# B.Sc. COMPUTER SCIENCE

## SEMESTER V

## CORE VII - OPERATING SYSTEMS

**UNIT – I**:

Introduction - History of operating system- Different kinds of operating system - Operation system concepts - System calls-Operating system structure.

**UNIT – II:**

Processes and Threads: Processes - threads - thread model and usage - inter process communication.

**UNIT – III**:

Scheduling - Memory Management: Memory Abstraction - Virtual Memory - page replacement algorithms.

**UNIT - IV**:

Deadlocks: Resources- introduction to deadlocks - deadlock detection and recovery - deadlocks avoidance - deadlock prevention. Multiple processor system: multiprocessors - multi computers.

**UNIT – V**:

Input / Output: principles of I/O hardware - principles of I/O software. Files systems: Files - directories - files systems implementation - File System Management and Optimization.

**TEXT BOOK**

1. Andrew S. Tanenbaum, "Modern Operating Systems", 2ndEdition, PHI private Limited, New Delhi, 2008.

**REFERENCE BOOKS**

1. William Stallings, "Operating Systems - Internals & Design Principles",5thEdition, Prentice - Hall of India private Ltd, New Delhi, 2004.
2. Sridhar Vaidyanathan, "Operating System", 1st Edition,Vijay Nicole Publications, 2014.

# UNIT – I

## INTRODUCTION:

- Modern general-purpose computers, including personal computers and mainframes, have an operating system to run other programs, such as application software. Examples of operating systems for personal computers include Microsoft Windows, Mac OS (and Darwin), Unix, and Linux.
- The lowest level of any operating system is its kernel. This is the first layer of software loaded into memory when a system boots or starts up. The kernel provides access to various common core services to all other system and application programs.
- A modern computer system consists of one or more processors, some main memory, disks, printers, a keyboard, a display, network interfaces, and other input/output devices. For this reason, computers are equipped with a layer of software called the **operating system,** whose job is to manage all these devices and provide user programs with a simpler interface to the hardware.

- At the bottom is the hardware, which, in many cases, is itself composed of two or more levels (or layers). The lowest level contains physical devices, consisting of integrated circuit chips, wires, power supplies, cathode ray tubes, and similar physical devices.
- **Microarchitecture level,** in which the physical devices are grouped together to form functional units. Typically this level contains some registers internal to the CPU (Central Processing Unit) and a data path containing an arithmetic logic unit.
- In each clock cycle, one or two operands are fetched from the registers and combined in the arithmetic logic unit (for example, by addition or Boolean AND). The result is stored in one or more registers. On some machines, the operation of the data path is controlled by software, called the **microprogram.**

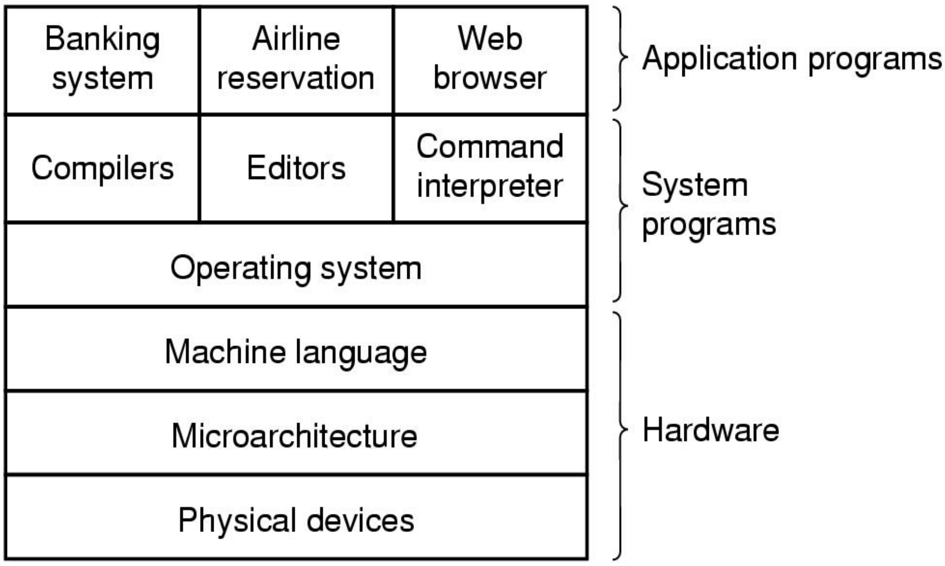


**Figure 1-1.** A computer system consists of hardware, system programs, and application programs.

- The purpose of the data path is to execute some set of instructions. Some of these can be carried out in one data path cycle; others may require multiple data path cycles. Together, the hardware and instructions visible to an assembly language programmer form the **ISA (Instruction Set Architecture)** level. This level is often called **machine language.**

The machine language typically has between 50 and 300 instructions, mostly for moving data around the machine, doing arithmetic, and comparing values. In this level, the input/output devices are controlled by loading values into special **device registers.**

- On top of the operating system is the rest of the system software. Here we find the command interpreter (shell), window systems, compilers, editors, and similar application-independent programs.
  - It is important to realize that these programs are definitely not part of the operating system, even though they are typically supplied by the computer manufacturer.
  - The operating system is (usually) that portion of the software that runs in **kernel mode** or **supervisor mode.**

**Definition of Operating System:**

- An operating system (OS) is a software program that manages the hardware and software resources of a computer. The OS performs basic tasks, such as controlling and allocating memory, prioritizing the processing of instructions, controlling input and output devices, facilitating, networking, and managing files.
- An Operating system is a program that acts as an intermediary between a user of a computer and the computer hardware.
- It acts as an interface between applications and the computer hardware.
- An Operating System can be viewed as the programs, implemented in either software that makes the hardware usable.

**The Operating System as an Extended Machine:**

- The function of the operating system is to present the user with the equivalent
- of an **extended machine** or **virtual machine** that is easier to program than the underlying hardware.
- How the operating system achieves this goal is a long story, which we will study in detail throughout this book.
- To summarize it in a nutshell, the operating system provides a variety of services that programs can obtain using special instructions called system calls.

**The Operating System as a Resource Manager**

- Resource management includes multiplexing (sharing) resources in two ways: in time and in space.
- When a resource is time multiplexed, different programs or users take turns using it.
- First one of them gets to use the resource, then another, and so on.
- For example, with only one CPU and multiple programs that want to run on it, the operating system first allocates the CPU to one program, then after it has run long enough, another one gets to use the CPU, then another, and then eventually the first one again.
- Determining how the resource is time multiplexed — who goes next and for how long — is the task of the operating system.
- Another example of time multiplexing is sharing the printer.
- When multiple print jobs are queued up for printing on a single printer, a decision has to be made about which one is to be printed next.

**HISTORY OF OPERATING SYSTEM:**

- **The first generation (1945 – 55):** Vacuum Tubes and Plug boards.
- **The second generation (1955-65):** Transistors and Batch systems
- **The Third generation (1965 -80):** ICs and Multiprogramming
- **The fourth generation (1980 – present):** Personal computer

**First Generation (1945 -55):**

In 1940 Howard Aiken At Harvard, John von Neumann at the Institute for Advanced study in Princeton and Konrad Zuse in Germany, among others all succeeded in building Calculating engines.

**Vacuum Tubes:** Machines were enormous filling up entire rooms with tens of thousands of Vacuum tubes.

**Drawback:** millions of times slower than Personal computers available today.

**Plug boards:** All programming was done in absolute machine language. Wiring up plug boards to control the machine's basic functions.

**Drawback:** Programming Languages were unknown. Operating systems were unheard

**Process of Plug board**

- Programmer signup sheet on the wall
- Come down to the machine room
- Insert plug board into the computer
- Wait until the completion of process

**The following task are done by using plug board**

- Numerical calculations
- Grinding out tables of sin, cos and logarithms
- 1950 – Punched cards were introduced.
- It is used to write programs on Carts and read them

**Second Generation (1955 – 65)**

In 1950s – Transistor was introduced. Two main systems are

- Mainframes
- Batch System

**Mainframes**

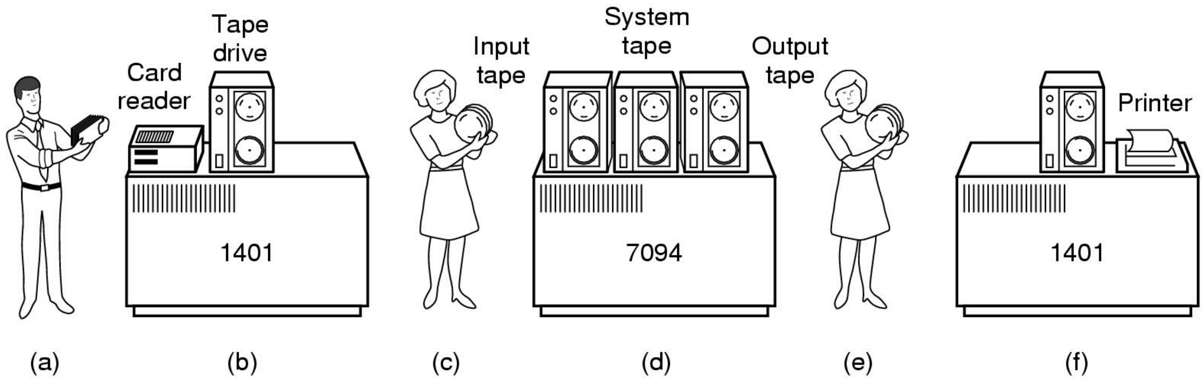
- To run a job first write the program on paper
- Punch the Program on card
- Bring the card deck down to the input room
- Hand it to one of the operators
- After finished the Job, it carry the job from output room.
- Finally the Programmer collects the result.

**Drawback:** Most of computer time was wasted while operators were walking around the machine room.

**Batch System:**

- It was to collect a tray full of jobs in the input room.
- Read them onto a magnetic tape using small inexpensive computer (IBM 1401)
- IBM 1401 is very good at reading cards, copying tapes, Printing Output.

**Drawback:** Not at all good at numerical calculation



- Second generation computers are used for scientific and engineering calculation.

**Third Generation (1965 -1980)**

**System/360:**IBM introduced the system/360 was a series of software compatible machines ranging form 1401 sized to much more powerful than the 7094.

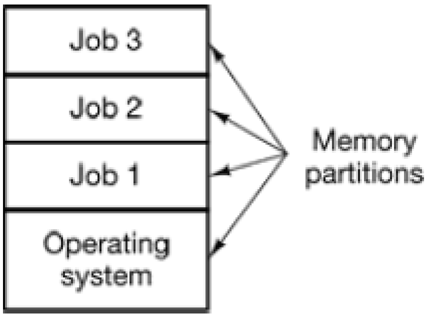
**Advantage:**

- Better prize and Performance
- Maximum memory
- Speed Processor
- High number of I/O devices permitted.

It was the designed to handle scientific and commercial computing.

**Multiprogramming:**

- It refers to a computer system's ability to support more than one process( program) at the same time.
- Multiprocessing operating systems enables several programs to run concurrently.



**Spooling** (**Simultaneous Peripheral Operation On Line**) :

- In multiprogramming system whenever a running job finished, the operating system could load new job from the disk into the now – empty partition and run it. This technique is called spooling (**Simultaneous Peripheral Operation On Line**)

**Time Sharing System:**

- A set of programs are executed in fixed time scheduled manner.
- The computer can provide fast, interactive service to a number of users and also work on big batch jobs in the background when the CPU is otherwise idle.
  - The first timesharing system – **CTSS – (Compatible Time Sharing System)**
  - The Second Timesharing system – **MULTICS – (Multiplexed Information and computing services)**
  - MULTICS is a "computer Utility".
  - MULTICS hundreds of simultaneous timesharing users.

**Features:**

- Ability to read jobs form cards onto the disk as soon as they were brought to the computer room.
- It is well suited for big scientific calculation and commercial data processing.

**The fourth generation(1980 – present)**

- **Microcomputer:** Large scale Integration circuit chip containing Thousands of transistors on a square centimeter of silicon.
- The first general purpose 8 bit CPU DOS(Disk Operating system)
- MS – Dos(Microsoft disk operating system) Revised system of DOS
- CP/M MS_DOS and other operating systems for early microcomputers were all based on users typing in commands form the keyboard. In 1960s Engelbart invented the GUI(Graphical user Interface) Complete with windows, Icons, menus and mouse.
- In 1995 a freestanding version of Windows, Windows 95 was released. In 1998 a slightly modified version of this system, called Windows98 was released.
- Another Microsoft operating system is Windows NT (New Technology) Compatible with Windows 95. It is full 32 bit system.
- In 1999 Windows 2000 is developed. It was intended to be the successor to both Windows 98 and Windows NT. Windows 98 called Windows MC (Millennium edition)
- On Pentium Based computers, Linux is becoming a popular alternative to Windows.
- Unix systems support a windowing systems X Windows system.
- It handles the basic window management allowing users to create, delete, move and resize windows using a mouse.
- In 1980 growth of networks of personal computers running Network Operating systems and distributed operating systems.

**Network Operating systems:**

The users are aware of the existence of multiple computers and can log in to remote machines and copy files from one machine to another.

**Distributed Computing:**

Computing has become distributed phenomenon rather than centralized one. It can be used to increase transmission speed or to distribute dedicated computers to the work side. Computation will be distributed to various processors; each processor will have its own local disk. The processors communicate with one another through various communication lines, such as high-peed disk buses or telephone lines, to share the computation.

Reasons for using distributed systems are,

- Resource sharing
- Computation speed up
- Reliability
- Communication

**Parallel Computing:**

To increase the computing power beyond the range of today’s sequential processes is to switch to parallel architectures in which, many processors function concurrently, and it will use common memory, i.e. trying a thousand-billion-instruction-per-second. Computer together in parallel and let them work on problems concurrently. Future computer systems will exhibit massive parallels;

they will have huge number of processors, so many in fact that any part of computation which may be performed in parallel.

**Multi tasking:**

- Multitasking is the ability to execute more than one task at the same time , a task being a program.
- Multiprocessing some times implies that more than one CPU is invoked, in multitasking only one CPU is involved.

## DIFFERENT KIND OF OPERATING SYSTEM

### Mainframe operating system

- ✓ The operating system for the mainframes those room sized computer still found in major corporate data centers.
- ✓ These computers distinguish themselves from personal computers in terms of their I/O capacity.
- ✓ A personal Mainframe with 1000 disks and thousands of gigabytes of data is not unusual.
- ✓ Mainframes are also making something of a comeback as high end web servers.
- ✓ The operating systems for Mainframes are heavily oriented toward processing many jobs at once, most of which need prodigious amount of input and output.

**Three kinds of services:**

- ▪ Batch
- ▪ Transaction Processing
- ▪ Timesharing

**Batch:**

- ✓ In batch processing data's are accumulated and processed in groups.
- ✓ Processes routine jobs without any interaction or interactive user present

**Transaction Processing:**

- ✓ It handles large numbers of small requests
- ✓ Each unit of work is small but the system must handle hundreds or thousands per second.
- ✓ Check processing system at a bank or airline reservation is an example.

**Timesharing:**

- ✓ It allows multiple remote users to run jobs on a computer at once.
- ✓ An example for mainframe operating system is OS/390, a descendant of OS/360.

### Server operating system

- ✓ A server operating system is software that was especially developed to serve as a platform for running multi-user computer programs, applications that are networked and programs critical to business computing.
- ✓ Server operating systems 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.
- ✓ Server can provide print service, file service or web service
- ✓ An example for server operating systems is UNIX and WINDOWS 2000.

## Multiprocessor operating system

- ✓ When more than one process work on a system simultaneously , it is called as multiprocessor operating system.
- ✓ A **shared-memory multiprocessor** (or just multiprocessor henceforth) is a computer system in which two or more CPUs share full access to a common RAM. A program running on any of the CPUs sees a normal (usually paged) virtual address space.
- ✓ These systems are called parallel computers, multi computer, or multiprocessors.
- ✓ It consists of some special features for communication and connectivity.

## Personal operating system.

- ✓ In order to provide an efficient technical service for desktop computers, a clear specification for desktop computer operating systems is required.
- ✓ Microsoft Windows Operating Systems are the main operating systems
- ✓ Its job is to provide a good interaction to a single user.
- ✓ They are widely used for word processing spreadsheets and internet access.
- ✓ **Example:** Windows98 ,Windows2000,Macintosh, Linux
- ✓

## Real time operating systems

- ✓ The systems are characterized by having time as a key parameter.
- ✓ **Example:** In industrial process control system real time computers have to collect data about the production process and use it to control machine in the factory.

  **Two types**

  - Hard real time
  - Soft real time

**Hard real time:**

- ✓ If the action absolutely must occur at a certain moment ( or within certain range).

**Soft real time system:**

- ✓ In soft real time system, missing an occasional deadline in acceptable such as digital audio or multimedia systems come under this category.
- ✓ **Example:** Digital audio Multi media system

## Embedded operating system

- ➢ An **embedded operating system** is an operating system for embedded computer systems. These operating systems are designed to be compact, efficient at resource usage, and reliable, forsaking many functions that non-embedded computer operating systems provide, and which may not be used by the **specialized applications** they run.
- ➢ Continuing on down to smaller and smaller system, we come to palmtop computers and embedded 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.
- ➢ Palmos is an example for Embedded operating system

## Smart card Operating system

- ➢ The smallest operating systems run on smartcards, 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 single function called electronic payments, but others can handle multiple functions on a same smart card.
- Some smart cards are java oriented ROM on the smart card holds an interpreted for the Java Virtual Machine.
- Java applets 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.

**OPERATING SYSTEM CONCEPTS:**

**All operating system have certain basic concepts like:**

- **Processes**
- **Memory**
- **Files**
- **Deadlocks**
- **Input/output**
- **Security**
- **The shell**

**Processes:**

- A key concept in all operating system is the process.
- Process means program in execution. Each process has its own address space.

**Address space:**

- A list of memory locations from some minimum to some maximum which the process can read and write.
- The address space contains the executable program, the program data, and its stack.
- Each process is some of registers.
- The registers include program counter, status pointer, and hardware register.

**Execution of Process:**

- Operating system executes the process in sequential order.
- Operating system decides to stop running process and start running another process, when the first one has more than its share of CPU time in the part second.
- When a process is suspended temporarily, it must be restarted in exactly the same state it had when it was stopped. All information about the process must be explicitly saved during suspension.
- Read call is executed after the process is restarted and it will read the proper data.
- All information about each process is stored in an operating system table called Process table.
- It is an array of structure.
- A suspended process consists of its address space usually called core image and its process table entry contains registers.

**Command Interpreter or shell**

A Process called command Interpreters or Shell reads commands from a terminal. The user has typed a command requesting that a program be compiled. The Shell must now create a new process that will run the compiler. When that process has finished the compilation, it executes a system call or terminates itself.

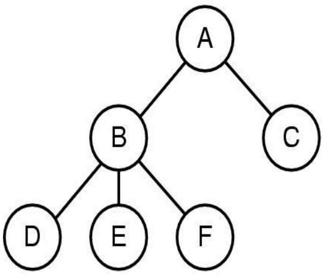
**Child Process**

If a process can create a new one or more other process referred to as child processes and there processes can create child processes.

**Interprocess communication**

Related process needs to communicate with one another and synchronize their activities. This communication is called Interprocess communication.

**Process Tree**



Process A creates two child process B and C. B creates D, E, and F.

**Alarm signal**

When the specified number of seconds has elapsed the operating system sends an alarm signal to the process. The signal causes the process to temporarily suspend whatever it was doing save its registers on the stacks and start running a special signal handling procedure.

**UserID and password**

Each Person authorized to use a system is assigned a VID by the system administrator. A child process has the same VID as its parent.

**Group Identification**

Users can be members of groups, each of which has a GID (group identification)

**Superuser**

One user ID called superuser, has special power and may violate many of the protection rules.

**Deadlock:**

A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set.

- In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process.
- None of the processes can run, none of them can release any resources, and none of them can be awakened.

**1. A traffic Deadlock**

A number of automobiles are attempting to drive through a busy section of the city, but the traffic has become completely snarled. Traffic comes to a halt, and it is necessary for the police to unwind the jam by slowly and carefully backing cars out of the congested area.

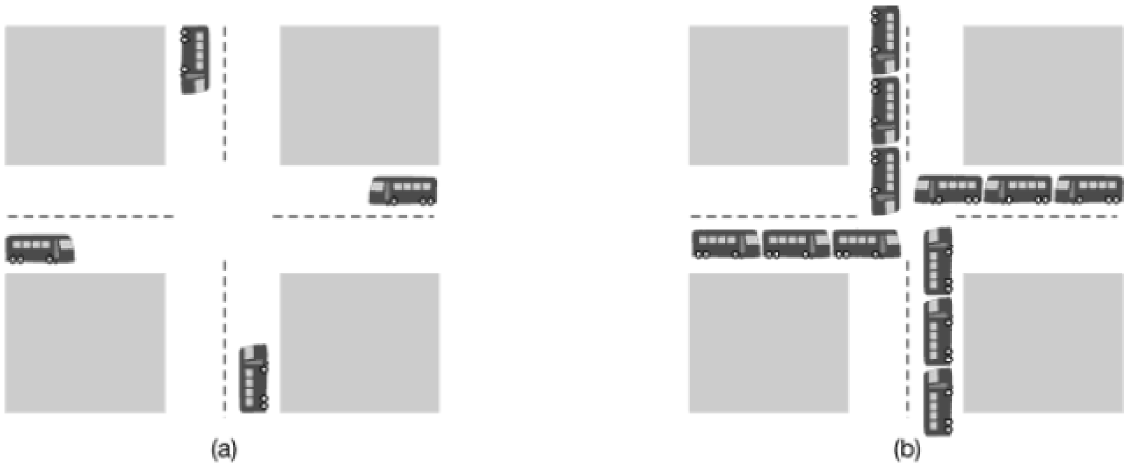
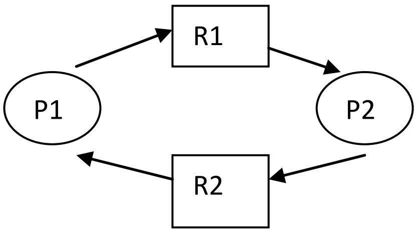


**Figure 1-13.** (a) A potential deadlock. (b) An actual deadlock.

## 2. A Simple Resource Deadlock:

The system is said to be in deadlock, when each process holds a resource being requested by the other process and neither process is willing to release the resource it holds.



## Example:

- Two processes each need to produce a CD-ROM from data on a tape. Process
- 1 requests and is granted the tape drive.
- Next process 2 requests and is granted the CDrecorder.
- Then process 1 requests the CD-recorder and is suspended until process 2 returns it.
- Finally, process 2 requests the tape drive and is also suspended because process 1 already has it.

## Memory Management:

- Every computer has some main memory to hold executing programs.
- Protection mechanism helps the process to safeguard its content, by not allowing other process to interfere in its job.
- Each process has some set of addresses; it can use typically running from 0 up to some maximum.
- The maximum amount of address space process has is less than the main memory.
- Many computer addresses are 32 or 64 bits giving an address space of 232 or 264 bytes.
- If a process has more address space that the computer main memory, then a technique called virtual memory exists.
- Virtual memory is the concept of increasing the capacity of the main memory without increasing the size of the same.
- The operating system keeps part of the address space in main memory and part or disk and shuttle pieces back and forth between them as needed.
- The needs of users can be met best by a computing environment that supports modular programming and the flexible use of data.
- System managers need efficient and orderly control of storage allocation.
- The operating system, to satisfy these requirements, has five principal storage management responsibilities:

**Simple Operating System:**

- ✓ In case of simple operating system only one program can run at a time in memory.

**Sophisticated Operating System:**

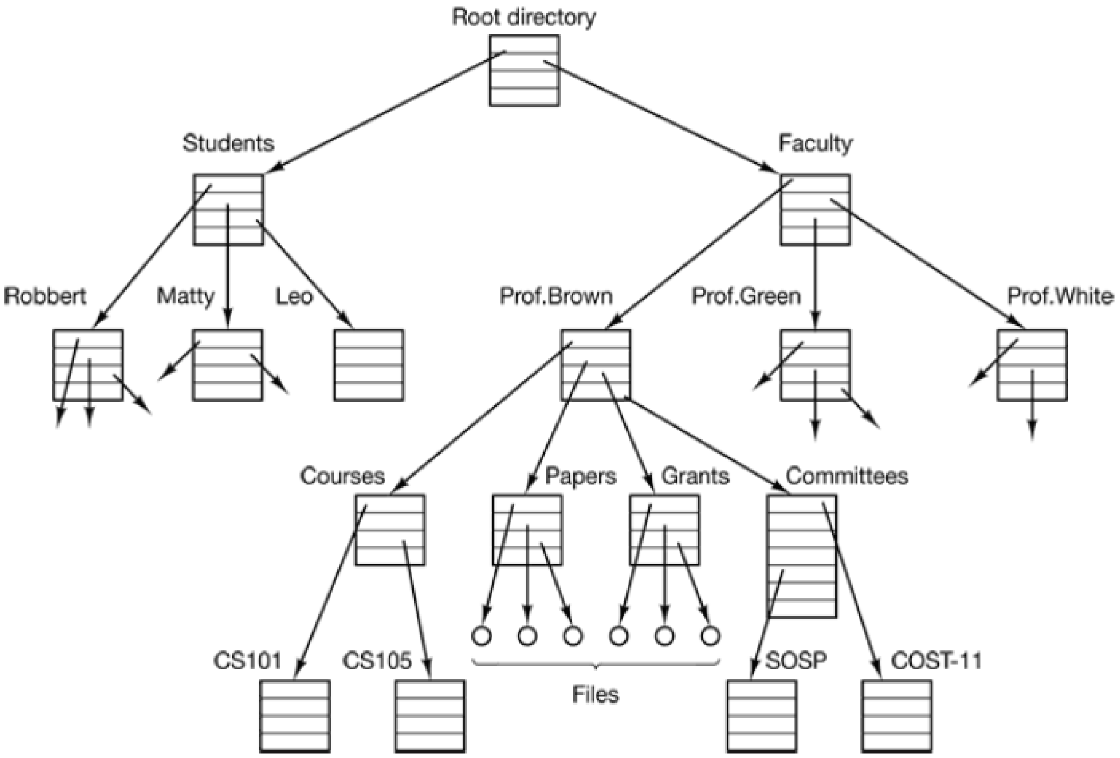
- ✓ Multiple programs can be executed in memory at the same time

**Input/Output device:**

- ➢ All computers have physical devices for acquiring input and producing output. Input and Output devices is Keyboard, monitor, printer.
- ➢ Every operating system has an I/O subsystem for managing its I/O devices.
- ➢ Some of the I/O software's are device Independent.

**Files:**

- ➢ User should be able to create, modify and delete files. User should be able to share each other's file.
- ➢ The mechanisms for sharing files should provide various types of controlled access such as read access, write access execute access.
- ➢ User should be able to structure their files in a manner most appropriate for each application.
- ➢ User should be able to order the transfer information between files.
- ➢ Backup and recovery capabilities must be provided to prevent accidental loss or mal practice.
- ➢ A major function of the operating system is to hide the peculiarities of the disk and other I/O devices.
- ➢ System calls are need to create file, Remove files, read files and write files. File is a collection of records.
- ➢ A file can be read, it must be located or disk and opened, and after it has been read it should be close, so calls are provided to do these things.
- ➢ To provide a place to keep files, most operating system have the concept of a directory as a way of grouping files into single place.
- ➢ System calls are needed to remove or create directory calls are provide to put an existing file in a directory and to remove a file a from a directory.
- ➢ Directory entries may be either files or other directory.



**Path name and root directory**

- ✓ Every file within the directory hierarchy can be specified by giving its path name from the top of the directory hierarchy, the root directory.
- ✓ Absolute name consists of list of directory that must be traversed form the root directory to get to the file, with slashes separating the components.

**Students/Leo/Paper/filename**

- ✓ The leading slash indicates that the path is absolute, that is starting at the root directory.

**Working directory:**

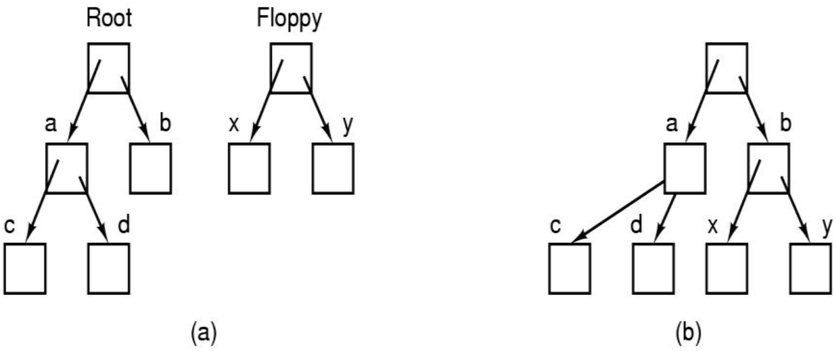
- ✓ Each process has a current working directory in which path names not beginning with a slash are looked for.

**File descriptor:**

- ✓ Before a file can be read or written, it must be opened, at which time the permissions are checked.
- ✓ If the access is permitted the system returns a small integer called a file descriptor to subsequent operations.
- ✓ If the access is prohibited, an error code is returned.

**Mounted file system**

- ✓ Mounted file system is an important concept in UNIX.
- ✓ It allows the file system on floppy disk to be attached to the tree.
- ✓ The file system on a floppy has been mounted on directory b, thus allowing access to files.



**Special file:**

- ✓ They are used to provide I/O devices look like files.
- ✓ They can be read and written using the same system calls as are used for reading and writing files.

**Two kinds of files:**

- **Blocks files**
- **Character Special files**

**Blocks files:**

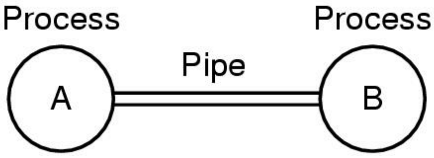
- ✓ It is used to model devices that consists of a collection of randomly addressable blocks such as disks

**Character Special files:**

- ✓ Used to model Printers, modems and other devices that accept or output a character stream.

**Pipe**

- ✓ A Pipe is a pseudo file that can be used to connect two processes.
- ✓ If Process A and B wish to talk using a Pipe they must set it up in advance.
- ✓ When process A wants to send data to process B, it writes on the pipe as through it were an input file.



**<u>Security:</u>**

Computers contains large amount of information that users after want to keep confidential information like

- ★ Electronic mail
- ★ Business plan
- ★ Tax return

- It is up to the operating system to manage the system security.
- Files are only accessible by authorized users.
- Files in UNIX are protected by assigning each one 9-bit binary protection code.
- The protection code consists of 3-bit fields:
  - **One for the owner**
  - **One for the other members of owner group**
  - **One for everyone else**
  -

Each field has a bit read access, write access and executes access

| Person | Performing operation |
|---|---|
| Owner | Read, write, execute |
| Group Member | Read and execute |
| Everyone else | execute |

**bits=rwx** **bits**

**r=read**

**w=write**

**x=execute**

**Protection code=rwx r-x- - x**

**rwx=owner**

**r-x person in owner group**

**--x everyone else**

Protecting the system from unwanted intruders, both human and nonhuman is one of them.

**<u>The Shell:</u>**

- Shell is a UNIX term for the interactive user interface with an operating system.
- The shell is the layer of programming that understands and executes the commands a user enters.

**Example**

**Sh, Csh, Ksh and bash**

- The shell has a terminal as standard Input and standard Output.
- It starts out by typing the prompt, a character such as a dollar sign, which to accept a command.
- If the user types

**Date**

- The shell creates a child process and run the date program as the child.
- While the child process running the shell waits for it to terminate.
- 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**

Standard Input can be redirected

**Sort<file1>flie2**

- It invokes sort program with Input takes from file1 and output sent to file2.
- The output of one program can be used as the input for another program by connecting them with a pipe

**Cat file1 file2 file3|sort>/dev/lp**

**<u>SYSTEM CALLS</u>**

- The interface between the operating system and 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.

- System calls are of two types namely
  - **Vague generalities**
    - Operating systems have system calls for reading files
  - **Some specific system**
    - It has 3 parameters namely

    - One to specify the file
    - One to tell where the data are to be put and
    - One to tell how many bytes to be read

- If a process is running a user program in user mode and needs a system call 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.
    - System calls has three parameters
    - First one specifying the file
    - Second one pointing to the buffer
    - Third one giving the number of bytes to read
    - **EG: Count=read( fd,buffer,nbytes );**
- The system call returns the number of bytes actually read in count.
- This value is normally the same as n bytes, but may be smaller.
- If the system call cannot be carried out, either due to an invalid parameter of disk error, count is set to -1, and the error number is put in a global variable errno.
- Program checks the result of a system call to check if any error has occurred.

**Read(fd,buffer,nbytes)**

First push the parameter onto the stack

There are 11 steps in making the system call

1. **Push n bytes**
2. **Push &buffer**
3. **Push fd:**

   First and third parameters are called by values.

   Second parameter is passed by reference

4 **Call read:**

Procedure call instruction used to call all procedures.

5. **Register :**

   Puts the system call number in a place where the operating system expects it such as register.

6. **TRAP:**

   Executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within kernel.

7. **Dispatches:**

   The kernel code examines the call number and then dispatches to the correct system call handler. (Table of pointer)

8. **System call handler**

   At that point the system call handler runs

9. **Return to caller**

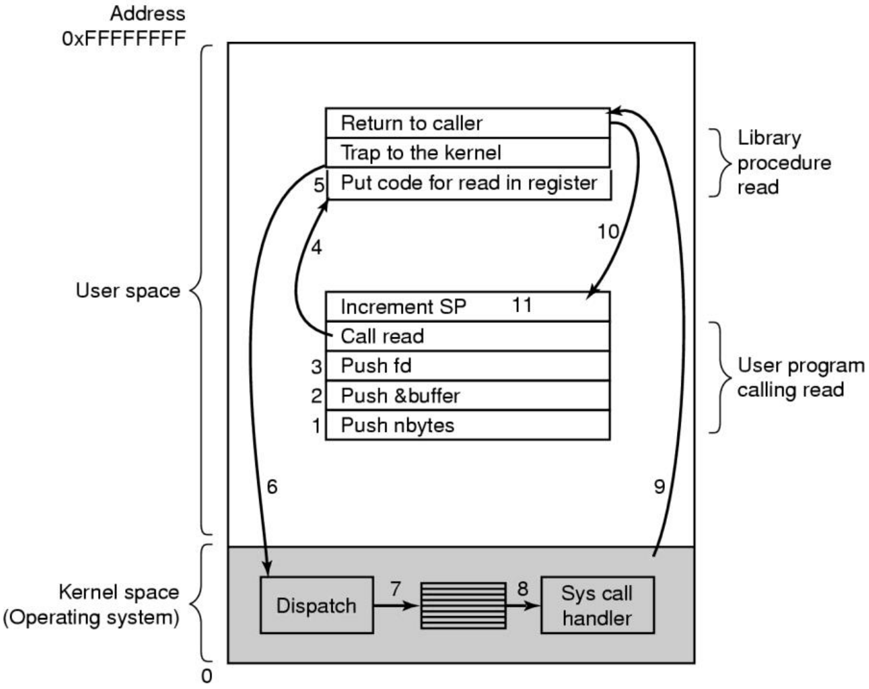
   Once the system call handler has be returned to the user – space library procedure.

10. **Increment SP**

    Return to the user program in the usual way procedure calls return.

11. User program has to clean up the stack it does after any PC.

**read(fd,buffer,nbytes)**



Some of the major POSIX system calls. Four types of system caller.

- ✶ **Process Management**
- ✶ **File Management**
- ✶ **Directory and file system management**
- ✶ **Miscellaneous**

**System calls for process management:**

**<u>Fork system call:</u>**

- It is used to create a new process in UNIX.
- It creates an exact duplicates of the original process including all the file descriptors, registers
- All the variables have identical values
- The fork call return a value, which is zero in the child and equal to the child's process identifier or PID in the parent.

| **Call** | **Description** |
|---|---|
| **pid=fork()** | Create a child process identical to the parent |
| **pid=waitpid(pid,&statloc,options)** | Wait for a child to terminate |
| **s=execve(name, argv, environp)** | Replace a process core image |
| **exit(status)** | Terminate process execution and return status |

**<u>Waitpid(pid, &statloc, option) :</u>**

**Waitpid** executes the command and then reads the next command

- ♦ Waitpid can wait for a specific child or for any old child by setting the first parameters to -1.
  - o **Pid=** process identification number
  - o **Statloc =** child exit status(Normal, abnormal, exit value)
  - o **Option =** Various options

**Program:**

```
# define true 1
while (true)
{                    Type_prompt();
                     Read_command(command,parameters);
                              If (fork()!=0)
                              {
                                       Waitpid(-1,&status,0)
                              }
                             else
                             {
                           execve(command,parameters,0)} }
```

**Execve(name,argv,environp)**

- It is used to execute the child process. It has **three parameters**,
  - **Name =** name of the file to be executed
  - **argv =** a pointer to a array argument
  - **environp =** a pointer to a environment array

Consider the case of a command such as

**Cp file1 file2**

It is used to copy file1 to file2. After the shell has forked, the child process located and executes the files cp and passes located and executes the files cp and passes to it the names of the source and target files. The main program of cp contains the declarations

**main(argc,argv,envp)**

**Argc** counts the number of items

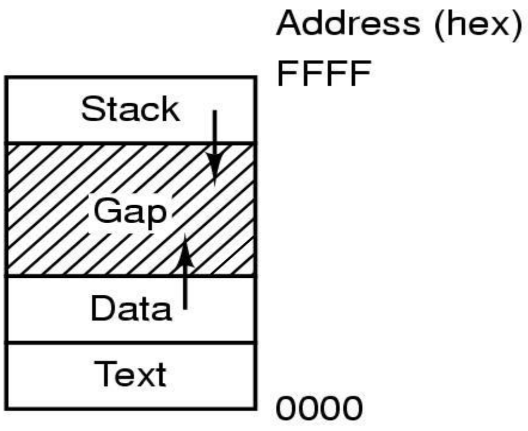
**Argv** is a pointer to an array

**Envp** is a pointer to the environment, an array of strings containing assignments of the form name= value used to pass information.

**Exit =** which processes should use when they are finished executing.

**Processes in UNIX have their memory divided up into three segment:**

- **The text segment :** It contains the program code.
- **The data segment :**
  - It contains the variable.
  - Data segment grows upward.
  - Brk system call used to expand data segment
- **The stack segment :**
  - It contains the data structure.
  - The stack grows into the gap automatically.
  - Stack segment grows downward



**System calls for file management**

- To read or write a file, the file must be opened first using **open.**
- This call specifies the file may be opened, either as an absolute path name or relative to the working directory, and a code of O_RDONLY, O_WRONLY, or O_RDWR meaning open for reading writing and both.
- To create a new file **O-CREAT** is used.
- The file descriptor returned can then be used for reading or writing.
- The file can be closed by **CLOSE**, which makes the file descriptor available for reuse on a subsequent open.
- The **LSEEK** call changes the value of the position pointer, so that subsequent calls to read or write can begin anywhere in the file.

| **Call** | **Description** |
|---|---|
| **Fd=open(file,how)** | Open a file for reading writing or both |
| **S=create(fd)** | Close an open file |
| **N=read(fd,buffer,nbytes)** | Write data from a buffer into a file |
| **Position=lseek(fd,offset,whence)** | Move the file pointer |
| **S=stat(name,&buf)** | Get a files status information |

**LSEEK has three parameters**. **Lseek(fd, offset,whence)**

- The first is the file descriptor for the file.
- The second is a file position.
- The third tells whether the file position is relative to the beginning of the file, the current position, or the end of the file.

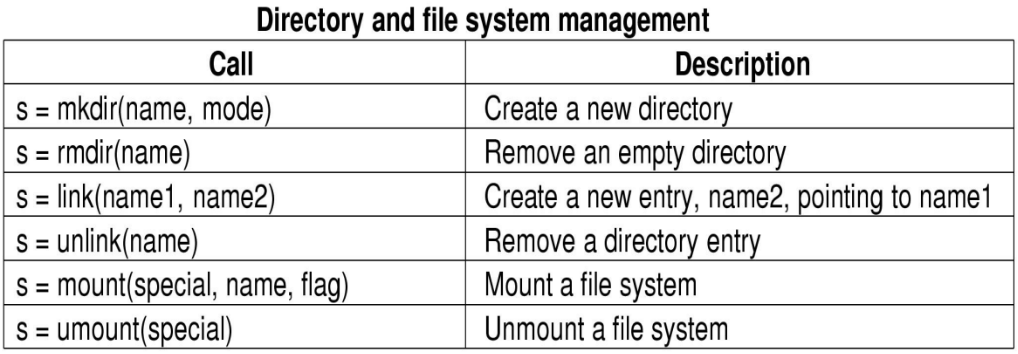
UNIX keeps track of the file mode size, time of last modification, and other information

**Stat (name, &buf)**

- ♦ Name= specifies the file name to be inspected.
- ♦ Buf = Pointer to a structure.

**System calls for directory management.**

- First two calls, **mkdir** and **rmdir** create and remove empty directories.
- The next call is link used to allow the same file to appear under tow or more names, often in different directories.



Directory and file system management

| Call | Description |
|---|---|
| s = mkdir(name, mode) | Create a new directory |
| s = rmdir(name) | Remove an empty directory |
| s = link(name1, name2) | Create a new entry, name2, pointing to name1 |
| s = unlink(name) | Remove a directory entry |
| s = mount(special, name, flag) | Mount a file system |
| s = umount(special) | Unmount a file system |

- o It allows several members to share common files.
- o Shared file means that changes that any member of the team makes are visible to other members.
- o There is only one file.
- o When copies are made of a file, subsequent changes made to one copy do not affect the other ones.

**The system call :**

**Link("/usr/sam/memo","usr/samp/ex1");**

The file memo in sam directory is not entered into samp directory under the name.

**Ex Both refers same file.**

**/usr/samp/ usr/sam/ /usr/samp/ /usr/sam/**

| | |
|---|---|
| 16 | mail |
| 81 | games |
| 40 | test |
| | |

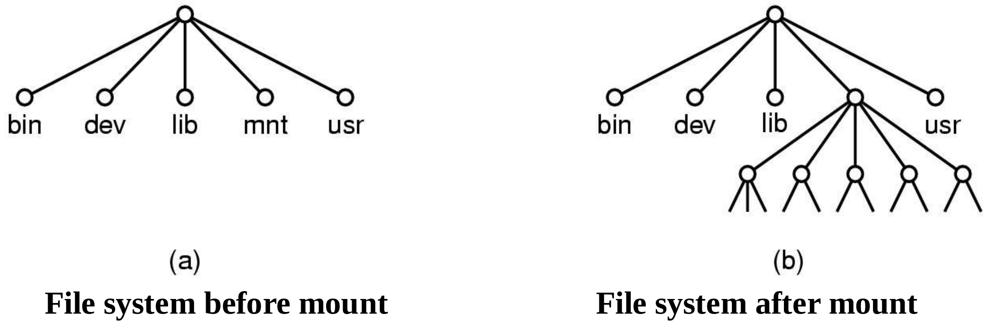
| | |
|---|---|
| 31 | bin |
| 70 | memo |
| 59 | fc |
| 38 | Prog1 |

| | |
|---|---|
| 16 | mail |
| 81 | games |
| 40 | test |
| 70 | ex |

| | |
|---|---|
| 31 | bin |
| 70 | memo |
| 59 | fc |
| 38 | Prog1 |

**Two directory before linking** **Same directory after linking**

- o By executing the mount system call, the floppy disk file system can be attached to the root file system mount ("/dev/fdo" , "mnt",0)
- o The mount system call allows two file systems to be merged into one.



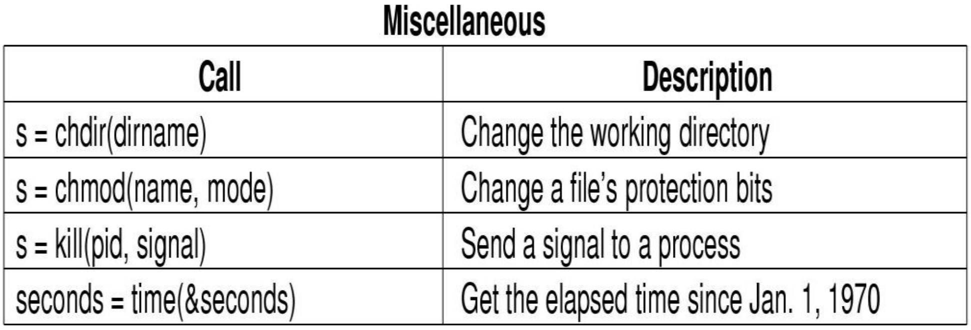
**File system before mount** **File system after mount**

**Miscellaneous system calls**

- o The **chdir** call changes the current working directory.
- o After the call **Chdir("/usr/sam/test");** an open on the file xyz will open /usr/sam/test/xyz.
- o The concept of a working directory eliminates the need for typing absolute path names all the time.
- o In UNIX every file has a mode used for protection.

- The mode includes the **read – write – execute** bits for the owner, group and others.
- The **chmod** system call makes it possible to change the node of the file

  **Chmod(“file”,0644)**
- The **kill** system call is the way users and user processes send signals.
- If a process is prepared to catch a particular signal, then when it arrives a signal handler is run.
- If the process is not prepared to handle a signal, the its arrival kills the process



Miscellaneous

| Call | Description |
|---|---|
| s = chdir(dirname) | Change the working directory |
| s = chmod(name, mode) | Change a file’s protection bits |
| s = kill(pid, signal) | Send a signal to a process |
| seconds = time(&seconds) | Get the elapsed time since Jan. 1, 1970 |

**The Windows Win32 API**

- Microsoft has defined a set of procedures called the win32 **API(Application program Interface)**
- WIN32 means the interface supported by all versions of windows.
- In windows it is impossible to see system call and then the user space library call.
- The win32 API has a huge number of calls for managing windows, geometric figures, text, fonts, scrollbar, dialog boxes, menu and other GUI features.

**Create Process :**

- It helps to create a new process.
- There is no concept of a parent process and a child process.
- After a process is created, the creator and create are equals.

**Wait for single object:**

- It is used to wait for an event**.**

**Exit Process:**

- It is used to terminate the completed process.

| **UNIX** | **WINDOWS** | **DESCRIPTION** |
|---|---|---|
| Fork | Create process | Create a new proess |
| Waitpid | Wait for single object | Can wait for a process to exit |
| Execve | None | Create process=fork+execve |
| Exit | Exitprocss | Terminate execution |
| Open | Create file | Create a file or open an existing file |
| Close | Closehandle | Close a file |
| Read | Readfile | Read data from file |
| Write | Writefile | Write data to a file |
| Lseek | Setfilepointer | Move the file pointer |
| Stat | Getfileattributeex | Get various file attributes |

| | | |
|---|---|---|
| Mkdir | Create directory | Create a new directory |
| Rmdir | Remove directory | Remove an empty directory |
| Link | None | Win 32 does not support links |
| Unlink | Delete file | Destroy an existing file |
| Mount | None | Win32 does not support mount |
| Unmount | None | Win32 does not support mount |
| Chdir | Set current directory | Change the current working directory |
| Chmod | None | Win32 does not support security |
| Kill | None | Win32 does not support signals |
| Time | Getlocaltime | Get the current time. |

- File can be read, open, close, write.
- The **SetFile pointer and GetFileAttributesEx calls set the file position and get some of the file** attributes**.**
- Windows has directories and they are created with **CreateDirectory and RemoveDirectory**, is used to remove the existing directory respectively.
- There is also a notion of a current directory, set by **SetCurrect** directory.
- The current time is acquired using **GetLocalTime.**
- The win32 interface does not have links to files, mounted file systems, security or signals.

## STRUCTURE OF OPERATING SYSTEM:

It is classified into

- **Monolithic system**
- **Layered system**
- **Virtual machines**
- **Exokernel**
- **Client – server system**

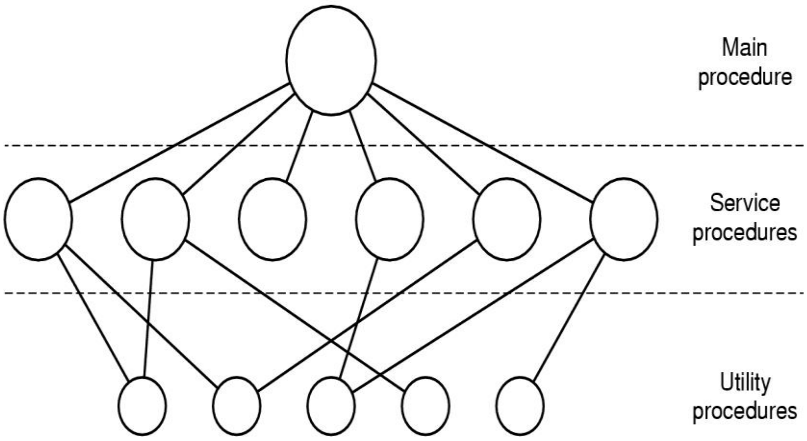
### Monolithic system (Big Mess):

- The operating system is written as a collection of **procedures**, each of which can call any of the other ones whenever it needs to.
- Each process or procedure in the system has a well – defined interfaces interms of parameters and results.
- Each one is free to call any other one.
- To construct the actual object program of the operating system when this approach is used, one first complies all the individual procedures, or file containing the procedures and then binds them all together into a single object file using the system links.
- Every procedure is visible to every other procedure.

### Basic structure for the operating system

- A main program that invokes the requested service procedure.
- A set of service procedures that carry out the system calls,
- A set of utility procedures that help the service procedures.
- This division of the procedure into 3 layers.

  - Main procedures
  - Service procedures
  - Utility procedures



## Layered System

The system was a simple batch system for a dutch computer. The system had 6 layers

| 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 |

**Layer 0 :**

- ♦ It helps in allocation of processor, and also switches in between the processes when interrupt occurs or when timers expire.

**Layer 1 :**

- ♦ It helps in memory Management.
- ♦ It allocates space for processes in main memory and on a 512 K word drum is used for holding a part of processes.

**Layer 2:**

- ♦ It handles communication between each process and the operator console.

**Layer 3 :**

- ♦ It takes care of managing the I/O devices and buffering the information streams to and from them.

**Layer 4 :**

- ♦ It is found when the user programs are identified.

**Layer 5 :**

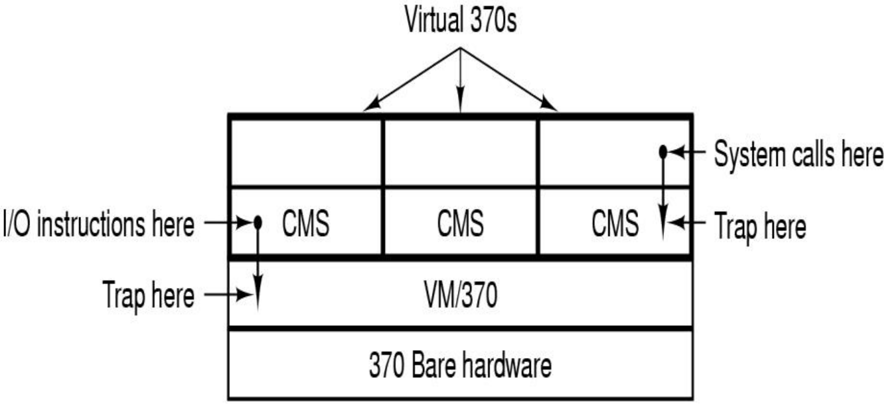
- ♦ The system operator process are located.

- A further generalization of the layering concept was present in the **MULTICS** system**.**
- **MULTICS** was described as having a series of **concentric rings,** with the inner ones being more privileged than the outer ones.
- When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to make the equivalent of a system call that is a TRAP instruction whose parameters were carefully checked for validity before allowing the call to proceed.

- The **advantage** of the ring mechanism is that it can easily be extended to structure user subsystems.

**Virtual Machines:**

- The heart of the system, known as the virtual machine monitor, runs on **the bare hardware** and does the multiprogramming, providing not one, but several virtual machines to the next layer up.
- Virtual machines are not extended machines, with files and other nice features.
- They are exact copies of the bare hardware including kernel/user mode, I/O, interrupts and everything else the real machines has.
- Each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the bare hardware.



**CMS (Conversational Monitor System**):

- It is an Interactive timesharing system.
- Multiprogramming or extended machine with more convenient interface than the bare hardware.
- When a CMS program executes a system call, the call is trapped to the operating system in its own virtual machine.
- CMS then issues the normal hardware I/O instruction for reading its virtual disk.

When sun Microsystems invented the Java Programming Language, it also invented a virtual machine. The Java complier producer code for **JVM** which then typically is executed by software JVM interpreted.

**Advantage:**

The advantage of this approach is that the JVM code can be shipped over the internet to any computer that has a JVM interpreter and run there.

**Exokernels :**

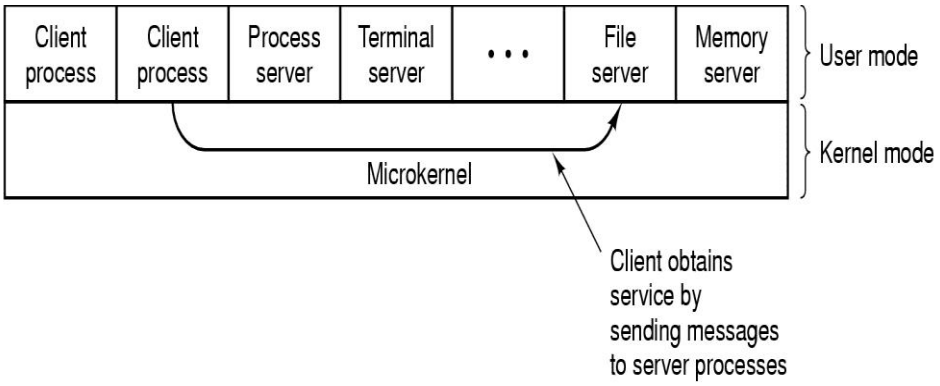
- Bottom layer, running in kernel mode is a program called the **Exokernel.**
- Its job is to allocate resource to virtual machine and then check attempts to use them to make sure no machine is trying to use others resources.

**Advantage:**

- The **advantage** of **exokernel** scheme is that it **saves a layer of mapping**.
- No remapping is needed.
- Separates the multiprogramming program from the user operating system code, but with less overhead,
- The exokernel need only to keep track of which virtual machine has been assigned to which resource.

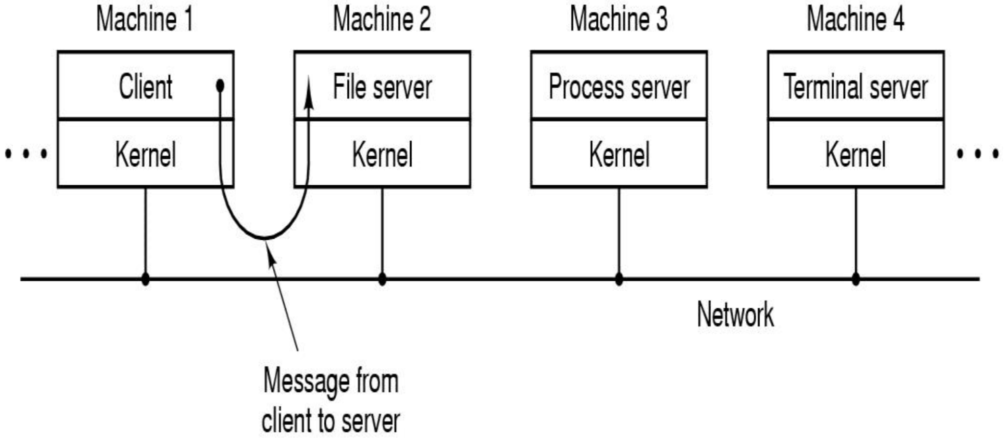
## Client – Server Model

- A trend in modern operating system is to take the idea of moving code up into higher layers even further and remove as much as possible from kernel mode leaving a minimal microkernel.
- The usual approach is to implement most of the operating system in user processes to request a service, such as reading a block of a file, a user process sends the request to a server process, which then does the work and sends back the answer.
- **Client :** Sends request to the server
- **Server :** Sends response to the client



### Advantage:

- By splitting the Operating system up into parts each of which only handles one fact of the system, such as file service, process service, terminal service or memory service each part becomes small and manageable.
- It has a good adaptability to use in distributed system.
- If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine or whether it was sent across a network to a server on a remote machine.



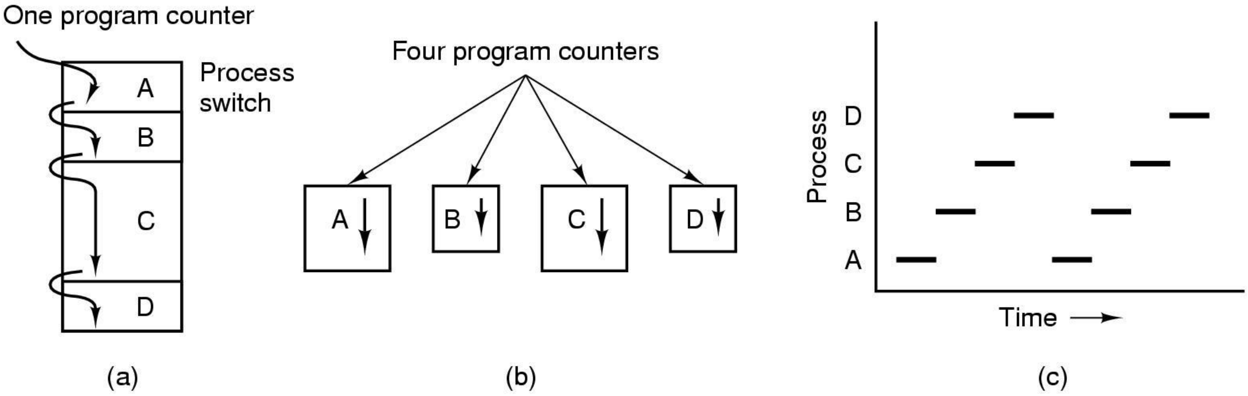
# UNIT – II

**Processes and Threads:**

- Processes are oldest and important abstractions that operating systems provide.
- It supports the ability to have concurrent even when there is only one CPU available.
- They turn a single CPU onto multiple virtual CPUs.

**Processes:**

- A process is an instance of a program running in a computer. It is close in meaning to task , a term used in some operating systems.
  - When a system is booted, many processes are started secretly.
  - In any multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds.
- **The Processes Model:**
- The process model as we have discussed it thus far is based on two independent concepts: resource grouping and execution. Sometimes it is useful to separate them; this is where threads come in.
- One way of looking at a process is that it is way to group related resources together.
- A process has an address space containing program text and data, as well as other resources. These resources may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them together in the form of a process, they can be managed more easily.
- All runnable software on the computer, sometimes including the OS is organized into a number of sequential processes.A process is just an instance of an executing program, including the current values of the program counter, registers and variables.
- The CPU switches back and forth from process to process, this rapid switching back and forth is called multiprogramming.
- A computer multiprogramming four programs in memory.



- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant
- A process is an activity of some kind. It has a program, input, output and a state.
- A single processor may shared among several processes, with some scheduling algorithm being used to determine when to stop work on one process and service a different one.
- If a program is running twice, it counts as two processes.
- **Process Creation:**
- OS need some way to create processes.

- In simple systems, it may be possible to have all the processes that will ever be needed be present when the system comes up.
- In general purpose systems, it needed to created and terminate processes as needed during operation.
- Four principal events – 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, several processes are created.
- Foreground processes – processes that interact with user and perform work for them.
- Background processes – which are not associated with particular users, but have some function.
- Processes in background to handle some activity like e-mail, web pages, news, printing etc are called daemons.

  UNIX – ps program, to list running processes.

  Windows – task manager.
- In interactive systems, user can start a program by typing a command or clicking an icon. These actions start a new process and run the selected program in it.
- UNIX – new process takes over the Windows
- Microsoft Windows – no window, but create one.
- In both systems, users may have multiple windows open at once.
- Processes are created applies only to the batch systems found on large mainframe.
- When the OS decided that it has the resource to run another job, it created a new process and runs the next job from the input queue in it.
- A new process is created by having an existing process execute a process creation system call.
- The process may be a running user process, a system process invoked from the keyboard or mouse or a batch manager process.
- UNIX – only one system call to create a new process: **fork.** After the fork two processes, the parent and the child have the same memory image, same environment strings, and the same open files.
- Windows – a single Win32 function call, **Create Process,** handles both process creation and loading the correct program into the new process.
- In both UNIX and Windows, after a process is created, the parent and child have their own distinct address spaces,
- In UNIX, the child's initial address space is a copy of the parent's, but they are definitely two distinct address spaces involved.
- **Process Termination:**
- After a process has been created, it starts running.
- Later the new process will terminate, 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)

1. **Normal exit:**

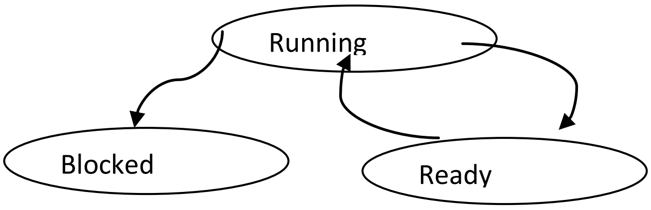
   * Most processes terminate because they have done their work.

   * When a compiler has compiled the program given to it, the compiler executes a system call to tell the OS that it is finished.
   * UNIX – exit
   * Exit Process -- Windows.

2. **Error exit:**
   * Most processes terminate by the process, often due to a program bug.
   * Ex: referencing nonexistent memory, divided by Zero.
3. **Fatal error:**
   * The user types the command **cc foo.c**
     to compile the program foo.c and no such files exists, the compiler simply exits.
   * Do not exit when bad parameters, they give pop-up dialog box and ask for user to try again.
4. **Killed by another process:**
   * A process executes a system call telling the OS to kill some other process.
   * UNIX - ***kill***
   * Win32 - ***Terminate Process.***

- **Process Hierarchies:**
  - Parent creates a child process, child processes can create its own process
  - Forms a hierarchy
  - UNIX calls this a "process group"
  - Windows has no concept of process hierarchy
  - all processes are created equal
  - When a process creates another process, the parent process and child process continue to be associated in certain ways.
  - The child process can itself create more processes, forming a process hierarchy.
  - In UNIX, a process and all of its children and further descendants together form a process group.
  - Each process can catch the signal, ignore the signal or take the default action, which is to be killed by the signal.
  - All the processes in the whole system belong to a single tree, with ***init*** at the root.
  - In windows no concept of a process hierarchy. All processes are equal.
  - When a process is created, the parent is given a special token that it can use to control the child.
- **Process States:**
  - 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.
  - Shell command **cat chapter1 chapter2 chapter3 | grep tree**
    1st process → running cat, concatenates and outputs three files.
    2nd process → running grep, selects all lines containing the word “tree”.
  - state diagram – the three states a process may in:

1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run)
3. Blocked (unable to run until some external event happens)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available.

- Transition 1 occurs when the operating system discovers that a process cannot continue right now.
- The process can execute a system call, such as pause, to get into blocked state
- Including UNIX, when a process reads from a pipe or special and there is no input available, the process is automatically blocked.
- Transitions 2 and 3 are caused by the process scheduler, a part of the operating system, without the process even knowing about them.
- Transition 2 occurs when the scheduler decides that the running process has run long enough and it is time to let another process have some CPU time.
- Transition 3 occurs when all the other processes have their fair share and it is time for the first process to get the CPU to run again.
- Transition 4 occurs when the external event for which a process was waiting happens.

## Implementation of Processes:

- The OS maintains a table called the Process table, with one entry per process.
- Entry contains important information about the process state, including its program counter, stack pointer, memory allocation, the status of its open files.

| Process management | Memory management | File management |
|---|---|---|
| Register<br>Program counter<br>Program status word<br>Stack pointer<br>Process state<br>Priority<br>Scheduling parameters<br>Process ID<br>Parent process<br>Process group<br>Signals<br>Time when process started<br>CPU time used | Pointer to next segment info<br>Pointer to data segment info<br>Pointer to stack segment info | Root directory<br>Working directory<br>File descriptors<br>User ID<br>Group ID |

**Some of the fields of a typical process table entry.**

- Each I/O device class (e.g., floppy disks, hard disks, timers, terminals) is a location (often near the bottom of memory) called the **interrupt vector.**
- It contains the address of the interrupt service procedure.
- Suppose that user process 3 is running when a disk interrupt occurs.
- User process 3's program counter, program status word, and possibly one or more registers are pushed onto the (current) stack by the interrupt hardware.
- The computer then jumps to the address specified in the disk interrupt vector.
- The interrupt vector contains the address of the interrupt service procedure

**Threads:**

- A thread is the smallest unit of processing that can be performed in an OS.
- In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads.
- In OS, each process has an address space and a single thread of control.

## The Thread Model:

- The process model is based on two independent concepts:

  **i) Resource grouping and ii) execution.**

i) **Resource grouping** is a way of group related resources together.

These resources may include open files, child processes, pending alarms, signal handlers, accounting information and more.

By putting them together in the form of a process, they can be managed more easily.

ii) The concept a process has is a thread of execution, is termed as **thread.**

- Thread has a program counter that keeps track of which instruction to execute next.
- It has registers, which hold its current working variables.
- It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from.
- The threads share an address space, open files, and other resources.
- In the latter case, processes share physical memory, disks, printers, and other resources.
- Because threads have some of the properties of processes, they are sometimes called lightweight processes.
- The term multithreading is also used to describe the situation of allowing multiple threads in the same process.
- In Fig. 2-6(a) we see three traditional processes. Each process has its own address space and a single thread of control.
- In contrast, in Fig. 2-6(b) we see a single process with three threads of control

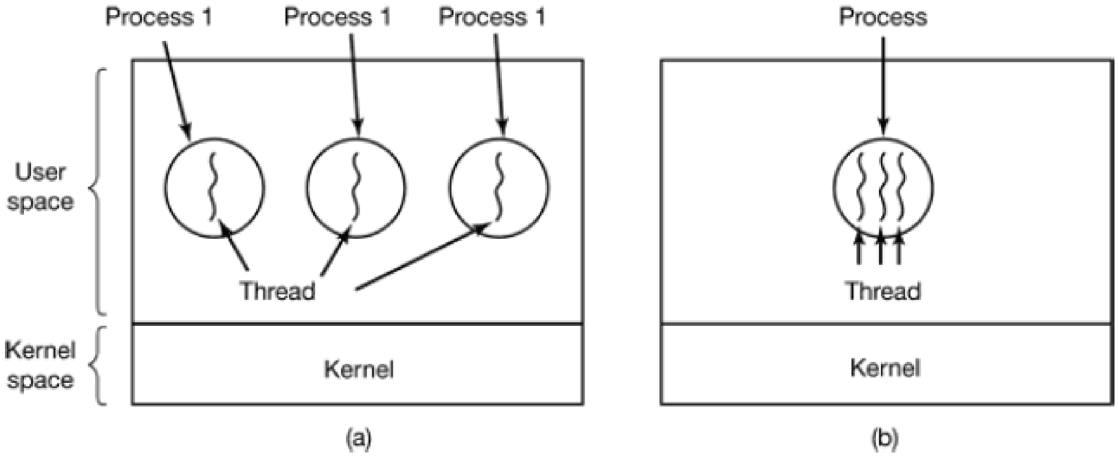


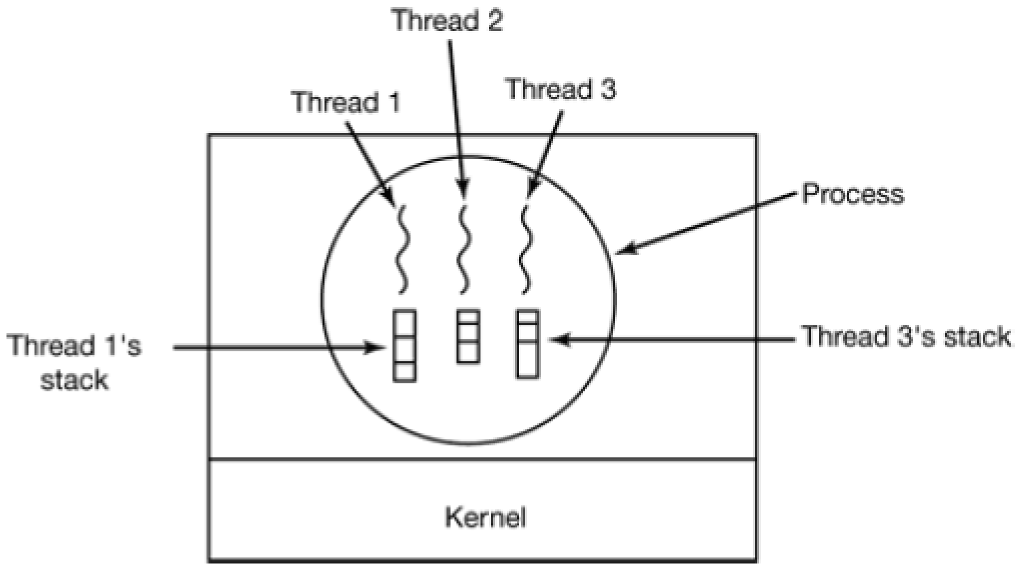
**Figure 2-6.** (a) Three processes each with one thread. (b) One process with tree threads.

- When a multithreaded process is run on a single-CPU system, the threads take turns running.
- Different threads in a process are not as independent as different processes.
- All threads have exactly the same address space, which means that they also share the same global variables.
- Items shared by all threads in a process.

**Per process items**
Address space
Global variables
Open files
Child processes
Pending alarms
Signals and signal handlers
Accounting information

**Per thread items**
Program counter
Registers

- Items private to each thread
- A thread can be in any one of several states: running, blocked, ready or terminated.
- A running thread currently has the CPU and it's active.
- A blocked thread is waiting for some event to unblock it.
- A ready thread is scheduled to run and will as soon as its turn comes up.
- The transitions between thread states are the same as the transitions between process states.



- A parameter ***to thread – create*** typically specifies the name of a procedure for the new thread to run.
- When a thread has finished its work, it can exit by calling a library procedure, ***thread_exit.***
- One thread can wait for a thread to exit by calling a procedure***, thread_join***.
- Thread call is ***thread_yield,*** which allows a thread to voluntarily give up the CPU to let another thread run.
- Other calls allow one thread to wait for another thread to finish some work.

## Thread Usage

- There are several reasons for having these miniprocesses, called threads.
- **The main reason for having threads is :**
  1. That in many applications, multiple activities are going on at once.
     By decomposing such an application into multiple sequential threads that run in parallel, the programming model becomes simpler.

- We have seen this argument before. It is precisely the argument for having processes. Instead of thinking about interrupts, timers, and context switches, we can think about parallel processes.
- Only now with threads we add a new element the ability for the parallel entities to share an address space and all of its data among themselves.
- This ability is essential for certain applications, which is why having multiple processes (with their separate address spaces) will not work.
- A second argument for having threads is that since they do not have any resources attached to them, they are easier to create and destroy than processes.
- In many systems, creating a thread goes 100 times faster than creating a process. When the number of threads needed changes dynamically and rapidly, this property is useful.
- A third reason for having threads is also a performance argument. 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.

Example:

- User deletes a page from 800 page of a document. After checking the changed page, now user wants to change in 600th page.
- The word processor is forced to reformat the entire document up to page 600. So there may be delay to display the 600th page.
- Thread help here. If word processor is written as a two threaded program.
  **One thread interacts with the user and the other handles reformatting in the background.**
  **Many word processors have a feature of automatically saving the entire file to disk for every few minutes to protect the user data from system crash or power failure.**
  **The third thread can handle the disk backups without interfering with other two**.

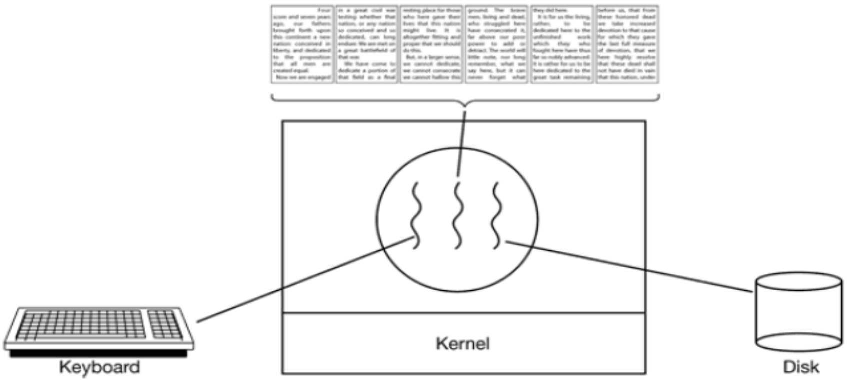
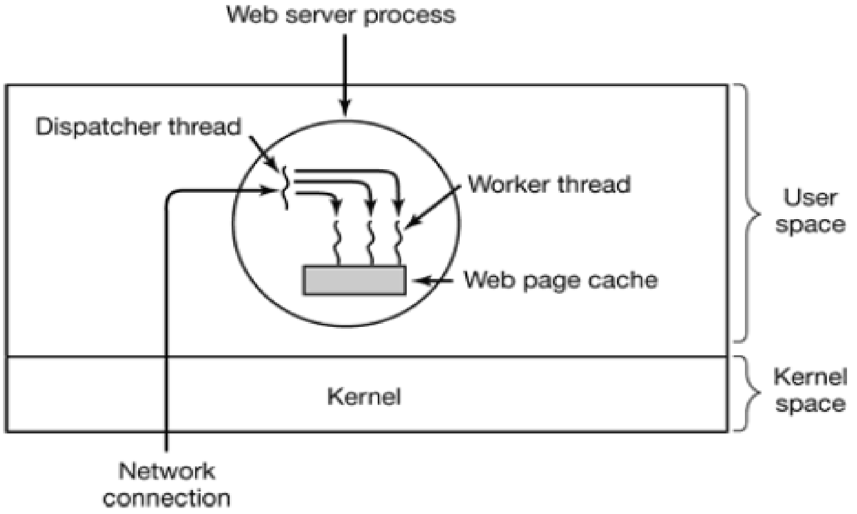


**Figure 2-9.** A word processor with three threads.

- If the program were single-threaded, then whenever a disk backup started, commands from the Input device are ignored until the back is finished.
- Three separate processes would not work here because all three threads need to operate on the document.
- By having three threads instead of three processes, they share a common memory to access the document.

**Another example: A server for a WWW site.**

- Request for pages come in and requested page is sent back to client.
- One thread, the dispatcher, reads incoming request for work from the network.
- An idle worker thread takes the request. The dispatcher wake up the sleeping worker, moving it from blocked to ready state.



**A multithread Web Server**

- When the worker wakes up, it checks to see if the request can be satisfied from the web page cache, to which all thread have access.

## Implementing Threads in User Space:

- There are two main ways to implement a threads package: in user space and in the kernel.
- The first method is to put the threads package entirely in user space. The kernel knows nothing about them.
- The threads run on top of a run-time system, which is a collection of procedures that manage threads.
- We have seen four of these already: thread_create , thread_exit , thread_wait , and thread_yield , but usually there are more.
- When threads are managed in user space, each process needs its own private thread table

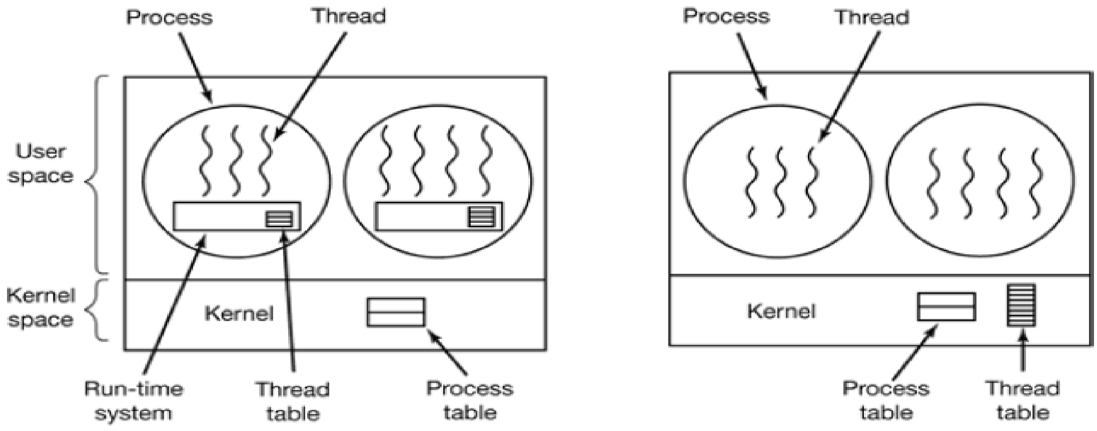


**Figure 2-13.** (a) A user-level threads package. (b) A threads package managed by the kernel.

- User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm

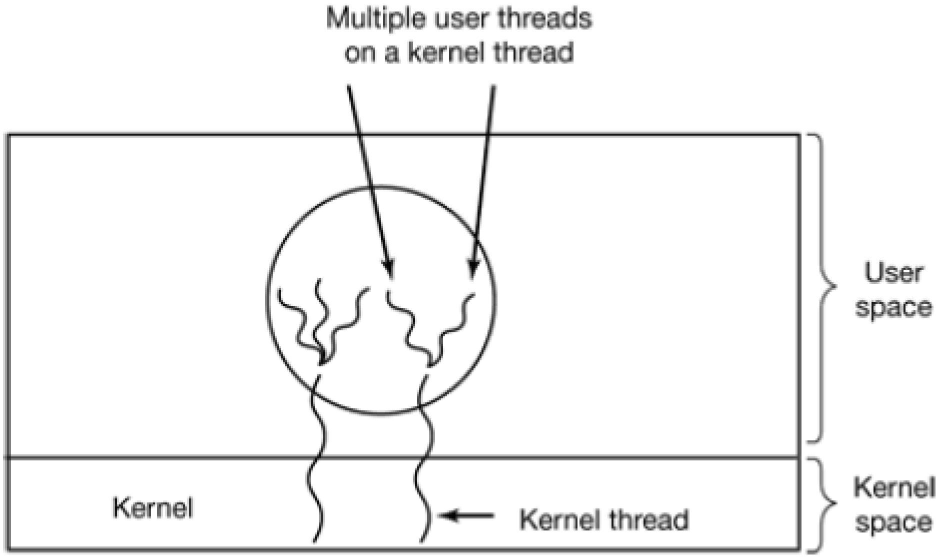
## Implementing Threads in the Kernel:

- The kernel’s thread table holds each thread’s registers, state, and other information.
- The information is the same as with user-level threads, but it is now in the kernel instead of in user space (inside the run-time system).

- This information is a subset of the information that traditional kernels maintain
- about each of their single-threaded processes, that is, the process state.
- In addition, the kernel also maintains the traditional process table to keep track of processes.

## Hybrid Implementations:

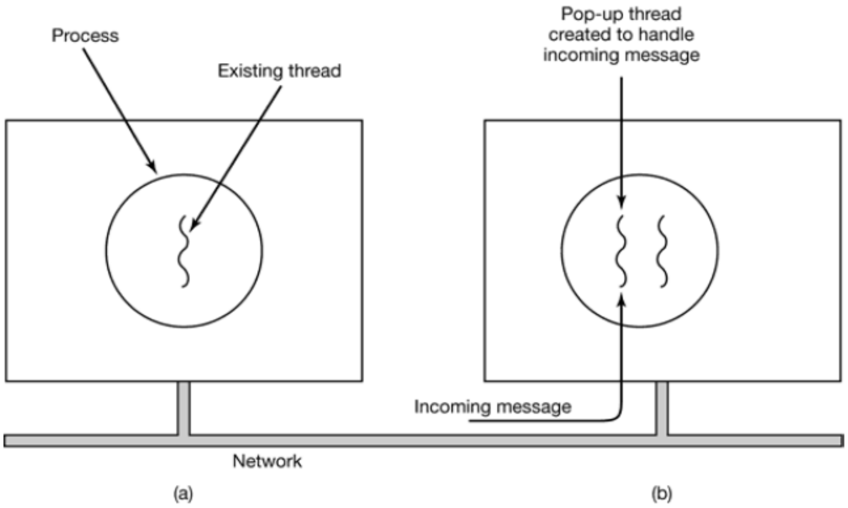
- Various ways have been investigated to try to combine the advantages of user-level threads with kernel-level threads.
- One way is use kernel-level threads and then multiplex user-level threads onto some or all of the kernel threads



**e 2-14.** Multiplexing user-level threads onto kernel-level threads.

## Pop-Up Threads:

- Threads are frequently useful in distributed systems. An important example is how incoming messages, for example requests for service, are handled.
- The traditional approach is to have a process or thread that is blocked on a receive system call waiting for an incoming message.
- When a message arrives, it accepts the message and processes it.
- completely different approach is also possible, in which the arrival of a message causes the system to create a new thread to handle the message. Such a thread is called a **pop-up thread**



## Making Single-Threaded Code Multithreaded:

- Many existing programs were written for single-threaded processes. Converting these to multithreading is much trickier than it may at first appear.
- Below we will examine just a few of the pitfalls.
- As a start, the code of a thread normally consists of multiple procedures, just like a process.
- These may have local variables, global variables, and procedure parameters.
- Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program do.
- These are variables that are global in the sense that many procedures within the
- thread use them (as they might use any global variable), but other threads should logically leave them alone.

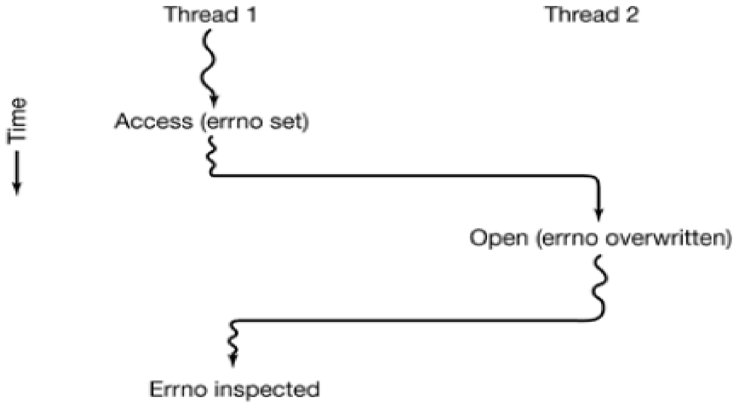


**Figure 2-16.** Conflicts between threads over the use of a global variable.

- In this way, each thread has its own private copy of *errno* and other global variables, so conflicts are avoided.
- In effect, this decision creates a new scoping level, variables visible to all the procedures of a thread, in addition to the existing scoping levels of variables visible only to one procedure and variables visible everywhere in the program.

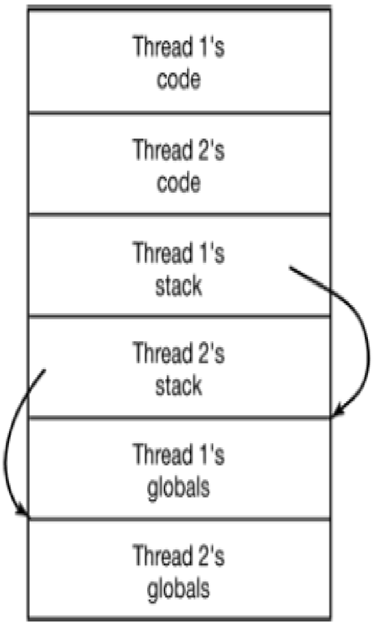


**Figure 2-17.** Threads can have private global variables.

**Interprocess Communication (IPC):**

- Processes frequently need to communicate with other processes.
- Ex: a shell pipeline, the output of the first process must be passed to the second process, and so on down the line.
- There is a need for communication between processes.
- Three issues :
  1. How one process can pass information to another?
  2. Making sure two or more processes do not get in each other's way.
  3. Proper sequencing when dependencies are present.
- Two of these issues apply equally well to threads.
  1. Passing information – is easy for threads – they share a common address space.
  2. Keeping out of each other's hair and proper sequencing

**Race conditions:**

- A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly.
- **In OS,** processes that are working together may share some common storage that each one can read and write.
- The shared storage may in main memory or it shared file.
- Ex: a print spooler.
  - When a process wants to print a file, it enters the file name in a **special spooler directory.**
  - Another process, **the printer daemon**, periodically checks to see if there are any files
  - To be printed, and if there are, it prints them and then removes their names from the Directory.
  - Imagine that our spooler directory has a very large number of slots, numbered 0, 1, 2, each one capable of holding a file name.
  - Tow shared variables, ***out,*** which points to the next file to be printed, and ***in***, which points to the next free slot in the directory.

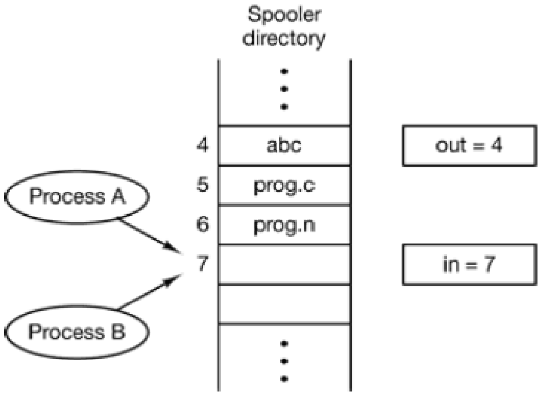


**Figure 2-18.** Two processes want to access shared memory at the same time.

  - Slots 0 to 3 are empty and slots 4 to6 are full processes A and B decide they want to queue a file for printing

- Process A reads ***in*** and stores the value, 7, in a local variable called ***next_free_slot.***
- CPU decides that process A has run, it switches to process B.
- Process B also reads ***in,*** and also gets a 7.It too stores it in its local variable ***next_ free_ slot.***
- Process B now continues to run. It stores the name of its file in slot 7 and updates ***in*** to be an 8.
- Process A runs again, starting from the place it left off. It looks at, ***next_free_slot*** finds a 7 there, and writes its file name in slot 7.
- It computers ***next_free_slot*** + 1, which is 8, and sets in to8.
- Where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are **called race conditions.**

- **Critical Regions**
  - We need is mutual exclusion, that is, some way of making sure that if one process is using a stared variable or file , the other processes will be exclude from doing the same thing
  - **A process has to access shared memory or files, or do other critical things that can lead to that part of the program where the shared memory is accessed is called the critical region or critical section.**

**Four conditions to hold to have a good solution**;

1. No tow 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.

- Process A enters its critical region at time t1. A little later, at time T2 process B attempts to enter its critical region but fails because another process is already in its critical region.
- B is temporarily suspended until time T3 when A leaves its critical region, allowing B to enter immediately.

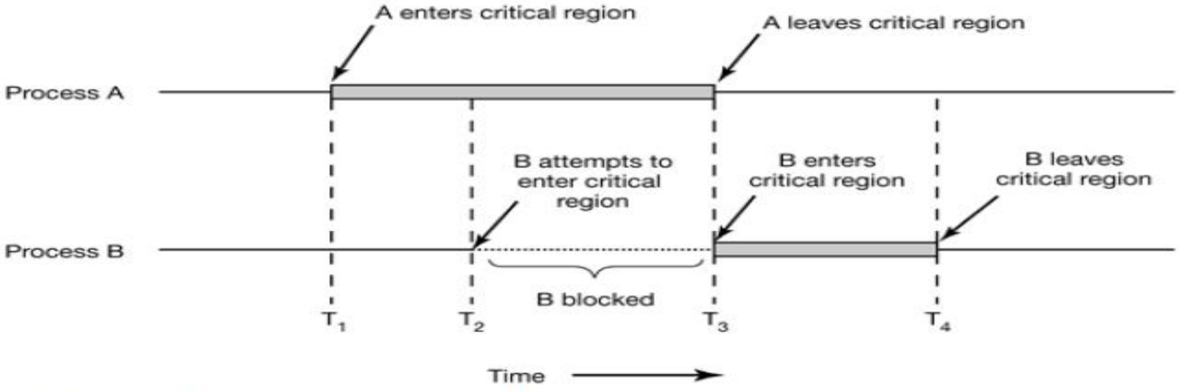


**Figure 2-19.** Mutual exclusion using critical regions.

- **Mutual Exclusion with Busy Waiting**

**Disabling interrupts**

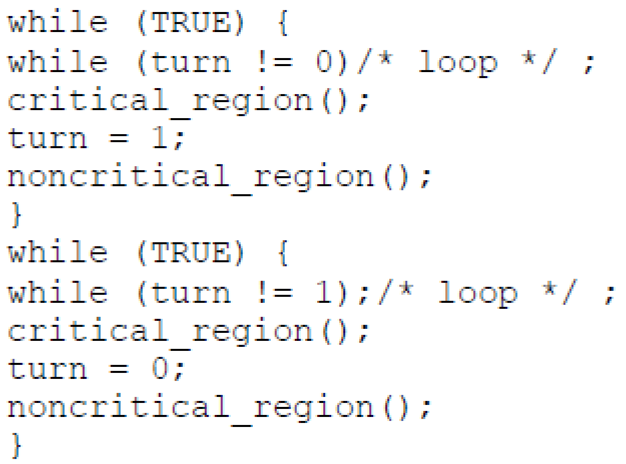
- On a single – processor system, the simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it.
- With interrupts disabled, no clock interrupts can occur.
- Once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

### Lock variables

- Single, shared variables, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region.
- If the lock is already 1, the process just waits until it becomes 0.
- A 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

## Strict Alternation

- A third approach to the mutual exclusion problem
- the integer variable turn , initially 0, keeps track of whose turn it is to enter the critical region and examine or update the shared memory.
- Initially, process 0 inspects turn , finds it to be 0,and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.
- 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 .

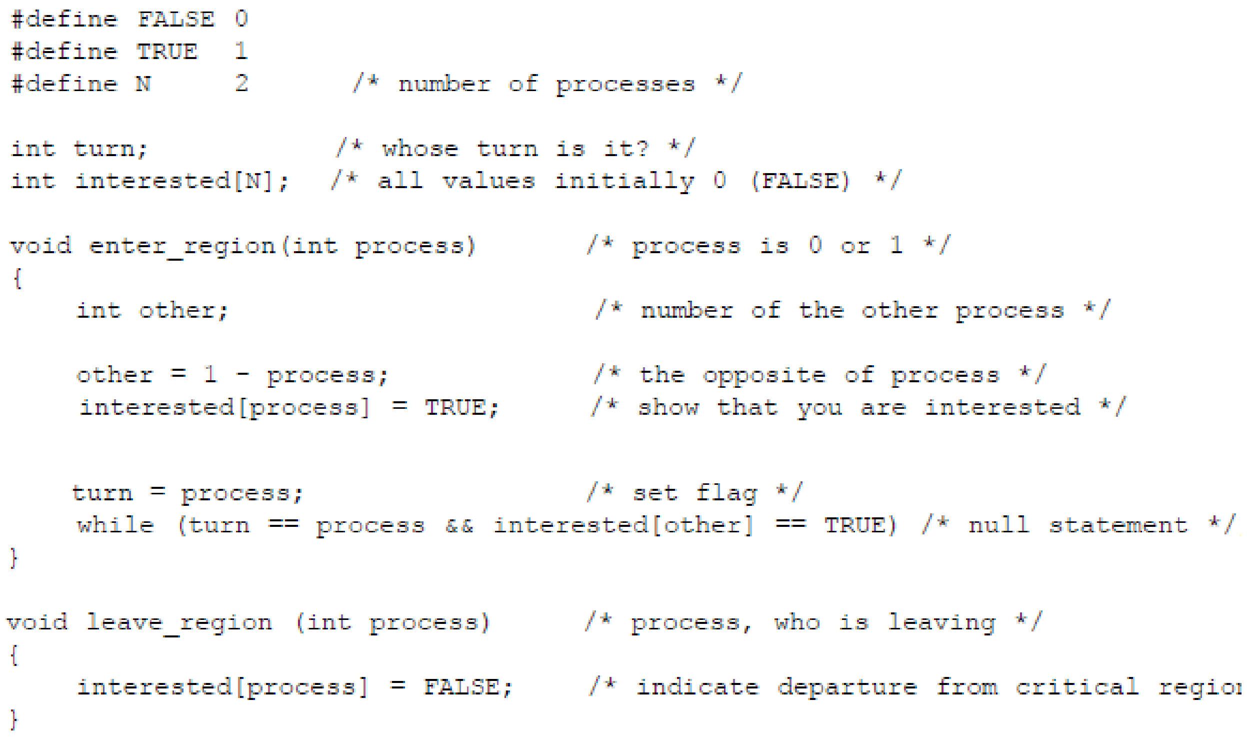


```
while (TRUE) {
while (turn != 0)/* loop */ ;
critical_region();
turn = 1;
noncritical_region();
}
while (TRUE) {
while (turn != 1);/* loop */ ;
critical_region();
turn = 0;
noncritical_region();
}
```

- When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region.

  Suppose that process 1 finishes its critical region quickly, so both processes are in their noncritical regions, with turn set to 0.
- Now process 0 executes its whole loop quickly, exiting its critical region and setting turn to 1.
- At this point turn is 1 and both processes are executing in their noncritical regions.

## Peterson’s Solution:

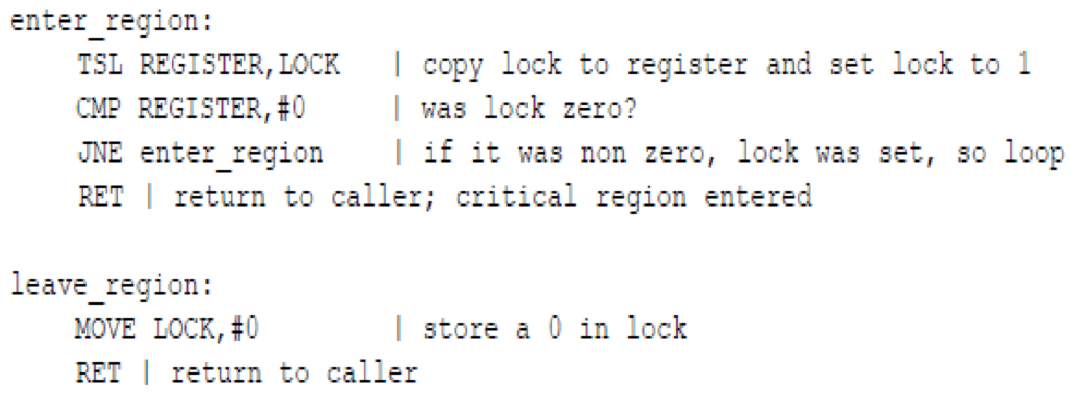


```
#define FALSE  0
#define TRUE   1
#define N      2       /* number of processes */

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

void enter_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 */
}

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

## The TSL(Test and Set Lock) Instruction

TSL REGISTER, LOCK

- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.
- It is important to note that locking the memory bus is very different from disabling interrupts.



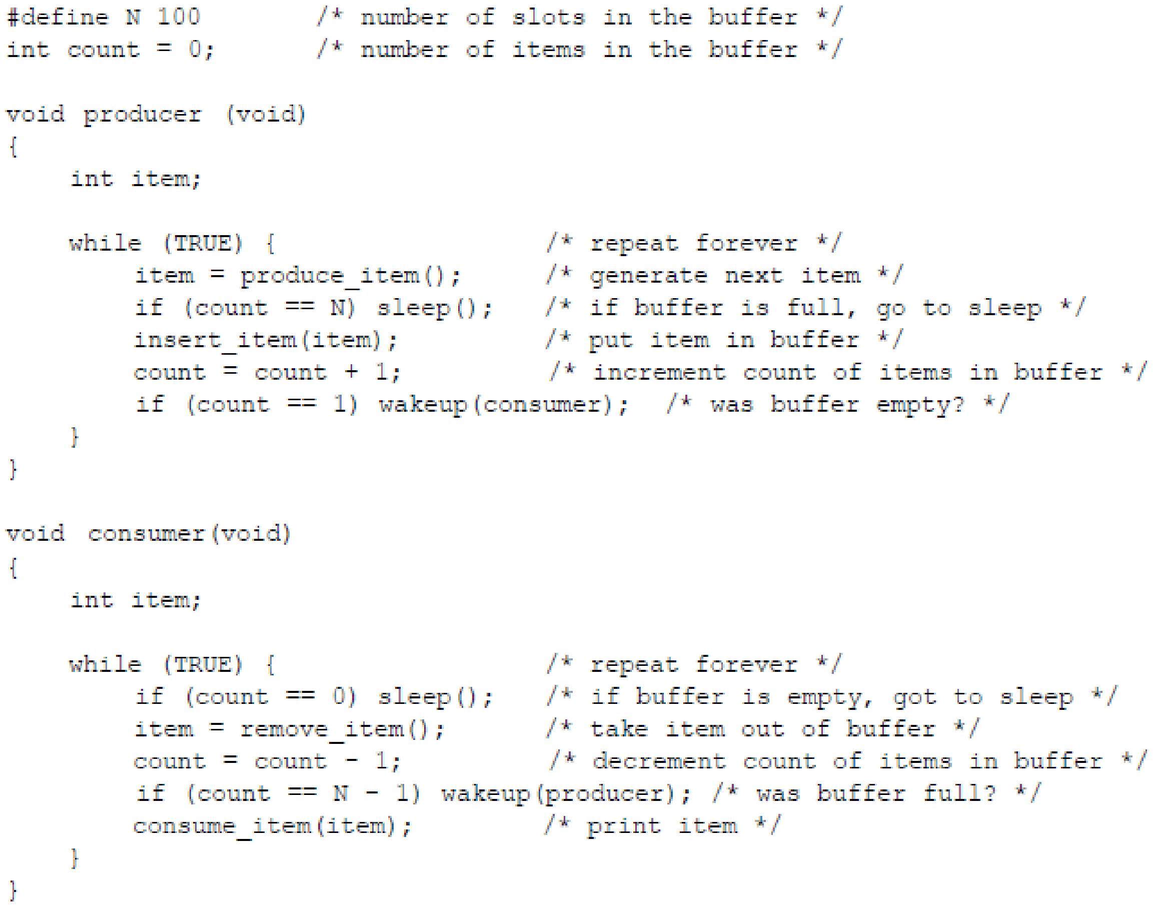
```
enter_region:
    TSL REGISTER,LOCK    | copy lock to register and set lock to 1
    CMP REGISTER,#0      | was lock zero?
    JNE enter_region     | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

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

- A process calls enter –region, which dose 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.
- The processes must call enter-region and leave-region at the correct times for the method to work.
- An alternative instruction to TSL is XCHG, which exchanges the contents of two locations atomically.

- **Sleep and Wakeup**
    - When a process wants to enter its critical region, it checks to see if the entry is allowed.
    - It is not, the process just sits in a tight loop waiting until it is.
    - Not only dose this approach waste CPU time, but it can also have unexpected effects
    - 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 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:

- 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.
- To keep track of the number of items in the buffer, we will need a variable, count .
- If the maximum number of items the buffer can hold is N , the roducer's code will first test to see if count is N . If it is, the producer will go to sleep; if it is not, the producer will add an item and increment count .
- The consumer's code is similar: first test count to see if it is 0. If it is, go to sleep, if it is nonzero, remove an item and decrement the counter.
- Each of the processes also tests to see if the other should be awakened, and if so, wakes it up.



```
#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 */
     }
}
```

**Figure 2-23.** The producer-consumer problem with a fatal race condition.

## Semaphores.

- E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use
- Dijkstra proposed having two operations, down and up
- The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value
- The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation.
- Semaphores are devices used to help with synchronization. If multiple processes share a common resource, they need a way to be able to use that resource without disrupting each other. You want each process to be able to read from and write to that resource uninterrupted.
- A semaphore will either allow or disallow access to the resource, depending on how it is set up. One example setup would be a semaphore which allowed any number of processes to read from the resource, but only one could ever be in the process of writing to that resource at a time.
-
    - A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.
    - The down operation on a semaphore checks to see if the value is greater than 0.if so, it decrements the value and just continues.
    - If the value is 0, the process is put to sleep without completing the down for the moment.
    - Once a semaphore operation has started, no other process can access the semaphore until the operation has completing or blocked.
    - This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.
    - Atomic actions, in which a group of related operations are either all performed without interruption or not performed at all.

## Mutexes:

- In computer programming, a mutex (mutual exclusion object) is a program object that is created so that multiple program thread can take turns sharing the same resource, such as access to a file. Typically, when a program is started, it creates a mutex for a given resource at the beginning by requesting it from the system and the system returns a unique name or ID for it.
    - When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutexes.
    - Mutexes are good only for managing mutual exclusion to some shared resource or piece of code.
    - A mutex is a variable that can be in one of tow states; unlocked or locked.
    - Only 1 bit is required to represent it, with 0 meaning unlocked and all other values meaning locked.
    - When a thread needs access to a critical region, ***it mutex _lock.***
    - On the other hand, if the mutex is already locked, the calling thread is blocked until the thread in the critical region is finished and calls ***mutex_ unlock.***
    - The code for ***mutex_lock and mutex_unlock*** for use with a user- level threads package.

    **Mutexes in Pthreads**

    - Pthreads provides a number of functions that can be used to synchronize threads.
    - The basic mechanism uses a mutex variable, which can be locked or unlocked, to guard each critical region.

| Thread Call | Description |
|---|---|
| Pthread_mutex_init | Create a mutex |
| Pthread_mutex_destory | Destory an existing mutex |
| Pthread_mutex_lock | Acquire a lock or block |
| Pthread_mutex_trylock | Acquire a lock or fall |
| Pthread_mutex_unlock | Release a lock |

**Monitors**

- A monitor is a collection of procedures, variable and data structures that are all grouped together in a special kind of module or package.
- Monitors have an important property that makes them useful for achieving mutual exclusion: only one process can be active in a monitor at any instant.
- Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls.

```
monitor example
        integer i;
        condition c;

        procedure producer()
        .

        end;
        procedure consumer();
        end;
        end monitor;
```

**Message Passing**

- **Message passing** - method of interprocess communication uses two primitives, **send and receive,** which , like semaphores and unlike monitors, are system calls rather than language constructs.

  ***Send(destination, &message);***

  ***And***

  ***Receive(source, &message);***

**Design Issues For Message-Passing Systems:**

Message passing systems have many challenging problems and design issues, especially if the communicating processes are on different machine s connected by a network.

* Message can be lost by the network.

* To guard against lost messages, sender and receiver can agree that as soon as a message has been received, the receiver will send back the acknowledgement message

* If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

**Barriers**

- Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase.
- This behavior may be achieved by placing a barrier at he end of each phase.
- When a process reaches the barrier, it is blocked until all processes have reached the barrier

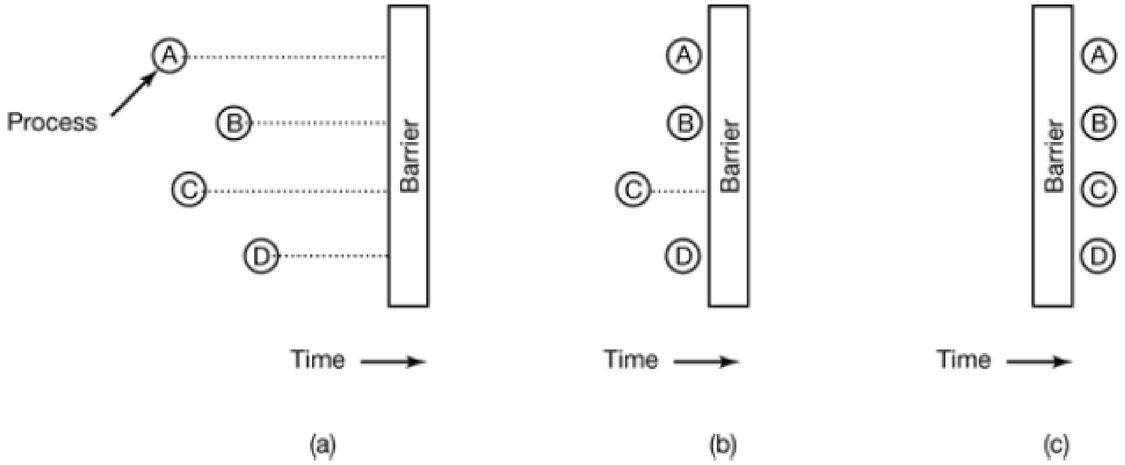


**Figure 2-30.** Use of a barrier. (a) Processes approaching a barrier. (b) All processes but blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

(a) Processes approaching a barrier
(b) All processes but one blocked at the barrier
(c) When the last process arrives at the barrier, all of them are let through

# UNIT - III

## SCHEDULING:

- When a computer is multiprogrammed, it frequently has multiple processes competing for the CPU at the same time.
- This situation occurs whenever two or more processes are simultaneously in the ready state.
- If only one CPU is available, a choice has to be made which process to run next.
- The part of the operating system that makes the choice is called the **scheduler** and the algorithm it uses is called the **scheduling algorithm**

## Introduction to Scheduling

- Back in the old days of batch systems with input in the form of card images on a magnetic tape, the scheduling algorithm was simple: just run the next job on the tape.
- With timesharing systems, the scheduling algorithm became more complex because there were generally multiple users waiting for service.
- CPU time is a scarce resource on these machines, a good scheduler can make a big difference in perceived performance and user satisfaction.
- Consequently, a great deal of work has gone into devising clever and efficient scheduling algorithms.

## Process Behavior

- When the system call completes, the CPU computes again until it needs more data or has to write more data and so on.
- Note that some I/O activities count as computing.
- For example, when the CPU copies bits to a video RAM to update the screen, it is computing, not doing I/O, because the CPU is in use.
- I/O in this sense is when a process enters the blocked state waiting for an external device to complete its work.

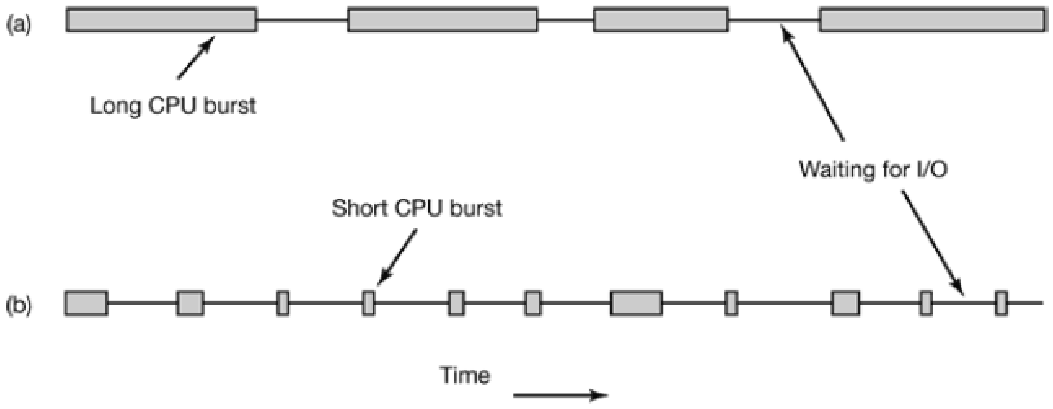


**Figure 2-37.** Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bo process. (b) An I/O-bound process.

## When to Schedule

- A key issue related to scheduling is when to make scheduling decisions. It turns out that there are a variety of situations in which scheduling is needed.
- First, when a new process is created, a decision needs to be made whether to run the parent process or the child process.
- Since both processes are in ready state, it is a normal scheduling decision and it can go either way, that is, the scheduler can legitimately choose to run either the parent or the child next.

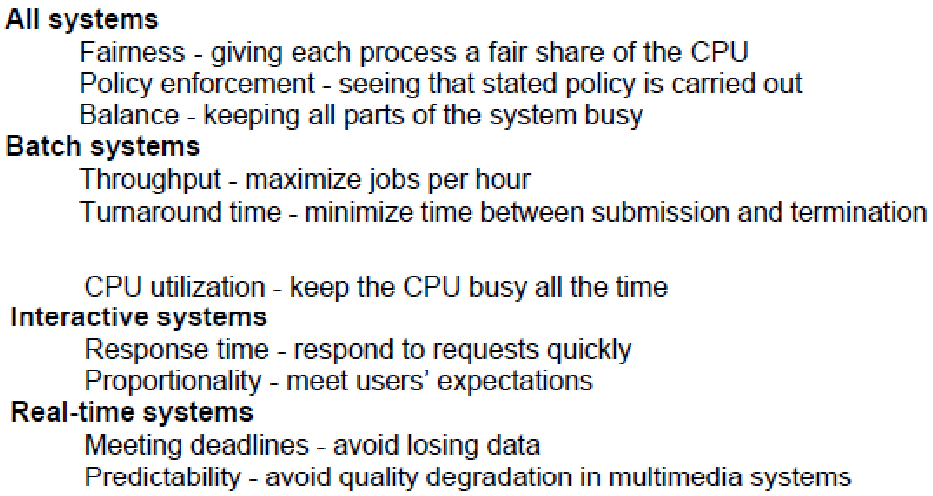
- Second, a scheduling decision must be made when a process exits. That process can no longer run (since it no longer exists), so some other process must be chosen from the set of ready processes.
- Third, when a process blocks on I/O, on a semaphore, or for some other reason, another process has to be selected to run.
- Fourth, when an I/O interrupt occurs, a scheduling decision may be made.

## Categories of Scheduling Algorithms:

1. Batch.
2. Interactive.
3. Real time.

- In batch systems, there are no users impatiently waiting at their terminals for a quick response.
- In an environment with interactive users, preemption is essential to keep one process from hogging the CPU and denying service to the others.
- In systems with real-time constraints, preemption is, oddly enough, sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly.

## Scheduling Algorithm Goals:



**All systems**

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

**Batch systems**

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination

CPU utilization - keep the CPU busy all the time

**Interactive systems**

Response time - respond to requests quickly
Proportionality - meet users' expectations

**Real-time systems**

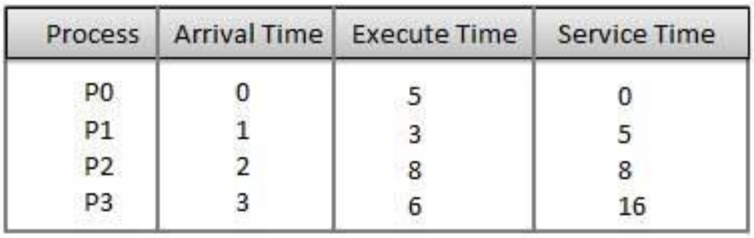
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

**Figure 2-38.** Some goals of the scheduling algorithm under different circumstances.

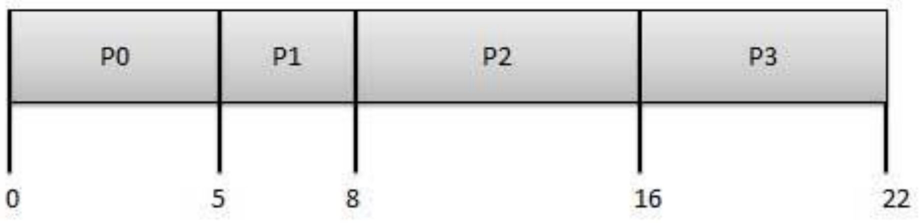
## Scheduling in Batch Systems:

## First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.



| Process | Arrival Time | Execute Time | Service Time |
|---|---|---|---|
| P0 | 0 | 5 | 0 |
| P1 | 1 | 3 | 5 |
| P2 | 2 | 8 | 8 |
| P3 | 3 | 6 | 16 |



Wait time of each process is as follows −

| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | 0 - 0 = 0 |
| P1 | 5 - 1 = 4 |
| P2 | 8 - 2 = 6 |
| P3 | 16 - 3 = 13 |

- Average Wait Time: (0+4+6+13) / 4 = 5.75

### Shortest Job First

- The scheduler picks the **shortest job first** .
- Look at Fig. 2-39. Here we find four jobs *A* , *B* , *C* , and *D* with run times of 8, 4, 4, and 4 minutes, respectively.
- By running them in that order, the turnaround time for *A* is 8 minutes, for *B* is 12 minutes, for *C* is 16 minutes, and for *D* is 20 minutes for an average of 14 minutes.

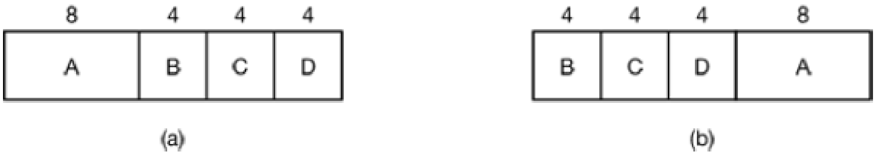


**Figure 2-39.** An example of shortest job first scheduling. (a) Running four jobs in the origi order. (b) Running them in shortest job first order.

- Now let us consider running these four jobs using shortest job first, as shown in Fig. 2-39(b).
- The turnaround times are now 4, 8, 12, and 20 minutes for an average of 11 minutes. Shortest job first is provably optimal.
- Consider the case of four jobs, with run times of *a* , *b* , *c* , and *d* , respectively.
- The first job finishes at time *a* , the second finishes at time *a* + *b* , and so on.
- The mean turnaround time is (4*a* + 3*b* + 2*c* + *d* )/4. It is clear that *a* contributes more to the average than the other times, so it should be the shortest job, with *b* next, then *c* , and finally *d* as the longest as it affects only its own turnaround time.

### Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

## Three-Level Scheduling:

- From a certain perspective, batch systems allow scheduling at three different levels,
- As jobs arrive at the system, they are initially placed in an input queue stored on the disk.
- The **admission scheduler** decides which jobs to admit to the system. The others are kept in the input queue until they are selected.
- A typical algorithm for admission control might be to look for a mix of compute-bound jobs and I/O-bound jobs.

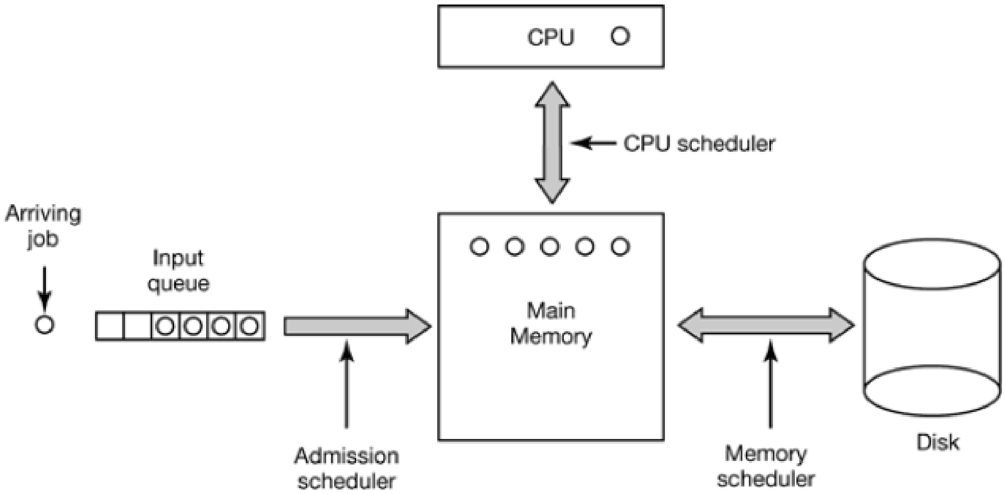


**Figure 2-40.** Three-level scheduling.

- The second level of scheduling is deciding which processes should be kept in memory and which ones kept on disk. We will call this scheduler the **memory scheduler**
- The third level of scheduling is actually picking one of the ready processes in main memory to run next.
- Often this is called the **CPU scheduler** and is the one people usually mean when they talk about the "scheduler."

## Scheduling in Interactive Systems:

### Round Robin Scheduling:

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a quantum.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

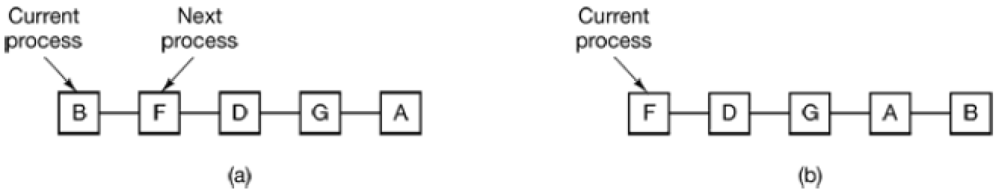
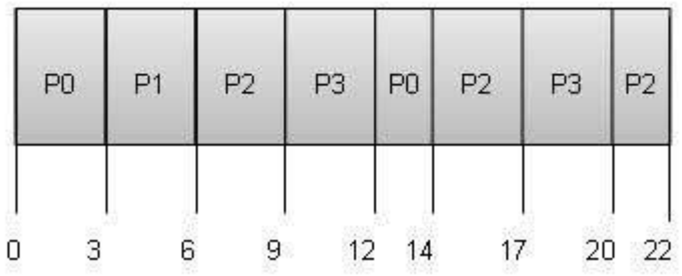


**Figure 2-41.** Round-robin scheduling. (a) The list of runnable processes. (b) The list of runna processes after *B* uses up its quantum.

Quantum = 3



Wait time of each process is as follows −

| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | (0 - 0) + (12 - 3) = 9 |
| P1 | (3 - 1) = 2 |
| P2 | (6 - 2) + (14 - 9) + (20 - 17) = 12 |
| P3 | (9 - 3) + (17 - 12) = 11 |

Average Wait Time: (9+2+12+11) / 4 = 8.5

## Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

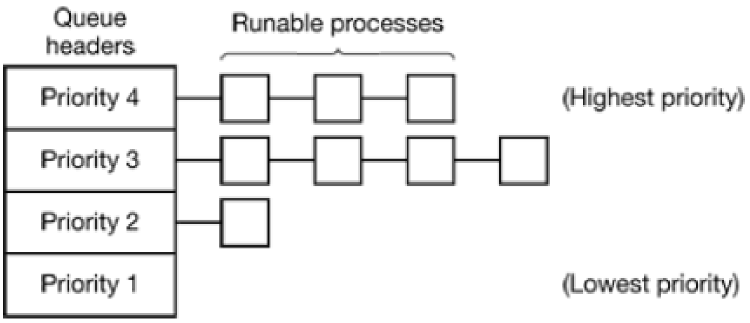
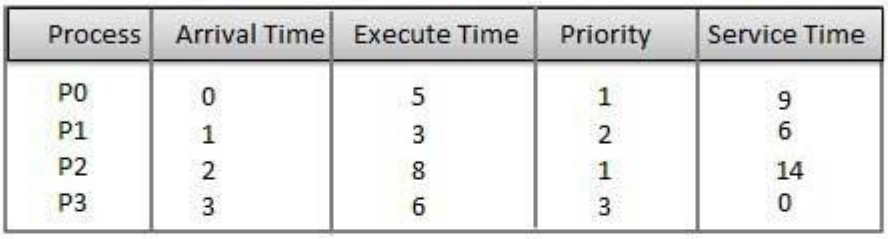
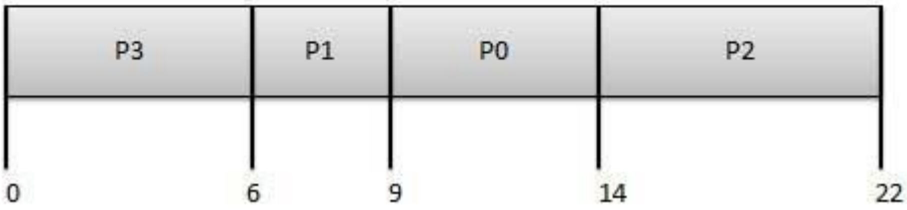


**Figure 2-42.** A scheduling algorithm with four priority classes.



| Process | Arrival Time | Execute Time | Priority | Service Time |
|---|---|---|---|---|
| P0 | 0 | 5 | 1 | 9 |
| P1 | 1 | 3 | 2 | 6 |
| P2 | 2 | 8 | 1 | 14 |
| P3 | 3 | 6 | 3 | 0 |



Wait time of each process is as follows −

| Process | Wait Time : Service Time - Arrival Time |
|---|---|
| P0 | 9 - 0 = 9 |
| P1 | 6 - 1 = 5 |
| P2 | 14 - 2 = 12 |
| P3 | 0 - 0 = 0 |

Average Wait Time: (9+5+12+0) / 4 = 6.5

## Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

## Shortest Process Next:

- Now suppose its next run is measured to be *T* 1 . We could update our estimate by taking a weighted sum of these two numbers, that is, *aT* 0 + (1 - *a* )*T* 1 .
- Through the choice of *a* we can decide to have the estimation process forget old runs quickly, or remember them for a long time. With *a* = 1/2, we get successive estimates of *T* 0 , *T* 0 /2 + *T* 1 /2, *T* 0 /4 + *T* 1 /4 + *T* 2 /2, *T* 0 /8 + *T* 1 /8 + *T* 2 /4 + *T* 3 /2
- After three new runs, the weight of *T* 0 in the new estimate has dropped to 1/8. The technique of estimating the next value in a series by taking the weighted average of the current measured value and the previous estimate is sometimes called **aging** .

## Lottery Scheduling:

- The basic idea is to give processes lottery tickets for various system resources, such as CPU time.
- Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource.
- When applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize.

## Fair-Share Scheduling:

- If user 1 starts up 9 processes and user 2 starts up 1 process, with round robin or equal priorities, user 1 will get 90% of the CPU and user 2 will get only 10% of it.
- To prevent this situation, some systems take into account who owns a process before scheduling it.
- In this model, each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it.
- Thus if two users have each been promised 50% of the CPU,

# Scheduling in Real-Time Systems:

- Real-time systems are generally categorized as
- **hard real time** , meaning there are absolute deadlines that must be met,
- **soft real time** , meaning that missing an occasional deadline is undesirable, but nevertheless tolerable.

**MEMORY MANAGEMEN T**

### Introduction:

- ➢ Main memory (RAM) is an important resource that must be carefully managed.
- ➢ Computers have a memory hierarchy with
  – A few megabytes of very fast, expensive, volatile cache memory.
  -A few gigabytes of medium-speed, medium-priced, volatile main memory, and
- ➢ – A few terabytes of slow, cheap, nonvolatile disk
  storage, DVDs, and USB sticks.
- ➢ The operating system must abstract this hierarchy into a useful model, then manage the memory model.

It is the job of the operating system that manages (part of) the memory hierarchy is called the memory.

- ▪ Job is to efficiently manage memory: keep track of which part of memory are of memory are in use.
- ▪ Allocation memory to processes when they need it when they are done.

➢ **No Memory Abstraction**

- ➢ Mainframe computers before 1960, minicomputers before 1970, and personal computers before 1980 had no memory abstraction.
- ➢ Every program saw the physical memory as it was.
- ➢ When a program executed an instruction like
- ➢ **MOV REGISTER1, 1000**
- ➢ the computer just moved the contents of physical
- ➢ memory location **1000** to **REGISTER1**.
- ➢ It was not possible to have two running programs in memory at the same time.

- Three variation are:
    - The operating system may be at the bottom of memory in RAM
    - (Random Access Memory)
    - It may be in ROM (Read Only Memory) at the top of memory, as shown in (b.)
    - The device drive may be at the top of memory in a ROM and the system in RAM down, below, as shown in (-c.)

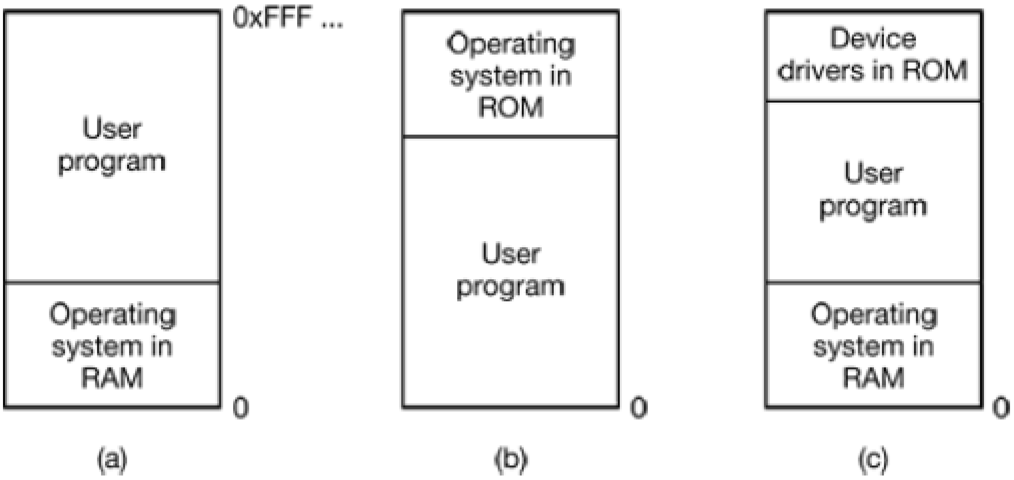


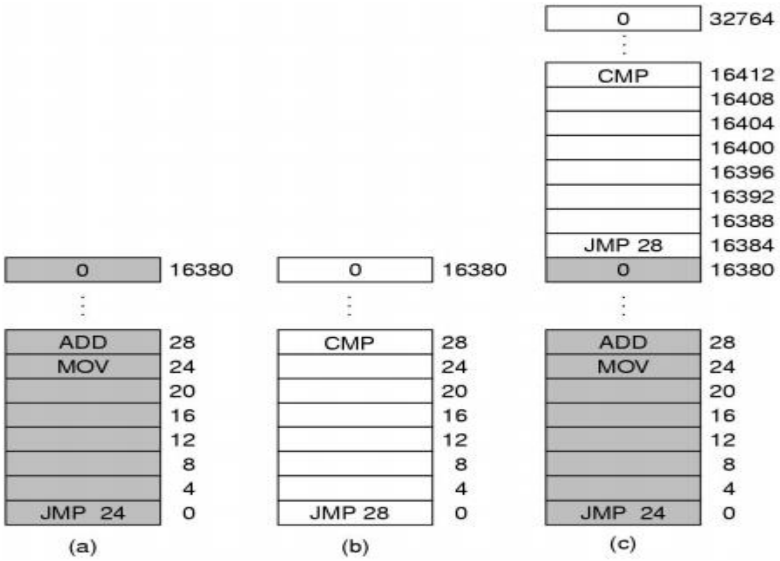
Figure-1: Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist.

- The first model was used on mainframes and minicomputers.
- The second model is used on some handheld computers and embedded system.
- The Third model was used by early personal computer.
- e.g.: running (MS-DOS), where the portion of the system in the ROM is called the BIOS (Basic Input Output System).
- Models (a) and(c) have the disadvantage that a bug in the user program can wipe out the operating system.

### Running Multiple programs without a memory Abstraction

- It is possible to run multiple programs at the same time. With the addition of some special hardware, it is possible to run multiple programs concurrently, even without swapping.
- Memory was divided into 2-KB blocks and each one was assigned a 4-bit protection key held in special registers inside the CPU.

.



•

- A machine with a 1-MB memory needed only in special registers for a total of 256bytes of key storage.
- The first program starts out by jumping to address 24, which contains a MOV instruction. The second program start out by jumping  to address 28, which contains a CMP instruction
- After the programs are loaded they can be run. Since they have different memory keys, neither one can damage the other.
- When the first program starts, it executes the JMP 24 instruction, which jumping to the instruction, as expected. This program functions normally.
- After the first program has run long enough, the operating system may decide to run the second program, which has been loaded above the first **one**, at address 16,384.
- The first instruction executed is JMP 28, which jumps to the ADD instruction in the first program, instead of the CMP instruction it is supposed to jump to. The

➢ **A Memory Abstraction :Address Spaces**

- Physical memory to processes has several major drawbacks.
- First, if user programs can address every bytes of memory, they can easily trash the operating system.
- This problem exists even if only one user program (application) is running.
- Second it is difficult to have multiple programs running at once.

- **Address Space:-**

  Two problems to solve to have multiple applications in memory without their interfering with each other:

  Protection and Relocation

  **Protection :-**

  Create a new abstraction for memory: the address space, the unique set of addresses that a process can use to address memory.

  Examples of address spaces

  – Telephone numbers: 0,000,000 to 9,999,999, some numbers not used

  – I/O ports on the Pentium: 0 to 16383

**Relocation :-**

**Address Space**

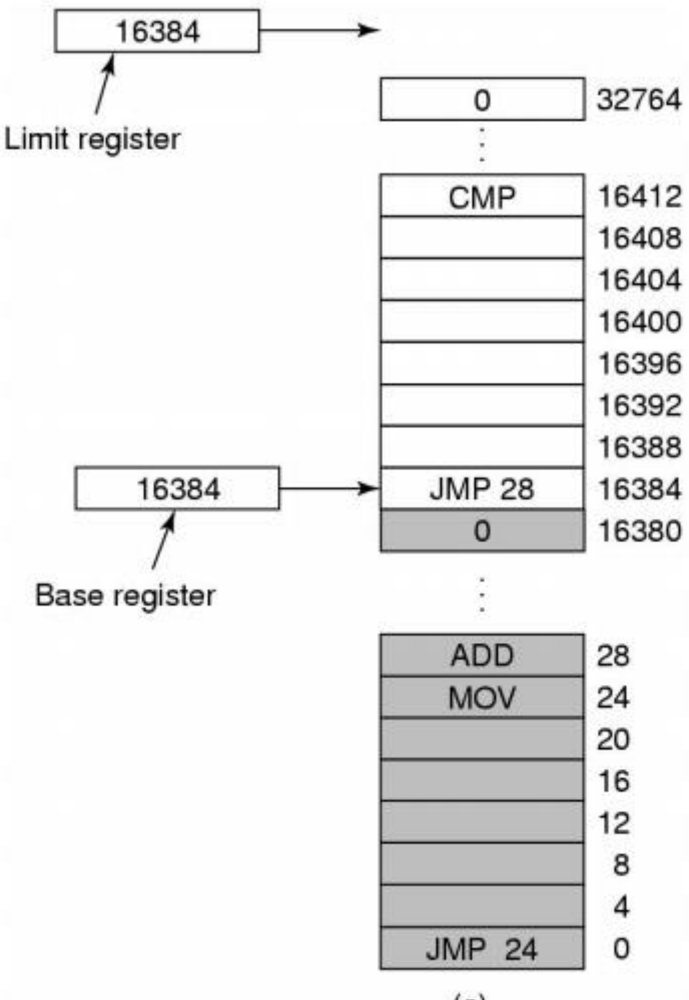
Give each program its own independent address space, so address 28 in one program means a different physical location than address 28 in another program.

**The Notion of an Address Space**

- The process concept create creates a kind of abstract CPU to run programs, the address space creates a kind of abstract memory for programs to live in.
- An **address space** is the set of address that a process can use to address memory. Each process has its own address space, independent of those belonging to other proc-process.
- The concept of an address space is very general and occurs is many contexts.
- A local telephone number is usually a 7-digit number.
- The address space for telephone number thus runs from0, 000,000 to 9,999,999.
- Address space do not have to be numeric. The set of.com Internet domain is also an address space.
- This address space consists of all string of length 2 to 63 character s that can be made using letters , numbers, and hyphens, followed by.com

**Base and Limit Registers:**

- Two special hardware registers, usually called the **base** and **limit** register.
    - When a process is run, the base register is loaded with the beginning physical address and the limit register is loaded with the program length.
    - The base and limit values that would be loaded into these registers for the first program are 0 and 16384, respectively.
    - The values used when the second program is loaded are 16384 and 16384, respectively.



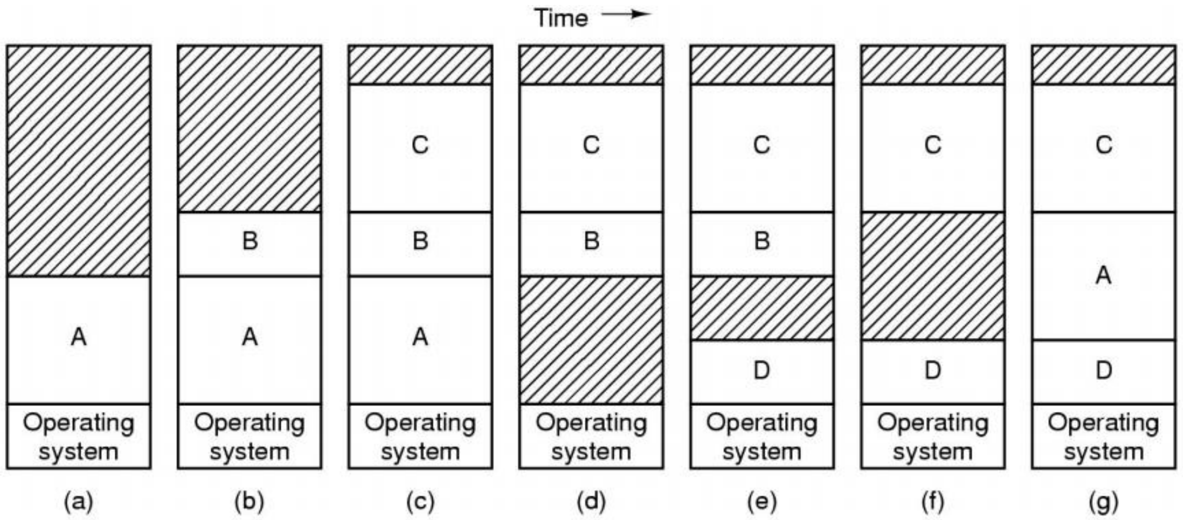
(c)

- When base and limit registers are used, programs are loaded into consecutive memory locations wherever there is room and without relocation during loading.
- When a process is run, the base register is loading with the physical address where its program beings in memory and the limit register is loaded with the length of the program.

**A disadvantage** of relocation using base and limit registers is the need to per-form and addition and a comparison on every memory reference.

**Swapping**

- **Swapping,** consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk.
- Idle processes are mostly stored on disk, so they do not take up any memory when they are not running
- **Virtual memory,** allows programs to run even when they are only partially in main memory.
- B goes out. Finally A comes in again. .
- The operation of a swapping system only process A is in memory.
- Then processes B and C are created or swapped in from disk. In fig: 1.(d) A is swapped out to disk.

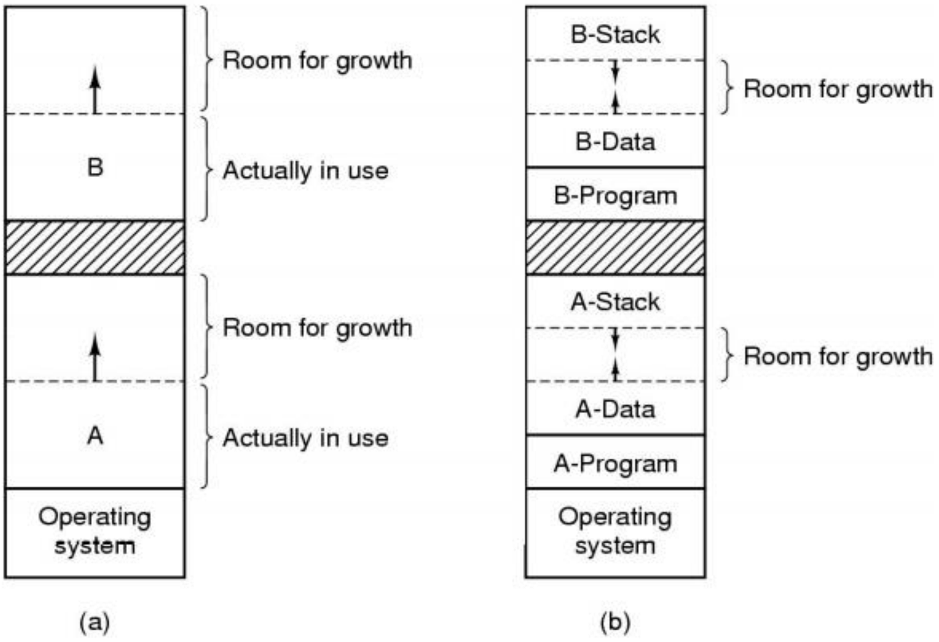


- Then D comes in and B goes out. Finally A comes in again.
- Since A is now at a different location, addresses contained in it must be relocated, either by software when it is swapped in or (more likely) by hardware during program execution.

- When swapping creates multiple holes in memory, it is possible to combine them all into one by one by moving all the processes downward as far as possible. This technique is known as **memory compaction.**
- When swapping processes to disk, only the memory actually in use should be swapped; it is wasteful to swap the extra memory.

Allocating space for a growing data segment

Allocating space for growing stack and data segment



### Managing Free Memory

- When memory is assigned dynamically, the operating system must manage it.
- There are two ways to keep track of memory usage: **bitmaps and free lists.**

### Memory Management with Bitmaps

- With a bitmap, memory is divided into allocation units as small as a few words and 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.
- The size of the allocation unit is an important design issue. The smaller the allocation unit, the larger the bitmap.
- A memory of 32n bits – use n map bits, so the bitmap will take up only 1/33 of memory.
- The bitmap provide a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit.

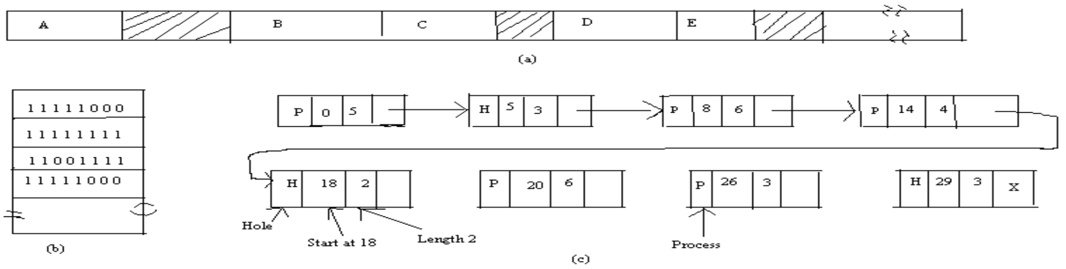
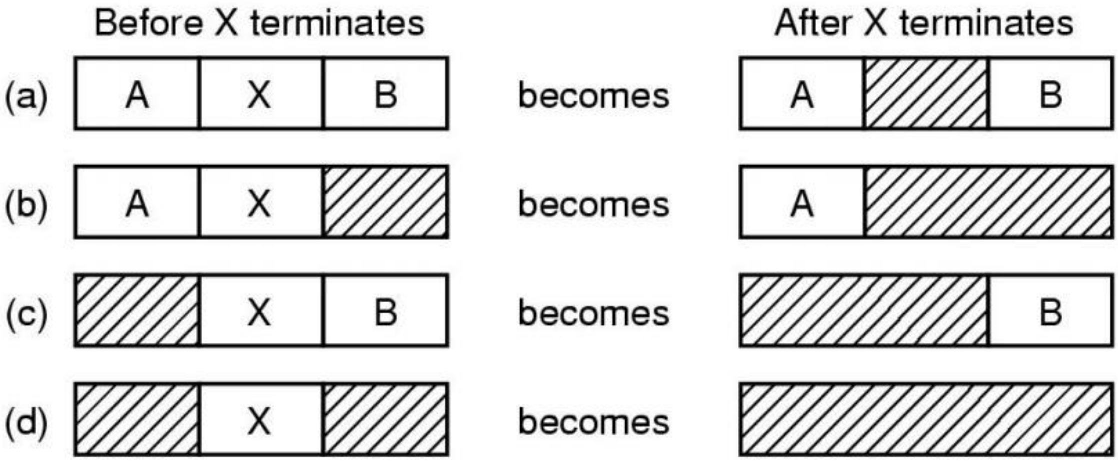


figure-1: (a) A part of memory with five processes and three hole. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

### Memory Management with Linked Lists

- Keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment either contains a process or is an empty hole between two processes.
- The segment list is kept sorted by address. Sorting this way has the advantage that when a process terminates or is swapped out, updating the list is straightforward.



(a) Updating the list requires replacing a P by an H.
(b) & (c) two entries are coalesced into one and the list becomes one entry shorter.
(4) Three entries are merged and two items are removed from the list.

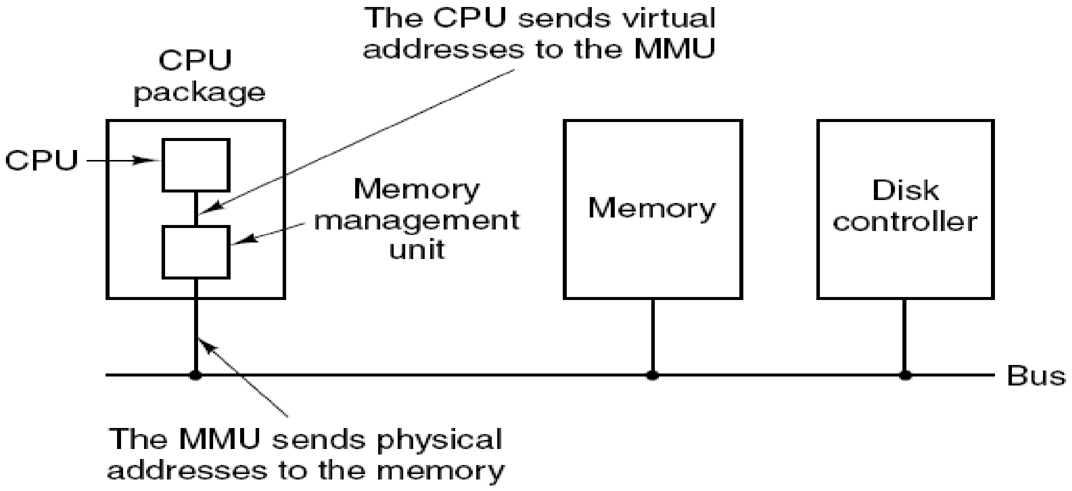
## VIRTUAL MEMORY

Virtual memory is a feature of an operating system that enables a process to use a memory (RAM) address space that is independent of other processes running in the same system, and use a space that is larger than the actual amount of RAM present, temporarily relegating some contents from RAM to a disk, with little or no overhead.

- The basic idea behind virtual memory is that each program has its own address space,which is broken up into chunks called **pages.**

**Paging:**

- Paging is a memory management technique in which the memory is divided into fixed size pages. Paging is used for faster access to data. When a program needs a page, it is available in the main memory as the OS copies a certain number of pages from your storage device to main memory. Paging allows the physical address space of a process to be noncontiguous.


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

- Virtual memory systems use a technique called paging. On any computer, programs reference a set of memory addresses when a program executes an instruction like.
- The contents of memory address 1000 to REG addresses can be generated using indexing, base registers, segment registers.
- This program –generated addresses are called virtual addresses and form the virtual address space.
- On computer without virtual memory the virtual address is put directly onto the memory bus.
- When virtual memory is used the virtual addresses do not go directly to the memory bus.
- Instead they go to a mmu that maps the virtual addresses onto the physical memory addresses as illustrated in.
- The virtual addresses space is divided into fixed-size. The corresponding units in the physical memory are called **page frames.**
- The pages and page frames are generally the same size.
- When the program tries to success address 0. For example using the instruction

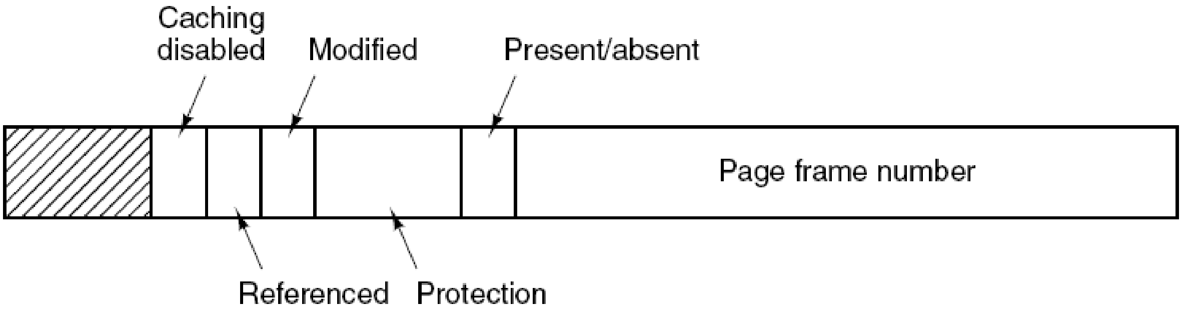
  **MOVE REG, 0**
- Virtual address 0 is sent to the MMU. The MMU sees that this virtual address falls in page 0[0 to 4095].
- **Move REG, 8192** is effectively transformed into **Move REG, 24576**
- A present/absent bit keeps track of which pages are physically present in memory.
- The MMU notices that the page is unmapped and causes the CPU to trap to the OS. This Trap is called a **Page Fault.**
- The page number is used as an index into the **page table**. Yielding the number of the page frame corresponding to that virtual page
- If the present/absent bit is 0, a trap to the operating system is caused.

➢ **Page Tables**

- The virtual address is spilt into a virtual page number and an offset.
- The virtual page number is used as an index into the page table to find the entry for that virtual page.
- From the page table entry, the page frame number is found.
- The page frame number, to the high-order end of the offset, replacing the virtual page number to form a physical address that can be sent to the memory.

➢ **Structure Of A Page Table Entry**

- 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 output this value. Next to it we have the present/absent bit.
- If this bit is l, 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 0cause a page fault.



The protection bits tell what kinds of success are permitted. In the simper from this field contains 1 bit. With 0 for read/write and l for read only.

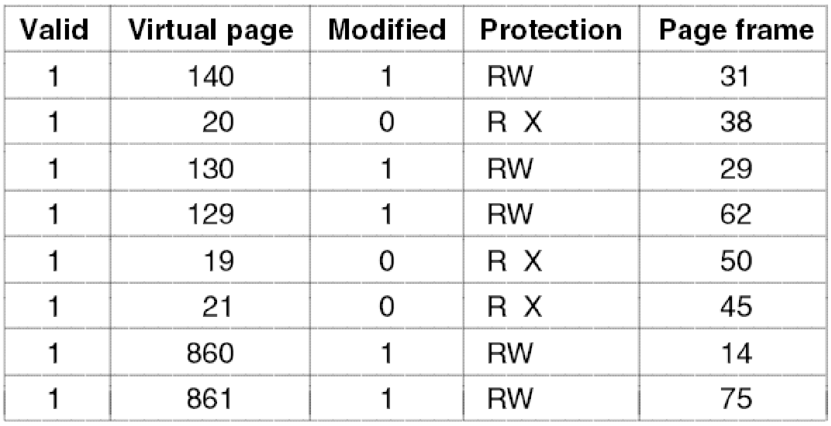
- The modified and referenced bits keep track of page usage when a page is written to the hardware automatically sets the modified.
- The referenced bit is set whenever a page is referenced either for reading of writing.
- Its value is to help the operating system choose a page to evict when a page fault occurs.
- This feature is important for pages that map onto device registers rather than memory.

**Speeding Up Paging**

- In any paging system, two major issues must be faced;
  1. The mapping from virtual address to physical address must be fast.
  2. If the virtual address space is large the page table will be large.
- The first point is a consequence of the fact that the virtual –to –physical mapping must be done on every memory reference.
- If an instruction execution takes, say 1 nsec, the page table lookup must be done in under 0.2 noses to avoid having the mapping become a major bottleneck.
- The second point follows from the fact that all modern computer use virtual addresses of at least 32 bits with 64 bits becoming increasingly common.
- The page table must have 1 million entries. And remember that each process needs its own page table.
- The need for large fast page mapping is a significant constraint on the way computer are built.
- During process execution no more memory reference are needed for the page table
- Another is that having to load the full page table at every context switch hurts performance.
- All the hardware needs then is a single register that point to the start of the page table.
- This design allows the virtual –to –physical map to be changed at a context switch by reloading one register

**Translation Lookaside Buffers (TLB)**

- The starting point of most optimization techniques is that the page table is in memory.
- This design has an enormous impact on performance
- A 1-byte instruction that copies one register to another. In the absence of paging, this instruction makes only one memory reference, to fetch the instruction.
- With paging at least one additional memory reference will be needed to access the page table.



| 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 |

- Since execution speed is generally limited by the rate at which the CPU can get instruction and data out of the memory.
- Computer designers have known about this problem for years and have come up with a solution.

- Their solution is based on the observation that most programs tend to make a large number of reference to a small number of pages, and not the other way around.
- Thus only a small fraction of the page table entries are heavily read the rest are barely used at all.
- The solution that has been devise is to equip computers with a small hardware device for mapping virtual addresses to physical addresses without going through the page table.
- The device called a TLE or sometimes an **associative memory**.
- The TLE function when a virtual addresses is presented to the MMU for translation the hardware first checks to see if its virtual page number is present in the TLU by comparing it to all the entries simultaneously.
- If a veiled match is found and the access dose not violates the protection bits the page frame is taken directly from the TLB, without going to the page table.
- If the virtual page number is present in the TLE but the instruction is trying to write on a read-only page, a protection fault is generated.
- The MMU detects the miss and dose an ordinary page table lookup.
- It then evicts one of the entries from the TLU and replaces it with the page table entry just looked up.

**Software TLB Management**

- Assumed that every machine with paged virtual memory has page tables recognized by the hardware plus a TLB.
- TLB management and handing TLB faults are done entirely by the MMU hardware.
- Traps to the operating system occur only when a page is not in memory.
- In the past this assumption was true.
- Many modern RISC machines including the SPARC, MIPS, and HP PA do nearly all of this page management in software.
- On these machines the TLB entries are explicitly loaded by the operating system.
- When a TLB miss occurs instead of the MMU just going to the page tables to find and fetch the needed page reference it just generates a TLB fault and tosses the problem into the lap of the operating system.
- The system must find the page, remove an entry from the TLB enter the new one and restart the instruction that faulted.
- When software TLB management is used it is essential to understand the difference between two kinds of misses.
- A soft miss occurs when the page reference is not in the TLB but is in memory.
- Al that is needed here is for the TLB to be updated.
- No disk I/O is needed, typically a soft miss takes 10-20 machine instruction to handle and can be completed in a few nanoseconds.
- Hard miss occurs when the page itself is not in memory a disk access is required to bring in the page which takes several milliseconds.
- A hard miss is easily a million times slower than a soft miss.

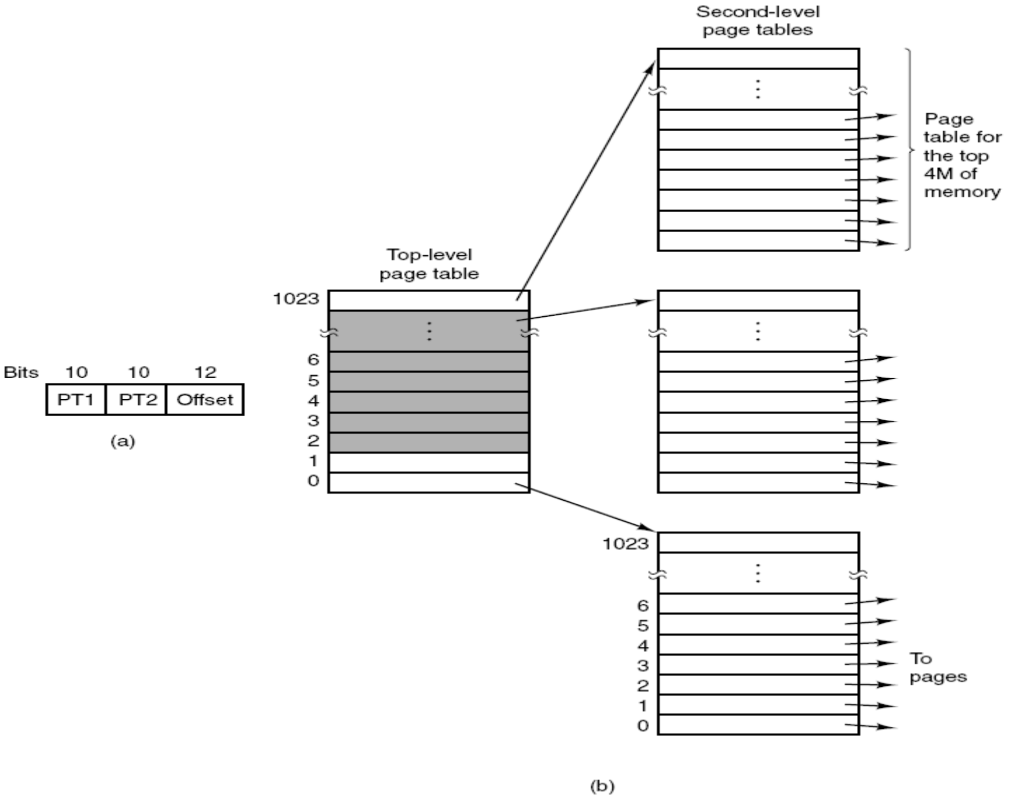
**Page Table For Large Memories**

- TLBs can be used to speed up virtual address to physical address translation over the original page-table –in –memory scheme.

**Multilevel Page Tables**

- A multilevel page table. A simple example is shown in fig. 3-13[a] we have a 32 bits virtual address that is partitioned into a 10 bit PT1 field a 10-bit PT2 field and a 12 bit offset field.
- Since offsets are 12 bits pages are 4 KE and there are a total of $2^{20}$ of then.

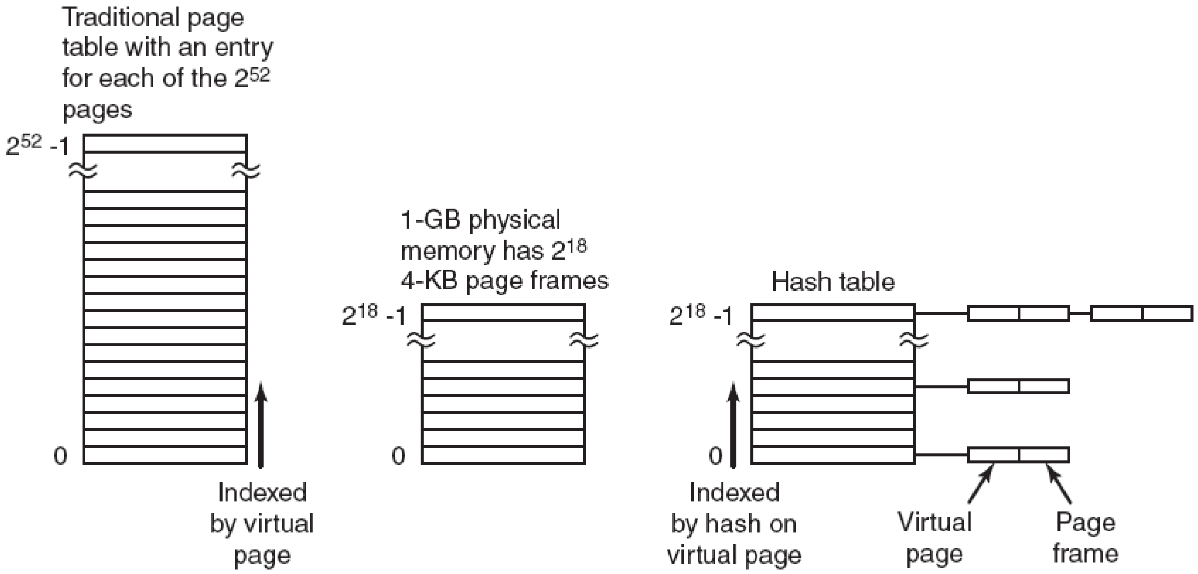
- The secret to the multilevel page table method is to avoid keeping all the page tables in memory all the time.
- In particular those that are not needed should not be kept around.



- The two-level page table works in this example.
- On the left we have the top-level page table with 1024 entries corresponding to the 10-bit PT1filed.
- When a virtual address is presented to the MMU it first extracts the PT1 filed and uses this value as an index into the top-level page table.
- The entry located by indexing into the top-level page table yields the address or the page frame number of a second –level page table.
- Entry 0 of the top-level page table point to the page table for the program text entry 1 point to the page table for the data and entry 1023 point to the page table for the stack.
- If page is not in memory the present/absent bit in page table entry will be zero causing a page fault.
- If the page is in memory the page frame number taken from the second –level page table is combined with the offset [4] to construct the physical address.
- This address is put on the bus and sent to memory.
- Four page tables are actually needed the top-level table and the second –level tables for 0 to 4M and the top 4M.
- The present/absent bits in 1021 entries of the top-level page table are set to 0, forcing a page fault if they are ever accessed.

### Inverted Page Tables

- There is one entry per page frame in real memory rather than one entry per page of virtual address space.


Traditional page table with an entry for each of the 2^52 pages
2^52 -1
0
Indexed by virtual page
1-GB physical memory has 2^18 4-KB page frames
2^18 -1
0
Hash table
2^18 -1
0
Indexed by hash on virtual page
Virtual page
Page frame

- For example with 64-bits virtual addresses a 4-KE page and 1 GE of RAM an inverted page table only enquires 262, 144 entries.
- The entry keeps track of which is located in the page frame.
- Although inverted page tables save vast amounts of space at least when the virtual address space is much larger then the physical memory they have a serious downside virtual –to-physical translation becomes mush harder.
- When process n reference virtual page p, the hardware can no longer find the physical page by using p as an index into the page table.
- Search is to have a hash table hashed on the virtual address.
- All the virtual pages currently in memory that have the same hash value are chained together as shown in fig. 3-14.
- If the hash table has as many slots as the machine has physical pages, the average chain will be only one entry long, greatly speeding up the mapping

### PAGE REPLACEMENT ALGORITHMS

- When a page fault occurs the operating system has to choose a page to evict to make room for the incoming page.
- If the page to be removed has been modified while in memory it must be rewritten to the disk to bring the copy up to data.
- Web server the server can keep a certain number of heavily used web pages in its memory cache.
- When the memory page to evict.

### The Optimal Page Replacements Algorithm

- The optimal page replacement algorithm says that the page with the highest label should be removed.
- If one page will not be used for 8 million instructions and another page will used for 6 million instructions removing the former pushes the page fault that will fetch it back as far into the future as possible.
- Computers, like people try to put off unpleasant events for as long as they can.
- The only with this algorithm is that it is un realizable.

- At the time of the page fault the operating system has no way of knowing when each of the pages will be referenced next.
- To avoid any possible confusion it should be made clear that this log of page referenced refers only to the one program just measured and then with only one specific input.
- The pages replacement algorithm derived from it is thus specific to that one program and input data.

### The Not Recently Used Page Replacement Algorithm

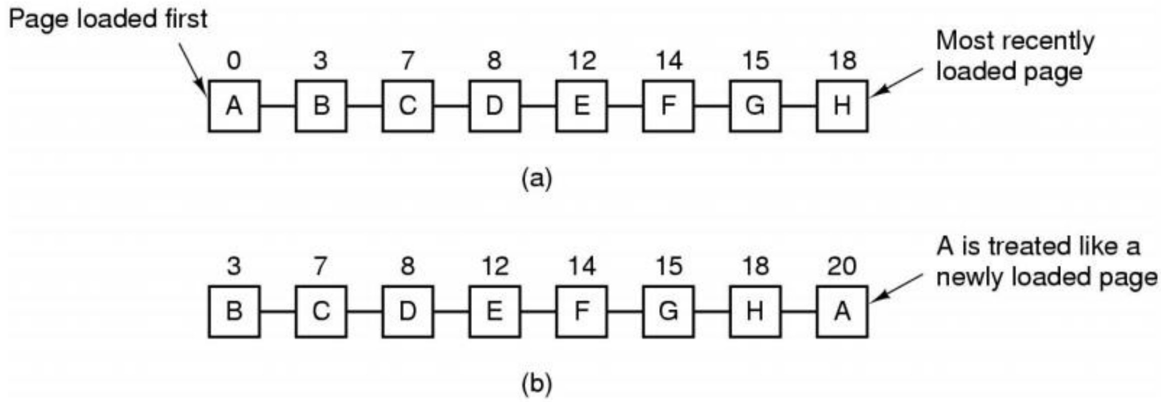
- R is set whenever the page is referenced M is set when the page is written to the bits are contained in each page table entry as shown in fig 3-11.
- It is important to realize that these bits must be updated on every memory reference.
- The operating system then sets the R bit changes the page table entry to points to the correct page with mode READ ONLY and restarts the instruction.
- If the page is subsequently modified another page fault will occur allowing the operating system to set the M bit and change the page’s mode to **READ/ WRITE.**
- When a page fault occurs, the OS inspects all the pages and divides them into 4 categories based on the current values of their R and M bits:
  Class 0: not referenced, not modified
  Class 1: not referenced, modified
  Class 2: referenced, not modified
  Class 3: referenced, modified.
- The NRU algorithm removes a page at random from the lowest- numbered nonempty class
- Implicit in this algorithm is the idea that it is better to remove a modified page that not been reference in at least one clock tick than a clean page that is in heavy use.
- The main attraction of NRU is that it is easy to understand moderated efficient to implement and gives a performance that while certainly not optimal may be adequate.

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

- FLFO algorithm. Consider a supermarket that has enough shelves to display exactly K different products.
- It is an immediate success so our finite supermarket has to get rid of one old product that in order to stock it.
- As a page replacement algorithm, the same idea is applicable.
- The operating system maintains a list of all pages currently in memory, with the most recent arrival at the tail and the least recent arrival at the head.
- On a page fault the page at the head is removed and the new page added to the tail of the list.
- When applied to stores FLFO might remove mustache wax but it might also remove flour salt or butter.
- When applied to computers the same problem arises.

### The Second – Chance Page Replacement Algorithm

- A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the R bit of the oldest pages.
- If it is 0, the page is both old and unused so it is replaced immediately.
- If the R bit is 1 the bit is cleared the page is put onto the end of the list of pages and its load time is updated as though it had just arrived in memory.
- Then the search continues. The operation of this algorithm called second chance.

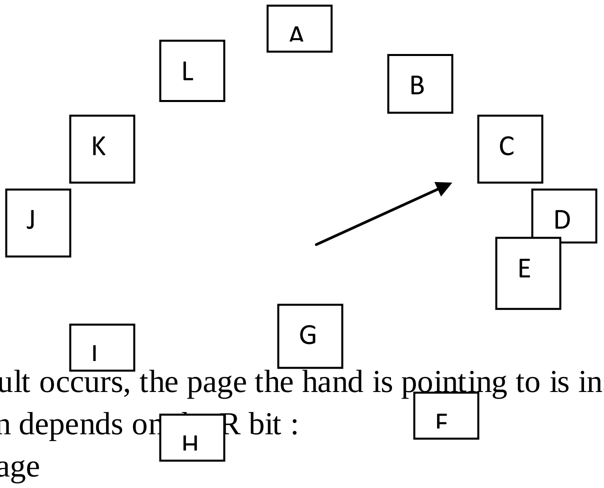


(a) pages sorted in FIFO order.
(b) page list if a page fault occurs at time 20 and A has its R bit set.

- Supposes that a page fault occurs at time 20 the oldest page is A which arrived at time 0, when the process started.
- If A has the R bits cleared it is evicted from memory either by being written to the disk or just abandoned.
- What second chance is looking for is old page have been referenced in the most recent click interval.
- If all the pages have been referenced second chance degenerates into pure FIFO.
- Specifically imagine that all the pages in fig. 3-15 have their R bit set.
- One by one the operating system moves the pages to the end of the list clearing the R bit each time it appends a page to the end of the list.

**The Clock Page Replacement Algorithm**

- To keep all the page frames on a circular list in the form of a clock.
- When a page fault occurs, the page being pointed to by the hand is inspected.
- If its R bits are 0, the page is evicted, the new page is inserted into the clock in its place and the hand is advanced one position.
- If R is 1, it is cleared and the hand is advanced to the next page.
- This process is repeated until a page is found with R = 0. This algorithm is called Clock.

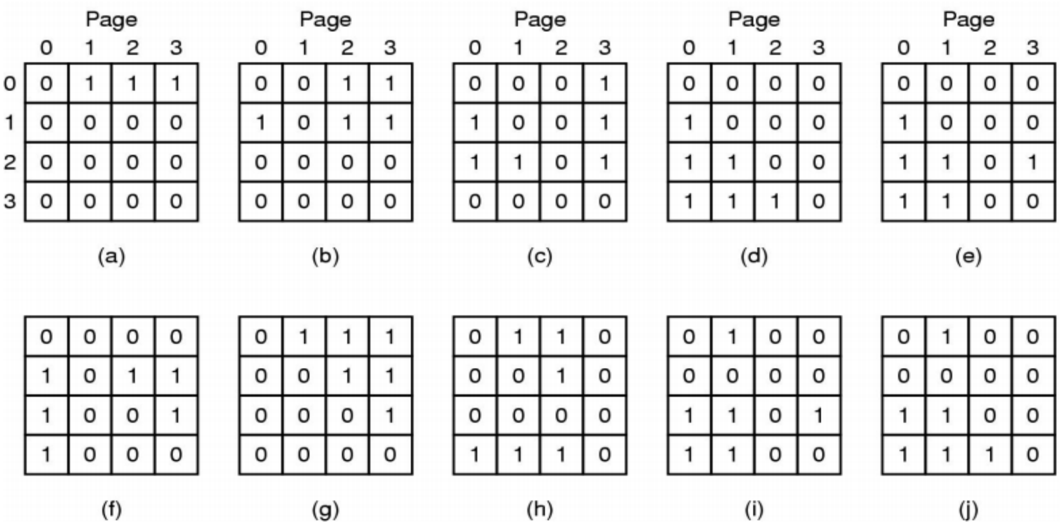


When a page fault occurs, the page the hand is pointing to is inspected.
The action taken depends on the R bit :
R=0:evict the page
R=1:clear R and advance hand

The clock Page replacement algorithm

**The Least Recently Used (LRU) Page Replacement Algorithm**

- When a page fault occurs, throw out the page that has been unused for the longest time. This is called LRU (Least Recently Used) paging.
- LRU is not cheap, it is necessary to maintain a linked list of all pages in memory, with the most recently used page at the front and the least recently used page at the rear.
- Difficulty: list must be updated on every memory reference.
- Finding a page in the list, deleting it and then moving it to the front is a very time consuming operation even in hardware
- There are other ways to implement LRU with special hardware.
- Equipping the hardware with a 64-bits counter, C, that is automatically incremented after each instruction.
- A machine with n page frames, the LRU hardware can maintain a matrix of nxn  bits initially all zero.
- Whenever page frame k is referenced the hardware first sets all the bits of row k to 1, and then sets all the bits of column l to 0.
- At any instant of times, the row whose binary value is lowest is the least recently used the row whose value is next lowest is next least recently used and so forth.
- The workings of this algorithm are given in fig.3-17 for four page frames and page references in the order 0123210323.



Page

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |

(a)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

(b)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

(c)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(d)

Page

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(e)

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 |

(f)

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 |

(g)

| | | | |
|---|---|---|---|
| 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |

(h)

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 |

(i)

| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

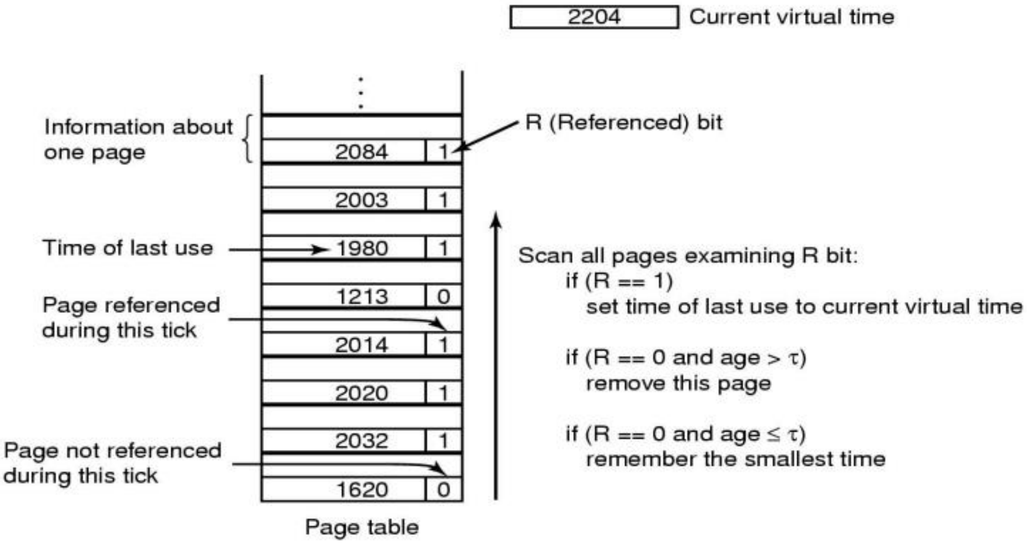
(j)

**Simulating LRU In Software**

- LRU algorithms machines have the required hardware.
- Instead, a solution that can be implemented in software is needed.
- One possibility is called the NFU (Non Frequently Used) algorithm.
- I require a software counter associated with each page initially zero.
- At each clock interrupt the operating system scans all the pages in memory.
- For each page the R bit which is 0or 1, is added to the counter the counters roughly keep track of how often each pages has been referenced.
- When a pages fault occurs, the page with the lowest counter is chosen for replacement.
- Main problem with FRU - it never forgets anything.
  - i. A small modification to NFU makes it able to simulate LRU quite well.
  - ii. Modification 2 parts: 1st counters are each shifted right 1 bit before the R bit is added in. 2nd the R bit is added to the leftmost rather than the rightmost bit.

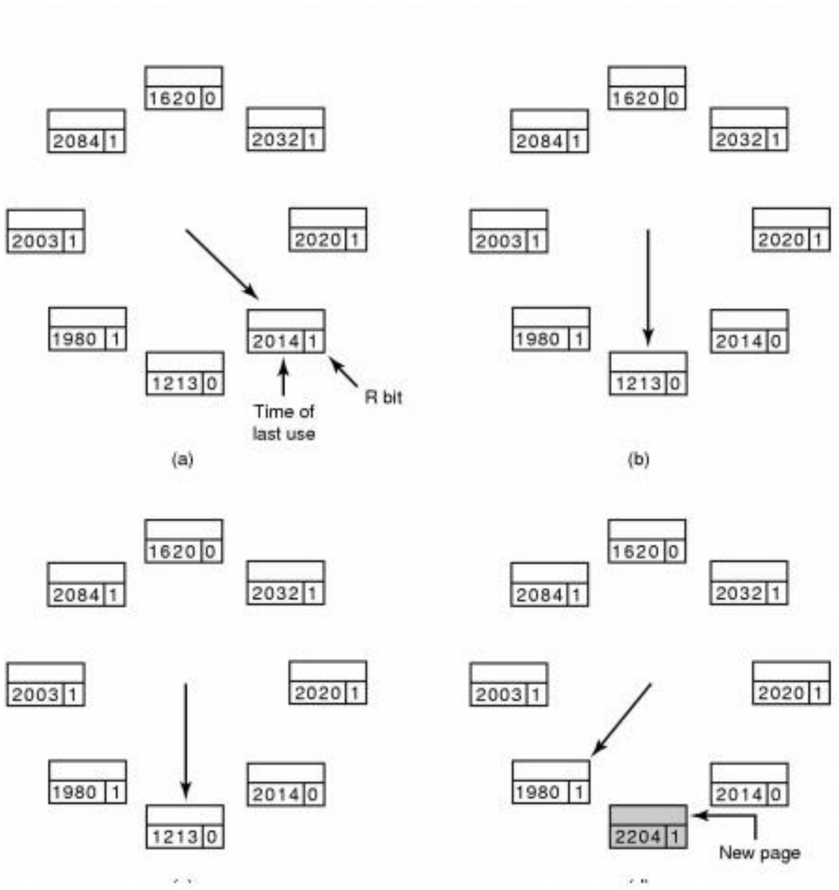
### The Working Set Page Replacement Algorithm

- The CPU tries to fetch the first instruction it gets a pages fault causing the operating system to bring in the page containing the first instruction.
- Other page fault for global variables and the stack usually follow quickly.
- After a while the process has most of the pages it needs and settles down to run with relatively few page faults.
- This strategy is called demand paging because pages are loaded only on demand not in advance.
- A locality of reference meaning that during any phase of execution the process references only a fraction of its pages.
- Each pass of a multipass compiler, references only a fraction of all the pages.
- The set of pages that a process is currently using is known as its working set.
- If the entire working set is in memory the process will run without causing many faults until it moves into another execution phase.
- At a rate of one or tow instructions per 10 milliseconds it will take ages to finish.
- A program causing page faults every few instructions is said to be thrashing.
- Many paging systems try to keep track of each process working set and make sure that it is in memory before letting the process run.
- This approach is called the working set model.
- It is designed to greatly reduce the page fault rate.
- Loading the pages before letting processes run is also called prepaging.
- The amount of CPU time a process has actually used since it started is often called its current virtual time


2204
Current virtual time
Information about one page
R (Referenced) bit
2084 1
2003 1
Time of last use
1980 1
Page referenced during this tick
1213 0
2014 1
2020 1
Page not referenced during this tick
2032 1
1620 0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

### The WSClock Page Replacement Algorithm

- The basic working set algorithm is cumbersome since the entire page table has to be scammed at each page fault until a suitable candidate is located.
- An improved algorithm that is based on the clock algorithm but also uses the working set information is called Schlock.
- The data structure needed is a circular list of page frames as in the clock algorithm and as shown



- Each entry contains the time of last use field from the basic working set algorithm as well as the R bit and the M bit.
- If the R bit is set to 1 the page has been used during the current tick so it is not an ideal candidate to remove.
- The R bit is then set to 0 the hand advanced to the next page and the algorithm repeated for that page.
- There are two cases to consider ;
  1. At least one write has been scheduled.     2. No writes have been scheduled.
- In the first case the hand just keeps moving looking for a clean page.
  Since one or more writes have been scheduled eventually some write will complete and its page will be marked as clean. The first clean page encountered is evicted.
- In the second case all pages are in the working set otherwise at least one write would have been scheduled.

**Summary Of Page Replacement Algorithms**

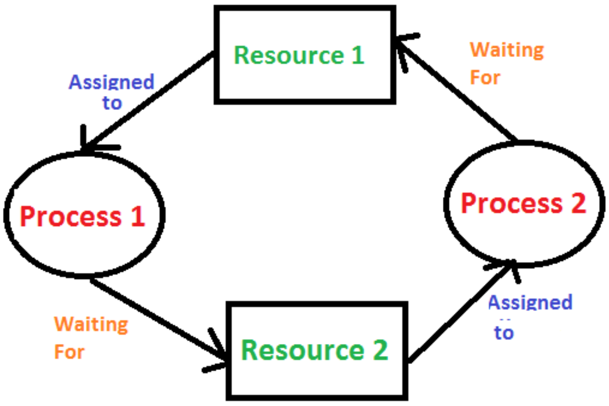
- The NRU algorithm divides page into four classes depending on the state of the R and M bits.
- A random page from the lowest- numbered class is chosen.
- FIFO keeps track of the order in which pages were loaded into memory by keeping them in a linked list.
- Removing the oldest page then be comes trivial but that page might still be in use so FIFO is a bad choice.

| Algorithm | Comment |
|---|---|
| **Optimal** | **Not implementable, but useful as a benchmark** |
| **NRU (Not Recently Used)** | **Very crude approximation of LRU** |
| **FIFO (First-in, First-out)** | **Might throw out important pages** |
| **Second chance** | **Big improvement over FIFO** |
| **Clock** | **Realistic** |
| **LRU (Least Recently Used)** | **Excellent, but difficult to implement exactly** |
| **NFU (Not Frequently Used)** | **Fairly crude approximation to LRU** |
| **Aging** | **Efficient algorithm that approximates LRU well** |
| **Working set** | **Somewhat expensive to implement** |
| **WSClock** | **Good efficient algorithm** |

# UNIT – IV

## Deadlocks:

**A deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.**



- Two processes each want to record a scanned document on a CD.
- Process *A* requests permission to use the scanner and is granted it.
- Process *B* is programmed differently and requests the CD recorder first and is also granted it.
- Now *A* asks for the CD recorder, but the request is denied until *B* releases it. Unfortunately, instead of releasing the CD recorder *B* asks for the scanner.
- At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

## Resources:

- A resource can be a hardware device or a piece of information. Computers have many different resources that can be acquired.

**Resources – 2 types:**

1. **Preemptable Resources**
2. **Nonpreemptable Resources:**

- 1. **Preemptable Resources** is one that can be taken away from the process owing it with no ill effects, Memory is an example of a preemptable resource.
- **Ex:**
  - A system with 32 MB of user memory, one printer, and two 32-MB processes that each want to print something.
  - Process A requests and gets the printer, then starts to compute the values to print.
  - Before it has finished with the computation, it exceeds its time quantum and is swapped out.
- **2. Nonpreemptable Resources** is one that cannot be taken away from its current owner without causing the computation to fall.

**If a process has begun to burn a CD-ROM, suddenly taking the CD recorder away from it and giving it to another process will result in a garbled CD.**

- Deadlocks involve nonpreemptable resource.
- Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another.
- Sequence of events required to use a resource:

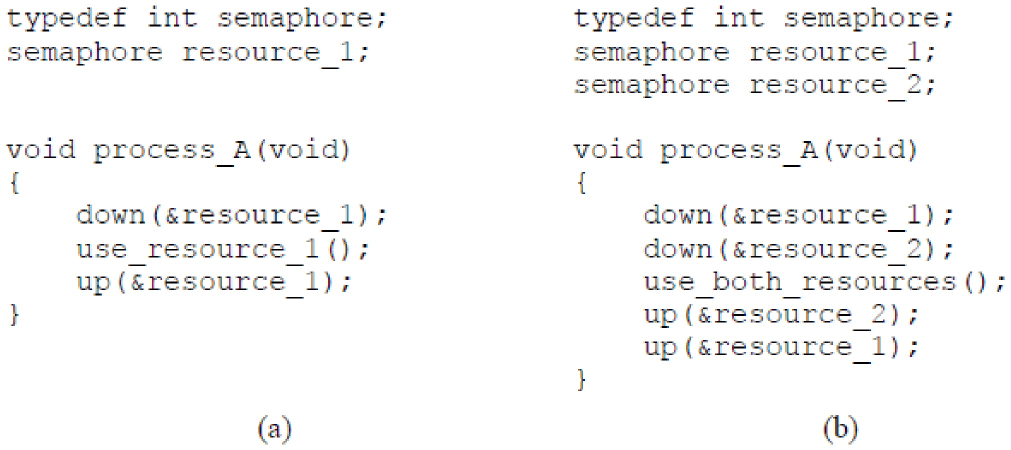
i. Request the resource.

ii. Use the resource.

iii. Release the resource.

**Resource Acquisition**

- Resource like records in a database system, it is up the user processes to manage resource usage themselves.
- One way of allowing user management of resources is to associate a semaphore with each resource.



```
typedef int semaphore;
semaphore resource_1;

void process_A(void)
{
     down(&resource_1);
     use_resource_1();
     up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void)
{
     down(&resource_1);
     down(&resource_2);
     use_both_resources();
     up(&resource_2);
     up(&resource_1);
}
```

(b)

**Figure 3-1.** Using a semaphore to protect resources. (a) One resource. (b) Two resources.

## Introduction to deadlocks:

- Deadlocks can be defined as ***"A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause"***
- Each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process.
- The number of processes and the number and kind of resources possesses and are unimportant.
- This result holds for any kind of resource, including both hardware and software.
- This kind of deadlock is called a resource deadlock.

## Conditions for Resource Deadlocks

- Four condition for deadlocks.

i. **Mutual exclusion condition**. Each resource is either currently assigned to exactly one process or is available.

ii. **Hold and wait condition.** Processes currently holding resources that were granted earlier can request new resources.

iii. **No preemption condition.** Resources previously granted cannot be taken away from a process. They must be explicitly released by the process holding them.

iv. **Circular wait condition**. There must be a circular chain of two or ore processes, each of which is waiting for a resource held by the next member of the chain.

- All 4 conditions must be present for a resource deadlock to occur.

## Deadlock modeling

- The four conditions can be modeled using directed graphs.
- Graphs – two kinds of nodes:

  i. Processes, as circle

  ii. Resources, as squares.

- A directed arc from a resource node to a process node means that the resource has previously requested by granted to and is currently held by that process.

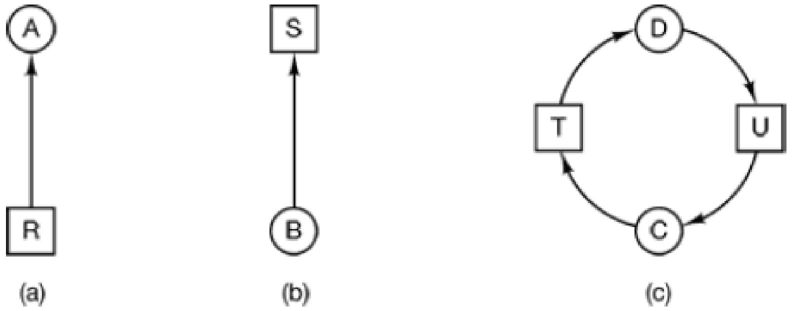
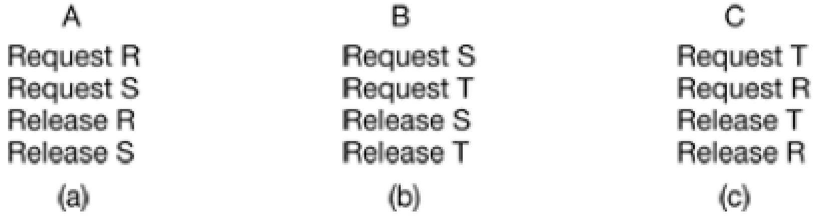


**Figure 3-3.** Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- Fig. 3-3(a), resource R is currently assigned to process A.
- An arc from a process to a resource means that the process is currently blocked waiting for that resource.
- In Fig. 3-3(b), process B is waiting for resource S.
- In Fig. 3-3(c) we see a
- deadlock: process C is waiting for resource T, which is currently held by process D.
- Process D is not about to release resource T because it is waiting for resource U, held by C. Both processes will wait forever.
- In this example, the cycle is *C–T–D–U–C.*

- Three processes, *A*, *B*, and *C*, and three resources *R*, *S*, and *T*. The requests and releases of the three processes are given in Fig. 3-4(a)-(c).
- The operating system is free to run any unblocked process at any instant, so it could decide to run *A* until *A* finished all its work, then run *B* to completion, and finally run C.
- Fig. 3- 4(d). If these six requests are carried out in that order, the six resulting resource graphs are shown in Fig. 3-4(e)-(j),
- After request 4 has been made, *A* blocks waiting for *S*, as shown in Fig. 3-4(h).
- In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 3-4(j).



| A | B | C |
|---|---|---|
| Request R | Request S | Request T |
| Request S | Request T | Request R |
| Release R | Release S | Release T |
| Release S | Release T | Release R |
| (a) | (b) | (c) |

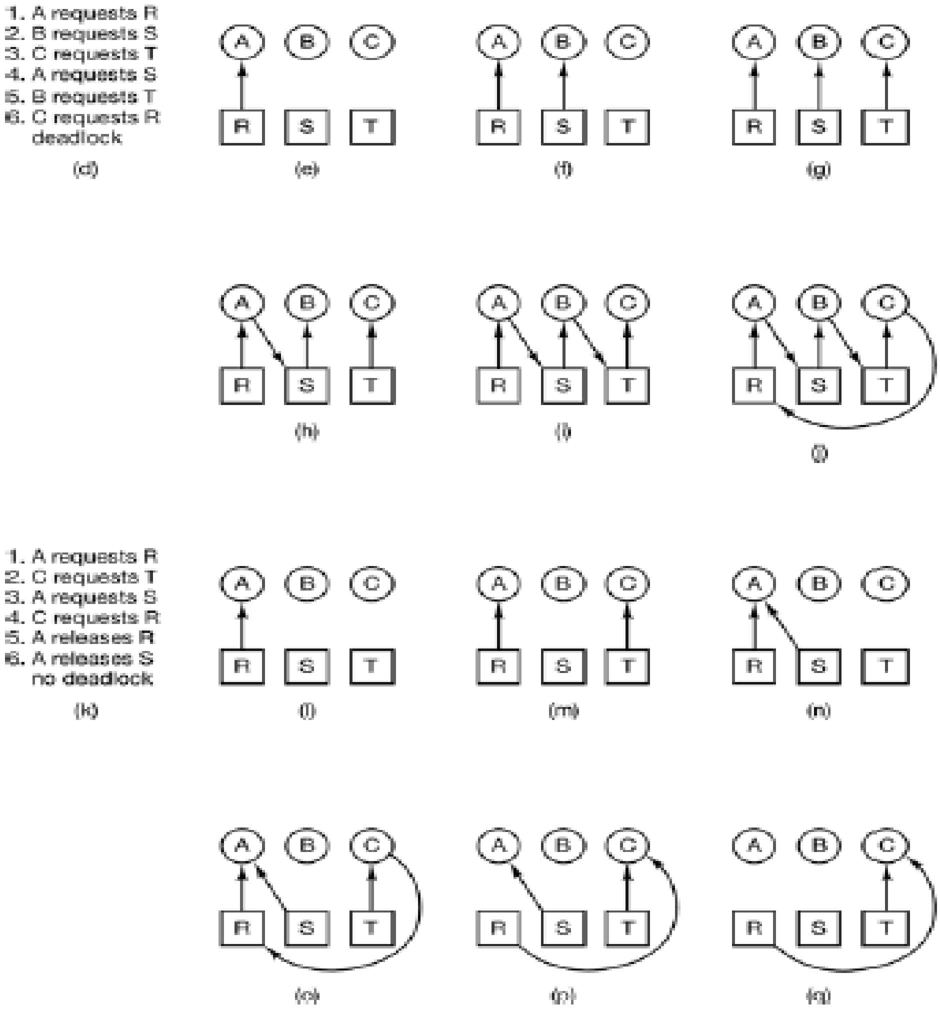


**Figure 3-4.** An example of how deadlock occurs and how it can be avoided.

## Deadlock detection and recovery:

- A second technique is detection and recovery. When this technique is used, the system does not attempt to prevent deadlocks from occurring.

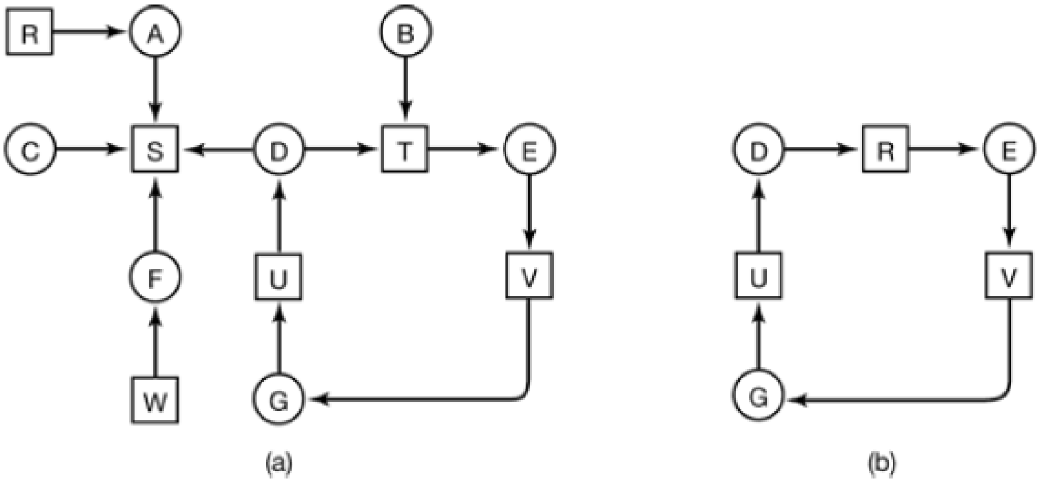
**Deadlock Detection with One Resource of Each Type**

- The simplest case only one resource of each type exists.
- Such a system might have one scanner, one CD recorder, one plotter, and one tape drive, but no more than one of each of resource.
- If this graph contains one or more cycles, a deadlock exists.
- Any process that is part of a cycle is deadlocked .if no cycles exists, the system is not deadlocked.
- A more complex system than the ones we have looked at so far, consider a system with seven processes, A though G, and six resources, R though W.
- The state of which resources are currently owned and which ones are currently being requested is as follows;

1. Process *A* holds *R* and wants *S*.
2. Process *B* holds nothing but wants *T*.
3. Process *C* holds nothing but wants *S*.
4. Process *D* holds *U* and wants *S* and *T*.
5. Process *E* holds *T* and wants *V*.

6. Process *F* holds *W* and wants *S*.

7. Process *G* holds *V* and wants *U*.



- Fig. 3-5(a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig. 3- 5(b).
- From this cycle, we can see that processes *D*, *E*, and *G* are all deadlocked.
- Processes *A*, *C*, and *F* are not deadlocked because *S* can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete.

**The algorithm operates by carrying out the following steps as specified;**

1. For each node, N in the graph, perform the following five steps with N as the starting node.

Initialize L to the empty list, and designate all the arcs as unmarked.

3. Add the current node to the end of L and check to see if the node now appears in L two times. If it dose, the graph contains a cycle and the algorithm terminates.

4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.

5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step3.

6. If this node is initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3.

**Deadlock Detection with Multiple Resources of Each Type**

- When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks.
- Detecting deadlock among n processes, $p_1$ through $p_n$. Number of resource classes be m, with $E_1$ resources of class 1, $E_2$ resources of class 2 and $E_i$ resources of class I ($1 < i < m$)
- E is the existing resource vector.
- At any instant, some of the resources are assigned and are not available. Let A be the available resource vector, with $A_i$ giving the number of instances of resource i.
- If both of our two tape drives are assigned $A_1$ will be 0.
- Two arrays, **C,** the Current allocation matrix, and **R,** the request matrix.
- The *i- th* row of C tells how many instances of each resource class $P_i$ currently holds. Thus $C_{ij}$ is the number of instances of resource *j* that are held by process *i*.
-

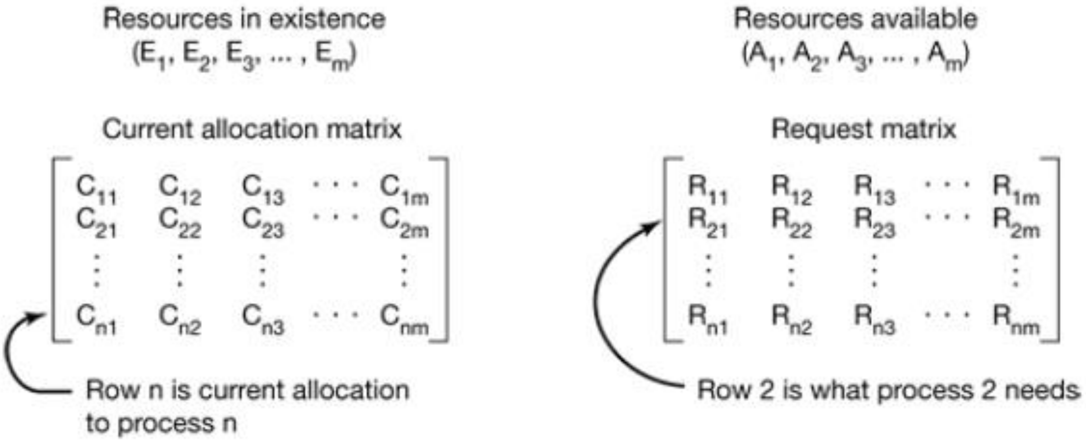

Resources in existence
$(E_1, E_2, E_3, \ldots, E_m)$
Resources available
$(A_1, A_2, A_3, \ldots, A_m)$
Current allocation matrix
$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$
Row n is current allocation to process n
Request matrix
$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$
Row 2 is what process 2 needs

**Figure 3-6.** The four data structures needed by the deadlock detection algorithm.

This observation means that

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

**Ex.:**

$$\begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ E = ( & 4 & 2 & 3 & 1\ ) \end{array} \qquad \begin{array}{lcccc} & \text{Tape drives} & \text{Plotters} & \text{Scanners} & \text{CD Roms} \\ A = ( & 2 & 1 & 0 & 0\ ) \end{array}$$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

**Figure 3-7.** An example for the deadlock detection algorithm.

**Recovery from Deadlock**

- Deadlock detection algorithm has succeeded and detected a deadlock. There are various ways of recovering from deadlock.

**Recovery through Preemption**

- It may possible to temporarily take a resource away from its current owner and give it to another process
- In many cases, manual intervention may be required, especially in batch processing operating systems running on mainframes.
- To take a laser printer away from its owner, the operator can collect all the sheets already and put them in a pile.Then the process can be suspended .at this point the printer can be assigned to another .process.

- When that process finishes, the pile the of printed sheets can be put back in the printer's output tray and the original process restarted.

**Recovery through Rollback**

- If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes check pointed periodically.
- Check pointing a process means that its state is written to a file so that it can be restarted later.
- The checkpoint contains not only the memory image, but also the resource state.
- New checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence accumulates.
- When a deadlock is detected, it is easy to see which resources are needed.
- To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired that resource by starting one of its earlier checkpoints.

**Recovery through killing processes**

- Simplest way to break a deadlock is to kill one more processes.
- One possibility is to kill a process in the cycle.
- A process not in the cycle can be chosen as the victim in order to release its resources.
- In this approach, the process to be killed is carefully chosen because it is holding resource that some process in the cycle needs.
- For example, one process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer.
- These two are deadlocked. A third process may hold another identical printer and another identical plotter and be happily running.
- Killing the third process will release these resources and break the deadlock involving the first two.

**Deadlock Avoidance**

- Deadlock can be avoided if certain information about processes are available to the operating system before allocation of resources, such as which resources a process will consume in its lifetime. For every resource request, the system sees whether granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock.

**Resource Trajectories**

- The main algorithms for doing deadlock avoidance are based on the concept of safe states.
- A model for dealing with two processes and two resources, for example, a printer and a plotter.
- The horizontal axis represents the number of instructions executed by process by process A.
- The vertical axis represents the number of instructions executed by process B.
- At $I_1$ A requests a printer; at $I_2$ it needs a plotter. The printer and plotter are released at $I_3$ and $I_4$ respectively .process B needs the plotter are from $I_5$ to $I_7$ and the printer from $I_6$ to $I_{8.}$

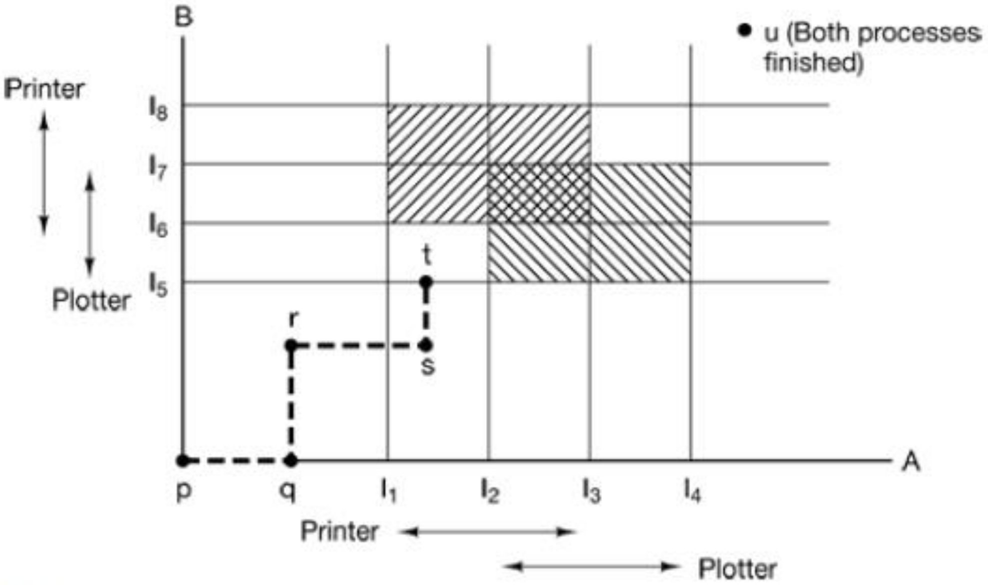
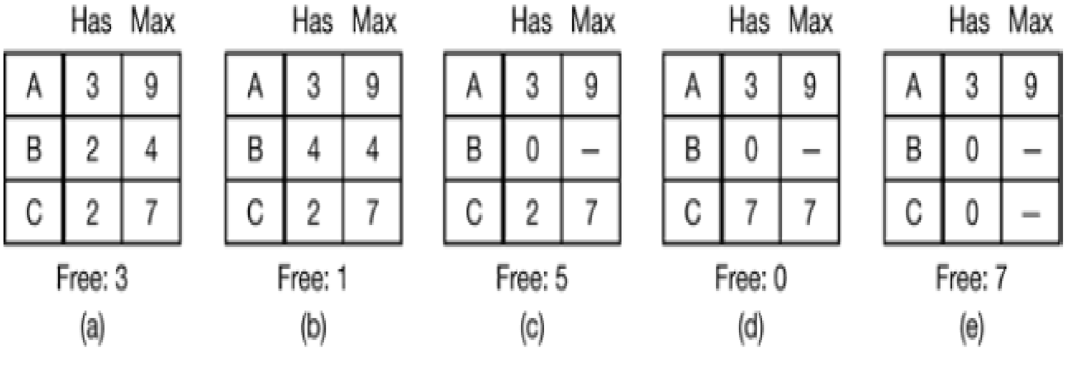


**Figure 3-8.** Two process resource trajectories.

- Every point in the diagram represents a joint state of the two processes.
- The state is at p, with neither process having executed any instructions.
- If the scheduler chooses to run A first, we get to point q, in which A has executed some number of instructions, but B has executed none.
- At point q the trajectory becomes vertical, indicating that the scheduler has chosen to run B.

**Safe and Unsafe States**

- The deadlock avoidance algorithms that we will study use the information.
- At any instant of time, there is a current state consisting of E, A, C and R.
- A state is said to be safe if there is some scheduling order in which every process can run to completion even if all of them suddenly request their max number of resource immediately.



(a)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

Free: 3

(b)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 4 | 4 |
| C | 2 | 7 |

Free: 1

(c)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 2 | 7 |

Free: 5

(d)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 7 | 7 |

Free: 0

(e)

| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 0 | – |
| C | 0 | – |

Free: 7

**Figure 3-9.** Demonstration that the state in (a) is safe.

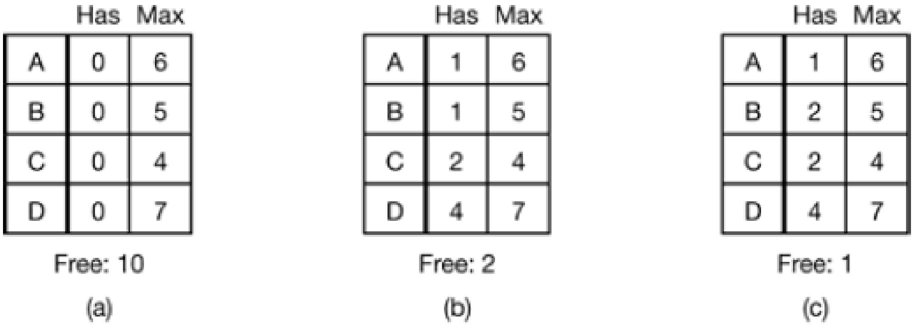
- The state in (a ) is safe because there exits a sequence of allocations that allows all process to complete.
- The scheduler could simply run B, until it asked for and got two more instances of the resource, leading to the state.
- When B completes, the state (c ) gets.Then the scheduler can run C.
- Complete C , it get (e)
- The A get the six instances of the resource it needs and also complete.

- The state (a) is safe because the system, by careful scheduling, can avoid deadlock.

**The Banker's Algorithm for a single Resource**

- A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965); it is known as the banker's algorithm and is an extension of the deadlock detection algorithm**.**
- It is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit.
- What the algorithm does is check to see if granting the request leads to an unsafe state.

Four customers A, B, C and D each of whom has been granted a certain number of credit units.



| | 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

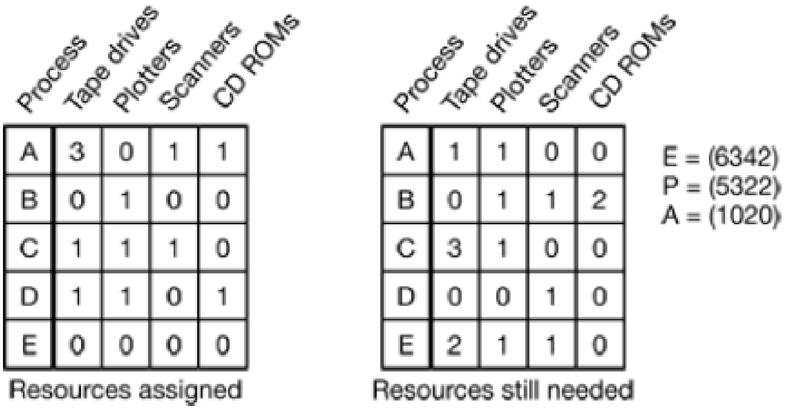
(c)

**Figure 3-11.** Three resource allocation states: (a) Safe. (b) Safe (c) Unsafe.

- The customers go about their respective businesses, making loan requests from time to time.
- At a certain moment the situation is as shown in fig. 6-11 [b]. this state because with two units left, the banker can delay any requests except, C' s, thus letting C finish and release all four of his resources.
- The banker's algorithm considers each request as it occurs, and sees if granting it leads to a safe state.

**The Banker's Algorithm For Multiple Resources**

- The banker's algorithm can be generalized to handle multiple resources.
- The one on the left shows how many of each resource are currently assigned to each of the five processes.
- The matrix on the right shows how many resources each process still needs in order to complete
- As in the single –resources case, processes must state their total resource needs before executing, so that the system can compute the right-hands matrix at each instant.



| Process | Tape drives | Plotters | Scanners | CD ROMs |
|---|---|---|---|---|
| A | 3 | 0 | 1 | 1 |
| B | 0 | 1 | 0 | 0 |
| C | 1 | 1 | 1 | 0 |
| D | 1 | 1 | 0 | 1 |
| E | 0 | 0 | 0 | 0 |

Resources assigned

| Process | Tape drives | Plotters | Scanners | CD ROMs |
|---|---|---|---|---|
| A | 1 | 1 | 0 | 0 |
| B | 0 | 1 | 1 | 2 |
| C | 3 | 1 | 0 | 0 |
| D | 0 | 0 | 1 | 0 |
| E | 2 | 1 | 1 | 0 |

Resources still needed

E = (6342)
P = (5322)
A = (1020)

**Figure 3-12.** The banker's algorithm with multiple resources.

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. if no such row exist, the system will eventually lock since no process can run to completion.

2. Assume the process of the row chosen requests all the resources it needs and finishes. Mark that process as terminated and add all its resources to the A vector.

3. Repeat steps 1and 2 until either all processes are marked terminated or no process is left whose resource needs can be met.

**Deadlock Prevention**

•Approach: Ensure that the necessary conditions for deadlocks are never satisfied

•P revent one of the following from becoming true

–M utual Exclusion n

–Hold and Wait

–No Preemption

–Circular Wait

**The Mutual Exclusion Condition**

- Removing the **mutual exclusion** condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.
- ▪ By spooling printer output, several processes can generate output at the same time.
- ▪ The only process that actually requests the physical printer is the printer daemon.
- ▪ Since the daemon never requests any other resources, we can eliminate deadlock for the printer.
- ▪ If the daemon is programmed to begin printing even before all the output is spooled, the printer might lie idle if an output process decides to wait several hours after the first burst of output.
- ▪ What would happen if two processes each filled up one half of the available spooling space with output and neither was finished producing its full output/
- ▪ In this case we have two processes that have each finished part, but not all, of their output, and cannot continue.
- ▪ Neither process will ever finish, so we have a deadlock on the disk.

**The hold and wait condition**

- The **hold and wait** or **resource holding** conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to request resources only when it has none. Thus, first they must release all their currently held resources before requesting all the resources they will need from scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely, as such a resource may always be allocated to some process, resulting in resource starvation.[1] (These algorithms, such as serializing tokens, are known as the *all-or-none algorithms*.)
- ▪ If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks.
- ▪ Goal is to require all processes to request all their resources before starting execution.
- ▪ If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, noting will be allocated and the processes would just wait.
- ▪ An immediate problem with this approach is that many processes do not know how many resources they will need until they started running.
- ▪ In fact, if they knew the banker’s algorithm could be used.

- Another problem is that resources will not be used optimally with this approach.
- An example a process that reads data from an input tape, analyzes it for an tour, and then writes an output tape as well as plotting the results.
- Some mainframe batch systems require the user to list all the resources on the first line of each job.
- The system then acquires all resources immediately and keeps them until the job finishes.

### The no preemption condition

The **no** **preemption** condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a *priority* algorithm. Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms andoptimistic concurrency control.

- Spooling printer output to disk and allowing only the printer daemon access to the real printer eliminates deadlock involving the printer, although it creates one for disk space.
- For example, records in databases or tables inside the operating system must be locked to be used and there in lies the potential for deadlock.
- Attacking the circular wait condition
- Only one condition is left. The circular wait can be eliminated in several ways.
- One way is simply to have a rule saying that a process is entitled only to single resources at any moment.
- If it needs a second one, it must release the first one. For a process that needs to copy a huge file form a tape to a printer, this restriction is unacceptable.
- Another way to avoid the circular wait is to provide a global numbering of all the resources as shown

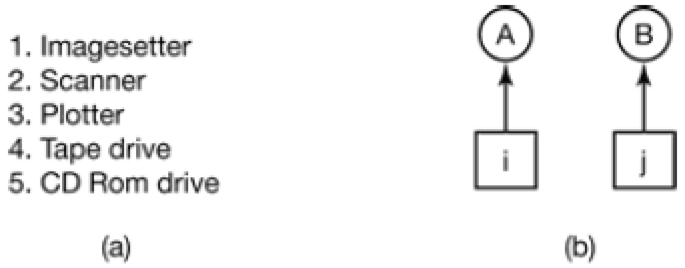


**Figure 3-13.** (a) Numerically ordered resources. (b) A resource graph.

- Now the rule is this processes can request resources whenever they want, to but all request must be made in numerical order.
- A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.
- With more than two processes the same logic holds. At every instant, one of the assigned resources will be highest.
- The process holding that resources will never ask for a resource already assigned.
- It will either finish, or at worst, request even higher numbered resources all of which are available.
- At this point, some other process will hold the highest resources and can also finish. In short there exists a scenario in which all processes finish so no deadlock is present.

**The Circular Wait condition:**

- The final condition is the **circular wait** condition. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration.Dijkstra's solution can also be used.

**Summary of approaches to deadlock prevention.**

| **Condition** | **Approach** |
|---|---|
| Mutual exclusion | Spool everything |
| Hold and wait | Request all resources initially |
| No preemption | Take resources away |
| Circular wait | Order resources numerically |

**Multiple Processor system:**

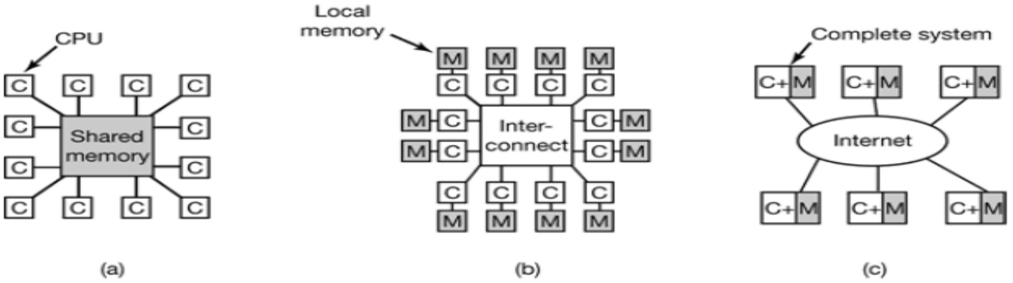


**Figure 8-1.** (a) A shared-memory multiprocessor. (b) A message-passing multicomputer. (c) A -wide area distributed system.

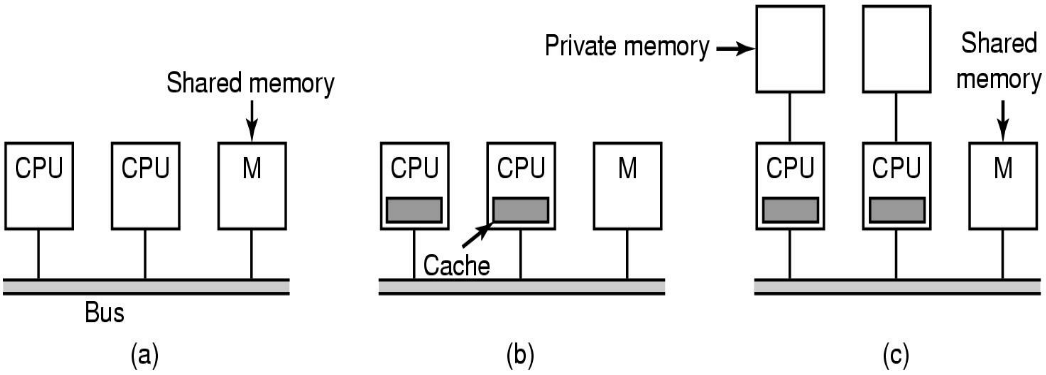
- **Multiprocessors:**
  - A shared-memory multiprocessor (or just multiprocessor henceforth) is a computer system in which two or more CPUs share full access to a common RAM.
  - A program running on any of the CPUs sees a normal(usually paged)virtual address space.
  - The basis of interprocessor communication: one CPU writes some data into memory, and another one reads the data out.
  - Multiprocessor operating systems handle system calls, do memory management, provide a file system and manage I/O devices.

➢ **Multiprocessor Hardware:**

- All multiprocessor have the property that every CPU can address all memory, some multiprocessor have the additional property that every memory word can be read as fast as every other memory word.
- UMA(uniform memory access) multiprocessors
- In contrast NUMA (Nonuniform memory access) multiprocessors do not have this property

**UMA multiprocessors with Bus-Based architectures**

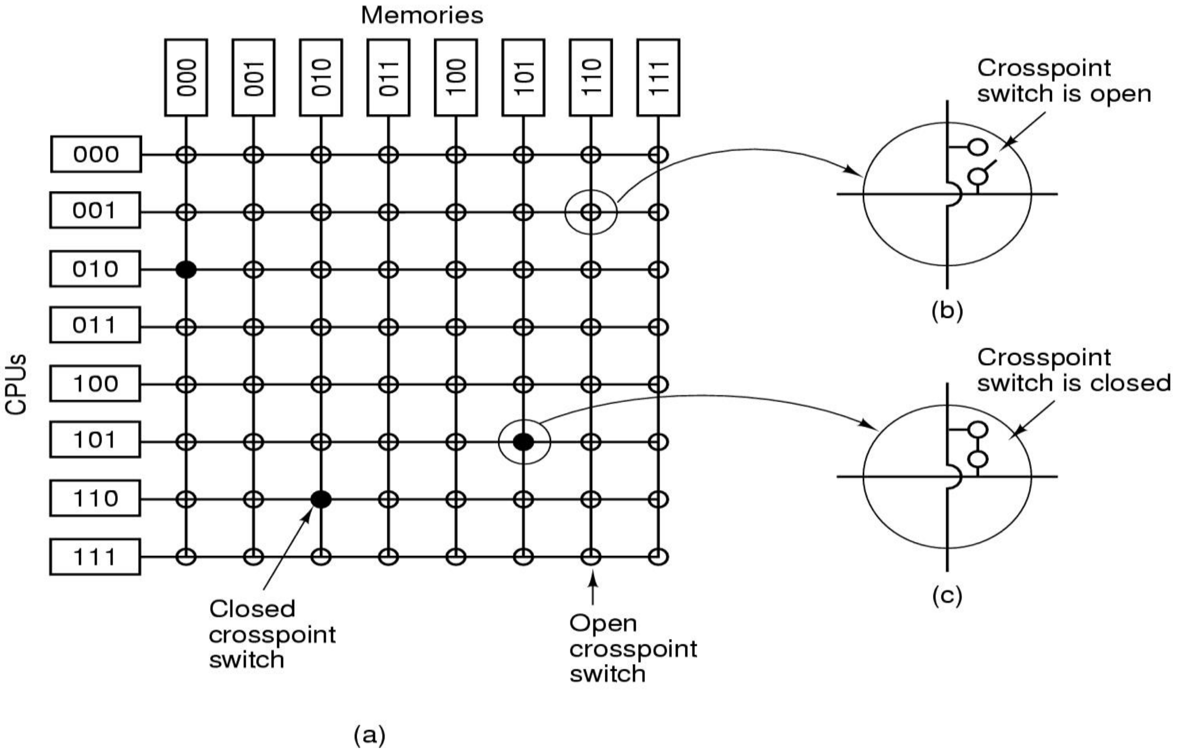
- The simplest multiprocessors are based on a single bus
- Two or more CPUs and one or more memory modules all use the same bus for communication
- When a CPU wants to read a memory word, it first checks to see i9f the bus is busy. If the bus is idle, the CPU puts the address of the word it wants on the bus, asserts a few control signals, and waits until the memory puts the desired word on the bus.
- If the bus is busy when a CPU wants to read or write memory, the CPU just waits until the bus becomes idle.
- Most of the CPUs will be idle most of the time. The solution to this problem is to add a cache to each CPU.
- The cache can be inside the CPU chip, next to the CPU chip, on the processor board, or some combination of all three.



- When a word is referenced, its entire block, called a cache line, is fetched into the cache of the CPU touching it.
- Each cache block is marked as being either read-only (in which case it can be present in multiple caches at the same time) or read-write.
- In other caches have a “clean” copy, that is, an exact copy of what is in memory, and they can just discard their copies and let the write fetch the cache block memory before modifying it.
- If some other cache has a “dirty” copy it must either write over the bus. This set of rules is called a cache –coherence protocol and is one of many.

**UMA Multiprocessors Using Crossbar Switches**

- At each intersection of a horizontal and vertical line is a crosspoint. A crosspoint is a small switch that can be electrically opened or closed, depending on whether the horizontal and vertical lines are to be connected or not.
- Three crosspoints closed simultaneously, allowing connections between the pairs (010,000), (101,101), and (110,010) at the same time. Many other combinations are also possible.
- The number of combinations is equal to the number of different ways eight rooks can be safely placed on a chess board.



- One of the nicest properties of the crossbar switch is that is a nonblocking network.
- Meaning that no CPU is ever denied the connection it needs because some crosspoint or line is already occupied (assuming the memory module itself is available).
- Furthermore, no advantage planning is needed.
- Even if seven arbitrary connections are already setup, it is always possible to connect the remaining CPU to the remaining memory.

**UMA Multiprocessors Using Multistage Switching Networks**

- A completely different multiprocessor design is based on the humble 2*2 switch.
- This switch has two inputs and two outputs.
- Messages arriving on either input line can be switched to either output line. For our purposes.
- Messages will contain up to four parts.

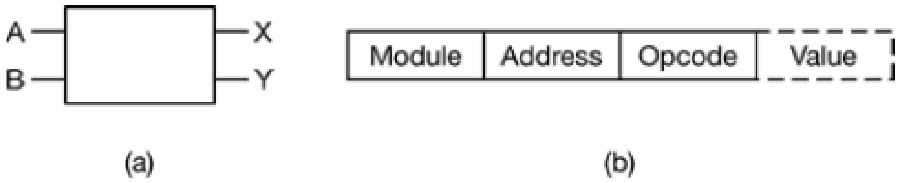
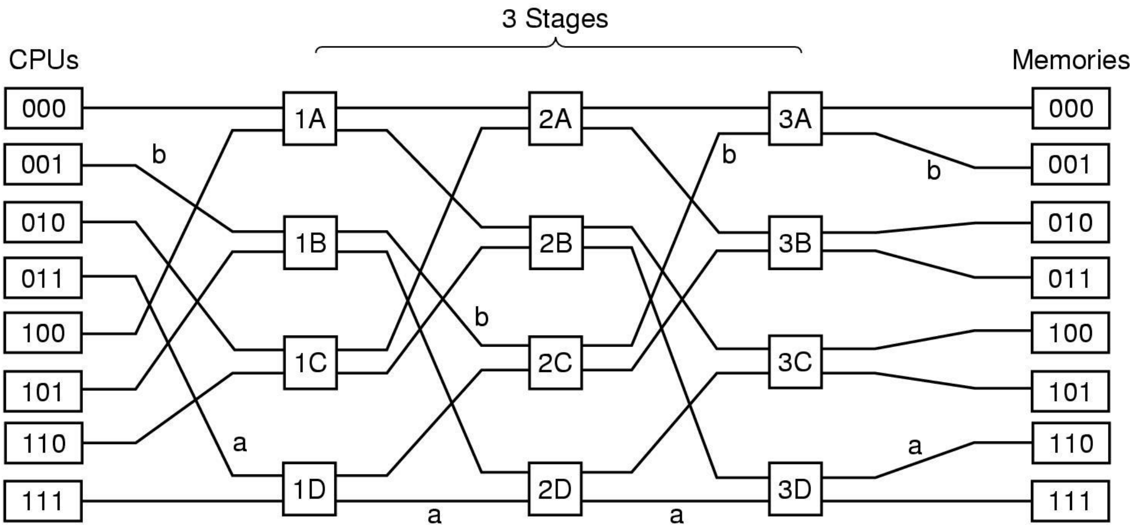


**Figure 8-4.** (a) A 2 2 switch. (b) A message format.

1. The module field tells which memory to use
2. The address specifies an address within a module.
3. The Opcode gives the operation, such READ or WRITE.
4. The Value field may contain an operand, it is optional.

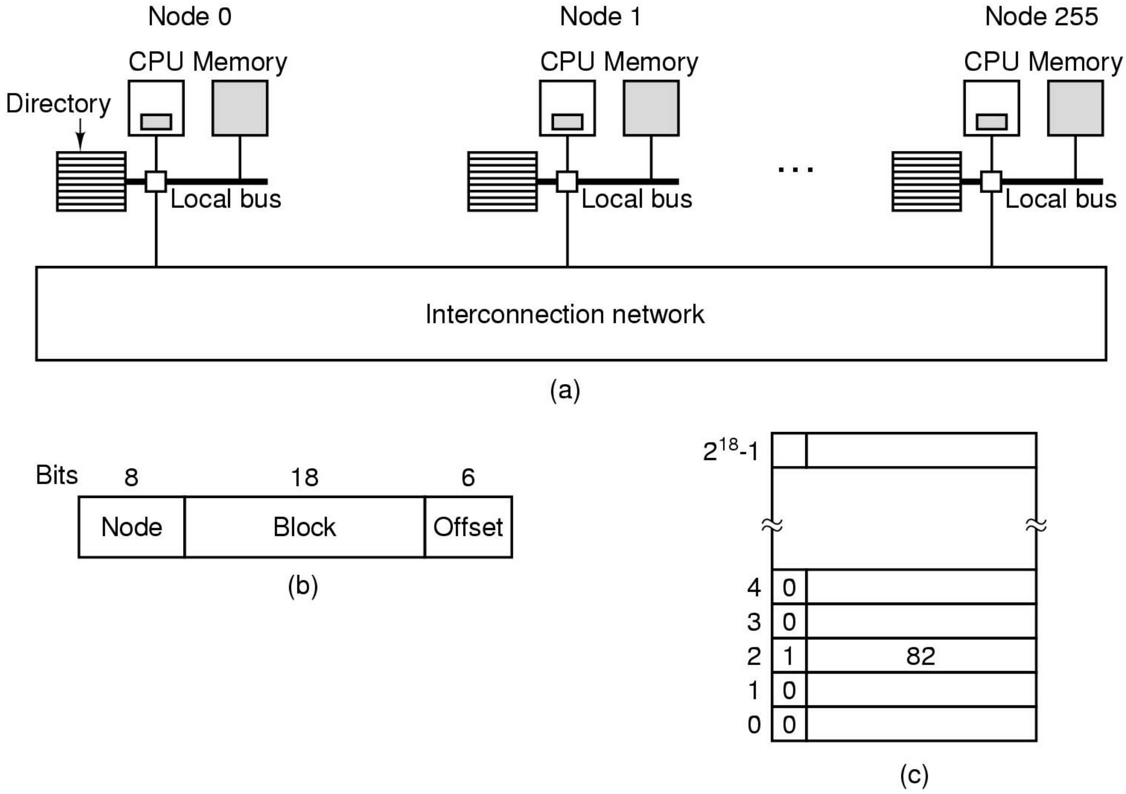
- The switch inspects the module field and uses it to determine if the message should be sent on X or on Y.
- Our 2x2 switches can be arranged in many ways to build larger multistage switching networks.
- Omega network, it connect eight CPUs to eight memories using 12 switches.



- Generally n CPU and n memories – $\log_2$ n stages, with n/2 switches per stage for a total of (n/2) log2n switches.
- Wiring pattern of the omega network is often called the perfect shuffle.
- The CPU sends a READ message to switch 1D containing the value 110 in the module field.
- The switch takes the first bit of 110 and uses it for routing.
- A 0 routes to upper output and a 1 route to lower one. Since the bit is 1, the message is routed via the lower output to 2D.
- All $2^{nd}$ stage switches, including 2D use the $2^{nd}$ bit for routing.
- It is 1, so message is now forwarded via the lower output to 3D.
- As the message moves through the switching network, the bits at the left-hand end of the module number are no longer needed.
- At the same time all this is going on, CPU 001 wants to write a word to memory module 001. An analogues process happens, here with the message routed via the upper, upper, and lower outputs, respectively, marked by the letter b.
- Its request would come into conflict with CPU 001's request at switch 3A.
- Unlike the crossbar switch the omega network is a **blocking network.**
- A memory system in which consecutive words are in different modules is said to be **interleaved.**
- Interleaved memories maximize parallelism because most memory references are to consecutive addresses.

**NUMA MULTIPROCESSORS:**

- NUMA machines have three key characteristics that all of them posses and which together distinguish them from other multiprocessors:
    1. There is a single address space visible to all CPU's.
    2. Access to remote memory is via LOAD and STORE information.
    3. Access to remote memory is slower than access to local memory.



- When the access time to remote memory is not hidden (because there is no caching), the system is called **NC-NUMA** (No Code NUMA). When coherent caches are present, the system is called **CC-NUMA** (Cache Coherent NUMA).
- The Most popular approach for building large CC-NUMA multiprocessors currently is the **directory-based multiprocessor.**
- When a cache line is referenced, the database is queried to find out where it is and whether it is clean or dirty.
- A 256 node system, each node consisting of one CPU and 16MB of RAM connected to the CPU via a local bus. The total memory is 2^32 bytes, divided up into 2^26 cache lines of 64 bytes each.

**MULTICORE CHIPS:**

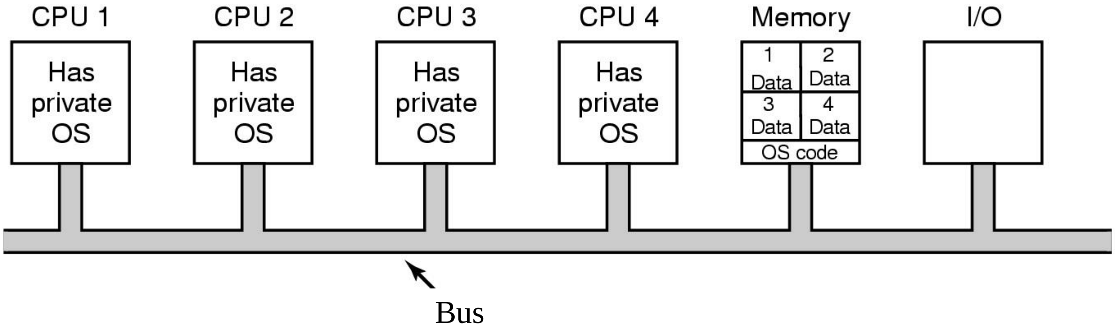
- Technology improves, transistors are getting smaller and smaller and it is possible to put more and more of them on a chip.
- This empirical observation is often called Moore's Law.
- Special hardware circuitry makes sure that if a word is present in two or more caches and one of the CPU's modifies the word, it is automatically removed from all the caches in order to maintain consistency. This process is known as snooping.
- Multicore chips are sometime called **CMPs (Chip-level Multiprocessors).**
- CMPs are not really that different from bus-based multiprocessors or multiprocessors that use switching networks.
- CMPs differ from their larger cousins is fault tolerance.
- Because the CPUs are so closely connected, failures in shared components may bring down multiple CPUs at once, likely in traditional multiprocessors.
- In addition to symmetric multicore chips, where all the cores are identical, another category of multicore chip is the **system on a chip**. These chips have one or more main CPUs, but also

special-purpose cores, such as video and audio decoders, crypto processors, network interfaces, and more, leading to a complete computer system on a chip.

## MULTIPROCESSOR OPERATING SYSTEM TYPES:

### Each CPU has its own operating system:

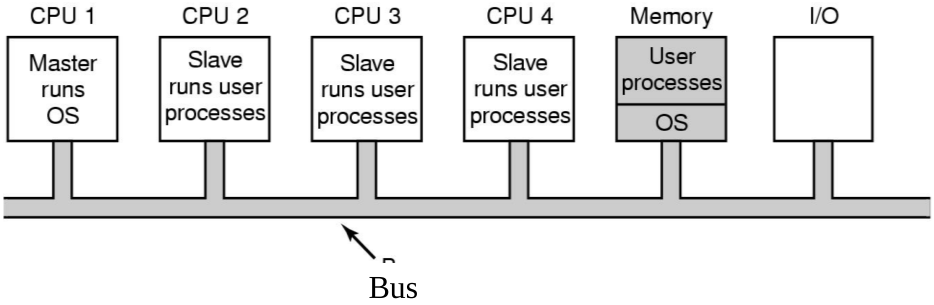
- The simplest possible way to organize a multiprocessor operating system is to statically divide memory into as many partitions as there are CPUs and give each CPU its own private memory and its own private copy of the operating system. In effect the n CPUs then operate as n independent computers.
- Optimization is to allow all the CPUs to share the operating system code and make private of only the operating system data structures.



- First, when a process makes a system call, the system call is caught and handled on its own CPU using the data structures in that operating system's table.
- Second, since each operating system has its own tables, it also has its own set of processes that it schedules by itself. There is no sharing of processes.
- Third, there is no sharing of pages. It can happen that CPU 1 has pages to spare while CPU 2 is paging continuously. There is no way for CPU 2 to borrow some pages from CPU 1 since the memory allocation is fixed.
- Fourth, and worst, if the operating system maintains a buffer cache of recently used disk blocks, each operating system does this independently of the other ones.
- The only way to avoid this problem is to eliminate the buffer caches.

## MASTER-SLAVE MULTIPROCESSORS:

- One copy of the operating system and its tables is present on CPU 1 and not on any of the others.
- All system calls redirected to CPU 1 for processing there. CPU 1 may also run user processes if there is CPU time left over. This model is called master-slave since CPU 1 is the master and all the others are slaves.

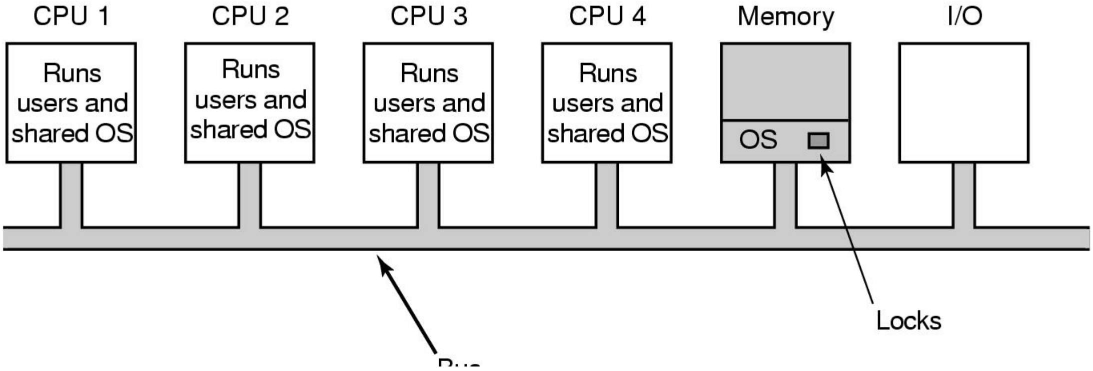

CPU 1
Master runs OS
CPU 2
Slave runs user processes
CPU 3
Slave runs user processes
CPU 4
Slave runs user processes
Memory
User processes
OS
I/O
Bus

- The master-slave model solves most of the problems of the first model. There is a single data structure that keeps track of ready processes.
- When a CPU goes idle, it asks the operating system on CPU 1 for a process to run and is assigned one.

- One CPU is idle while another is overloaded. Similarly, pages can be allocated among all the processes dynamically and there is only one buffer cache, so inconsistencies never occur.
- The problem with this model is that with many CPUs, the master will become a bottleneck. After all, it must handle all system calls from all CPUs.
- 10 CPUs will pretty much saturate the master, and with 20 CPUs it will be completely overloaded.
- Thus this model is simple and workable for small multiprocessors, but for large ones it fails.

**SYMMETRIC MULTIPROCESSORS**:

- The SMP (symmetric multiprocessors), eliminates this asymmetry. There is one copy of the operating system in memory, but any CPU can run it.
- When a system call is made, the CPU on which the system call was made traps to the kernel and processes the system call.
- This model balances processes and memory dynamically, since there is only one set of operating system tables.
- If two or more CPUs are running operating system code at the same time, disaster may well result.
- The simplest way around this problem is to associate a mutex (i.e. lock) with the operating system, making the whole system one big critical region.
- When a CPU wants to run operating system code, it must first acquire the mutex.
- Many part of the operating system are independent of one another. For example, there is no problem with one CPU running the scheduler while another CPU handling a file system call and a third one is processing a page fault.
- This observation leads to splitting the operating system up into multiple independent critical regions that do not interact with one another. Each critical region is protected by its own mutex, so only one CPU at a time can execute it.
- The process table is needed for scheduling, but also for the fork system call and also for signal handling. Each table that may be used by multiple critical regions needs its own mutex.

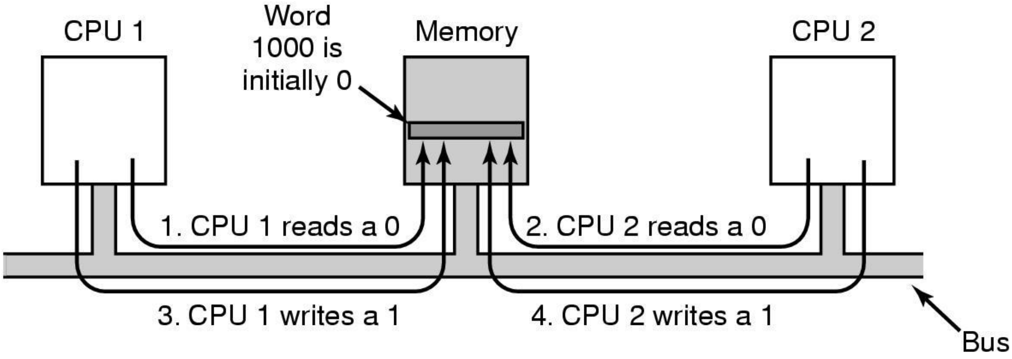


Bus

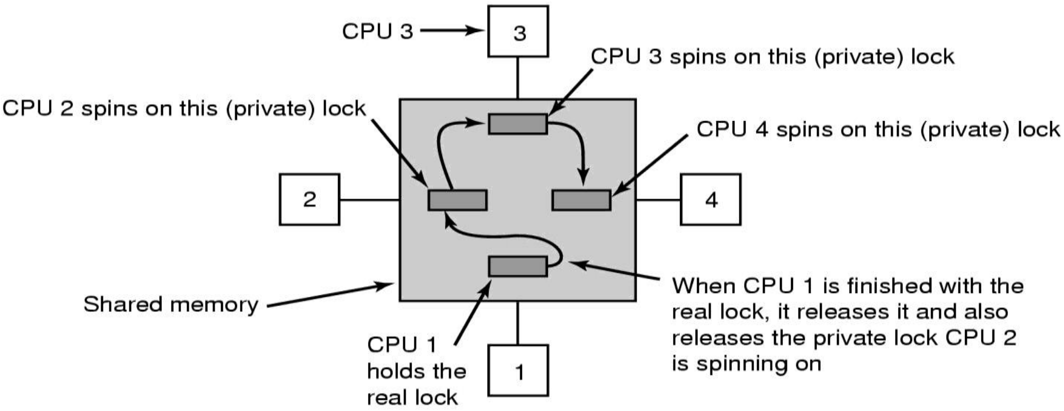
**MULTIPROCESSOR SYNCHRONIZATION:**

- The CPUs in a multiprocessor frequently need to synchronize.
- If a process on a uniprocessor machine makes a system call that requires accessing some critical kernel table, the kernel code can just disable interrupts before touching the table.
- On a multiprocessor, disabling interrupts affects only the CPU doing the disable. Other CPUs continue to run and can still touch the critical table.
- The worst-case timing, in which memory word 1000, being used as a lock, is initially 0.

- In step 1, CPU 1 reads out the words and gets a 0.
- In step 2, before CPU 1 has a chance to rewrite the word to 1, CPU 2 gets in and also writes a 1 into the word. Both CPUs got a 0 back from the TSL instruction, so both of them now have access to the critical region and the mutual exclusive fails.



- To prevent this problem, the THL instruction must first lock the bus, preventing other CPUs from accessing it, then do both memory accesses, then unlock the bus.
- Locking the bus is done by requesting the bus using the usual bus request protocol.


CPU 3
3
CPU 3 spins on this (private) lock
CPU 2 spins on this (private) lock
CPU 4 spins on this (private) lock
2
4
Shared memory
When CPU 1 is finished with the real lock, it releases it and also releases the private lock CPU 2 is spinning on
CPU 1 holds the real lock
1

Use of multiple locks to avoid cache thrashing.

**SPINING VERSUS SWITCHING:**

Spinning versus Switching

- In some cases CPU must wait
  - waits to acquire ready list
- In other cases a choice exists
  - spinning wastes CPU cycles
  - switching uses up CPU cycles also
  - possible to make separate decision each time locked mutex encountered

**Multiprocessor scheduling:**

**Scheduling:**

- Scheduling on a single processor is one dimensional (process)
- Scheduling on a multiprocessor is two dimensional (process & CPU)
- Unrelated processes
- Related in groups processes

**Process** Vs thread is not the only scheduling issue. On a uniprocessor, scheduling is one dimensional.

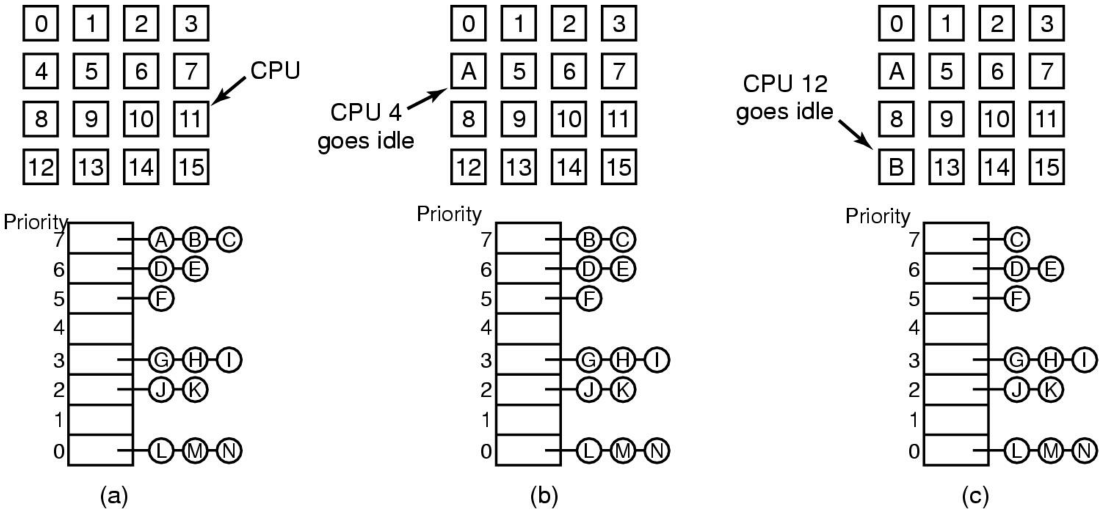
On a multiprocessor scheduling has tow dimensions.

- The scheduler has to decide which thread to run.
- Which CPU to run it on.

- An example of the former situation is a timesharing system in which independent users starup independent proceses.
- An example of latter situation occurs regularly in program development environments.large systems often consist of some number of hader files contsining macro, type definitions, and variables declarations thar are used by the actual code files.
- The program make is commonly used to manage development.
- When make is invoked, it starts the compilation of only those code files that must be recompiled on account of changes to the header or code files. Object files that are still valid are not regenerated.

**Timesharing:**

- The simplest scheduling algorithm for dealing with unrelated threads is to have a single system-wide data structure for ready threads, possibly just a list, but more likely a set of lists for threads for threads at different priorities as depicted in figure.
- To get around this anomaly, some systems use smart scheduling, in which a thread acquirering a spin lock sets a process-wide flag to show that it currently has a spin lock. when it releases the lock, it clears the flag.
- Some multiprocessors take this effect into account and use what is called affinity scheduling. The two basic idea here is to make a serious effort to have a thread run on the same CPU it ran on last time.



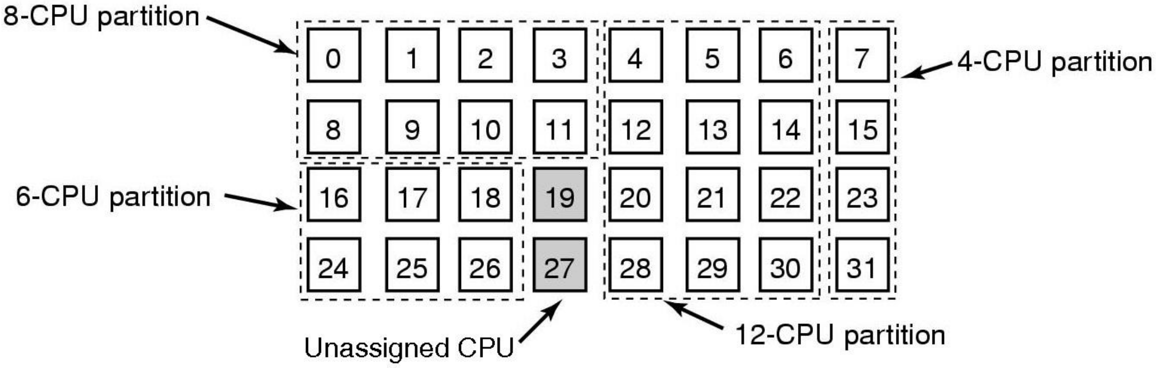
Using a single data structure for scheduling a multiprocessor.

- One way to create this affinity is to use a two-level scheduling algorithm.when a thread is created, it is assigned to a CPU, for example based on which one has the smallest load at that moment.
- The actual scheduling of the threads is the bottom level of the algorithm . it is done by each CPU sperately, using priorities or some other means.
- Two-level scheduling has three benefits:
  - It distributes the load roughly evenly over the available CPUs.
  - It taken of cache affinity where possible.
  - By giving each SPU its own ready list connection for the ready lists is minimized because attempts to use another CPU’s ready list are relatively infrequent.

**Space sharing:**

- The other general approach to multiprocessor scheduling can be used when threads are related to one another in some way.
- Scheduling multiple threads at the same time across multiple CPUs is called space sharing.

- The simplest space-sharing algorithm works like this. Assume that an entire group of related threads is created at once.
- Atany instant of time, the set of CPU is statically partitioned into some number of partitions, each one running the threads of one. In the following figure we have partitioned of sizes 4,6,8 and 12 CPUs, with 2 CPUs unassigned, for example .
- As time goes on, the number and size of the partitions will change as new threads are created and old ones finish and terminate.



A set of 32 CPUs split into four partitions, with two CPUs.

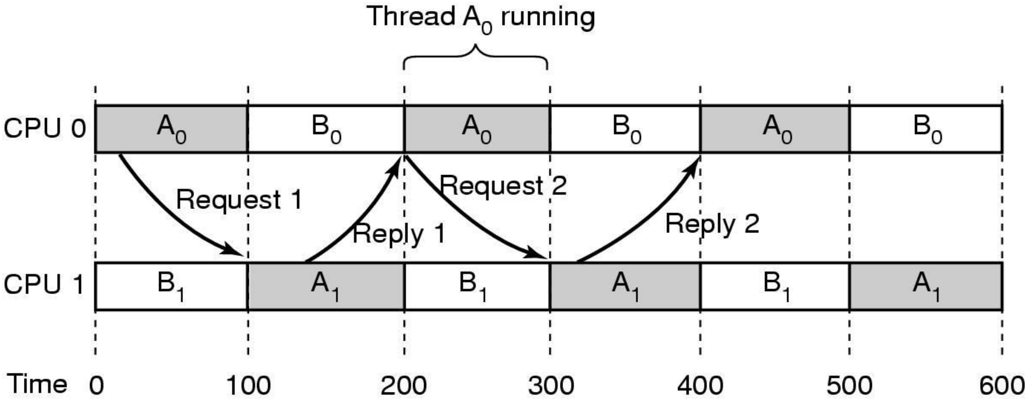
- Periodically,scheduling decisions have to be made. In uniprocessor systems, shortest job first is a well-known algorithm for batch scheduling.
- The analogous algorithm for a multiprocessor is to choose the process needing the smallest number of CPUs cycles, that is, the threads whose

CPU-count X run time is the smallest of the candidates.

**Gang scheduling:**

A clear advantage of space sharing is the elimination of multiprogramming, which eliminates the context switching overhead.

- To see the kind of problem that can occur when the threads of a process are independently scheduled,consider a system with threads A0 and A1 belonging to process a and threads B0 and B1 belonging to process B. threads A0 and B0 are timeshared on CPU 0; threads A1 and B1 are timesharing on CPU1.



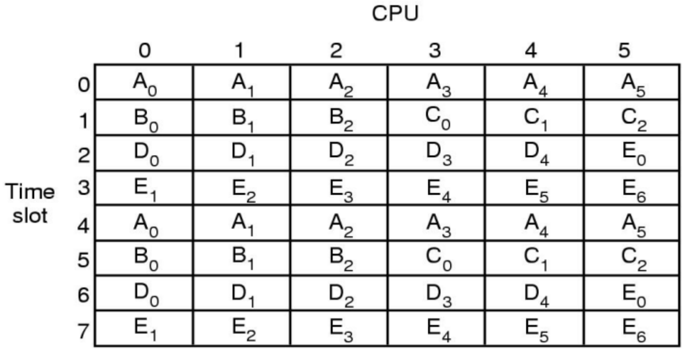
Communication between two threads belonging to thread A are running out of phase.

- The solution to this problem is gang scheduling, which is an out growth of **co-scheduling. Gang scheduling** has three parts:
    - Groups of related threads are scheduled as a unit, a gang.
    - All members of a gang run simultaneously, on different timeshared CPUs.

- All gang members start and end their time slices together.

All **CPUs** scheduled **synchronously.**

**Time** divided into **quanta.**



| Time slot | CPU 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
| 1 | $B_0$ | $B_1$ | $B_2$ | $C_0$ | $C_1$ | $C_2$ |
| 2 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $E_0$ |
| 3 | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ | $E_6$ |
| 4 | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ |
| 5 | $B_0$ | $B_1$ | $B_2$ | $C_0$ | $C_1$ | $C_2$ |
| 6 | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $E_0$ |
| 7 | $E_1$ | $E_2$ | $E_3$ | $E_4$ | $E_5$ | $E_6$ |

Gang scheduling

## MULTICOMPUTERS:

Multiprocessors are popular and attractive because they offer a simple communication model:all CPUs share a common memory.processors can write messages to memory that can be read by other processors.to get around these problems, much research has been done on multicomputers, which are tightly coupled CPUs that do not share memory.

These systems are also known as **Clustering computers**, and **COWS(clusters of workstations).**

Interconnection network is **crucial**

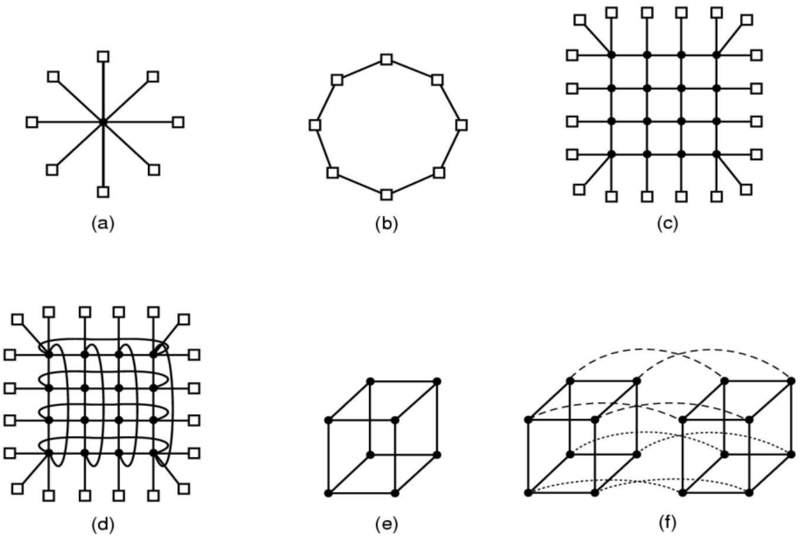
**Multicomputer hardware:**

- The basic node of a multicomputer consists of a CPU, memory, a network interface, and sometimes a hard disk.
- The node may be packaged in a standard PC case, but the graphics adapter, monitor, keyboard, and mouse are nearly always absent.

**Interconnection technology:**

There are six interconnection technologies which each node has a network interface card with one or two cables. They are.

- Star topology.
- Ring topology.
- Grid or mesh.
- Double torus.
- Cube.
- Hypercube.



**1. Star topology:**

In a small system there may be one system, there may be one switch to which all the nodes are connected in the star topology. Fig (a)

**2. Ring topology:**

While the two wires coming out the network interface card, one going into the node on the left and one going into the node on the right. Fig (b)

**3. Grid or mesh:**

Is a two-dimensional design that has been used in many commercial systems.it has a diameter, which is the longest path between any two nodes. Fig (c)

**4. Double torus:**

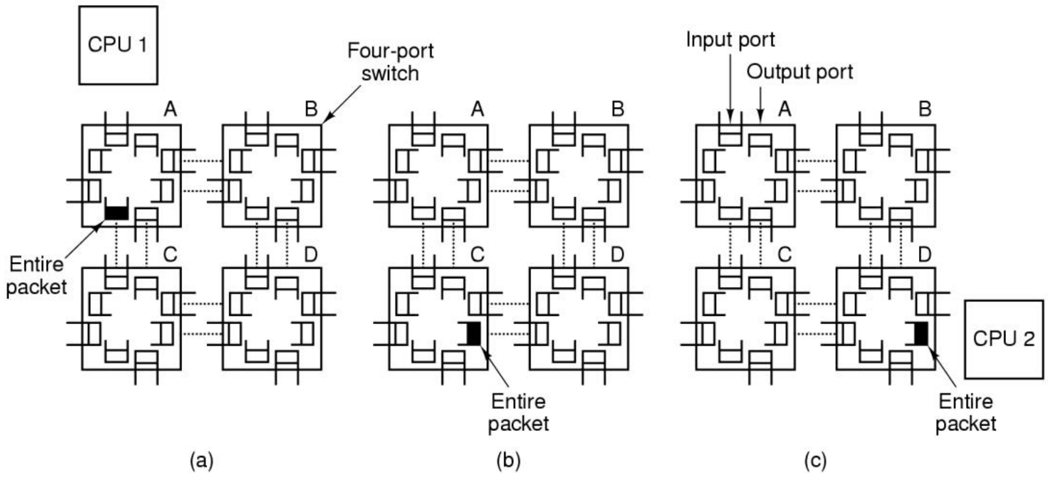
A variant of the grid is the double torus. Which is a grid with the edges connected. Fig (d)

**5. Cube:** Is a regular three dimensional topology. Fig (e)

**6. Hypercube:**

We have a four dimensional cube built from two three dimensional cubes with the corresponding nodes connected. We could make a five dimensional cube by cloning the structure. Fig (f)

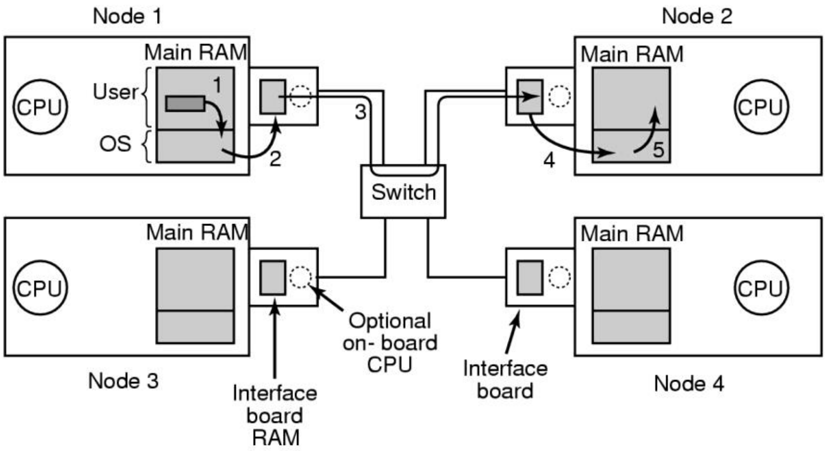
- Two kinds of switching schemes are used in multicomputers
  - First one each messages is first broken into a chunk of some maximum length called packet.the sitching scheme is called store and forward packet switching.
  - Second one is the other switching regime, circuit switching, consists of the switch first establish a path through all the switches to the destinations switch. A variation on the circuit switching, called wormhole routing.



Store and forward packet switching.

**Network interfaces:**

- The way these boards are built and how they connect to the main CPU and RAM have substantial iimplications for the operating systems.
- If the packet is in the main RAM, this continuous flow out onto the network cannot be guaranteed due to other traffic on the memory bus.
- Using a dedicated RAM on the interface board eliminates this problem. This design is shown as follows:



Position of the network interface boards in a multicomputer.

- Many interface boards have a full CPU on them possibly in addition to one or more DMA channels. They are called **Network processors.**

**Low-Level Communication Software :**

**Problems:**

- If several processes, running on node, need network access to send packets …?

One can map the interface board to all process that need it
but a synchronization mechanism is needed (difficult for multiprocessing)

- If kernel needs access to network …?

We can use two network boards

  - One mapped into user space, one into kernel space

- DMA uses physical address, user process virtual address.

It is needed to handle the problem without system calls.

**Node to network interface communication:**

Node to Network Interface Communication, with on-board **CPU**

- Use send & receive rings, with bitmap
- coordinates main CPU with on-board CPU

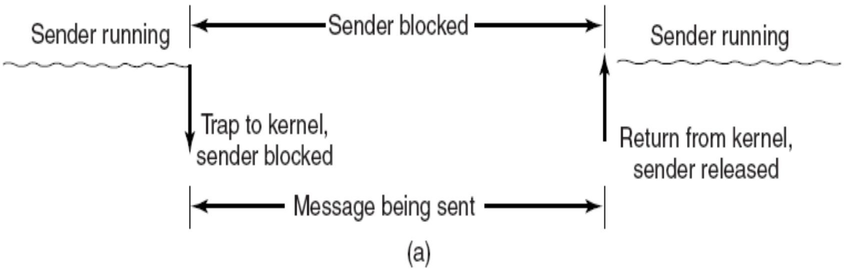
**User Level Communication Software:**

**Send and receive:**

- Multicomputers communicate through messages
  - Send(dest, &mptr) : send a message pointed by *mptr* to a process identified by *dest*
  - Receive(addr, &mptr) : *addr* is usually CPU number and a process or port number
- Send calls can be *blocked* or *nonblocked*
- Receive calls are always blocked. That is, the receiving process has to wait if the message has not been sent before

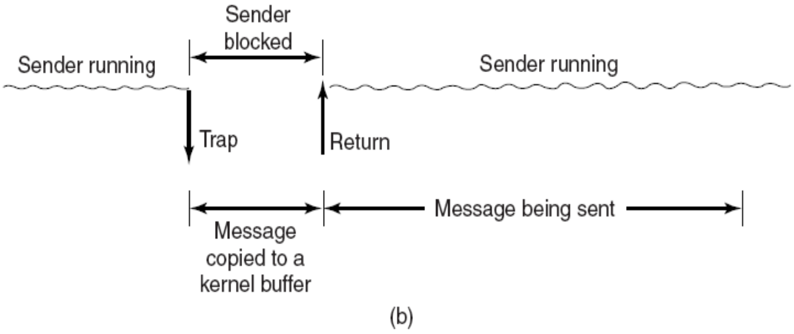
**Blocking versus non blocking calls:**

- **Blocking (synchronous) calls**
  - these are blocking (synchronous) calls
  - CPU idle during transmission


Sender running
Sender blocked
Sender running
Trap to kernel, sender blocked
Return from kernel, sender released
Message being sent
(a)

A blocking send call.

- **Non blocking calls(asynchronous)**
  - with copy
  - with interrupt
  - copy on write



A non blocking sends call.

- Yet another option is a scheme in which the arrival of a message causes a new thread to be created spontaneously in the receiving process address space. Such a thread is called a **pop-up thread.**
- The big win here is that no copying at all is needed. The handler takes the message from the interface board and processes to on the fly. This scheme is called **active message.**

**Remote Procedure Call:**

- The procedure send and receive are fundamentally engaged in doing I/O, and many people believe that I/O is the wrong programming model.
- No message passing or I/O at all is visible to the programmer. This technique is known as **RPC(Remote Procedure Call).**
- The idea behind the RPC, to call a remote procedure, the client program must be bound with a small library procedure called the **client sub.**
- That represents the server procedure in the clients address space. Similarly the server is bound with a procedure called **server stub.**
- The actuall step in making an **RPC** is shown in the figure.

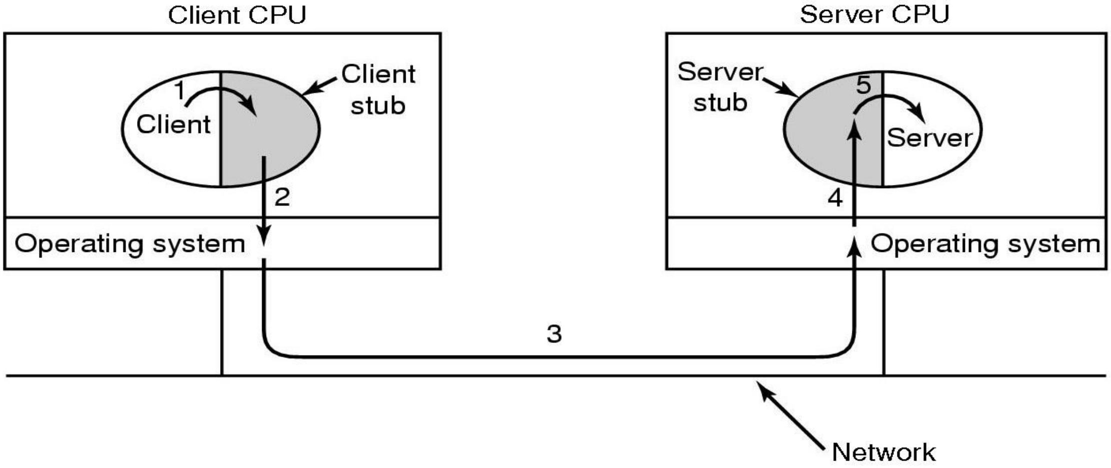
**Step 1:** the client calling the client stub.

**Step 2:** the client stub packing the parameters into a message and making a system Call to send the message.packing the parameters is called **marshaling**.

**Step 3:** the kernel sending the message from the client machine to the server.

**Step 4:** the kernel passing the incoming packet to the server machine.

**Step 5:** the server stub calling the server procedure.



Steps in making a remote procedure call. The stubs are shaded gray.

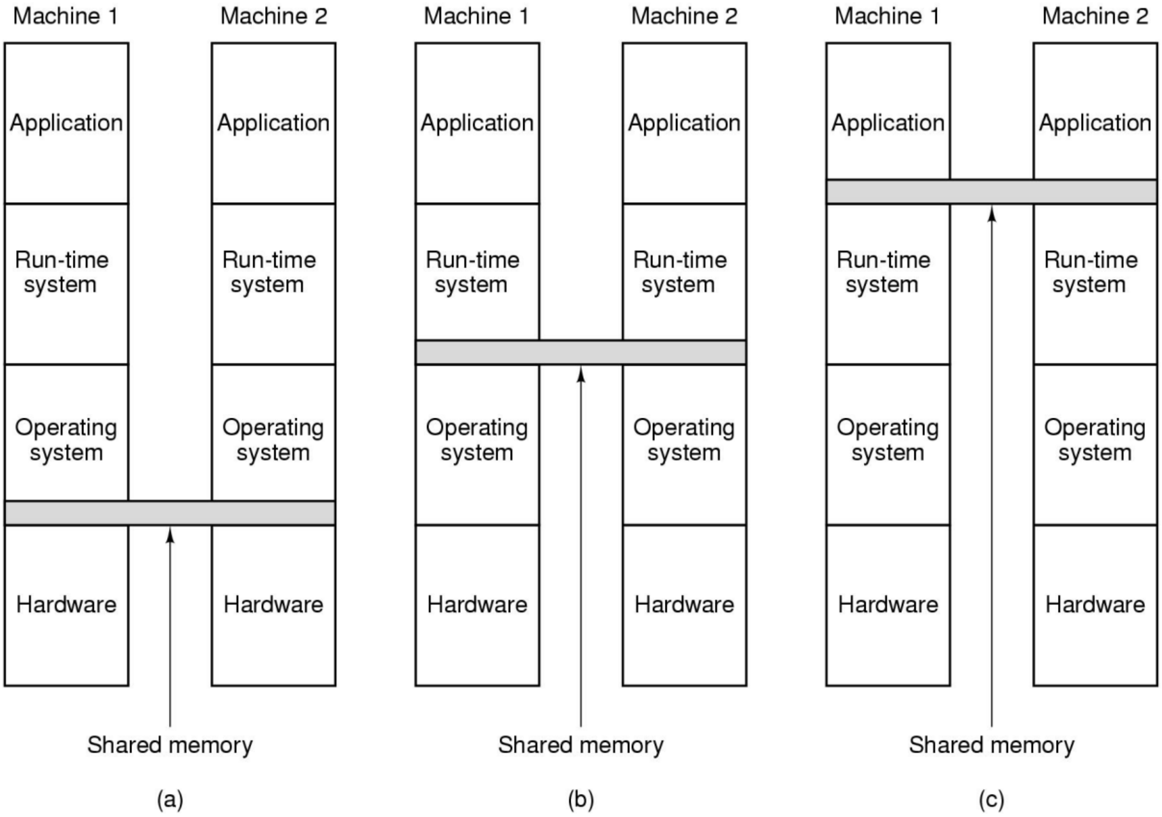
**Implementing issues:**

- Cannot pass pointers
  - call by reference becomes copy-restore (but might fail)
- Weakly typed languages
  - client stub cannot determine size
- Not always possible to determine parameter types
- Cannot use global variables
  - may get moved to remote machine

**Distributed Shared Memory :**

Many programmers still prefer a model of shared memory and would like to preserve the illusion of scared memory reasonably well, even when it does not actually exist, using a technique called **DSM(Distributed Shared Memory).**

The difference between actual shared memory and DSM is illustrated in figure.

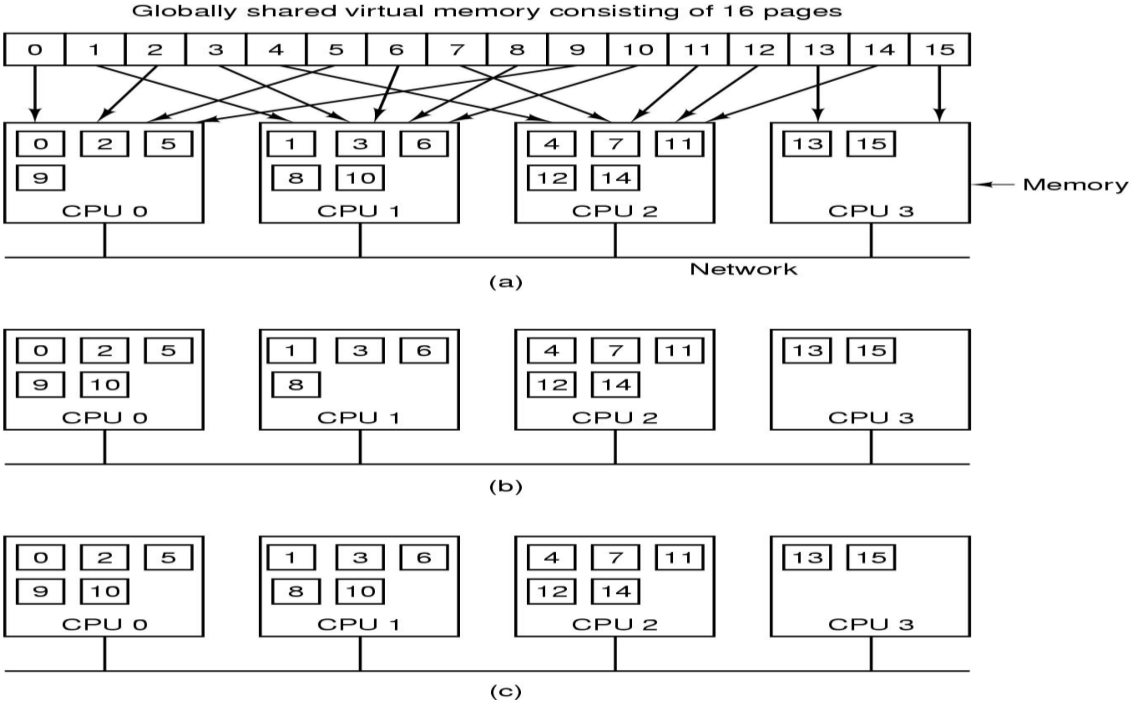


Various layers where shared memory can be implemented.

A. the hardware b. the operating system c. user-level software.

**Replication:**

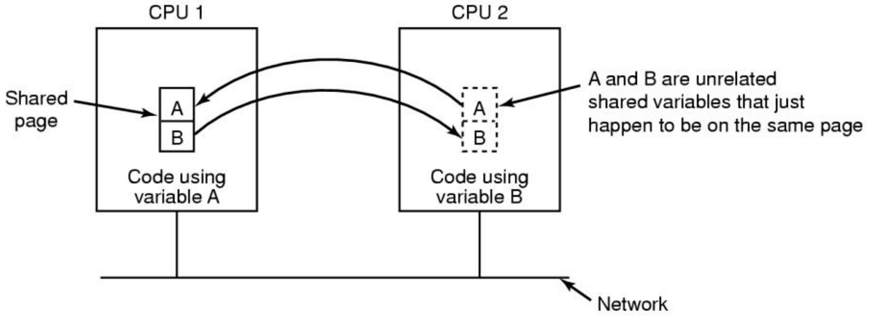
- *Pages* distributed on 4 *machines*
- CPU 0 references page 10
- CPU 1 reads (only) page 10 (replicated)



(a) pages of the address space distributed among four machines.
(b) Situation after CPU 1 references page 10 and the page is moved there.
(c) Situation if page 10 is read only and replication is used.

**False sharing:**

- Too large an effective page size introduces a new problem called **flase sharing**.
- DSM uses multiple of page size: *page size* is important
- False Sharing
- Must also achieve sequential consistency (write on replicated page)



CPU 1
CPU 2
Shared page
A
B
A
B
A and B are unrelated shared variables that just happen to be on the same page
Code using variable A
Code using variable B
Network

Flase sharing of page containing two unrelated variables.

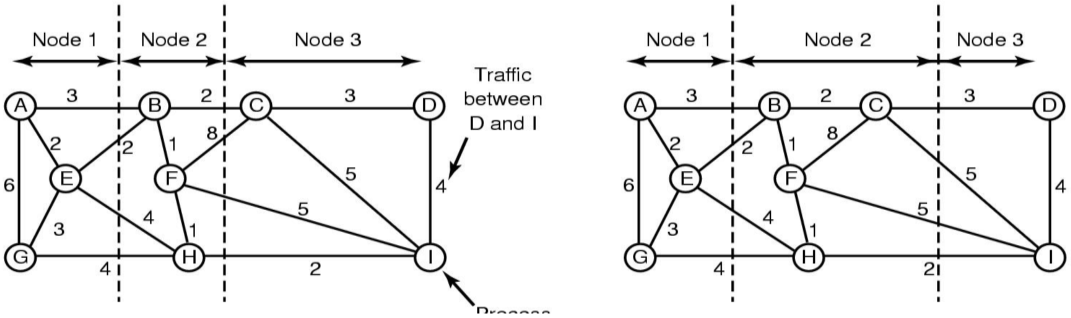
**Multicomputer scheduling:**

**Load balancing:**

- This is in contrast to multiprocessor systems. The algorithms and heuristics for doing this assignment are known as **processor allocation algorithms.**

**Graph-theoretic *deterministic algorithm:***

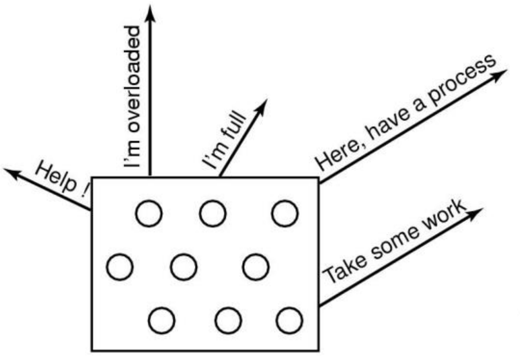
- Each process can only run on the CPU where it is located on. Choice parameters: CPU & memory. usage, comm. needs etc



Process

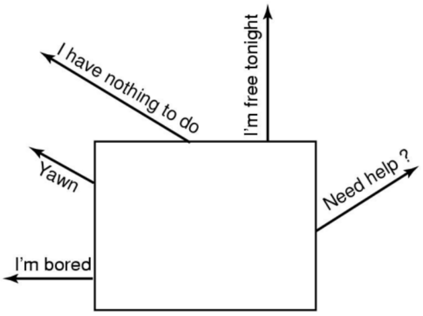
**Sender-initiated distributed *heuristic algorithm:***

- Processes run on the campus that have created them, unless overloaded



(a)

An overloaded node looking for a lightly loaded node to hand off processes to.



(b)

**Receiver-initiated distributed *heuristic algorithm:***

# UNIT - V

## PRINCIPLES OF I/O HARDWARE

- Programmers look at the interface presented to the software the commands the hardware accepts the function it carries out and errors that can be reported back.

### 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.
- All transfers are in units of one or more entire blocks.
- The essential property of a block device is that it is possible to read or write each block independently of all the other ones hard disks CD –ROMs and USE sticks are common block devices.
- A tape drive used for making disk backups tapes contain a sequence of blocks if the taps drive is given a command to read block.
- It can always rewind the tape and go forward until it comes to block.
- 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.
- I / O devices cover a huge range in speeds, which puts pressure on the software to perform well over many orders of magnitude in data rates.

| Device | Data rate |
|---|---|
| Keyboard | 10 by/sec |
| Mouse | 100by/sec |
| Scanner | 400KB/sec |
| USB 2.0 | 60 MB/sec |
| 52xCD-ROM | 7.8MB/sec |
| PCI bus | 528 MB/sec |
| Fast Ethernet | 12.5 MB/sec |

### Device Controllers

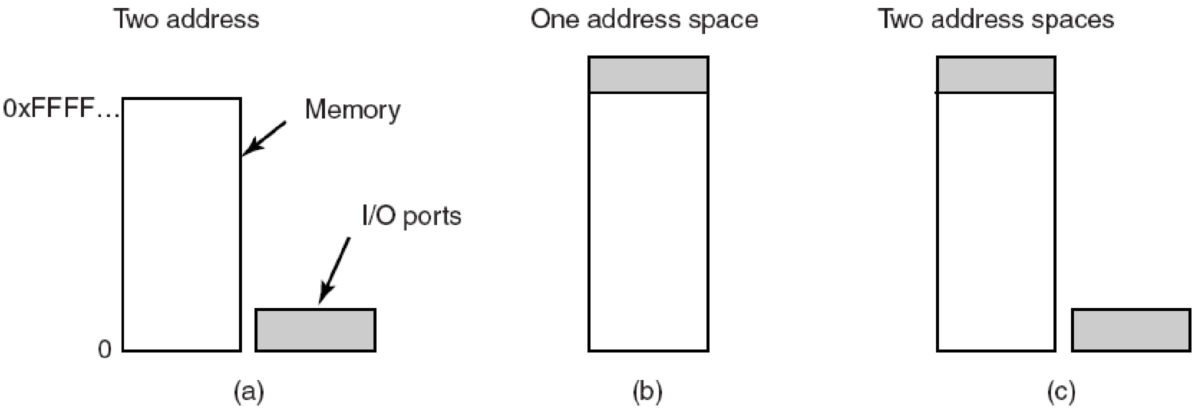
- I/O units typically consist of a mechanical component and an electronic component.
- It is often possible to separate the two portions to provide a more modular and general design.
- The electronic component is called the device controller or adapter.
- The controller card usually has a connector on it into which a cable leading to the device itself can be plugged.
- If the interface between the controller and device is a standard interface either an official ANSI, IEEE, or ISO standard or a de fact one, then companies can make controller or devices that fit that interface.
- Disk drives that match the IDE, SATA, SCSI, USB, or fire wire interface.
- The interface between the controller and the device is often a very low-level interface.
- A disk for example, might be formatted with 10,000 sectors of 512 bytes per track. Check sum is called as Error-Correcting Code (ECC).
- The preamble is written when the disk is formatted and contains the cylinder and sector number, the sector size and data.

- The controller for a monitor also works as a bit serial device at an equally low level.
- It reads bytes containing the character to be displayed from memory and generates the signals used to modulate the CRT beam to cause it to write on the scree

**Memory-Mapped I/O**

- Each controller has a few registers that are used for communicating with the CPU.
- By writing in to the registers, the OS can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action.
- Many devices have a data buffer that the OS can read and write.
- The issue thus arises of how the CPU communicates with the control registers and the device data buffers.
- Two alternatives exits.

1. Each control register is assigned an I/O port number, an 8- or 16-bit integer.
   The set of all the I/O ports form the I/O port space and is protected so that ordinary user programs cannot access it.
   Using a special I/O instruction IN REG, PORT,
   The CPU can read in control register PORT and store the result in CPU register REG.
   OUT PORT, REG - the CPU can write the contents of REG to a control register.
   The Address spaces for memory and I/O are different. The instructions
   IN RO, 4 and MOV RO, 4



2. Each control register is assigned a unique memory one address space address to which no memory is assigned. This system is a called memory-mapped I/0.A hybrid scheme, with memory-mapped I/O data buffers and separate I/O ports for the two address spaces control registers.

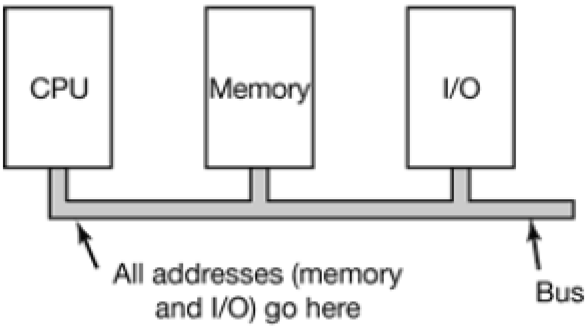
- Two scheme for addressing the controller – strength and weaknesses.

**Advantages:**

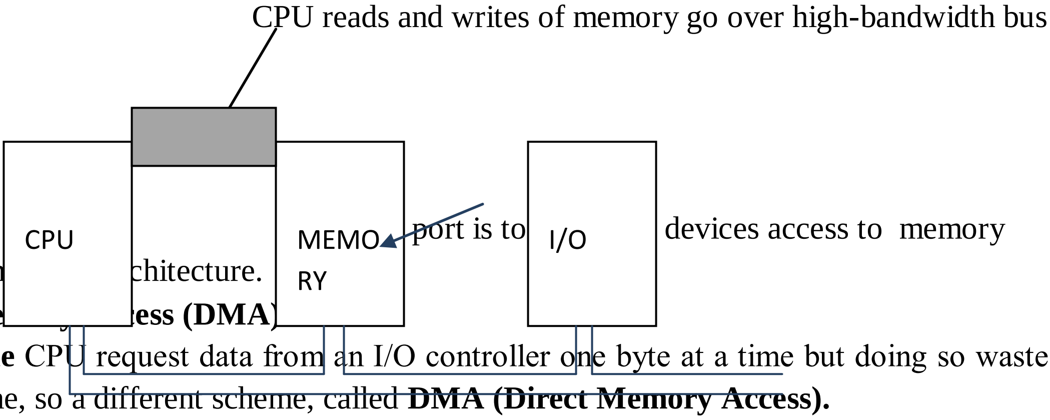
1. Special I/O instructions are needed to read and write the device control registers, access them – assembly code.
2. With memory-mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O.
3. With memory-mapped I/O, every instruction that can reference memory can also reference control registers.

- If the computer has a single bus, having everyone look at every address is straightforward.

(a) A single-bus architecture



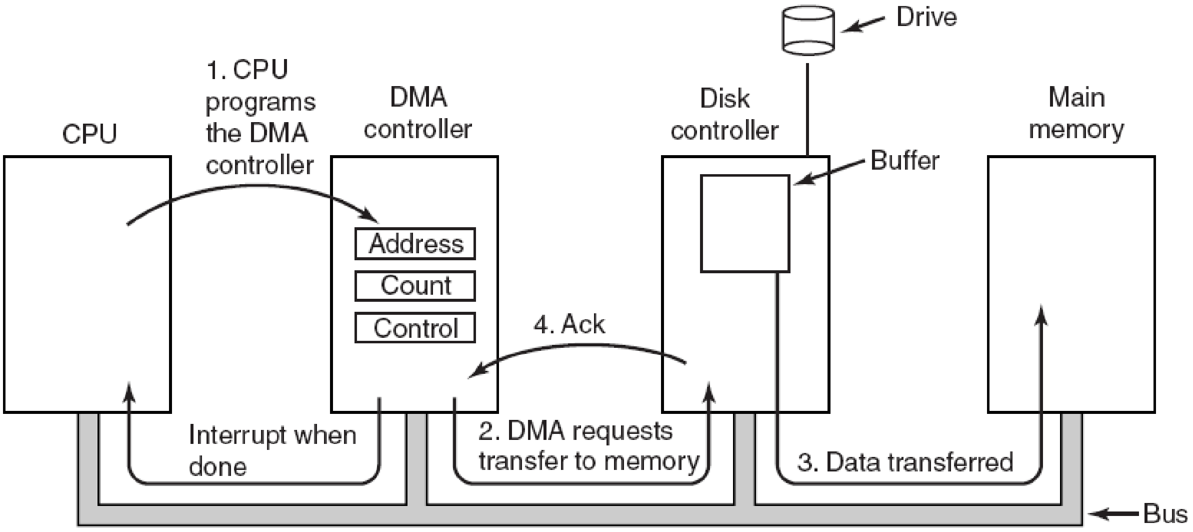
- PC have high-speed memory bus.



(b)A dual bus n chitecture.

**Direct Me ess (DMA)**

- **The** CPU request data from an I/O controller one byte at a time but doing so wastes the CPU's time, so a different scheme, called **DMA (Direct Memory Access).**
- A single DMA controller is available for regulating transfer to multiple devices.

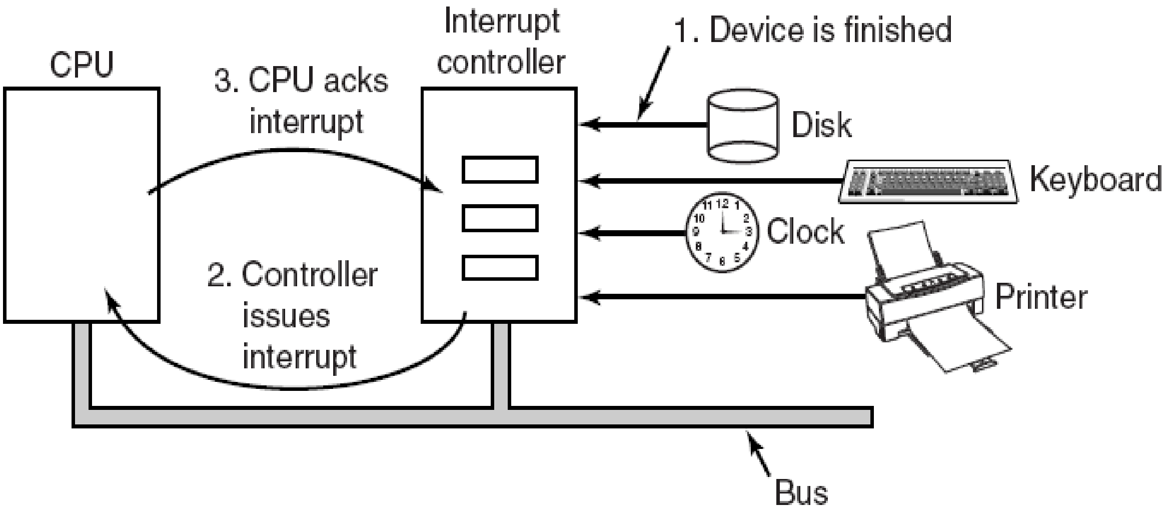


- A controller has access to the system bus independent of the CPU.
- These include a memory address register a byte count register and one or more control registers.
- The control registers specify the I/O port to use the direction of the transfer the transfer unit and the number of bytes to transfer in one burst.
- How DMA works DMA is not used the disk controller reads the block from the drive serially bit by bit until the entire block is in the controller internal buffer.
- It computers the checksum to verify that no read errors have occurred.
- Then the controller causes an interrupt.
- When the operating system starts running it can read the disk block from the controller buffer a byte or a word at a time by executing a loop with each iteration reading one byte or word from a controller device register and storing it in main memory.

- When DMA is used the CPU programs the DMA controller by setting its registers so it knows what to transfer.
- It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum.
- When valid data are in the disk controller buffer DMA can begin.
- The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller.
- The memory address to write to is on the bus address lines so when the disk controller fetches the next word from its internal buffer it knows where to write to memory is another standard bus cycle.
- When the write is complete the disk controller sends an acknowledgement signal to the DMA controller also over the bus .
- The CPU starts by loading each set of registers with the relevant parameters for its transfer.
- It may be set up to use a round –robin algorithm or it ay have a priority scene design to favor some devices over others

**Interrupts revisited**

- When an I/O devices has finished the work given to it it causes an interrupts  it does this by asserting a signal on a bus line that it has been assigned.
- The interrupt signal causes the CPU to stop what it is doing and start doing something else.
- This program counter points to the starts of the corresponding interrupt service procedure.
- The hardware always saves certain information before starting the service procedure.
- Which information is saved and where it is saved varies greatly from CPU to CPU.



**Precise and imprecise interrupts**

- An interrupt that leaves the machine in a wall-defined state is called a precise interrupt .

Such an interrupt has four properties;

- 1. The PC is saved in a known place.
- 2. All instructions before the one pointed to by the PC have fully executed
- 3.No instruction beyond the one pointed to by the  PC has been executed.
- 4. The execution state of the instruction pointed to by the PC is known.

**PRINCIPLES OF I/O SOFTWARE:**

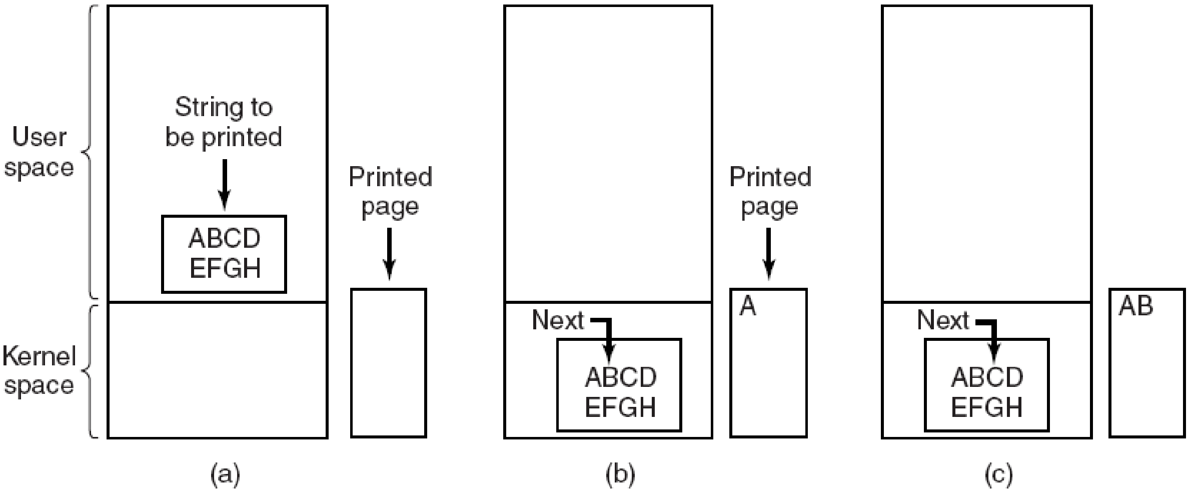
**Goals of the I/O software**

- A key concept in the design of I/O software is known as device independence.

- Example a program that reads a file as input should be able to read a file on a hard disk a CD-ROM a DVD or a USB stick without having to modify the program for each different device.
- A USB stick can be mounted on top of the directory so that copying a file to copies the fle to the USB stick.
- Another important issue for I/O software is error handing.
- Another issue for the I/O software is buffering.
- Often data that come off a device cannot be stored directly inits final destination.

**Programmed I/O**

- There fundamentally different ways that I/O can be performed.
- First one programmed I/O interrupt-driven I/O and I/O using DMA.
- It is simplest to illustrate programmed I/O by means of an example.
- Consider a user process that wants to print the eight-character string ABCDEFGH on the printer.
- The operating system then copies the buffer with the string to an array say P in kernel space.
- It then checks to see if the printer is currently available if not it waits until it is available.
- The first character has been printed and that the system has marked the B as the next character to be printed.
- At this point the operating system waits for the printer to become ready again when that happens it prints the next character as shown in fig 5-7



- This loop continues until the entire string has been printed then control returns to the user process.

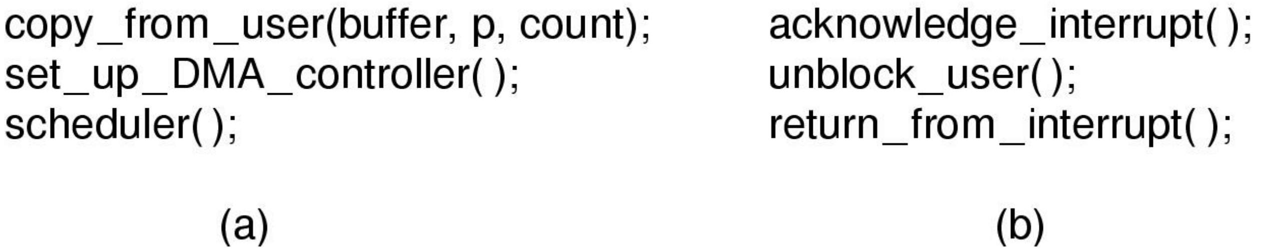
**Interrupt- driven I/O**

- If the printer can print say 100 characters/sec each character tahes 10 msec to print.
- This means that after every character is written to the printers data register the CPU will sit in an idle loop for 10msec waiting to be allowed to output the next character.
- When the system call to print the string is made the buffer is copied to kernel space as we showed earlier and the first character is copied to the printer as soon as it is willing to accpt a character.
- At that point the CPU calls the scheduler and some other process is run.
- When the printer has printed the character and prepared to accpt the next one it generates an interrupt.

**I/O using DMA**

- Interrupt take time so this scheme wastes a certain amount of CPU time.
- A solution is to use DMA.

- Here the idea is to let the DMA controller feed the character to the printer one at time without the CPU being bothered.
- The big win with DMA is reducing the number of interrupt from one per character to one per buffer printed.
- If there are many character and interrupts are slow this can be a major improvement.



```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

## Files Systems:

- 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.
- Magnetic disks have been used for years for long-term storage.
- Tapes and optical disks are also used, but lower performance.

**Files:**

- Files the most important concept related to OS.
- Files are logical units of information created by processes.
- Processes can read existing files and create new ones if need be.
- Part of the operating system dealing with files is known as the file system.

➢ **File Naming**

- Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later.
- When the process creates a file, it gives the file a name.
- When the process terminates, the file continues to exist and can be accessed by other processes using its name.
- Some file systems distinguish between upper and lower case letters, whereas others do not.
- UNIX in first category, MS-DOS in second.
- Ex: file with name: maria, Maria and MARIA.
- In **UNIX** - as **the different** but in **MS-DOS** – **same file name**
- Many operating systems support two-part file names, separated by a period, as **prg.c**
- The part after the period is called the file extension, which indicates something about the files.

| **Extension** | Meaning |
|---|---|
| file.bak | Backuo file |
| file.c | C source program |
| file.gif | Compuserver Gaphical Interchange Format image |
| file.hlp | Help file |
| file.html | WWW Hyper Text Markup Language document |
| file.jpg | Still picture encoded with the JPEG stsndard |

| | |
|---|---|
| file.mp3 | Music encoded with the MPEG standard |
| file.zip | Compressed archive |
| file.txt | General text file |
| file.pdf | Portable Document Format file |

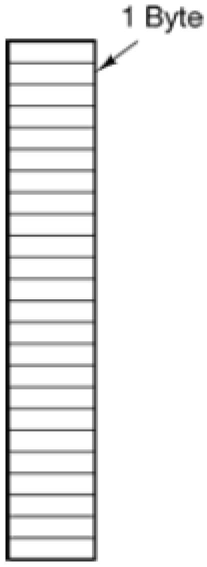
- In some systems (UNIX) file extensions are just convention and are not enforced by the OS.
- Windows is aware of the extensions and assigns meaning to them. When a user double clicks on a file name, the program assigned to its file extension is launched with the file.
- EX: double clicking on file.doc starts Microsoft word with ***file.doc*** as the initial **file to edit.**

**File Structure**

- Files can be structured in any of several ways.
  Three kinds of files.

**(a) Byte sequence**

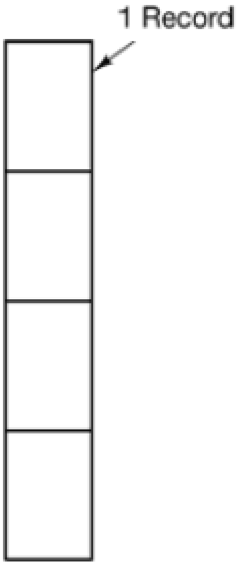
- The files are unstructured sequence of bytes
- The OS does not know what is in the file, it sees as bytes.
- The user programs can put anything they want in their files
  and any name for their need.
  UNIX, MS-DOS and Windows



(a)

**(b) Record sequence**

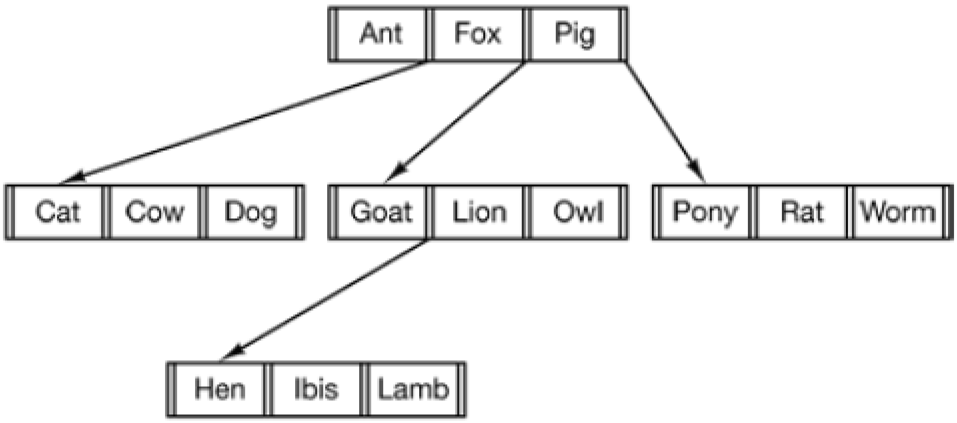
- A file is a sequence of fixed-length records, each with some internal structure.
- A file being a sequence of records is the idea that the read operation returns one record and write operation overwrites or appends one record



(b)

(c) Tree

- 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



(c)

- **File Types**:
- Many OS systems support several types of files.
- UNIX also has character and block special files.
- Regular files are the ones that contain user information.
- Directories are system files for maintaining the structure of the file system.
- **Character special files** are related to input/output and used to model serial I/O devices, like terminals, printers and networks.
- **Block special files** are used to model disks.
- Regular files are either ASCII files or binary files. ASCII files consist of lines of text.
  **Advantage** – they can display and printed as is and they can edit with any text editor.
- Other files are binary, which just means that they are not ASCII files.

- Ex:

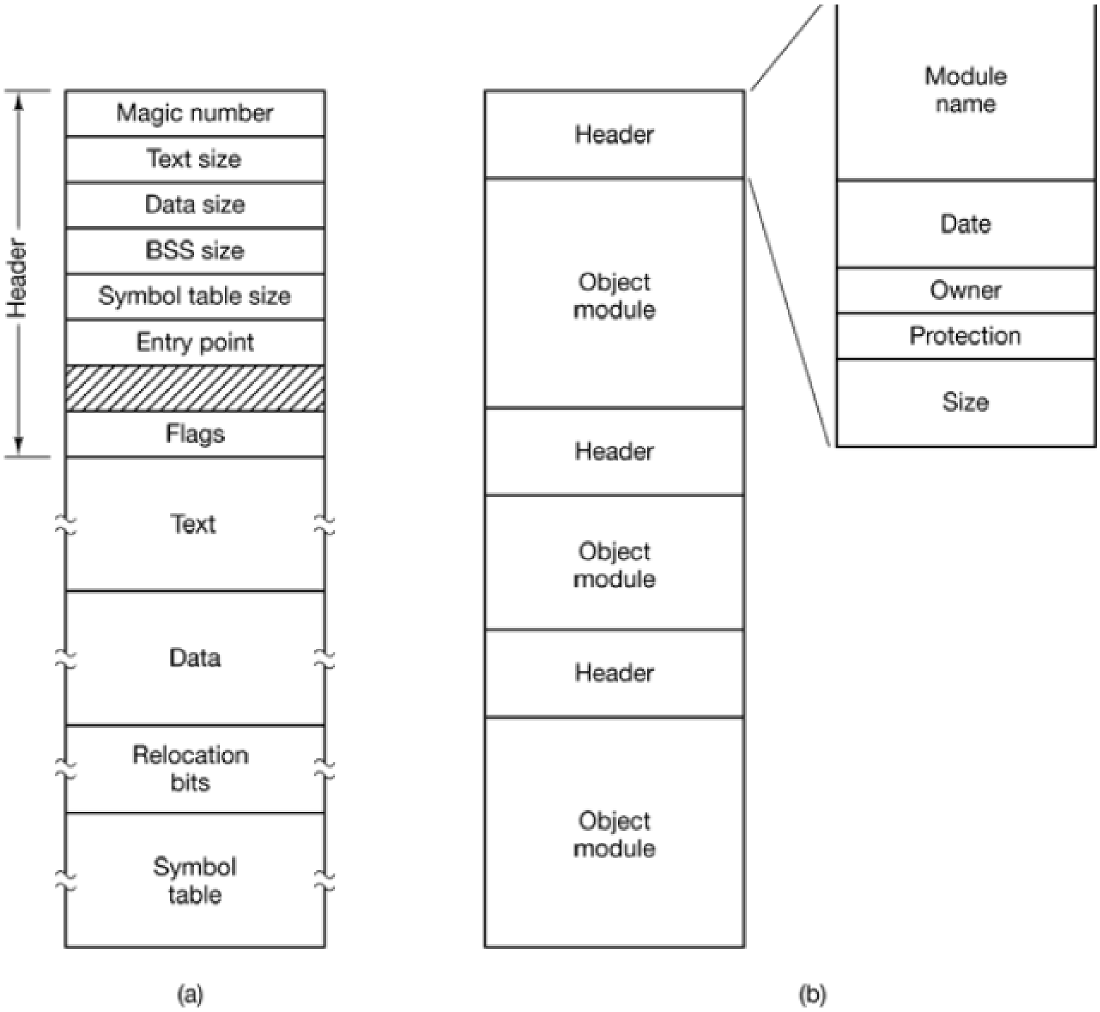


**Figure 6-3.** (a) An executable file. (b) An archive.

The file is just a sequence of bytes;

the OS will only execute a file if it has the proper format.

Five section: 1. header

2. Text

3. Data

4. Relocation bits

5.symbol table.

Header – starts with Magic number

Size - various pieces of the file, the address at which execution starts and some flag bits.

Text - program itself

Data

Relocation bits - program loaded into memory and relocated

Symbol table - used for debugging

Example 2:

- A binary file is an archive, also from UNIX.
- It consists of a collection of library procedures compiled but not linked.
- Each one is prefaced by a header telling its name, creation date, owner, protection cod and size.

**File Access:**

- OS provide a file access

1. **Sequential Access File**

- 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.
- Every read operation gives the position in the file to start reading at.

2. **Random Access File**

- When the disk is come into use for storing files, it become 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 records can be access in any order are called random access files.
- Ex: the airline seat reservation.
- A special operation seek, is provided to set the current position.

➢ **File Attributes:**

- Every file has a name and its data.
- All OS associate other information with each file, ex: the date, the time, last date of modified, file size.
- This data is also known **as metadata**.

| **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 files:1 for system file |
| System flag | 0 for ASCII file : 1 for binary file |
| Record length | Number of bytes in a 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 was last changed |
| Current size | Number of bytes in the file |
| Maximum size | Number of bytes the file may grow to |

- First four attribute- protection, who may access and who may not access.
- Flags are bits that control some specific property.

**File Operations:**

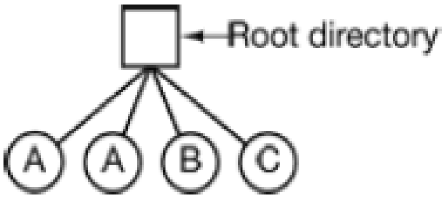
- Files exits to store information and allow it to be retrieved later.
- Different systems provide different operations to allow storage and retrieval.
  1. **Create:** The file is created with no data. It tell that the file is coming and to set some of the attributes.
  2. **Delete:** when the file is no longer needed, it has to be deleted to free up disk space.
  3. **Open:** It allow the system to fetch the attributes and list of disk address into main memory.
  4. **Close:** when all the accesses are finished, the attribute and disk addresses are no longer needed, so close it to free the internal table space.
  5. **Read:** Data are read from file. The bytes come from the current position.

6. **Write:** Data are written to the file again, usually at the current position. If current position is end of file, it increases the file size.
7. **Append:** It can only add data to the end of the file.
8. **Seek:** In random access file, to specify from where to take the data. A system call, seek, that repositions the file pointer to a specific place in the file.
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. Most of the flags are in this category.
11. **Rename:** It frequently happens that a user needs to change the name of an existing file.

**Directories:**

To keep track of files, file systems normally have directories or folders.

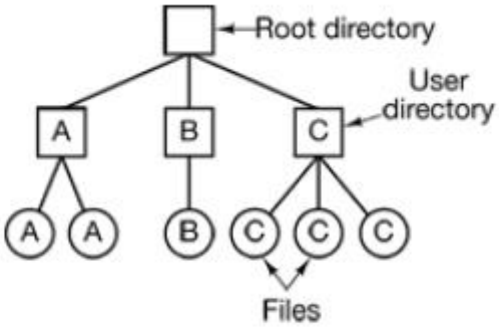
- **Single-Level Directory Systems:**
- Simple form of directory system is one directory containing all the files. It is called the root directory.



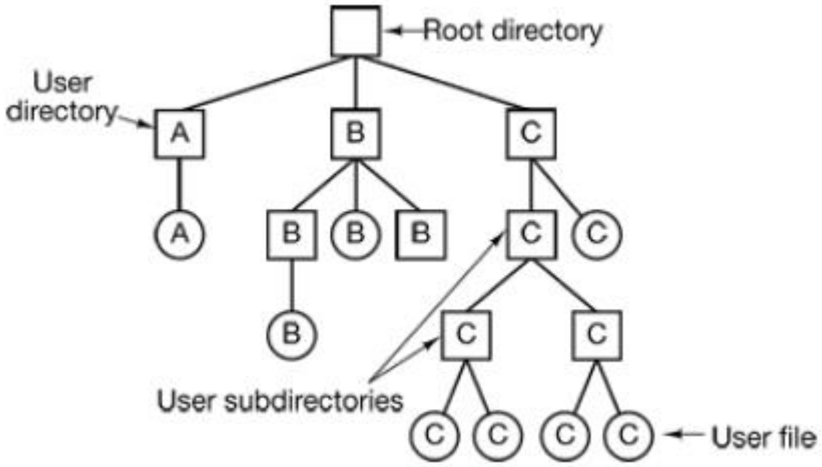
- Ex: a system with one directory
- Directory contains four files.
  - **Advantage:** It is simple and ability to locate files quickly- there is only one place to look.

## 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

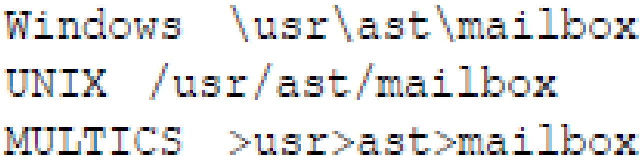


- **Hierarchical Directory Systems:**
- There can be as many directories as are needed to group the files in natural ways.
- If multiple users share a common file server, each user has a private root directory.
- The ability for users to create an arbitrary number of subdirectories provide a powerful structuring tool for users to organize their work.



- **Path names:**

- When the file system is organized as a directory tree.
- 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.



```
Windows  \usr\ast\mailbox
UNIX  /usr/ast/mailbox
MULTICS  >usr>ast>mailbox
```

- 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.
- 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.
- Each process has its own working directory,so when it changes its working directory and later exits,no other processes are affected and no traces of the change are left behind in the file system.
- Most operating systems that support a hierarchical directory system have two special entries in every directory,".''and''..'',generally pronounced "dot"and "dotdot".
- Dot refers to the current directory ;dotdot refers to its parent .
  Cp../lib/dictionary.
- The first path instructs the system to go upward then to go down to the directory lib to find the file dictionary.
- The second argument (dot)names the current directory .when the cp command gets a directory name(including dot)as its last argument,it copies all the files to that directory.

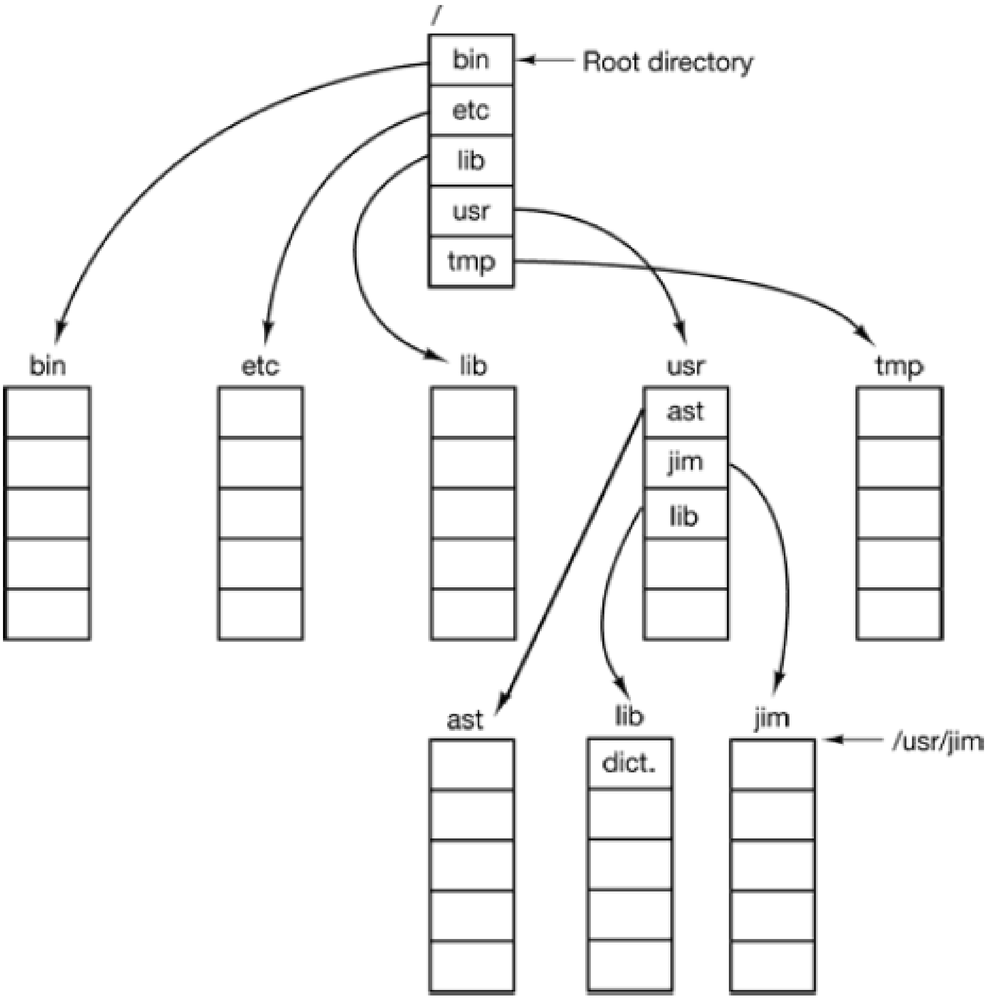


**Figure 6-10.** A UNIX directory tree.

**Directory Operations:**

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

- Create: a directory is created.
- Delete: a directory is deleted. Only on empty directory can be deleted.
- Opendir: directories can be read. Before a directory can be read, it must be opened, analogous to opening and reading a file.
- Closedir: when a directory has been read, it should be read, it should be closed to free up internal table space.
- Readdir: this call returns the next entry is an open directory. 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.
- Rename: in many respects, directories are just like files and can be renamed the same way files can be.
- Link: linking is a technique that allows a file to appear in more than one directory. This system a call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path.
- Uplink: a directory entry is removed. If the file being uplinked is only present in one directory, it is removed from the file system.
- A variant on the idea of linking files is the symbolic link.

**File System Implementation:**

**File system layout:**

- File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition.
- Sector 0 of the disk is called the MBR (master boot record) and is used to boot the computer.
- The end of the MBR contains the partition table. This table gives the starting and ending of each partition.
- One of the partitions in the table is marked as active.
- When the computer is booted, the BIOS read in and execute 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.
- Every partition starts with a boot block, even if it does not contain a bootable operating system.
- The layout of a disk partition varies a lot from file system to file system.
- Often the file system will contain some of the items shown.

**Possible file laylout**

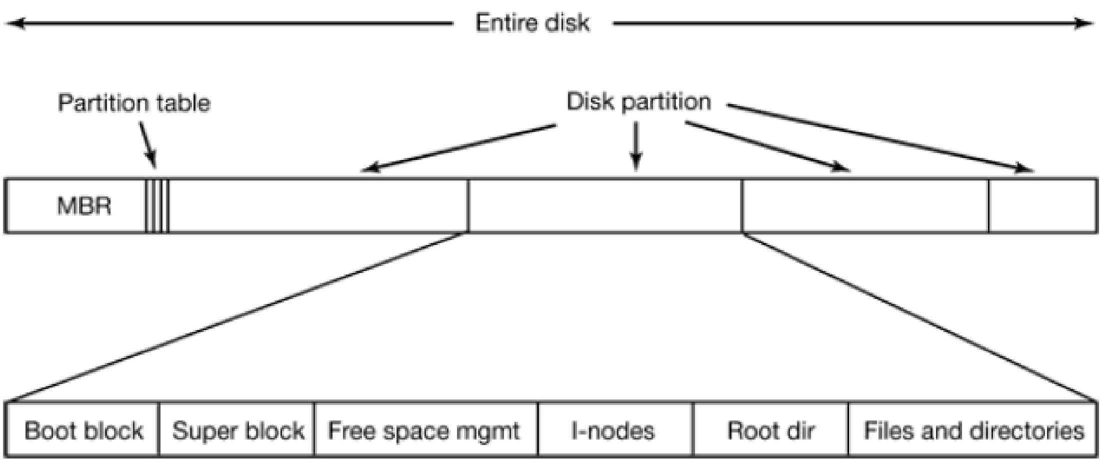


**Figure 6-11.** A possible file system layout.

- 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.
- Next about free blocks in the file system, for example in the form of a bitmap or a list of pointers.
- This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file.
- After that might come the root direvtory, which contains the top of the file system tree

- **Implementing Files**
  - Important issues in implementing file storage are keeping track of which disk blocks go with which file.
  - Various methods are used indifferent operating systems.

**Contiguous Allocation**

- The simplest allocation scheme is to store each file as a contiguous run of disk blocks.
- 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.
- Ex: the 40 disk blocks are shown, starting with block 0 on the left.
  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.
  Each file begins at the start of a new block, so that if file A was really 3 1/2 blocks.
  Some space is wasted at the end of the last block.

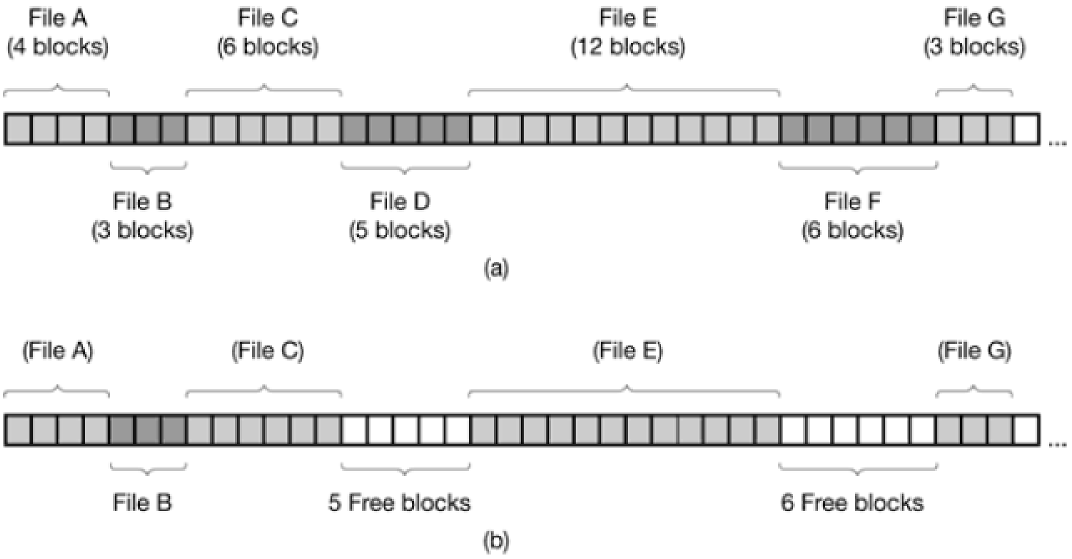


**Figure 6-12.** (a) Contiguous allocation of disk space for seven files. (b) The state of the

**Advantage:**

1. It is simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.
2. The read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed. Contiguous allocation is simple to implement and has high performance.

**Drawback:**

1. Over the course of time, the disk becomes fragmented.
2. Ex: two files, C & F, have been removed. When a file is removed, its blocks are naturally freed, leaving a run of free blocks on the disk.
   Disk is not compacted on the spot to squeeze out the hole.
   The disk ultimately consists of files and holes. Reusing the space requires maintaining a list of holes.

It is necessary to know its final size in order to choose a hole of the correct size to place it in.

There is one situation in which contiguous allocation is feasible and widely used on CD-ROMs.

**Linked List Allocation**

- Storing files is to keep each one as a linked list of disk.
- The first word of each block is used as a pointer to the next one. The rest of the block is for data.

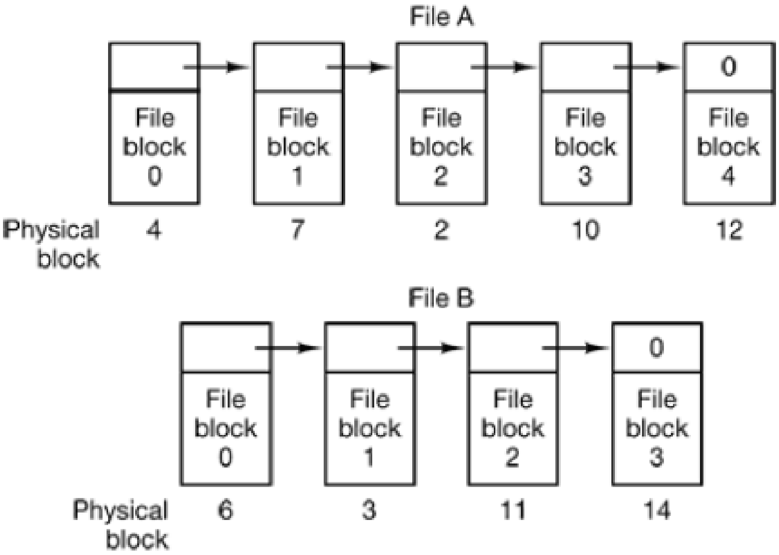


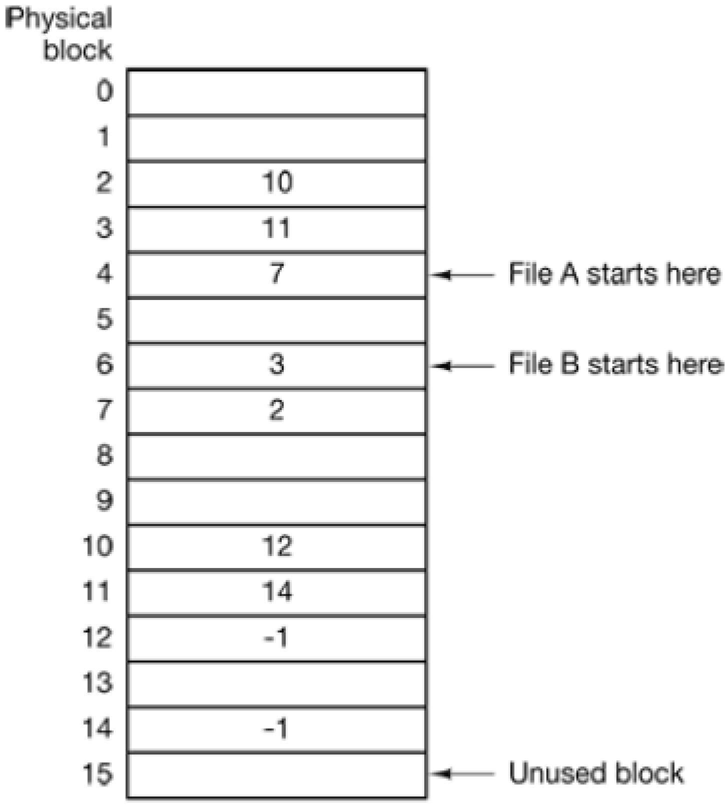
**Figure 6-13.** Storing a file as a linked list of disk blocks.

**Physic al block**

- No space is lost to disk fragmentation. It is sufficient for the directory entry to merely store the disk address of the first block.
- The rest can be found starting there. Reading a file sequentially is straightforward, random access is extremely slow.
- The amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes.
- With the first few bytes of each block occupied to a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks.

**Linked List Allocation Using a Table in Memory**

- Both disadvantages of the linked list allocation can be eliminated by 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.
- Start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6.
- Both chains are terminated with a special marker that is not a valid block number. Such a table in main memory is called a **FAT (File Allocation Table)**



| Physical block | | |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | 10 | |
| 3 | 11 | |
| 4 | 7 | ← File A starts here |
| 5 | | |
| 6 | 3 | ← File B starts here |
| 7 | 2 | |
| 8 | | |
| 9 | | |
| 10 | 12 | |
| 11 | 14 | |
| 12 | -1 | |
| 13 | | |
| 14 | -1 | |
| 15 | | ← Unused block |

- The entire block is available for data. The chain must still be followed to find a given offset within the file; the chain is in memory, so it can be followed without making any disk references.
- **Disadvantage:**
  The entire table must be in memory all the time to make it work.
  The table will take up 600 MB or 800 MB of main memory all the time, depending on whether the system is optimized for space or time.

**I-nodes**

- Keeping track of which blocks belong to which file is to associate with each file a data structure called an i-node (**Index-node**), which lists the attributes and disk addresses of the file's blocks.
- The i-node, it is then possible to find all the blocks of the file.

**Advantage:**

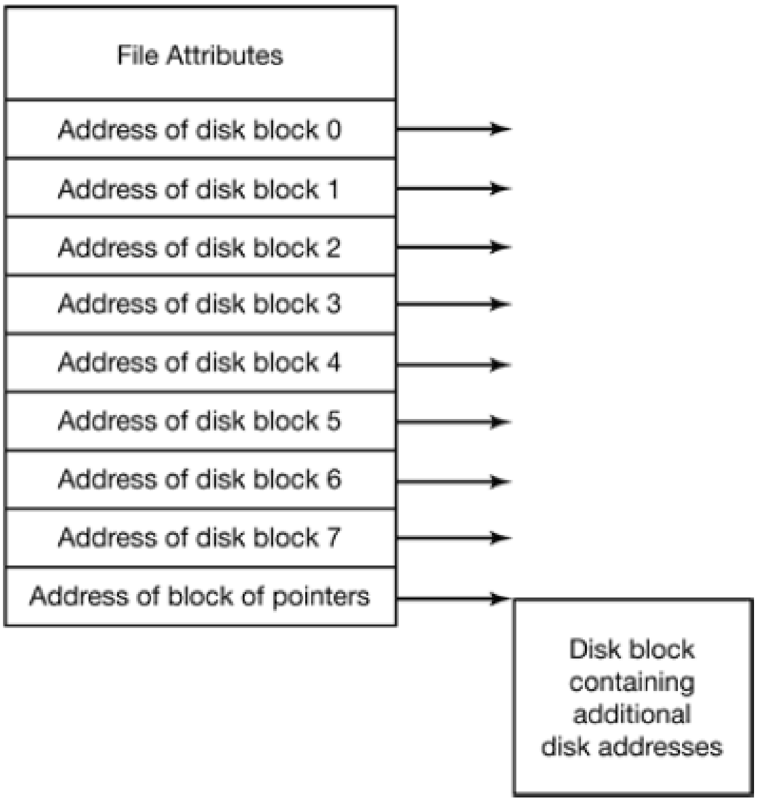
Over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open.

If each i-node occupies ***n*** **bytes** and a maximum of ***k*** **files** may be open at once, the total memory occupied is only ***kn*** **bytes**.

The table for holding the linked list of all disk blocks is proportional in size to the disk itself.

If the disk has ***n*** blocks, the table needs ***n*** entries. As disk grow larger, this table grows linearly with them.

In contrast, the i-node scheme requires an array in memory whose size is proportional o the maximum number of files that may be open at once.



ure 6-15. An example i-node.

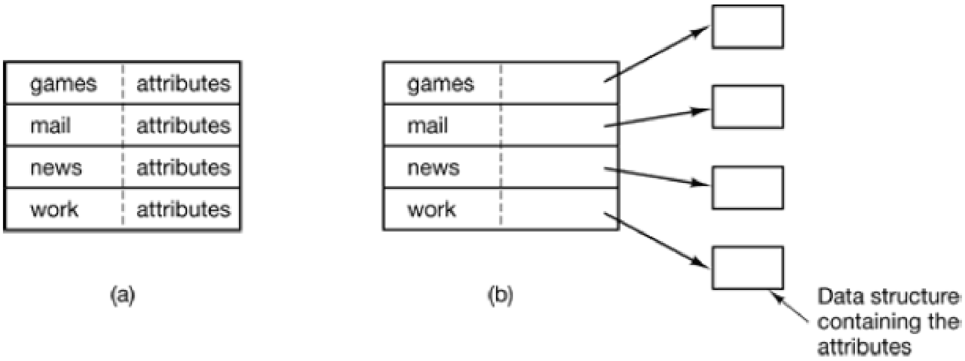
**Problem:**

If each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit?

**Solution:** reserve the last disk address not for a data block, but instead for the address of a block containing more disks block address.

➢ **Implementing Directories**

- To read a file it must be opened. When a file opened, the OS 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.
- Every file system maintains file attributes, such as each file's owner and creation time and they must be stored somewhere.
- Simple design, a directory consists of a list of fixed-size entries, one per file, containing a file name, a structure of the file attributes, one or more disk addresses telling where the disk blocks are.



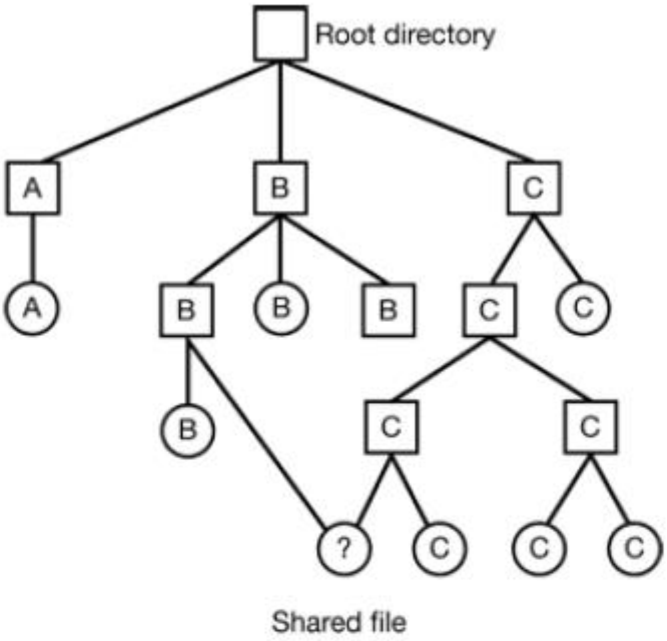
(a)A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry.(b) A directory in which each just refers to an i-node.

- For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries.
- Disadvantage:
  1. When a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit.
  2. A single directory entry may span multiple pages, so a page fault may occur while reading a file name.
- To handle variable-length names is to make the directory entries themselves all fixed length and keep the file name together in a heap at the end of the directory.

**Advantage:** when an entry is removed, the next file entered will always fit there.

- **Shared Files**
  - When several users are working together on a project, they often need to share files.
  - It is often convenient for a shared file to appear simultaneously in different directories belonging to different users.

**File system containing a shared file**



Ex: only with one of C’s files now present in one of B’s directories s well.

The connection between B’s directory and the share file is called a link.

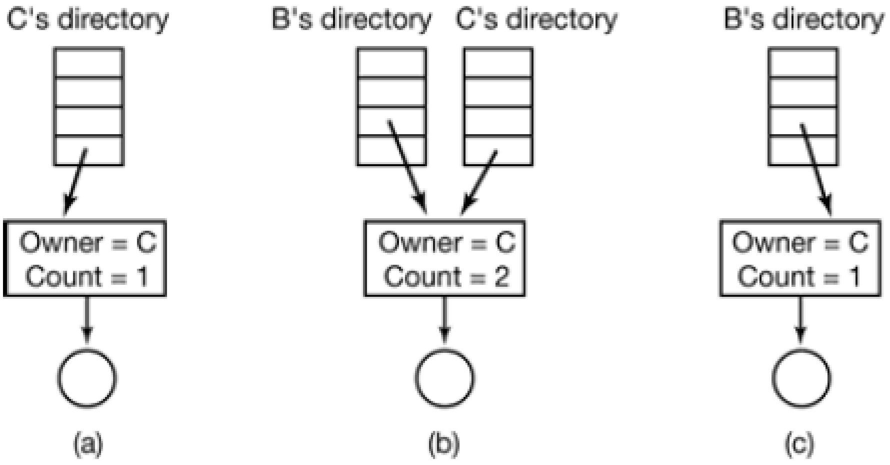
The file system itself is now a **Directed Acyclic Graph or DAG.**

- Sharing files is convenient, but it also introduces some problems.
- 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.

**Solved in two ways:**

1. 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.

2. 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 OS that the file begins read from is of type LINK. This approach is called **Symbolic linking.**
3. Creating a link does not change the ownership, but it does increase the link count in the i-node, so the system knows how many directory entries currently point to the file.



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

- The problem with symbolic links is the extra overhead required. The file containing the path must be read, then the path must be parsed and followed ,component by component ,until the i-node is reached
- All of this activity may require a considerable number of extra disk accesses
- Furthermore an extra i-node is needed for symbolic link as is an extra disk block to store the path name is short the system could store it in the i-node itself as a kind of optimization
- Symbolic links have the advantages that they can be used to link to files on machines anywhere in the world by simply providing the network address of the machine where the file resides in addition to its path on machine
- When links are allowed files can have two or more paths
- Programs that start at a given directory and find all the files in that directory and its subdirectories will locate a linked file multiple times

**Log-Structured File System**

- Changes in technology are putting pressure current file system
- In particular CPUs keep getting faster disks are becoming much bigger and cheaper and memories are growing exponentially in size
- File system, LFS (the log-structured file system)
- LFS design is as CPUs get fast and RAM memories get larger disk caches are also increasing rapidly
- It is now possible to satisfy a very substantial fraction of all read requests directly from the file system cache with no disk access needed
- The LFS designers re-implement the UNIX file system in such way as to achieve the full bandwidth of the disk even in the face of a workload consisting in large part of small random writes
- All writes are initially buffered in memory and periodically all the buffered writes are written to the disk in a single segment at the end of the log
- Opening a new file now consists of using the map to locate the i-node for the file

- Once the i-node has been located the addresses of the blocks can be found from it
- All of the blocks will themselves be in segments somewhere in the log
- If disks were infinitely large disks finite if a file is overwritten its i-node will now point to the new blocks but the old ones will still be occupying space in previously written segments
- LFS has a cleaner thread that spends its time scanning the log circularly to compact it
- It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there
- It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use
- If not that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment
- The original segment is then marked as free so that the log can use it for new data

**Journaling File System**

- The basic idea here to keep a log of what the file system is going to do before it does it so that if the system crashes before it can do its planned work upon rebooting the system can look in the log to see what was going on at the time of the crash and finish the job .such file system, called journaling file systems are actually in use.
- This operation (in UNIX)requires three steps:
  1. Remove the file from its directory.
  2. Release all the disk blocks to the pool of free disk blocks
  3. Return all the disk blocks to the pool of free disk blocks
- In the first step is completed and then the system crashes.
- The i-node and file blocks will not be accessible from any file, but will also not be available for reassignment they are just off in limbo somewhere decreasing the available resources if the crash occurs after the second step only the blocks are lost.
- The log entry is then written to disk.
- Only after the log entry has been written do the various operations begin after the operations compete successfully the log entry is erased.
- To make journaling work the logged operations must be idempotent which means they can be repeated as often as necessary without harm operations such as "update the bitmap to mark i-node k or block n as free" can be repeated until the cows come home with no danger.
- Journaling file systems have to arrange their data structures and loggable operations so they all of them are idempotent .under these conditions, crash recovery can be made fast and secure.
- For added reliability, a file system can introduce the database concept of an atomic transaction. When this concept is used, a group of actions can be bracketed by the begin transaction and end transaction operations.
- NTFS has a journaling system and its structure is rarely corrupted by system crashes.

# QUESTION BANK

## UNIT - I

**2 MARKS:**

1. What is mean by operating system?
2. Define Multiprogramming.
3. Define Spooling.
4. What is Time Sharing system?
5. Define Multitasking.
6. Define Processes.
7. What is mean by alarm signal?
8. What is mean by Deadlock?
9. Define file.
10. Define pipe.
11. What is mean by shell?
12. Define system call.
13. Define FORK().
14. Define LSEEK().
15. Define mount.

**5 & 10 MARKS:**

1.Explain about history of OS.

2.Discuss about type of OS.

3.Write short notes on files.

4.Explain about system call.

5.Discuss about structure of OS.

# UNIT - II

**2 MARKS:**

1. Define process.
2. Define threads.
3. How to create process?
4. Listout the condition for process termination.
5. Listout the process stated.
6. Define IPC.
7. What is mean by race conditions?
8. What is mean by spooler directory?
9. Define critical regions.
10. Expansion TSL.
11. Define bounded buffer problem.
12. What is mean by monitor?
13. How to pass message?

**5 & 10 MARKS:**

1.Expalin about process.
2.Discuss about process states.
3.Write short notes on threads.
4.Explain about IPC.
5.Discuss about producer and consumer problem.
6.Explain about semaphores.
7.Write short notes mutexes.
8.Discuss about message passing & Barriers.

# UNIT - III

**2 MARKS:**

1.Define scheduling.
2.What is mean by scheduler?
3.List out the categories of scheduling algorithm.
4.Write scheduling goals.
5.Expansion FCS & SRT.
6.Define admission scheduler.
7.What is mean by priority scheduling?
8.Define lottery scheduling.
9.what is mean by address space?
10.Define swapping.
11.What is mean by bitmaps?
12.Define paging.
13.Define pages.
14.What is mean page frame?
15.Define TLB.

**5 & 10 MARKS:**

1.Explain about scheduling.
2.Discuss about scheduling in batch system.
3.Write short notes on three level scheduling.
4.Discuss about scheduling in interactive system.
5.Explain about virtual memory.
6.Write short notes on page replacement algorithms.

## UNIT - IV

**2 MARKS:**

1.Define deadlock.
2.List out the resources types in deadlock.
3.What are all the events required to use resource?
4.List out the conditions for deadlock.
5.What are all the nodes used for deadlock?
6.How to recovery the deadlock?
7.Define multiple processor system.
8.Define multicomputers.
9.Expansion UMA & NUMA.
10.What is mean by cross points?
11.Define scheduling.
12.What is mean by timesharing system?
13.Define gang scheduling.
14.Listout the types of topologies.
15.Define RPC.
16.How to share the files?

**5 & 10 MARKS:**

1.Expalin about deadlock.
2.Discuss in detail about conditions for deadlock.
3.How to recovery from deadlock.
4.Explain about bankers algorithm.
5.Discuss about multicomputers.
6.Write short notes on gang scheduling.
7.Explain about space sharing.
8.Discuss about topologies.
9.Explain about RPC.

## UNIT - V

**2 MARKS:**

1.What is mean by device controllers?
2.Define DMA.
3.What is mean by Interrupts revisited **?**
4.List out the types of Interrupts.
5.Define File.
6.How to naming file?
7.List out the file extension.
8.List out the types of files.
9.What is mean by file attributes?
10.Define metadata.
11.List the file operation.
12.Define directories.
13.List the types of directories.
14.What is mean by path name?
15.List the directory operations.

**5 & 10 MARKS:**

1.Explain about principles of hardware.
2.Write short notes on memory mapped I/O.
3.Explain about DMA.
4.Discuss about principles of software.
5. Write short notes on programmed I/O.
6.Explain about file system.
7.Discuss about file structure.
8.Explain about file operations.
9.Write short notes on directories.