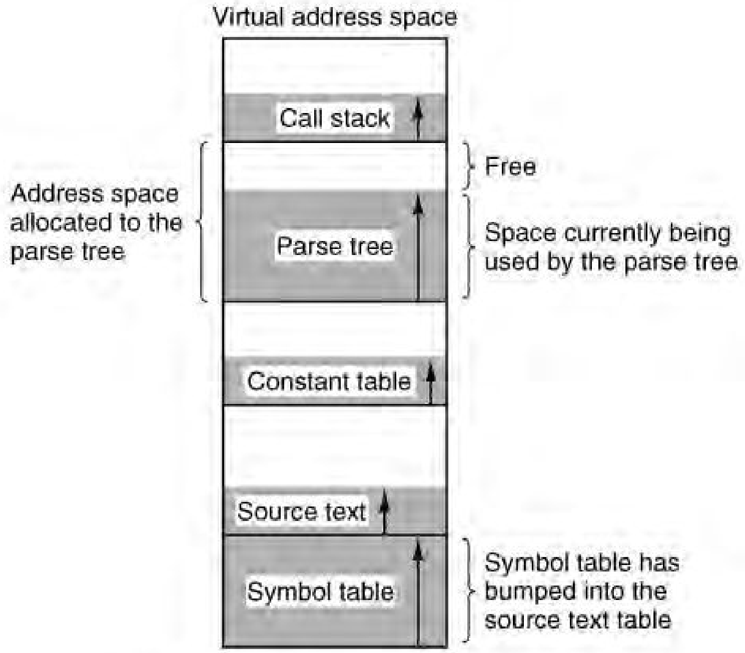


- ✓ If a page is in active use, it will be in working set
- ✓ If it is no longer used, it will drop from the working set

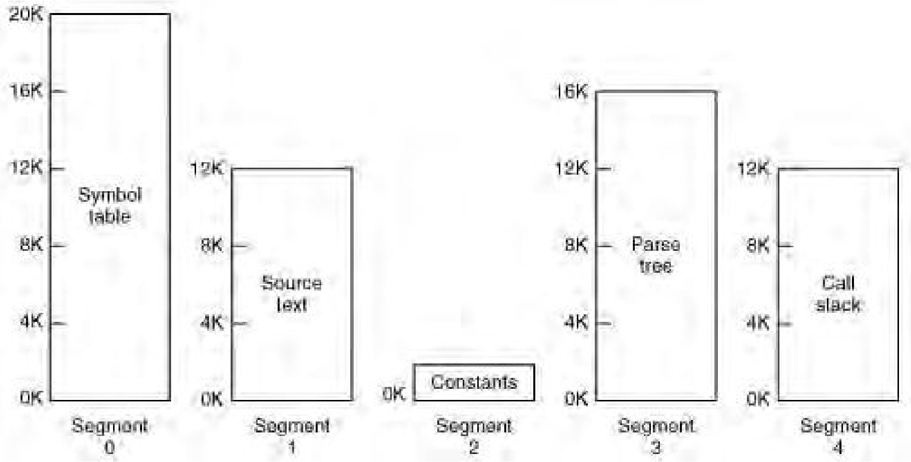
Segmentation

In paging memory will be treated as a linear address of bytes.
Suppose we need memory for compiler during execution, then paging describes memory as follows:

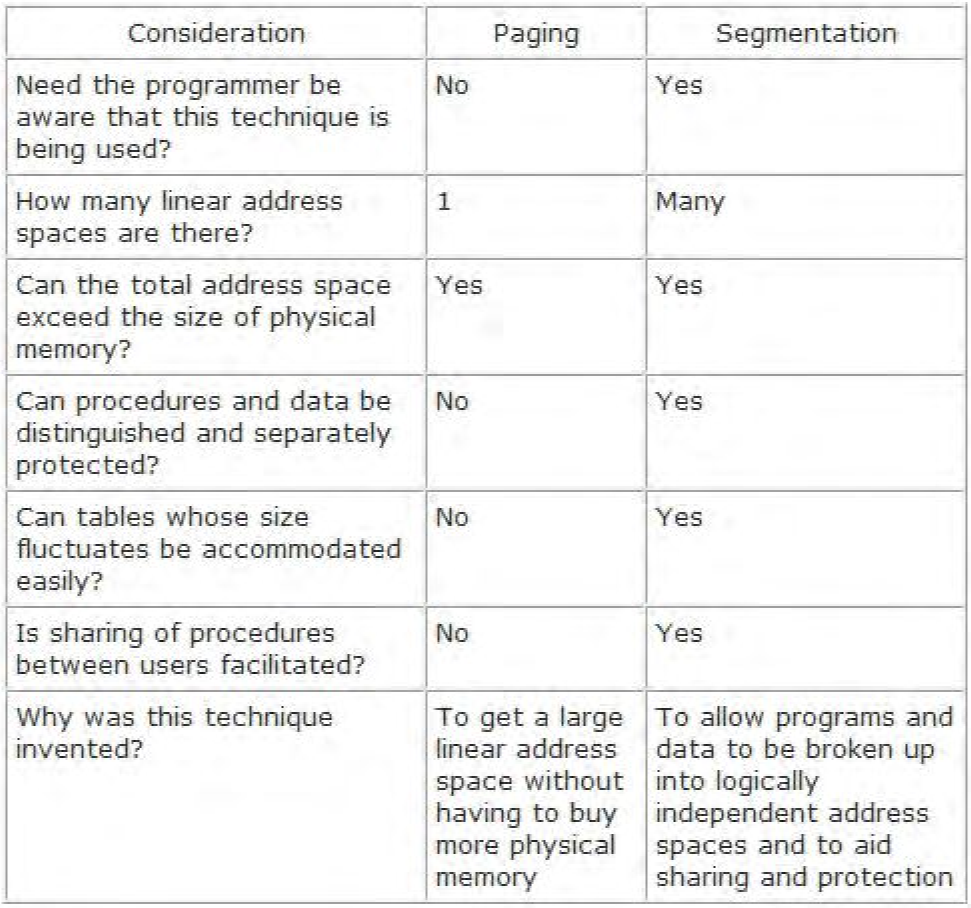
- ✓ Suppose the compiler needs a lot of variables such that the size of symbol table may not be sufficient.
- ✓ But there is a large free space in other tables.
- ✓ Compiler say the compilation cannot continue, but during so seem not be logical since there is free room.
- ✓ One general solution is to provide the machine with many completely independent address spaces called segments.


Virtual address space
Call stack
Free
Address space allocated to the parse tree
Parse tree
Space currently being used by the parse tree
Constant table
Source text
Symbol table
Symbol table has bumped into the source text table

Segmentation is the memory management technique in which memory is divided into variable sized chunks, which can be allocated to processes. Each chunk is called segment. Each segment consists of linear sequence of addresses from zero to some maximum. Different segments may have different lengths. In segmentation, a program may occupy more than one partition, and these partitions need not be contiguous. Also the segments lengths may vary during execution. Besides this the segment may be increased whenever something is pushed onto stack and decreased when something is popped out.



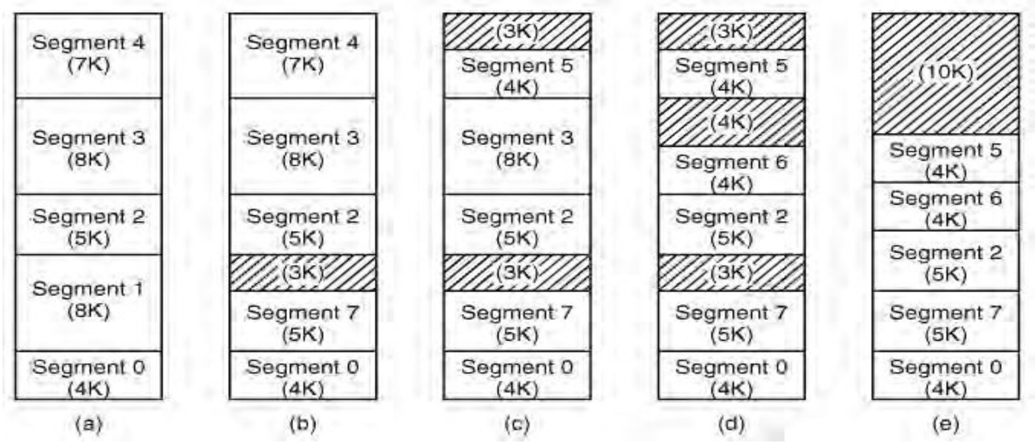
Comparison between paging and Segmentation



Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Implementation of Pure Segmentation

The implementation of segmentation differs from paging in an essential way: pages are fixed size and segments are not. Figure (a) shows an example of physical memory initially containing five segments. Now consider what happens if segment 1 is evicted and segment 7, which is smaller, is put in its place. We arrive at the memory configuration of Fig. (b). Between segment 7 and segment 2 is an unused area that is, a hole. Then segment 4 is replaced by segment 5, as in Fig. (c), and segment 3 is replaced by segment 6, as in Fig.(d). After the system has been running for a while, memory will be divided up into a number of chunks, some containing segments and some containing holes. This phenomenon, called checker boarding or external fragmentation, wastes memory in the holes. It can be dealt with by compaction, as shown in Fig. (e).



Segmentation with Paging: MULTICS

If the segments are large, it may be inconvenient, or even impossible, to keep them in main memory. This leads to the idea of paging them, so that only those pages that are actually needed have to be around.

Each MULTICS program has a segment table, with one descriptor per segment A segment descriptor contains an indication of whether the segment is in main memory or not. If any part of the segment is in memory, the segment is considered to be in memory, and its page table will be in memory. The descriptor also contains the segment size, the protection bits, and a few other items. The address of the segment in secondary memory is not in the segment descriptor but in another table used by the segment fault handler.

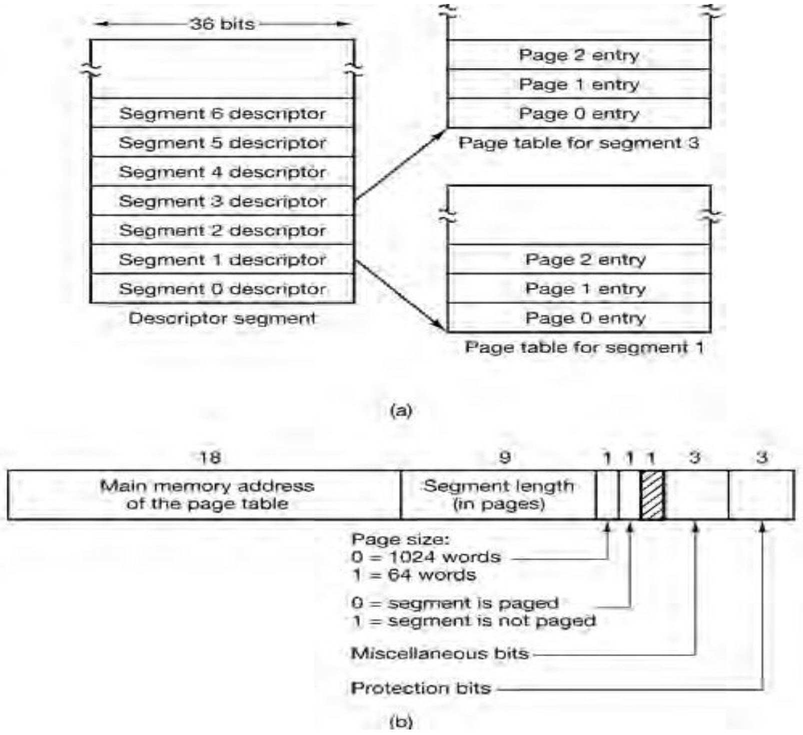


Figure: The MULTICS virtual memory. (a) The descriptor segment points to the page tables. (b) A segment descriptor. The numbers are the field lengths.

Each segment is an ordinary virtual address space and is paged in the same way as the non segmented paged memory. The normal page size is 1024 words (although a few small segments used by MULTICS itself are not paged or are paged in units of 64 words to save physical memory).

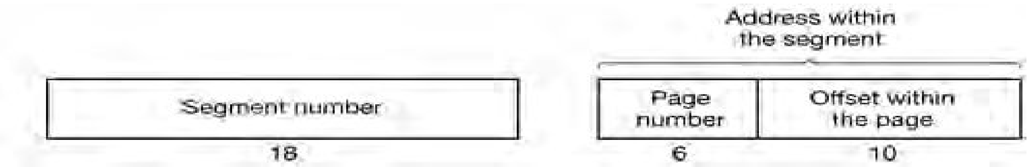


Figure : A 34-bit MULTICS virtual address.

An address in MULTICS consists of two parts: the segment and the address within the segment. The address within the segment is further divided into a page number and a word within the page. When a memory reference occurs, the following algorithm is carried out.

1. The segment number is used to find the segment descriptor.
2. A check is made to see if the segment's page table is in memory. If the page table is in memory, it is located. If it is not, a segment fault occurs. If there is a protection violation, a fault (trap) occurs.
3. The page table entry for the requested virtual page is examined. If the page is not in memory, a page fault occurs. If it is in memory, the main memory address of the start of the page is extracted from the page table entry.
4. The offset is added to the page origin to give the main memory address where the word is located.
5. The read or store finally takes place.

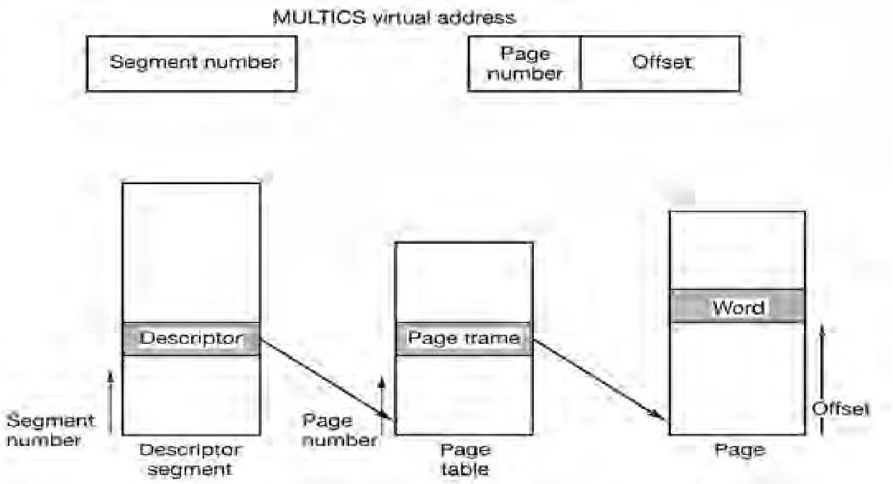


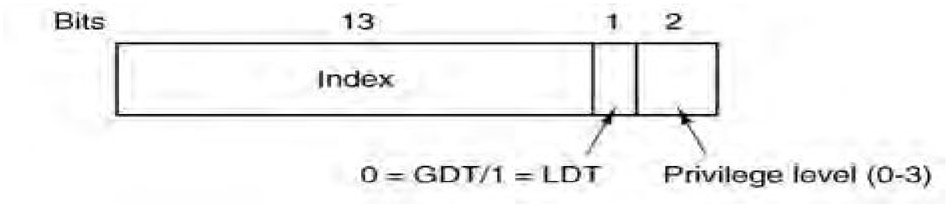
Figure: Conversion of a two-part MULTICS address into a main memory address.

Segmentation with Paging Intel Pentium

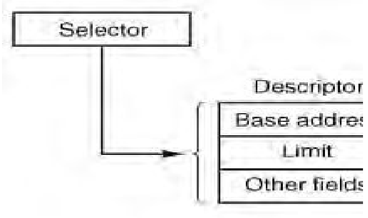
Pentium virtual memory consists of two types:

- ✓ LDT (Local Descriptor Table)→ Private
- ✓ GDT (Global descriptor Table) → public

Each program has its own LDT but there is a single GDT, shared by all the programs. LDT describes segments local to each program including its code, data, stack etc, whereas GDT describes system segments, including the operating system itself. To access a segment, a Pentium program first loads a selector for that segment. Each selector is a 16 bit program.



At the time the selector is loaded, its corresponding descriptor is fetched.

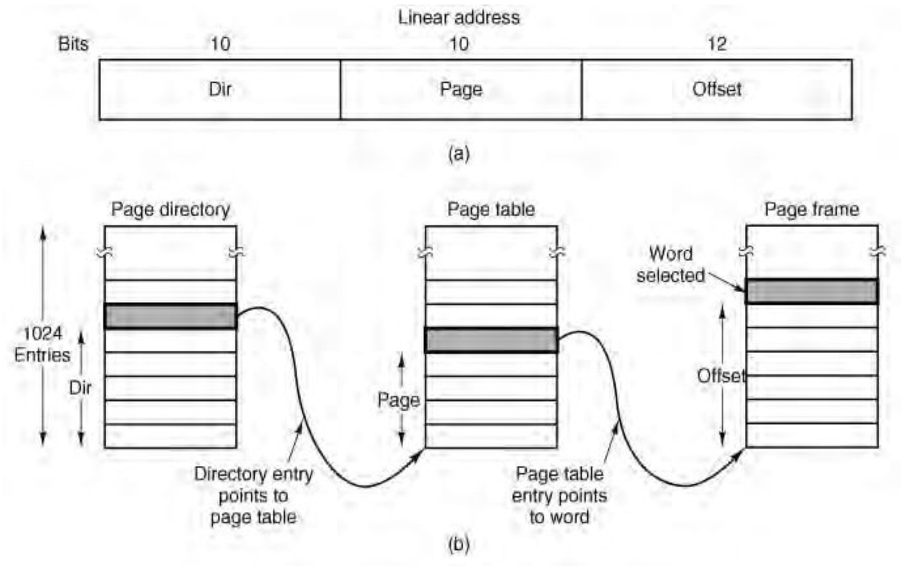


Case 1

If paging is disabled, then the linear address will be the physical address.

Case 2

If paging is enabled, further use the page table.



Page replacement Algorithm

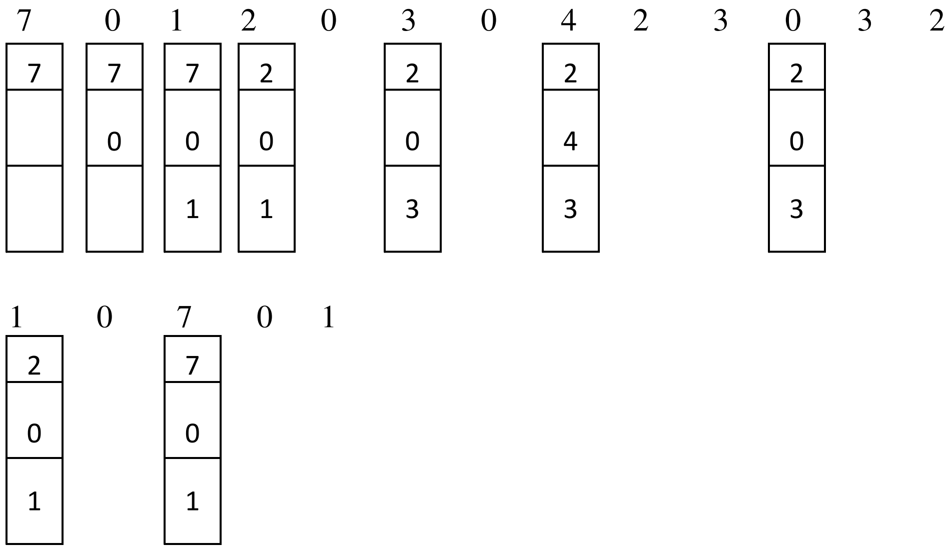
When a page fault occurs, the operating system has to choose a page to remove from memory to take room for the page that has to be brought in. If the page to be removed has been modified while in memory, it must be re-written to the disk. It would be possible to pick a random page but system performance is much better is a page that is not heavily used is chosen.

The Optimal Page Replacement Algorithm

The best possible page replacement algorithm is easy to describe but impossible to implement At the moment that a page fault occurs, replace the page that will not be used for the longest period of time. For example if a page will not be used for 8million instructions and another page will not be used for 6 million instructions remove the former page.

Example: For 3 page frames and 8 page system

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



Number of page faults = 9

Advantage: Guarantees the lower possible page fault rate.

Disadvantage: Unrealizable

The First-In, First-Out (FIFO) Page Replacement Algorithm

When page must be replaced, the oldest page is chosen.

7	0	1	2	0	3	0	4	2	3	0	3	2	1	0	7
7	7	7	2		2	2	4	4	4	0			0		0
	0	0	0		3	3	3	2	2	2			1		1
		1	1		1	0	0	0	3	3			3		7

0 1

Number of page fault = 12

Advantage: Distribute fair chance to all

Disadvantage: Likely to replace heavily used page

Try 7 2 1 0 5 3 0 1 4 1 2 0 (OP = 8, FIFO = 10)

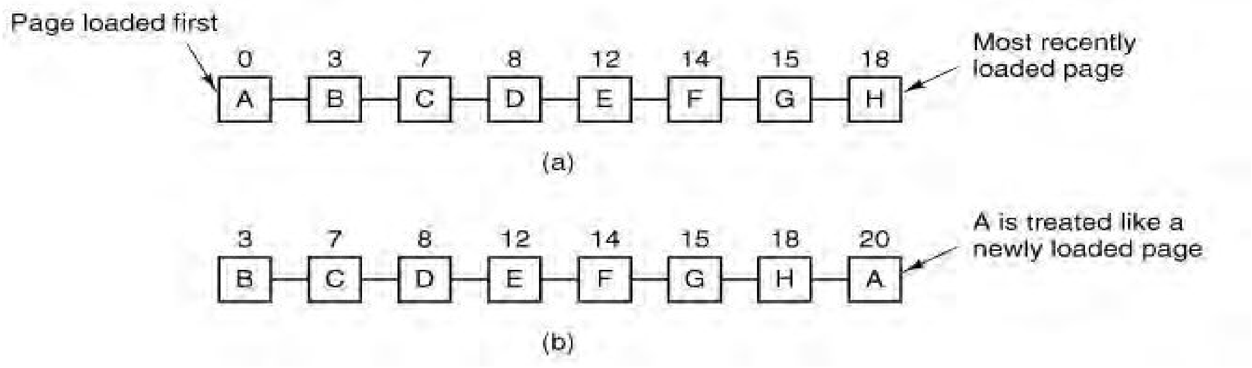
The Second Chance Page Replacement Algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page with the concept of reference R bit of the oldest page.

If R = 0, the page is old and unused, so replace it.

If R= 1, the page is put at the end of list of pages and set R= 0.

This operation is called second chance.

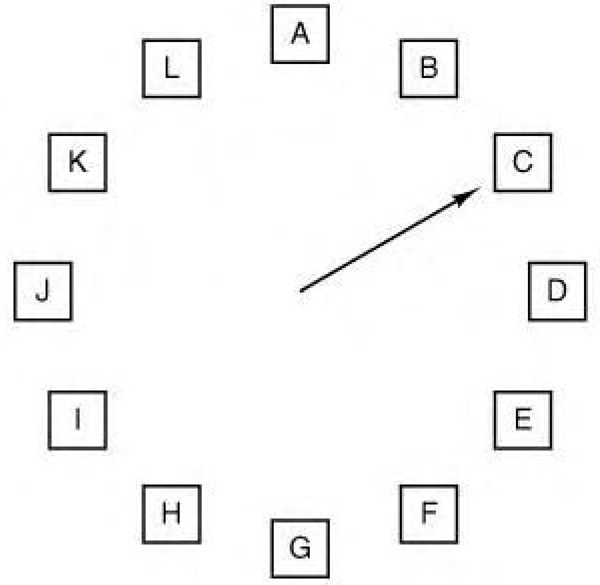


Advantage: Big improvement in FIFO

Disadvantage: If all pages have been referenced, second chance degenerate to pure FIFO.

Second Chance and Clock page replacement algorithm

Second chance is insufficient because it is constantly moving pages around the list. A better approach is to keep all the page frames on a circular list in the form of a clock. A hand points to the oldest page, when a page fault occurs, the page being pointed to by be inspected. If its R bit is 0, the page is evicted, the new page is inserted into the clock in its place and hand is advanced to the next page by setting its R bit to 0. This process is repeated until a page is found with R = 0.



When a page fault occurs,
the page the hand is
pointing to is inspected.
The action taken depends
on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

Not Recently Used (NRU)

Pages not recently used are not likely to be used in near future and they must be replaced with incoming pages. To keep useful statistics about which page are being used and which page are not, it uses two status bits, Reference bit(R) and modified bit (M).

- R is set when the page is referenced (read or write)
- M is set when the page is modified.

			R	M
➢ Class 0	Not referenced	Not modified	0	0
➢ Class 1	Not Referenced	Modified	0	1
➢ Class 2	Referenced	Not Modified	1	0
➢ Class 3	Referenced	Modified	1	1

Pages in the lowest numbered class should be replaced first; pages with in the same class are randomly selected.

The Least Recently Used (LRU) Page Replacement Algorithm

Recent past is the good indicator of the near future. Concept is pages that have not been used for ages probably may remain unused for long time. Throughout the page that has been unused for longest time.

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

7	0	1	2	0	3	0	4	2	3	0	3	2	1	0	7
7	7	7	2		2		4	4	4	0			1	1	1
	0	0	0		0		0	0	3	3			3	0	0
		1	1		3		3	2	2	2			2	2	7

0 1

Number of page fault = 12

Consider the following page reference string 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6 how many page faults occurs for the following page replacement algorithms assuming page frame size 3, 4, 5 respectively. Remember all frames are initially empty. (LRU, FIFO, OPR)

Answer:

No of Frames	LRU	FIFO	OPR
3	15	16	11
4	10	14	8
5	8	10	7

Belady's Anamoly

More frame size the memory has, the fewer page fault a program will get. Belady discovered a counter example in which FIFO caused more page faults, with four page frame than three which is known as Belady's anamoly. Try for 0 1 2 3 0 1 4 0 1 2 3 4

Stack Algorithm

A paging system can be characterized by three items

i. The reference string of the executing process
ii. The page replacement algorithm
iii. The number of page frames available in the memory

Maintain an internal array M_1 it has as many elements as virtual pages (n).

M is divided into two parts.

The top part with m entries contains all the pages that are currently in the memory. Initially M is empty (no any page have been referenced), as execution begins, the process begins emitting page in the reference string, one at a time. Check to see if the page is in memory, if not page fault, if the top part is empty, insert the page else a page replacement algorithm is invoked to remove a page from memory.

0 2 1 3 5 4 6 3 7 4 7 3 3 5 5 3 1 1 1 7 1 3 4 1

	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1	m=4
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4	n=4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3	Top
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7	part
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5	
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6	Bottom
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	Part
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
	P	P	P	P	P	P	P		P					P			P						P		

- ✓ Virtual address space has 8 pages and physical memory has 4 page frames.
- ✓ The first empty column reflects the state of M before execution begins.
- ✓ Newly loaded page is always entered at the top and everything else moved down.
- ✓ If the page was already in M, all the pages above it moved one position down.
- ✓ Such algorithm removes Belady's anamoly.

Questions

- ✓ Differentiate internal and external fragmentation?
- ✓ Explain segmentation?
- ✓ What hardware is required for paging?
- ✓ Write a note on virtual memory?
- ✓ Explain Page replacement algorithm in detail?
- ✓ Compare demand paging and segmentation?
- ✓ Explain page replacement algorithms?
- ✓ Define and explain Virtual Memory?
- ✓ Explain advantages and disadvantages Demand Paging
- ✓ Define and explain Performance of demand paging?
- ✓ Describe Page Replacement Algorithms?
- ✓ Explain various Allocation Algorithms?
- ✓ When do page fault occur? Describe the actions taken by an OS when a page fault occurs.
- ✓ A computer with 32 bit address uses a two level page table. Virtual addresses are split into a 9 bit top level page table field, 11-bit second level page table field and offset. How large the

pages? How much maximum space required when page tables loaded into memory if each entry required when page tables loaded into memory if each entry required when page tables loaded into memory if each entry required 4 bytes.

Ans: Twenty bits are used for the virtual page numbers, leaving 12 over for the offset. This yields a 4-KB page. Twenty bits for the virtual page implies 2^{20} pages.

- ✓ Can a page be in two working sets at the same time? Explain.

Ans: If pages can be shared, yes. For example, if two users of a timesharing system are running the same editor at the same time and the program text is shared rather than copied, some of those pages may be in each user's working set at the same time.

- ✓ If a page is shared between two processes, is it possible that the page is read-only for one process and read-write for the other? Why or why not?

Ans: It is possible. Assuming that segmentation is not present, the protection information must be in the page table. If each process has its own page table, each one also has its own protection bits. They could be different.

- ✓ If FIFO page replacement is used with four page frames and eight pages, how many page faults will occur with the reference string 0172327103 if the four frames are initially empty? Now repeat this problem for LRU.

Ans: FIFO yields 6 page faults; LRU yields 7.

INPUT/OUTPUT

One of the main functions of an operating system is to control all the computer's I/O (Input/Output) devices. It issues commands to the devices, catch interrupts, and handle errors.

PRINCIPLES OF I/O HARDWARE

I/O Devices

I/O devices can be roughly divided into two categories: **block devices** and **character devices**.

A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 bytes to 32,768 bytes. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. Disks are the most common block devices. If the tape drive is given a command to read block *N*, it can always rewind the tape and go forward until it comes to block *N*. Operation is analogous to a disk doing a seek, except that it takes much longer.

The other type of I/O device is the character device. A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices.

Categories of I/O Devices

1. Human readable

2. Machine readable

3. Communication

1. Human Readable is suitable for communicating with the computer user. Examples are printers, video display terminals, keyboard etc.

2. Machine Readable is suitable for communicating with electronic equipment. Examples are disk and tape drives, sensors, controllers and actuators.
3. Communication is suitable for communicating with remote devices. Examples are digital line drivers and modems.

Differences between I/O Devices

1. Data rate : there may be differences of several orders of magnitude between the data transfer rates.

2. Application: Different devices have different use in the system.

3. Complexity of Control: A disk is much more complex whereas printer requires simple control interface.

4. Unit of transfer: Data may be transferred as a stream of bytes or characters or in larger blocks.

5. Data representation: Different data encoding schemes are used for different devices.

6. Error Conditions: The nature of errors differs widely from one device to another.

POLLING

Protocol for interaction between the host and a controller. The controller indicates its state through the busy bit in the status register. (Recall that to set a bit means to write a 1 into the bit, and to clear a bit mean to write a 0 into it.)

The controller sets the busy bit when it is busy working, and clears the busy bit when it is ready to accept the next command. The host signals its wishes via the command-ready bit in the command register. The host sets the command-ready bit when a command is available for the controller to execute.

For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the Busy.
5. The controller reads the command register and sees the write command.
6. It reads the data-out register to get the byte, and does the I/O to the device.
7. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

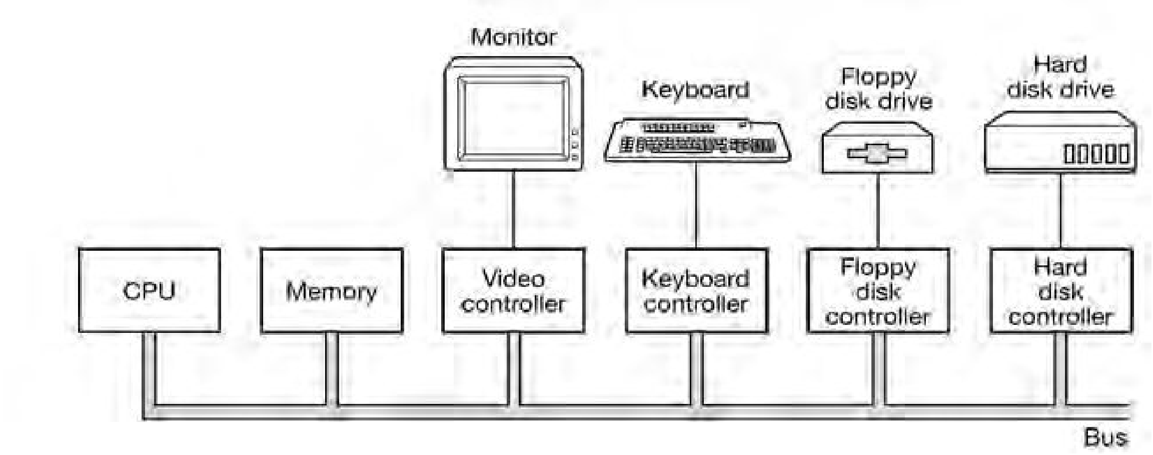
The host is busy-waiting or polling: It is in a loop, reading the status register over and over until the busy bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task

Device Controllers

I/O units consist of a mechanical component and an electronic component. The electronic component is called the **device controller** or **adapter**. The mechanical component is the device itself. The controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in the registers. Typically there are 4 types of registers for this task.

i. Status Register: Contains bits that can be read by the host, these bits indicate such as whether the current command has compiled, whether a byte has been available to read etc.
ii. Control Register: Can be written by the host to start a command
iii. Data in register: is read by host to get input.
iv. Data out register: is written by host to send output.

This arrangement is shown in Figure.



The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged. Many controllers can handle two, four, or even eight identical devices.

The controller's job is to convert the serial bit stream (starting with a **preamble**, then the 4096 bits in a sector, and finally a checksum, also called an **Error-Correcting Code (ECC)**. Preamble contains the cylinder and sector number, the sector size, and similar data, as well as synchronization information and written after disk formatting.) into a block of bytes and perform any error correction necessary. The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller. After its checksum has been verified and the block declared to be error free, it can then be copied to main memory.

A computer system contains a multitude of I/O devices and their respective controllers:

- ✓ Network card
- ✓ Graphics adapter
- ✓ Disk controller
- ✓ DVD-ROM controller
- ✓ Serial port
- ✓ USB
- ✓ Sound card

Direct Memory Access

A special control unit may be provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called Direct Memory Access (DMA).

DMA can be used with either polling or interrupt software. DMA is particularly useful on devices like disks, where many bytes of information can be transferred in single I/O operations. When used in conjunction with an interrupt, the CPU is notified only after the entire block of data has been transferred. For each byte or word transferred, it must provide the memory address and all the bus signals that control the data transfer.
Interaction with a device controller is managed through a device driver.

Device drivers are part of the operating system, but not necessarily part of the OS kernel. The operating system provides a simplified view of the device to user applications (e.g., character devices vs. block devices in UNIX). In some operating systems (e.g., Linux), devices are also accessible through the /dev file system.

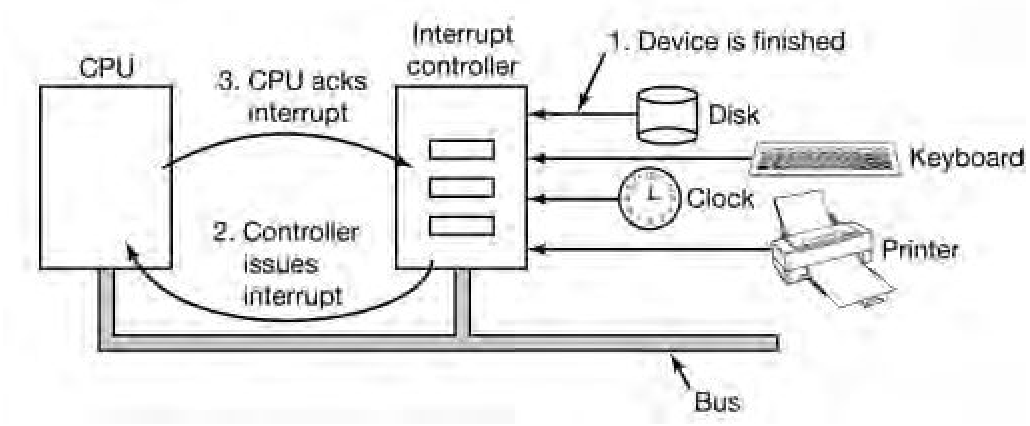
In some cases, the operating system buffers data that are transferred between a device and a user space program (disk cache, network buffer). This usually increases performance, but not always.

Interrupts

It may be more efficient to arrange for the hardware controller to notify the CPU when the device becomes ready for service, rather than to poll repeatedly until completion of I/O. The hardware mechanism that enables a device to notify the CPU is called interrupt.

Working Mechanism of Interrupts

When I/O device has finished the work given to it, it causes an interrupt. It does this by asserting a signal on a bus, that it has been assigned. The signal detected by the interrupt controller then decides what to do? If no other interrupts are pending, the interrupt controller processes the interrupts immediately, if another is in progress, then it is ignored for a moment.
To handle the interrupt, the controller puts a number of address lines specifying which device wants attention and asserts a signal that interrupt the CPU. The number of address lines is used as index to a table called interrupt vector to fetch a new program counter, this program counter points to the start of corresponding service procedure.



Non-Mask able interrupts

Reserved for events, such as unrecoverable memory errors.

Mask able interrupts

Can be turned off by CPU before the execution of critical instruction sequence that must not be interrupted.

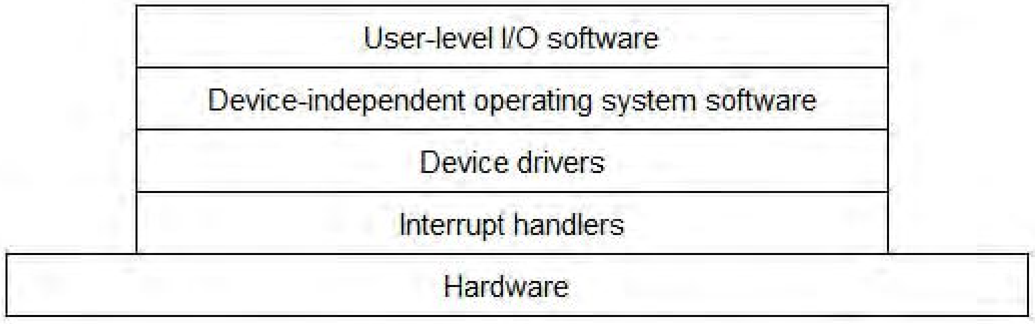
Goals of I/O Software

1. Device independence
2. Uniformed naming
3. Error handling

(Study in details)

Interrupt handlers

When interrupt happens, the interrupt does what it has to be in order to handle the interrupt. The process starts with unblocking the driver that started it; it is done by using semaphores or condition variables in monitors. The model works best if drivers are structured as kernel processes. Steps involved are as follows:



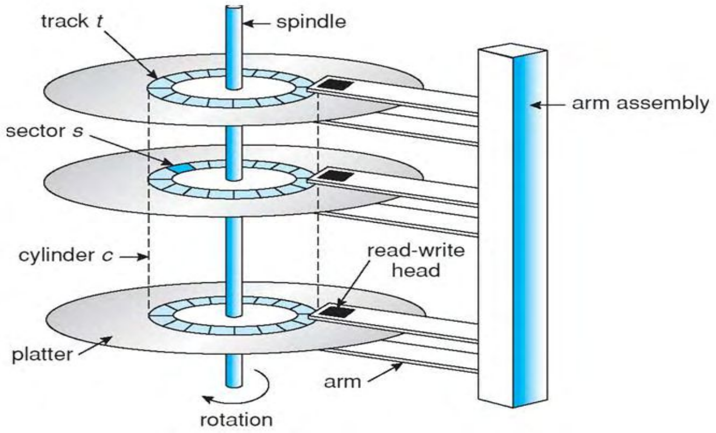
1. Save any register that have not already been saved by interrupt hardware.
2. Set up a context for the interrupt service procedure (TLB, MMU and a page table).
3. Set up a stack for interrupt service procedure.
4. Acknowledge interrupt controller. If there is no centralized interrupt controller, re-enable interrupts.
5. Copy the register from where they were saved (stack) to process table.
6. Run the interrupt service procedure (It will extract information from interrupting device controller register).
7. Choose which process to run next.
8. Set up MMU context for the process to run next (some TLB set may be needed).
9. Load new process register.
10. Start running new process.

Device Drivers

I/O device attached to a computer need some specific code to control it. This code is called device driver and is generally written by its manufacturer and delivered with device itself.

Disk Structure

Modern disk are addressed as large as one dimensional array of logical blocks. The size of logical block is mapped onto the sectors of disk sequentially; sector zero is the first sector of the first track on the outermost cylinder. As we move from outermost zone, the number of sectors per track decreases.



The drive increases its rotation speed as the head moves from the outer to inner tracks to keep the same rate of data moving under the head. Alternatively, the disk rotation speed can stay constant and the density of the bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disk and is known as constant angular velocity (CAV).

Disk Scheduling

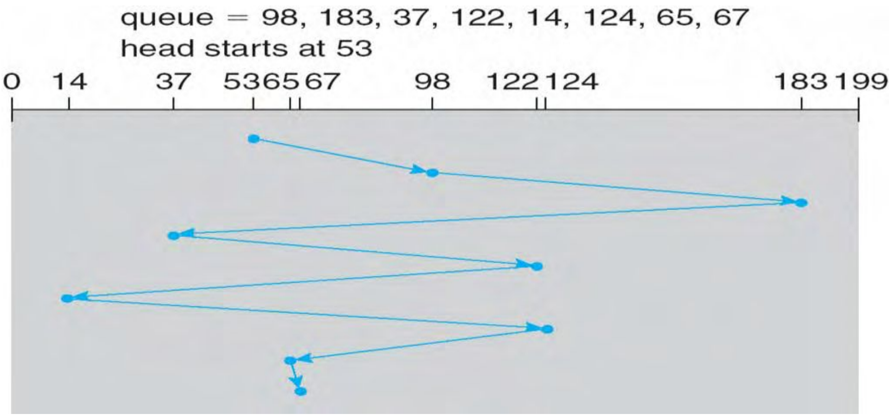
1. The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth
2. Access time has two major components
3. Seek time is the time for the disk are to move the heads to the cylinder containing the desired sector
4. Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head
5. Minimize seek time
6. Seek time » seek distance
7. Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer
8. Several algorithms exist to schedule the servicing of disk I/O requests
9. We illustrate them with a request queue (0-199)

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

Head pointer 53

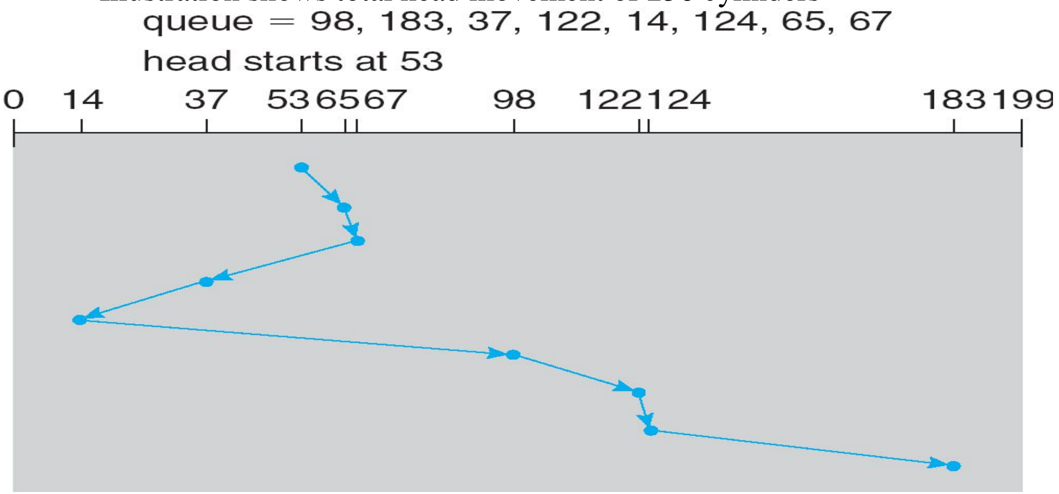
FCFS (First Come First Serve)

- The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order. This strategy has the advantage of being fair, because every request is honored and the requests are honored in the order received. With FIFO, if there are only a few processes that require access and if many of the requests are to clustered file sectors, then we can hope for good performance.
- Illustration shows total head movement of 640 cylinders



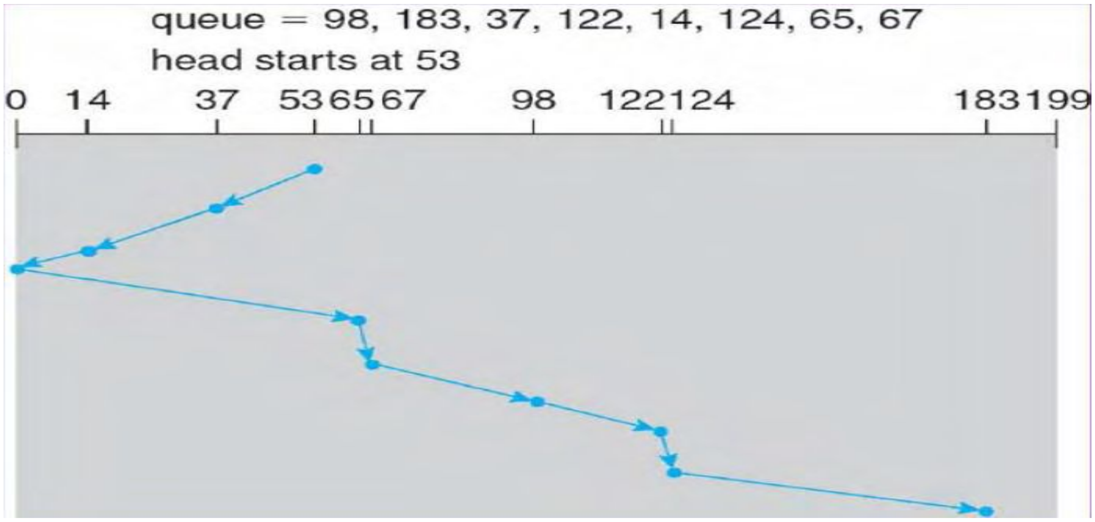
SSTF

- The SSTF policy is to select the disk I/O request the requires the least movement of the disk arm from its current position. Scan With the exception of FIFO, all of the policies described so far can leave some request unfulfilled until the entire queue is emptied. That is, there may always be new requests arriving that will be chosen before an existing request.
- The choice should provide better performance than FCFS algorithm.
- Under heavy load, SSTF can prevent distant request from ever being serviced. This phenomenon is known as starvation. SSTF scheduling is essentially a form of shortest job first scheduling. SSTF scheduling algorithms are not very popular because of two reasons.
    1. Starvation possibly exists.
    2. It increases higher overheads.
- Illustration shows total head movement of 236 cylinders



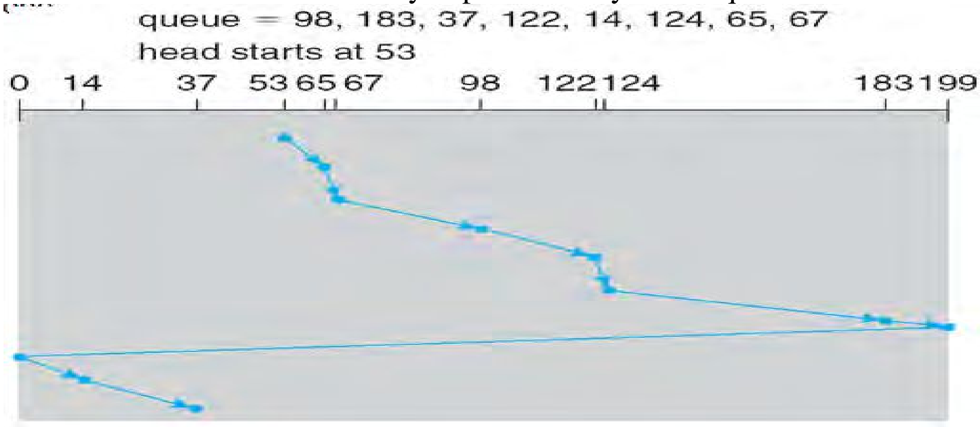
- The scan algorithm has the head start at track 0 and move towards the highest numbered track, servicing all requests for a track as it passes the track. The service direction is then reserved and the scan proceeds in the opposite direction, again picking up all requests in order.

- SCAN algorithm is guaranteed to service every request in one complete pass through the disk. SCAN algorithm behaves almost identically with the SSTF algorithm. The SCAN algorithm is sometimes called elevator algorithm.
- Illustration shows total head movement of 208 cylinders



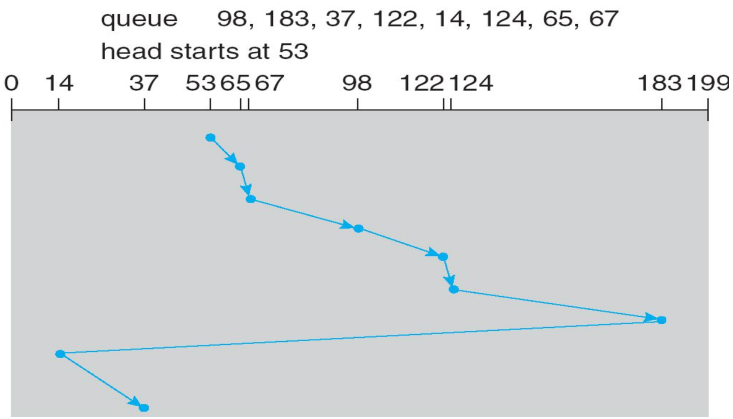
C-SCAN

- The C-SCAN policy restricts scanning to one direction only. Thus, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again.
- This reduces the maximum delay experienced by new requests.



LOOK

- Look same as SCAN and C-LOOK same as C-SCAN except
- Start the head moving in one direction. Satisfy the request for the closest track in that direction when there is no more request in the direction, the head is traveling, reverse direction and repeat.

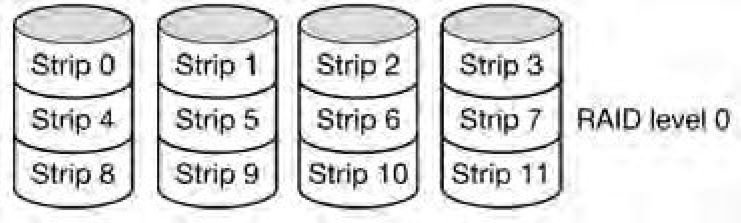


RAID Structure

- RAID – multiple disk drives provides reliability via redundancy
- Increases the mean time to failure
- Frequently combined with NVRAM to improve write performance
- RAID is arranged into six different levels
- Several improvements in disk-use techniques involve the use of multiple disks working co-operatively
- Disk striping uses a group of disks as one storage unit
- RAID schemes improve performance and improve the reliability of the storage system by storing redundant data
- Mirroring or shadowing (RAID 1) keeps duplicate of each disk
- Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance and high reliability
- Block interleaved parity (RAID 4, 5, 6) uses much less redundancy
- RAID within a storage array can still fail if the array fails, so automatic replication of the data between arrays is common
- Frequently, a small number of hot-spare disks are left unallocated, automatically replacing a failed disk and having data rebuilt onto them

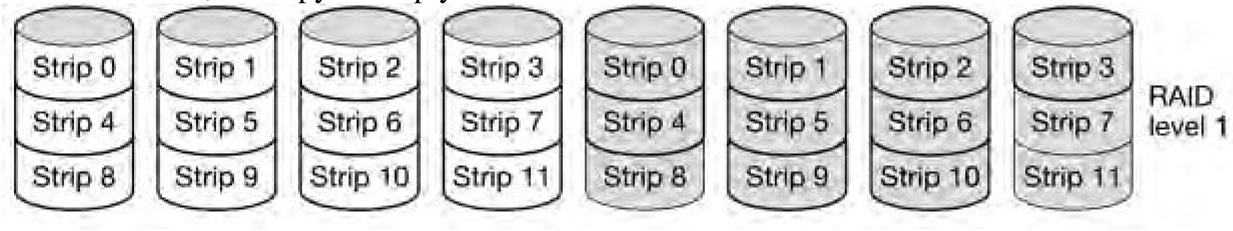
RAID Level 0

- Divided into strips of k sectors
- If software issues a command to read a data block, consisting four consecutive strips, the RAID controller will break this command up into four separate commands, one for each of four disks and then operate in parallel.
- If request is larger than the number of drives, some drives will get multiple requests. So that when they finish the first request, they start the second one.
- Problem (no redundancy)



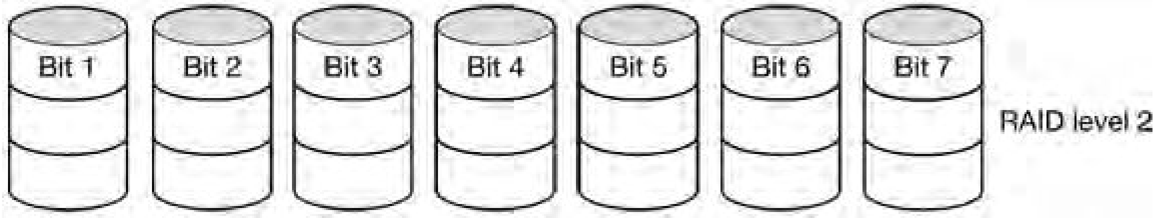
RAID Level 1

- Mirroring
- It duplicates all the disks, so there are four primary disk and four secondary disks.
- On write every strip is written twice
- On read either a copy can be read
- If a disk crashes, the copy is simply used instead

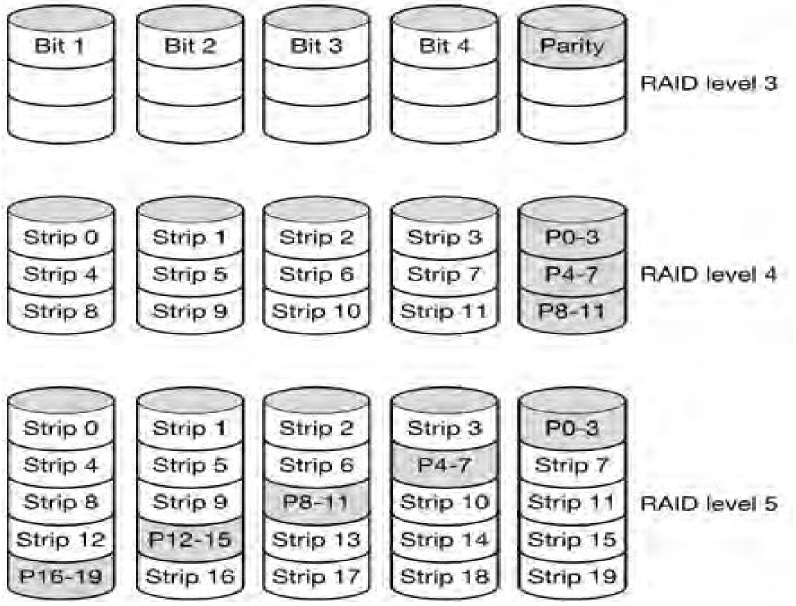


RAID Level 2

- Each byte in memory have parity bit associated with it that records whether the number of bits in the byte set to 1 is even (parity=0) or odd (parity=1)
- If one of the number of bits in the bytes get damaged, (either a 1 becomes 0 or 0 becomes 1) the parity of byte changes and thus will not match the computed parity.
- Thus all single bit errors are detected by the memory system.
- Can be corrected by error correcting code.
- Use four disks to store data and 3 parity disks.



RAID Level 3, 4 and 5

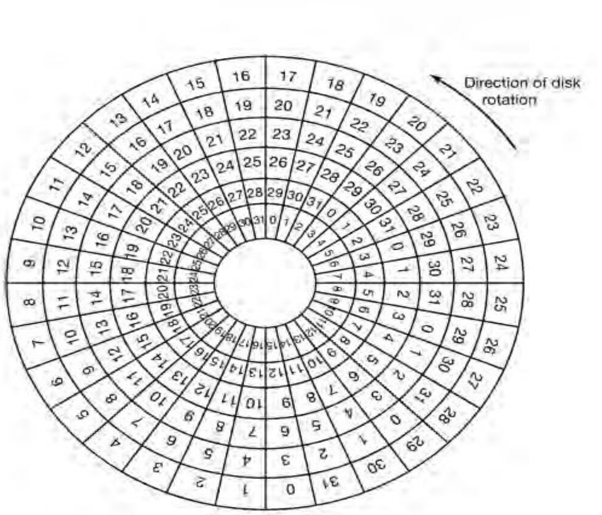


(Explain yourself)

Disk Formatting

A new magnetic disk is a blank state. Before a disk can store data it must be divided into sectors that disk controller can read and write. This process is called low-level or physical formatting.

- Disk must be formatted before storing data.
- Disk must be divided into sectors that the disk controllers can read/write.
- Low level formatting files the disk with a special data structure for each sector.
- Data structure consists of three fields: header, data area and trailer.
- Header and trailer contain information used by the disk controller.
- Sector number and Error Correcting Codes (ECC) contained in the header and trailer.
- For writing data to the sector – ECC is updated.
- For reading data from the sector – ECC is recalculated.
- Low level formatting is done at factory.



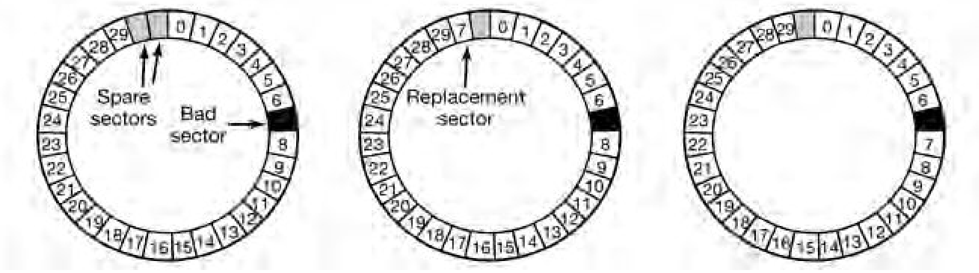
Since ECC contains enough information, if only few bits of data have been corrupted, controller identifies the bits and reports soft error. Then controller automatically does ECC processing whenever sector is read or write. The final step in preparing disk for use is to perform high- level formatting or Logical Formatting. It is carried on in two steps

1. Partitioning: Grouping of one or more cylinders. The operating system then treats it as a single disk.
2. Logical formatting: Creation of file system.

Note: On Pentium and most other computer, sector zero contains the master boot record, which contains some boot code and partition table. For Pentium, the partition table has room for four partitions.

Error Handling

Manufacturing defects introduce bad sectors; if defect is small (say few bits). Then the hard drive is shipped and ECC corrects the error every time the sector is accessed. If error is bigger it cannot be masked.



There are two ways for error correction

1. When one of the sectors is bad the controller remaps between the bad and spare sectors as in figure.
2. If controller cannot remap, the operating system does the same thing in software.

Stable Storage

Since some disks make errors, RAID can help this. But writing on bad sectors cannot be avoided by using RAID. We need a disk that works with no error but it is not achievable. One thing that can be done is when a write is issued; the disk either writes the data correctly or does nothing. This technique is known as stable storage and is implemented in software.

Stable storage uses a pair of identical disks with corresponding blocks working together to form one error free block. In absence of errors, both drives are the same. To achieve goal following three operations are defined.

1. **Stable Writes:** A stable write consists of first writing the block on drive 1, then reading it back to verify that it was written correctly. If it was not written correctly, the write and reread are done again up to *n* times until they work. After *n* consecutive failures, the block is remapped onto a spare and the operation repeated until it succeeds, no matter how many spares have to be tried. After the write to drive 1 has succeeded, the corresponding block on drive 2 is written and reread, repeatedly if need be, until it, too, finally succeeds.
2. **Stable reads:** A stable read first reads the block from drive 1. If this yields an incorrect ECC, the read is tried again, up to n times. If all of these give bad ECCs, the corresponding block is read from drive 2. Given the fact that a successful stable write leaves two good copies of the block behind, and our assumption that the probability of the same block spontaneously going bad on both drives in a reasonable time interval is negligible, a stable read always succeeds.
3. **Crash recovery:** After a crash, a recovery program scans both disks comparing corresponding blocks. If a pair of blocks is both good and the same, nothing is done. If one of them has an ECC error, the bad block is overwritten with the corresponding good block. It a pair of blocks are both good but different, the block from drive 1 is written onto drive 2.

Terminal Hardware

Terminals are hardware devices containing both a keyboard and display which communicates using a serial interface one bit at a time. These terminals use 9 pin or 25 pin connectors. Among these one is used for sending, one for receiving and one is grounded, others are used for various other control functions. Lines in which characters are sent 1 bit at a time are called serial line.

To send a character using RS 232 terminal, computer must transmit one bit at a time, prefixed by a start bit and followed by 1 or 2 stop bits to delimit the character. The RS 232 terminals are character oriented.

Since both computers and terminals work internally with whole characters but must communicate over a serial line a bit at a time, chips have been developed to do the character to serial and serial to character conversion. They are called UARTs (Universal Asynchronous Receiver Transmitters). UARTs are attached to computer by plugging RS-232 interface card into bus. On many computers one or two serial ports are built on parent board.

Input Software

The basic job of input software is to collect input and pass it to program. i.e. collect input and pass it upward as it is or corrects input for example dste → date.

The first task of input software is to collect characters (keyboard). If each keystroke causes interrupt, driver can acquire the characters, if interrupts are turned into low level software, it is possible to put newly acquired character into central buffer of pool and message is used to tell the driver that something has arrived.

The other approach is to do the buffering directly in the terminal data structure itself, with no central pool of buffers. Since it is common for users to type a command that will take a little while (say, recompiling and linking a large binary program) and then type a few lines ahead, to be safe the driver should allocate something like 200 characters per terminal. In a large-scale timesharing system with 100 terminals, allocating 20K all the time for type ahead is clearly overkill, so a central buffer pool with space for perhaps 5K is probably enough. On the other hand, a dedicated buffer per terminal makes the driver simpler (no linked list management) and is to be preferred on personal computers with only one keyboard.

Although the keyboard and display are logically separate devices, many users have grown accustomed to seeing the characters they have just typed appear on the screen. Some (older) terminals oblige by automatically displaying (in hardware) whatever has just been typed, which is not only a nuisance when passwords are being entered but greatly limits the flexibility of sophisticated editors and other programs. Fortunately, with most terminals, nothing is automatically displayed when a key is struck. It is entirely up to the software in the computer to display the character, if desired. This process is called **echoing**.

Output Software

Output is simpler than input. For the most part, the computer sends characters to the terminal and they are displayed there. Usually, a block of characters, for example, a line, is written to the terminal in one system call. The method that is commonly used for RS-232 terminals is to have output buffers associated with each terminal. The buffers can come from the same pool as the input buffers, or be dedicated, as with input. When a program writes to the terminal, the output is first copied to the buffer. Similarly, output from echoing is also copied to the buffer. After all the output has been copied to the buffer, the first character is output, and the driver goes to sleep. When the interrupt comes in, the next character is output, and so on.

FILE SYSTEMS

All computer applications need to store and retrieve information. Furthermore, it must not go away when a computer crash kills the process. A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time.

Thus we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information concurrently.

The usual solution to all these problems is to store information on disks and other external media in units called **files**. Information stored in files must be **persistent**. A file should only disappear when its owner explicitly removes it.

Files are managed by the operating system; part of the operating system dealing with files is known as the **file system**.

File Naming

Files are an abstraction mechanism. The most important characteristic of any abstraction mechanism is the way the objects being managed are named. When a process creates a file, it gives the file a name and can be accessed by other processes using its name.

The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names. Frequently digits and special characters are also permitted, so names like *2*, *urgent!*, and *Fig.2-14* are often valid as well. Many file systems support names as long as 255 characters. Some file systems distinguish between upper and lower case letters, whereas others do not

Many operating systems support two-part file names, with as in *prog.c*. The part following the period is called the **file extension** and usually indicates something about the file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in *prog.c.Z*, where *.Z* is commonly used to indicate that the file (*prog.c*) has been compressed using the Ziv-Lempel compression algorithm. Some of the more common file extensions and their meanings are shown in Figure.

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Computer serve Graphical Interchange Format image
file.hlp	Help file

file.html	World Wide Web Hyper Text Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

Figure Some typical file extensions.

File Structure

Files can be structured in any of several ways. Three common possibilities are depicted in Fig. (a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

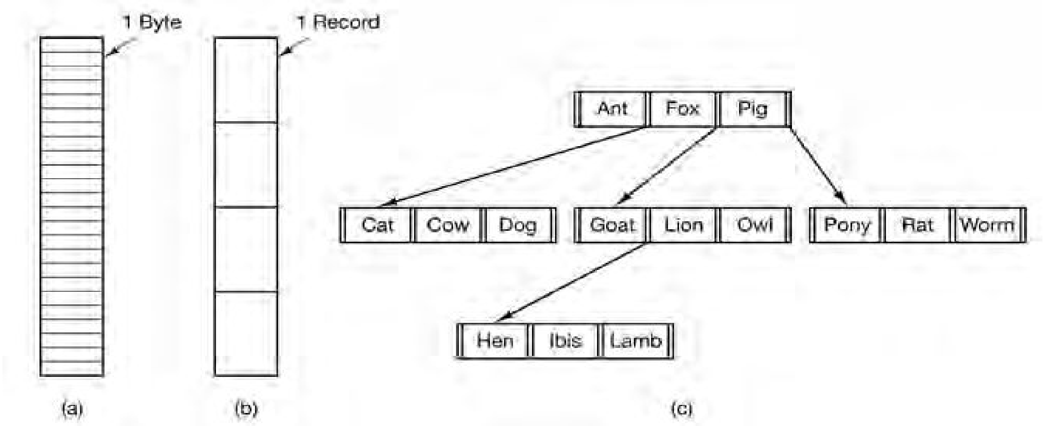


Figure Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

The first step up in structure is shown in (b). In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record.

The third kind of file structure is shown in (c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key** field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

File Types

Many operating systems support several types of files. A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts: a name and an extension. Following table gives the file type with usual extension and function.

File Type	Usual Extension	Function
Executable	exe, com, bin	Read to run machine language program.
Object	obj, o	Compiled, machine language, not linked
Source Code	c, cc, java, pas asm, a	Source code in various language
Text	txt, doc	Textual data, documents

File Access

Early operating systems provided only one kind of file access: **sequential access**. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order. Sequential files were convenient when the storage medium was magnetic tape, rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key, rather than by position. Files whose bytes or records can be read in any order are called **random access files**. They are required by many applications.

Modern operating systems do not make this distinction. All their files are automatically random access.

File Attributes

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was created and the file's size. These extra items the file's **attributes**. The list of attributes varies considerably from system to system.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner

Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure Some possible file attributes.

File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Most common system calls relating to files.

1. Create. The file is created with no data.
2. Delete. When the file is no longer needed, it has to be deleted to free up disk space.
3. Open. Before using a file, a process must open it.
4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space.
5. Read. Data are read from file.
6. Write. Data are written to the file, again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. Append. This call is a restricted form of write. It can only add data to the end of the file.
8. Seek. For random access files, a method is needed to specify from where to take the data.
9. Get attributes. Processes often need to read file attributes to do their work.
10. Set attributes. Some of the attributes are user settable and can be changed after the file has been created.
11. Rename. It frequently happens that a user needs to change the name of an existing file.

Memory-Mapped Files

Many programmers feel that accessing files as shown above is cumbersome and inconvenient, especially when compared to accessing ordinary memory. Conceptually, we can imagine the existence of two new system calls, map and unmap. The former gives a file name and a virtual address, which causes the operating system to map the file into the address space at the virtual address.

For example, suppose that a file, *f*, of length 64 KB, is mapped into the virtual address starting at address 512K. Then any machine instruction that reads the contents of the byte at 512K gets byte 0 of the file, and so on. Similarly, a write to address 512K + 21000 modifies byte 21000 of the file. When the process terminates, the modified file is left on the disk just as though it had been changed by a combination of seek and write system calls.

What actually happens is that the system's internal tables are changed to make the file become the backing store for the memory region 512K to 576K. Thus a read from 512K causes a page fault bringing in page 0 of the file. Similarly, a write to 512K + 1100 causes a page fault, bringing in the page containing that address, after which the write to memory can take place. If that page is ever evicted by the page replacement algorithm, it is written back to the appropriate place in the file. When the process finishes, all mapped, modified pages are written back to their files.

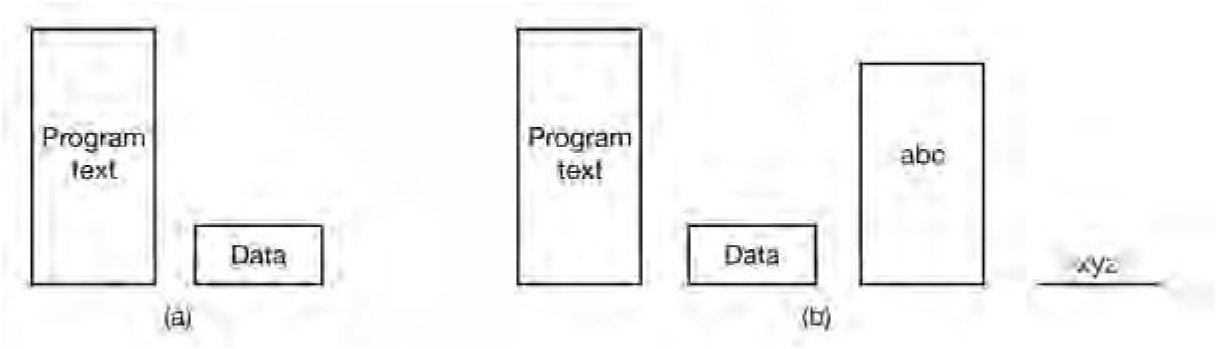


Figure (a) A segmented process before mapping files into its address space. (b) The process after mapping an existing file *abc* into one segment and creating a new segment for file *xyz.*

At this point the process can copy the source segment into the destination segment using an ordinary copy loop. No read or write system calls are needed. When it is all done, it can execute the unmap system call to remove the files from the address space and then exit. The output file, *xyz*, will now exist, as though it had been created in the conventional way.

Although file mapping eliminates the need for I/O and thus makes programming easier, it introduces a few problems of its own. First, it is hard for the system to know the exact length of the output file, *xyz*, in our example. It can easily tell the number of the highest page written, but it has no way of knowing how many bytes in that page were written. Suppose that the program only uses page 0, and after execution all the bytes are still 0 (their initial value). Maybe *xyz* is a file consisting of 10 zeros. Maybe it is a file consisting of 100 zeros. Maybe it is a file consisting of 1000 zeros. Who knows? The operating system cannot tell. All it can do is create a file whose length is equal to the page size.

A second problem can (potentially) occur if a file is mapped in by one process and opened for conventional reading by another. If the first process modifies a page, that change will not be reflected in the file on disk until the page is evicted. The system has to take great care to make sure the two processes do not see inconsistent versions of the file.

A third problem with mapping is that a file may be larger than a segment, or even larger than the entire virtual address space. The only way out is to arrange the map system call to be able to map a portion of a file, rather than the entire file. Although this works, it is clearly less satisfactory than mapping the entire file.

DIRECTORIES

To keep track of files, file systems normally have **directories** or **folders**, which, in many systems, are themselves files.

Single-Level Directory Systems

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the **root directory**, but since it is the only one, the name does not matter much. On early personal computers, this system was common, in part because there was only one user. Interestingly enough, the world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once. This decision was no doubt made to keep the software design simple.

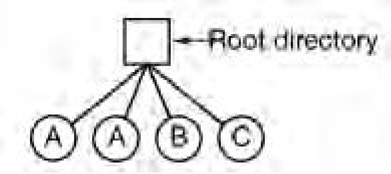
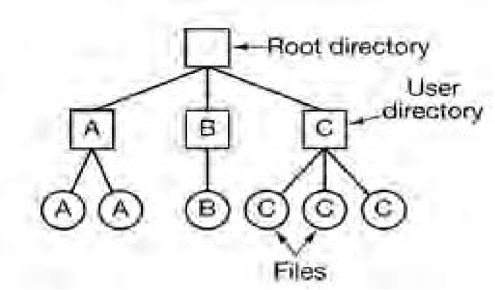


Figure A single-level directory system containing four files, owned by three different people, *A*, *B*, and *C*.

The **problem** with having only one directory in a system with multiple users is that different users may accidentally use the same names for their files

Two-level Directory Systems

To avoid conflicts caused by different users choosing the same file name for their own files, the next step up is giving each user a private directory. In that way, names chosen by one user do not interfere with names chosen by a different user and there is no problem caused by the same name occurring in two or more directories. This design could be used, for example, on a multiuser



computer or on a simple network of personal computers that shared a common file server over a local area network.

Hierarchical Directory Systems

The two-level hierarchy eliminates name conflicts among users but is not satisfactory for users with a large number of files. Even on a single-user personal computer, it is inconvenient. It is quite common for users to want to group their files together in logical ways. A professor for example, might have a collection of files that together form a book that he is writing for one course, a second collection of files containing student programs submitted for another course, a third group of files containing the code of an advanced compiler-writing system he is building, a fourth group of files containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on. Some way is needed to group these files together in flexible ways chosen by the user.

What is needed is a general hierarchy (i.e., a tree of directories). With this approach, each user can have as many directories as are needed so that files can be grouped together in natural ways.

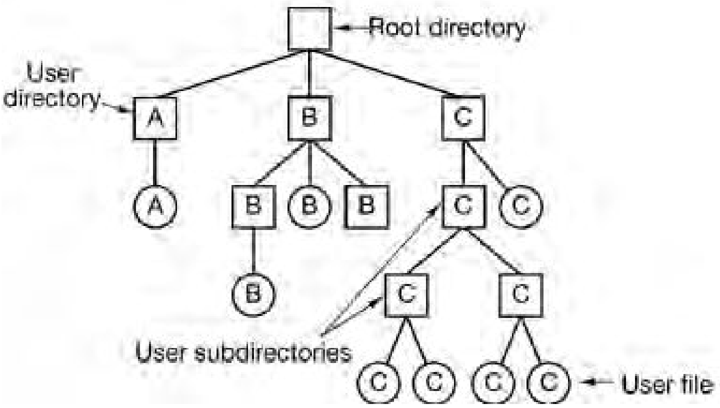


Figure A hierarchical directory system.

The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason, nearly all modern file systems are organized in this manner.

Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file is given an **absolute path name** consisting of the path from the root directory to the file. As an example, the path */usr/ast/mailbox* means that the root directory contains a subdirectory *usr*, which in turn contains a subdirectory *ast*, which contains the file *mailbox*. Absolute path names always start at the root directory and are unique.

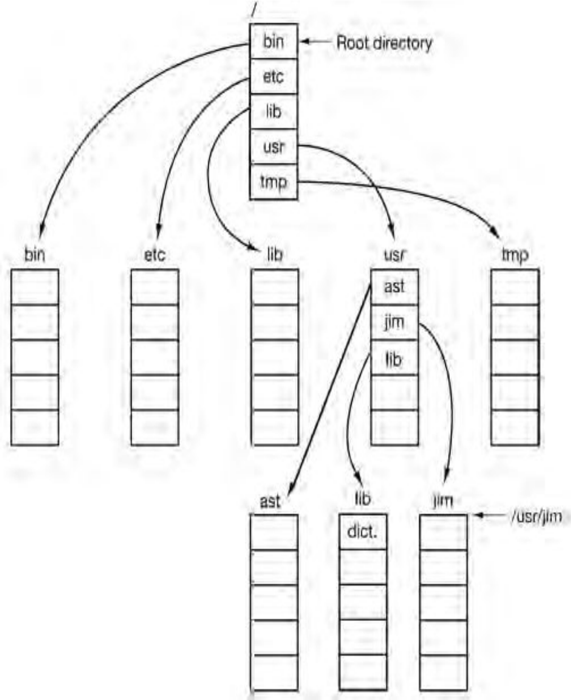


Figure A UNIX directory tree.

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the **relative path name**. This is used in conjunction with the concept of the **working directory** (also called the **current directory**). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is */usr/ast*, then the file whose absolute path is */usr/ast/mailbox* can be referenced simply as *mailbox*.

Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files.

1. **Create:** A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the *mkdir* program).
2. **Delete:** A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot usually be deleted.
3. **Opendir:** Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.
4. **Closedir:** When a directory has been read, it should be closed to free up internal table space.
5. **Readdir:** This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.
6. **Rename:** In many respects, directories are just like files and can be renamed the same way files can be.
7. **Link:** Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link**.
8. **Unlink**: A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

FILE SYSTEM IMPLEMENTATION

File System Layout

File systems are stored on disks. Sector 0 of the disk is called the **MBR** (**Master Boot Record**) which is used to boot the computer. The end of the MBR contains the partition table which gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition.

Other than starting with a boot block, the layout of a disk partition varies strongly from file system to file system. The first one is the **superblock**. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file system type, the number of blocks in the file system, and other key administrative information.

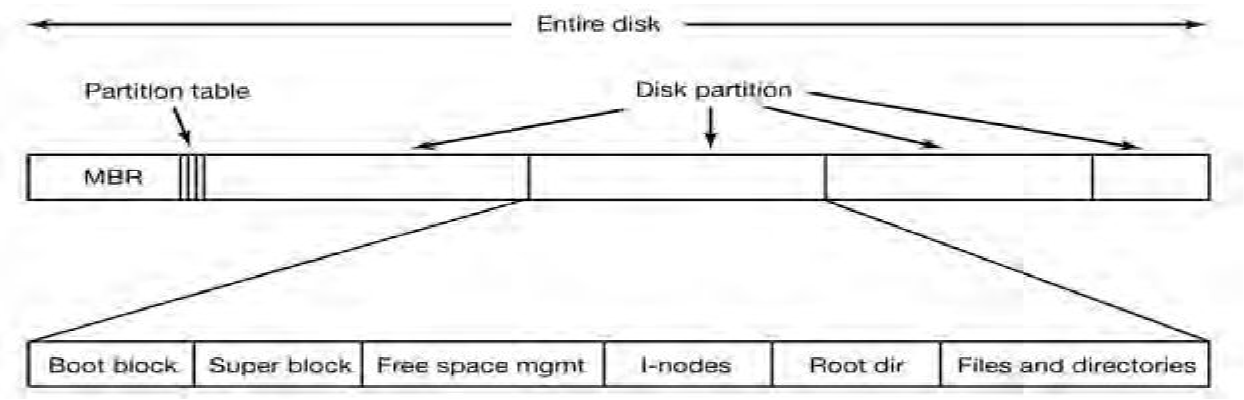


Figure A possible file system layout.

Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section, we will examine a few of them.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.

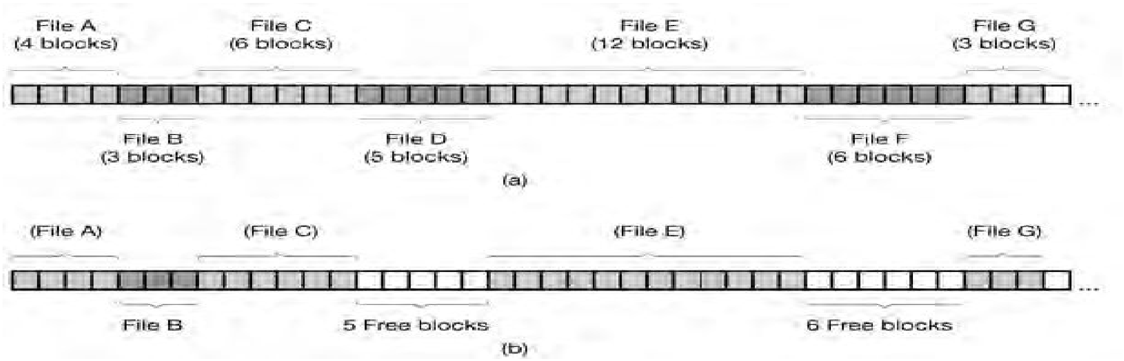


Figure (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file *A*, of length four blocks was written to disk starting at the beginning (block 0). After that a six-block file, *B*, was written starting right after the end of file *A*. Note that each file begins at the start of a new block, so that if file *A* was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one.

Advantage

- Simple to implement
- Read performance is excellent

Disadvantage

- The disk becomes fragmented

Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks. The first word of each block is used as a pointer to the next one. The rest of the block is for data. First few bytes are used to store the address of pointer so whole block cannot be used, as storage for data. i.e. we cannot use memory as power of two.

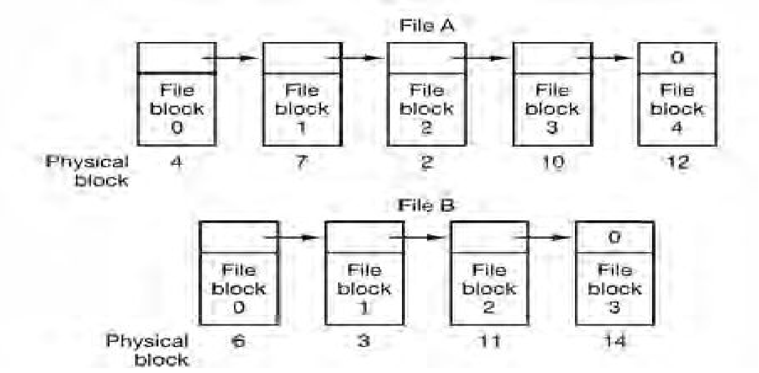


Figure Storing a file as a linked list of disk blocks.

Disadvantage

- External fragmentation on last block.

Linked List Allocation Using a Table in Memory

Taking the pointer word from each disk block and putting it in a table in memory. File *A* uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6, 3, 11, and 14, in that order. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., –1) that is not a valid block number. Such a table in main memory is called a **FAT** (**File Allocation Table**).

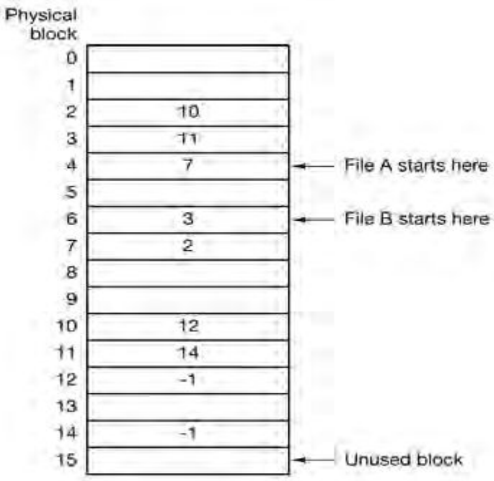


Figure Linked list allocation using a file allocation table in main memory.

Problem: The entire table must be in memory all the time to make it works.

I-nodes

List the attributes and disk address of the block. Which block belongs to which file, associate each file with a data structure called i-node. It is possible with i-node to find all the blocks of the file. The big advantage with this schema is that only i-node is in memory when its corresponding file is open.

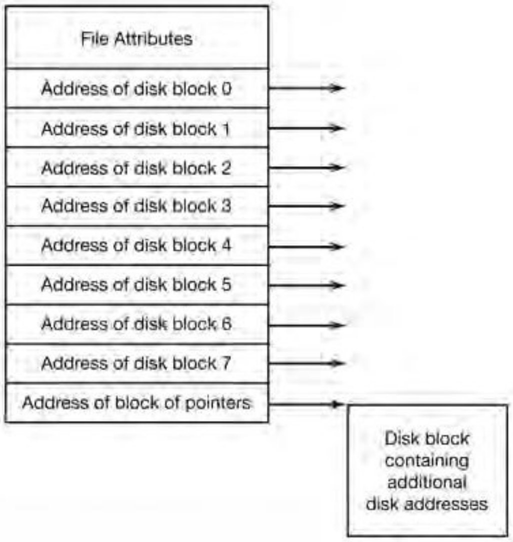


Figure An example of i-node.

Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (contiguous allocation), the number of the first block (both linked list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

Every file system maintains file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Many systems do precisely that. This option is shown in Fig (a).

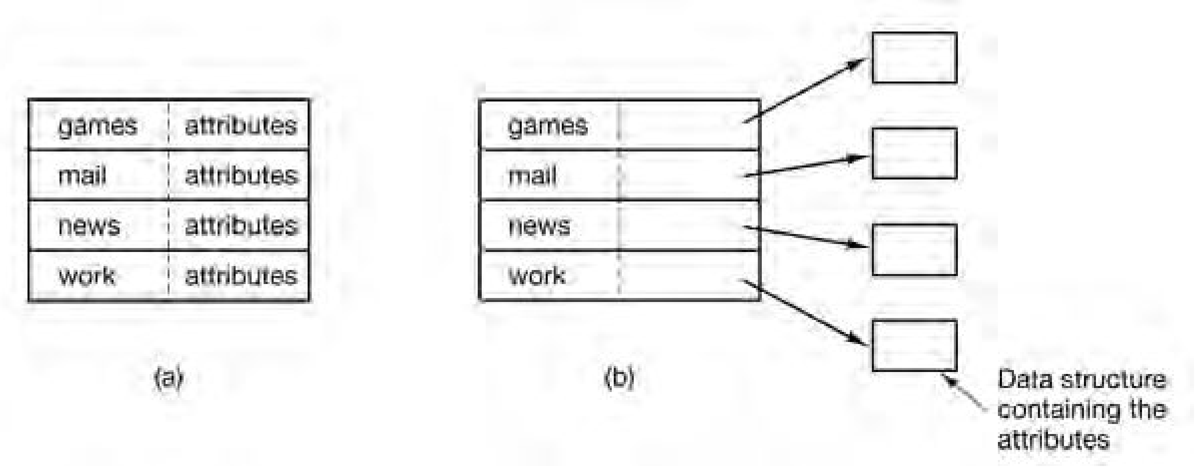


Figure (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries. In that case, the directory entry can be shorter: just a file name and an i-node number. This approach is illustrated in Fig. (b).

To handle the file with too long name following method can be used.

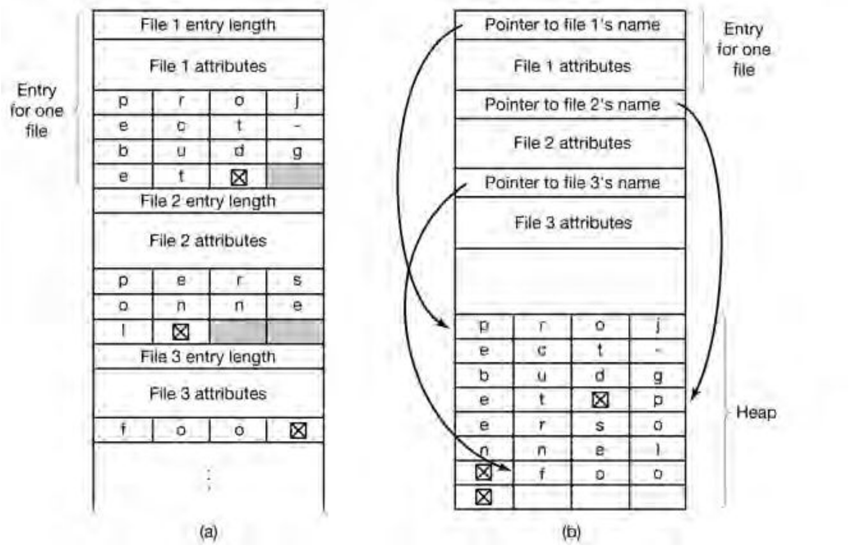
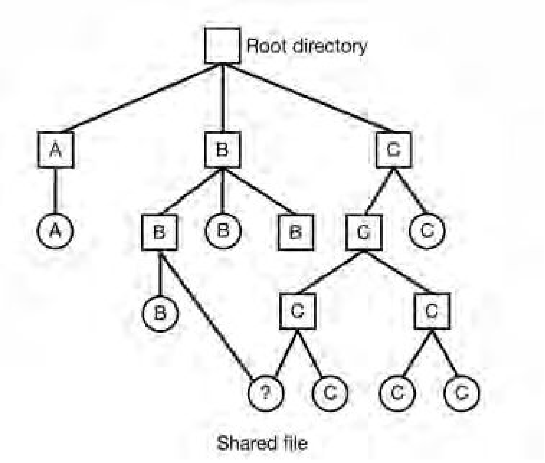


Figure Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

Shared Files



When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Sharing files is convenient, but it also introduces some problems. To start with, if directories really do contain disk addresses, then a copy of the disk addresses will have to be made in *B*'s directory when the file is linked. If either *B* or *C* subsequently appends to the file, the new blocks will be listed only in the directory of the user doing the append. The changes will not be visible to the other user, thus defeating the purpose of sharing.

This problem can be solved in two ways. In the first solution, disk blocks are not listed in directories, but in a little data structure associated with the file itself. The directories would then point just to the little data structure.

In the second solution, *B* links to one of *C*'s files by having the system create a new file, of type LINK, and entering that file in *B*'s directory. The new file contains just the path name of the file to which it is linked. When *B* reads from the linked file, the operating system sees that the file being read from is of type LINK, looks up the name of the file, and reads that file. This approach is called **symbolic linking**.

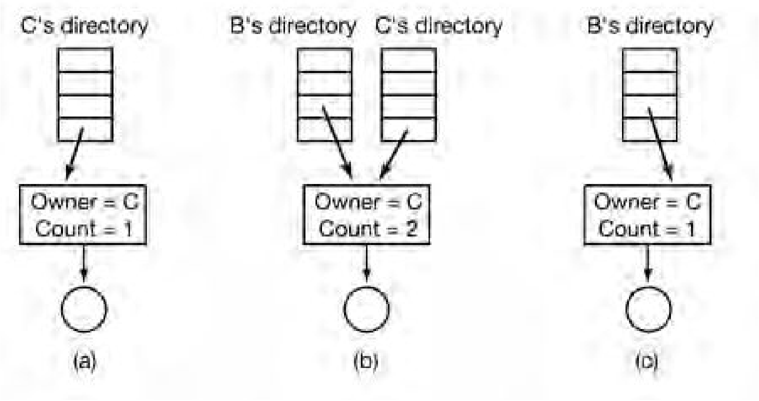


Figure (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

Disk Space Management

Files are normally stored on disk, so management of disk space is a major concern to file system designers. Two general strategies are possible for storing an *n* byte file: *n* consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. The same tradeoff is present in memory management systems between pure segmentation and paging.

As we have seen, storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises of how big the block should be. Having a large allocation unit, such as a cylinder, means that every file, even a 1-byte file, ties up an entire cylinder. On the other hand, using a small allocation unit means that each file will consist of many blocks. Reading each block normally requires seek and a rotational delay, so reading a file consisting of many small blocks will be slow.

As an example, consider a disk with 131,072 bytes per track, a rotation time of 8.33 msec, and an average seek time of 10 msec. The time in milliseconds to read a block of *k* bytes is then the sum of the seek, rotational delay, and transfer times:

$$10 + 4.165 + (k/131072) \times 8.33$$

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is needed for the pointer to the next block). A 16-GB disk needs a free list of maximum 16,794 blocks to hold all 2^{24} disk block numbers. Often free blocks are used to hold the free list.

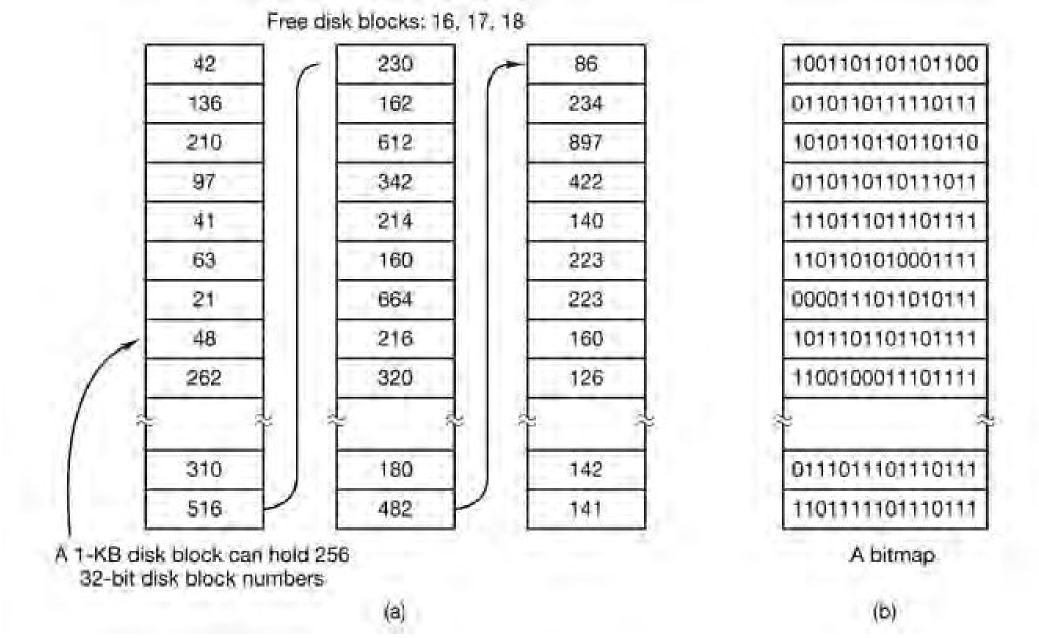


Figure (a) Storing the free list on a linked list. (b) A bitmap.

The other free space management technique is the bitmap. A disk with *n* blocks requires a bitmap with *n* bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa). A 16-GB disk has 2^{24} 1-KB blocks and thus requires 2^{24} bits for the map, which requires 2048 blocks. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked list scheme require fewer blocks than the bitmap. On the other hand, if there are many blocks free, some of them can be borrowed to hold the free list without any loss of disk capacity.

Disk Quotas

To prevent people from hogging too much disk space, multiuser operating systems often provide a mechanism for enforcing disk quotas. The idea is that the system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas

When a user opens a file, the attributes and disk addresses are located and put into an open file table in main memory. Among the attributes is an entry telling who the owner is. Any increases in the file’s size will be charged to the owner’s quota.

A second table contains the quota record for every user with a currently open file, even if the file was opened by someone else. It is an extract from a quota file on disk for the users whose files are currently open. When all the files are closed, the record is written back to the quota file.

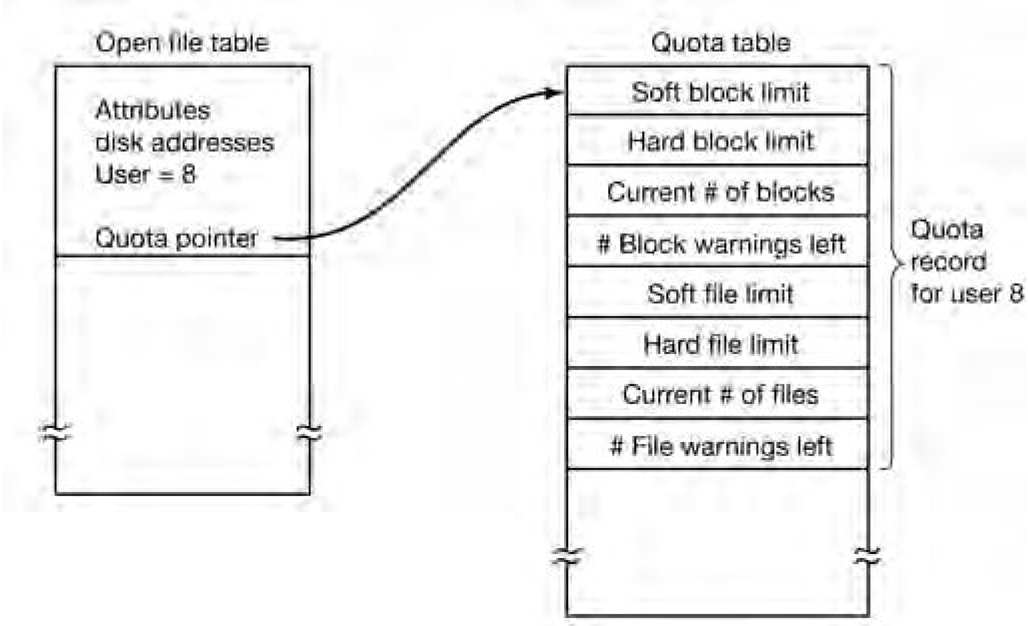


Figure Quotas are kept track of on a per-user basis in a quota table.

File System Reliability

Destruction of a file system is often a far greater disaster than destruction of a computer. If a computer is destroyed by fire, lightning surges, or a cup of coffee poured onto the keyboard, it is annoying and will cost money, but generally a replacement can be purchased with a minimum of fuss.

Floppy disks are generally perfect when they leave the factory, but they can develop bad blocks during use. Hard disks frequently have bad blocks right from the start; it is just too expensive to manufacture them completely free of all defects.

Backups

Most people do not think making backups of their files is worth the time and effort—until one fine day their disk abruptly dies, at which time most of them undergo a deathbed conversion. Companies, however, (usually) well understand the value of their data and generally do a backup at least once a day, usually to tape.

Backups to tape are generally made to handle one of two potential problems:

1. Recover from disaster.
2. Recover from stupidity.

The first one covers getting the computer running again after a disk crash, fire, flood, or other natural catastrophe. The second reason is that users often accidentally remove files that they later need again. This problem occurs so often that when a file is “removed” in Windows, it is not deleted at all, but just moved to a special directory, the **recycle bin**, so it can be fished out and restored easily later. Backups take this principle further and allow files that were removed days, even weeks ago, to be restored from old backup tapes.

File System Consistency

Another area where reliability is an issue is file system consistency. Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can be left in an inconsistent state. This problem is especially critical if some of the blocks that have not been written out are i-node blocks, directory blocks, or blocks containing the free list.

Two kinds of consistency checks can be made: **blocks and files**. To check for block consistency, the program builds two tables, each one containing a counter for each block, initially set to 0. The counters in the first table keep track of how many times each block is present in a file; the counters in the second table record how often each **block is present in the free list** (or the bitmap of free blocks).

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. (a). However, as a result of a crash, the tables might look like Fig. (b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they do waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

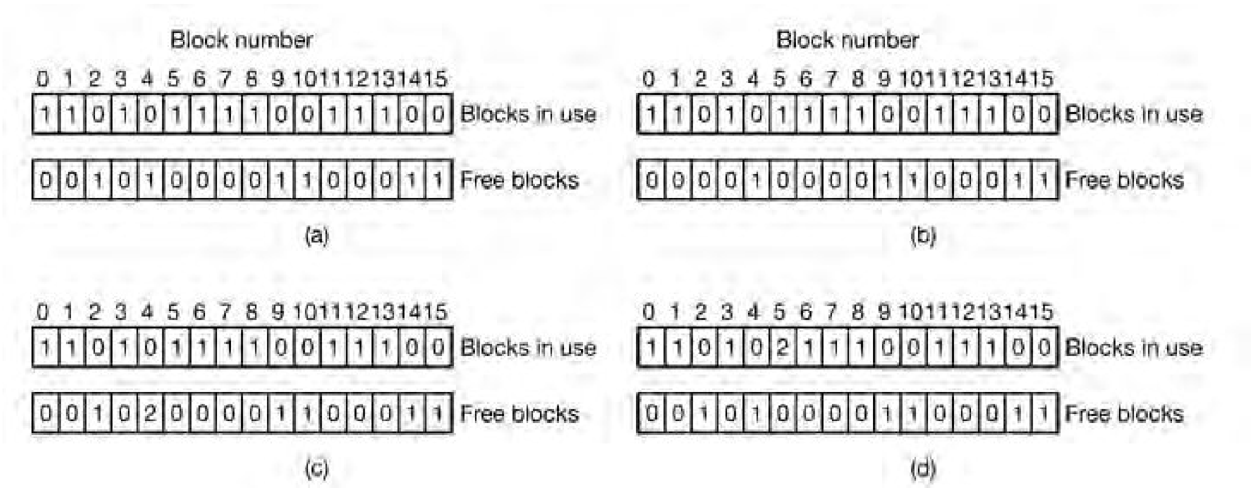

Block number
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0 Blocks in use
0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1 Free blocks
(a)
Block number
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0 Blocks in use
0 0 0 0 1 0 0 0 0 1 1 0 0 0 1 1 Free blocks
(b)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 0 Blocks in use
0 0 1 0 2 0 0 0 0 1 1 0 0 0 1 1 Free blocks
(c)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1 1 0 1 0 2 1 1 1 0 0 1 1 1 0 0 Blocks in use
0 0 1 0 1 0 0 0 0 1 1 0 0 0 1 1 Free blocks
(d)

Figure The system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

File System Performance

Access to disk is much slower than access to memory. Reading a memory word might take 10 nsec. Reading from a hard disk might proceed at 10 MB/sec, which is forty times slower per 32-bit word, but to this must be added 5-10 msec to seek to the track and then wait for the desired sector to arrive under the read head. If only a single word is needed, the memory access is on the order of a million times as fast as disk access.

Caching

The most common technique used to reduce disk accesses is the **block cache** or **buffer cache**. Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache, and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache.

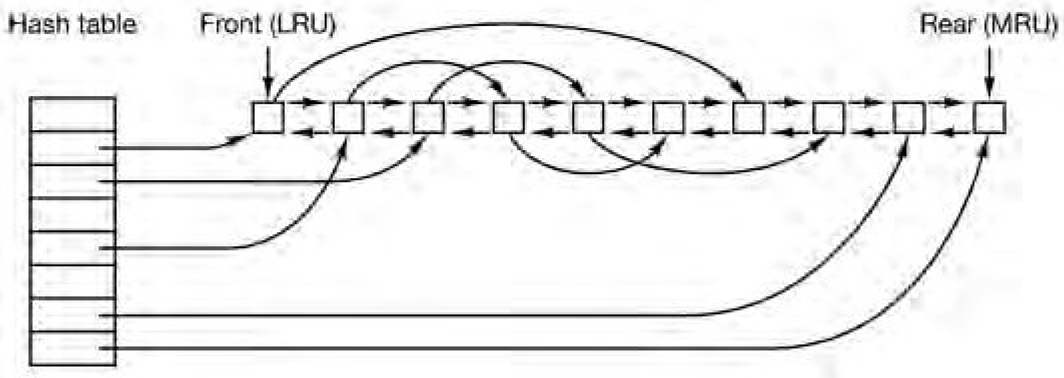


Figure The buffer cache data structures.

When a block has to be loaded into a full cache, some block has to be removed (and rewritten to the disk if it has been modified since being brought in). This situation is very much like paging, and all the usual page replacement algorithms, such as FIFO, second chance, and LRU, are applicable. One pleasant difference between paging and caching is that cache references are relatively infrequent, so that it is feasible to keep all the blocks in exact LRU order with linked lists.

Furthermore, some blocks, such as i-node blocks, are rarely referenced two times within a short interval. These considerations lead to a modified LRU scheme, taking two factors into account:

1. Is the block likely to be needed again soon?
2. Is the block essential to the consistency of the file system?

Block Read Ahead

A second technique for improving perceived file system performance is to try to get blocks into the cache before they are needed to increase the hit rate. In particular, many files are read sequentially. When the file system is asked to produce block *k* in a file, it does that, but when it is finished, it makes a sneaky check in the cache to see if block *k* + 1 is already there. If it is not, it schedules a read for block *k* + 1 in the hope that when it is needed, it will have already arrived in the cache.

Reducing Disk Arm Motion

Caching and read ahead are not the only ways to increase file system performance. Another important technique is to reduce the amount of disk arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder. When an output file is written, the file system has to allocate the blocks one at a time, as they are needed. If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together.