(1) What is operating System? Explain the abstract view of the components of a computer system.

- An operating system (OS) is a collection of software that manages computer hardware resources and provides various services for computer programs. It acts as an intermediary between the user of a computer and the computer hardware.

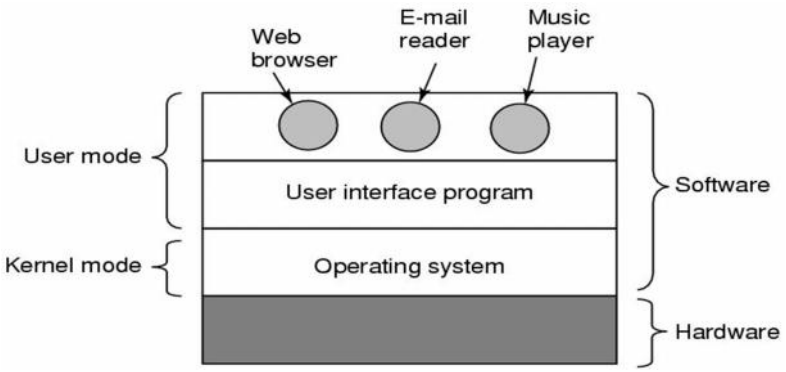


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

- The placement of the operating system is shown in Fig. 1-1. At the bottom is the hardware, which, consists of integrated circuit chips, wires, disks, a key board, a monitor and similar physical devices.
- On the top of the hardware is the software.
- Operating system runs on the bare hardware and it provides base for the rest of the software.
- Most computers have two modes of operation: kernel mode and user mode.
- The operating system is the most fundamental piece of software and runs in kernel mode.
- In this mode it has complete access to all the hardware and can execute any instruction that the machine is capable of executing.
- The rest of the software runs in user mode, in which only a subset of the machine instructions is available. Here we find the command interpreter (shell), compilers, editors, and other system programs.
- Finally, above the system programs are the application programs. These programs are purchased or written by the users to solve their particular problems, such as word processing, spreadsheets, web browser or music player.

- To hide complexity of hardware, an operating system is provided. It consists of a layer of software that (partially) hides the hardware and gives the programmer a more convenient set of instructions to work with.

(2) Give the view of OS as an extended machine.

Operating systems perform two basically unrelated functions: providing a clean abstract set of resources instead of the messy hardware to application programmers and managing these hardware resources.

Operating System as an Extended Machine

- The architecture (instruction set, memory, I/O, and bus structure) of most computers at the machine level language is primitive and awkward to program, especially for input / output operations.
- Users do not want to be involved in programming of storage devices.
- Operating System provides a simple, high level abstraction such that these devices contain a collection of named files.
- Such files consist of useful piece of information like a digital photo, e mail messages, or web page.
- Operating System provides a set of basic commands or instructions to perform various operations such as read, write, modify, save or close.
- Dealing with them is easier than directly dealing with hardware.
- Thus, Operating System hides the complexity of hardware and presents a beautiful interface to the users.

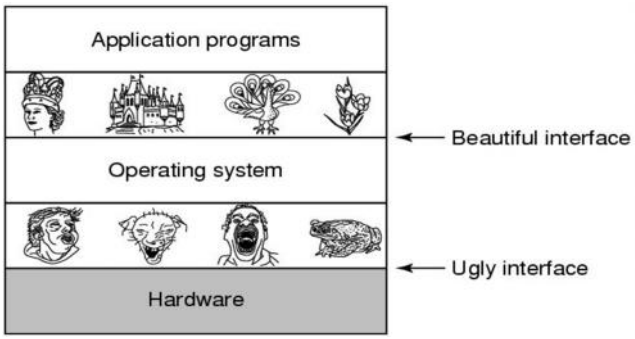


Figure 1-2. Operating Systems turn ugly hardware into beautiful abstractions.

- Just as the operating system shields (protect from an unpleasant experience) the programmer from the disk hardware and presents a simple file-oriented interface, it

also conceals a lot of unpleasant business concerning interrupts, timers, memory management, and other low level features.

- In each case, the abstraction offered by the operating system is simpler and easier to use than that offered by the underlying hardware.
- In this view, 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 work with than the underlying hardware.
- The operating system provides a variety of services that programs can obtain using special instructions called system calls.

(3) Give the view of OS as a Resource Manager.

- The concept of an operating system as providing abstractions to application programs is a top down view.
- Alternatively, bottom up view holds that the OS is there to manage all pieces of a complex system.
- A computer consists of a set of resources such as processors, memories, timers, disks, printers and many others.
- The Operating System manages these resources and allocates them to specific programs.
- As a resource manager, Operating system provides controlled allocation of the processors, memories, I/O devices among various programs.
- Multiple user programs are running at the same time.
- The processor itself is a resource and the Operating System decides how much processor time should be given for the execution of a particular user program.
- Operating system also manages memory and I/O devices when multiple users are working.
- The primary task of OS is to keep the track of which programs are using which resources, to grant resource requests, to account for usage, and to resolve conflicting requests from different programs and users.
- An Operating System is a control program. A control program controls the execution of user programs to prevent errors and improper use of computer.
- 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, CPU and printer are time multiplexed resources. OS decides who will use it and for how long.
- The other kind of multiplexing is space multiplexing, instead of the customers taking turns, each one gets part of the resource.
- For example, both primary and secondary memories are space multiplexed. OS allocates them to user programs and keeps the track of it.

(4) Explain different types of tasks done by OS. OR Write different services provided by operating system.

- Operating system services and facilities can be grouped into following areas:

Program development

- Operating system provides editors and debuggers to assist (help) the programmer in creating programs.
- Usually these services are in the form of utility programs and not strictly part of core operating system. They are supplied with operating system and referred as application program development tools.

Program execution

- A number of tasks need to be performed to execute a program, such as instructions and data must be loaded into main memory. I/O devices and files must be initialized.
- The operating system handles these scheduling duties for the user.

Access to I/O devices

- Each I/O devices requires its own set of instruction for operations.
- Operating system provides a uniform interface that hides these details, so the programmer can access such devices using simple reads and writes.

Memory Management

- Operating System manages memory hierarchy.
- It keeps the track of which parts of memory are in use and free memory.
- It allocates the memory to programs when they need it.
- It de-allocates the memory when programs finish execution.

Controlled access to file

- In the case of file access, operating system provides a directory hierarchy for easy access and management of files.
- OS provides various file handling commands using which users can easily read, write, and modify files.
- In case of system with multiple users, the operating system may provide protection mechanism to control access to file.

System access

- In case of public systems, the operating system controls access to the system as a whole.
- The access function must provide protection of resources and data from unauthorized users.

Error detection and response

- Various types of errors can occur while a computer system is running, which includes internal and external hardware errors. For example, memory error, device failure error and software errors as arithmetic overflow.
- In case, operating system must provide a response that clears error condition with least impact on running applications.

Accounting

- A good operating system collects usage for various resources and monitor performance parameters.
- On any system, this information is useful in anticipating need for future enhancements.

Protection & Security

- Operating systems provides various options for protection and security purpose.
- It allows the users to secure files from unwanted usage.
- It protects restricted memory area from unauthorized access.
- Protection involves ensuring that all access to system resources is controlled.

(5) Give the features of Batch Operating System.

- Batch operating system is one that processes routine jobs without any interactive user presents. Such as claim processing in insurance and sales reporting etc.

- To improve utilization, the concept of batch operating system was developed.
- Jobs with similar needs were batched together and were run through the computer as a group.
- Thus, the programmer would leave their program with operator, who in turn would sort program into batches with similar requirements.
- The operator then loaded a special program (the ancestor of today's operating system), which read the first job from magnetic tape and run it.
- The output was written onto a second magnetic tape, instead of being printed.
- After each job finished, the operating system automatically read the next job from the tape and began running it.
- When the whole batch was done, the operator removed the input and output tapes, replaced the input tape with the next batch, and brought the output tape for offline printing.
- With the use of this type of operating system, the user no longer has direct access to machine.
- Advantages:
  - Move much of the work of the operator to the computer.
  - Increase performance since it was possible for job to start as soon as the previous job finished.
- Disadvantages:
  - Large Turnaround time.
  - More difficult to debug program.
  - Due to lack of protection scheme one batch job can affect pending jobs.

(6) Explain the features of Time Sharing System.

- Time Sharing is a logical extension of multiprogramming.
- Multiple jobs are executed simultaneously by switching the CPU back and forth among them.
- The switching occurs so frequently (speedy) that the users cannot identify the presence of other users or programs.
- Users can interact with his program while it is running in timesharing mode.
- Processor's time is shared among multiple users. An interactive or hands on computer system provides online communication between the user and the system.
- A time shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time shared computer. Each user has at least one

separate program in memory.

- A time shared operating system allows many users to share computer simultaneously. Since each action or command in a time shared system tends to be short, only a little CPU time is needed for each user.
- Advantages :-
  - Easy to use
  - User friendly
  - Quick response time
- Disadvantages:-
  - If any problem affects the OS, you may lose all the contents which have stored already.
  - Unwanted user can use your own system in case if proper security options are not available.

(7) Explain the features of Real Time Operating System.

- A real time operating system is used, when there are rigid (strict) time requirements on the operation of a processor or the flow of data.
- It is often used as a control device in a dedicated application. Systems that control scientific experiments, medical imaging systems, and industrial control system are real time systems. These applications also include some home appliance system, weapon systems, and automobile engine fuel injection systems.
- Real time Operating System has well defined, fixed time constraints. Processing must be done within defined constraints or the system will fail.
- Since meeting strict deadlines is crucial in real time systems, sometimes an operating is simply a library linked in with the application programs.
- There are two types of real time operating system,
  - Hard real system:
    - ✓ This system guarantees that critical tasks complete on time.
    - ✓ Many of these are found in industrial process control, avionics, and military and similar application areas.
    - ✓ This goal says that all delays in the system must be restricted.
  - Soft real system:
    - ✓ In soft real-time system, missing an occasional deadline, while not desirable, is acceptable and does not cause any permanent damage.

- ✓ Digital audio or multimedia systems fall in this category.

- An example of real time system is e-Cos.

(8) Explain different types of OS.

Mainframe Operating Systems

- The operating system found in those room sized computers which are still found in major corporate data centers. These computers differ from personal computers in terms of their I/O capacity.
- They typically offer three kinds of services: batch, transaction processing, and timesharing.
- **Batch operating system** is one that processes routine jobs without any interactive user presents, such as claim processing in an insurance and sales reporting etc.
- **Transaction processing** system handles large numbers of small requests, for example check processing at a bank and airline reservation.
- **Time sharing** allows multiple remote users to run jobs on the computer at once, such as querying a database.
- An example mainframe operating system is OS/390 and a descendant of OS/360.

Server Operating Systems

- They run on servers, which are very large personal computers, workstations, or even mainframes.
- They serve multiple users at once over a network and allow the users to share hardware and software resources.
- Servers can provide print service, file service or web service.
- Typically server operating systems are Solaris, FreeBSD, and Linux and Windows Server 200x.

Multiprocessor Operating Systems

- An increasingly common way to get major group computing power is to connect multiple CPUs into a single system. Depending on precisely how they are connected and what is shared, these systems are called parallel computers, multicomputers, or multiprocessors. The operating systems used in this system are multiprocessor operating system.
- They need special operating systems, but often these are variations on the server

OS with special features for communication, connectivity and consistency.

- Multiprocessor operating systems includes Windows and Linux, run on multiprocessors.

Personal Computer Operating Systems

- The next category is the personal computer operating system. All Modern computers support multiprogramming, often with more than one programs started up at boot time. Their job is to provide good support to a single user.
- They are widely used for word processing, spreadsheets and Internet access.
- Common examples of personal computer operating system are Linux, FreeBSD, Windows Vista, and Macintosh operating system.

Handhelds Computer Operating Systems

- Continuing on down to smaller and smaller systems, we come to handheld computers. A handheld computer or PDA (personal digital assistant) is a small computer that fits in a pocket and performs a small number of functions, such as electronics address book and memo pad.
- The OS that runs on handhelds are increasingly sophisticated with the ability to handle telephony, photography and other functions.
- One major difference between handhelds and personal computer OS is that the former do not have multi gigabyte hard disks.
- Two of the most popular operating systems for handhelds are Symbian OS and Palm OS.

Embedded Operating Systems

- Embedded systems run on the computers that control devices that are not generally thought of as computers and which do not accept user installed software.
- The main property which distinguishes embedded systems from handhelds is the certainty that no untrusted software will ever run on it.
- So, there is no need for protections between applications, leading to some simplifications.
- Systems such as QNX and VxWorks are popular embedded operating system.

Sensor Node Operating Systems

- Networks of tiny sensor nodes are being deployed (developed) for numerous purposes. These nodes are tiny computers that communicate with each other and with a base station using wireless communication.

- These sensor networks are used to protect the perimeters of buildings, guard national borders, detect fires in forests, measure temperature and precipitation for weather forecasting, glean information about enemy movements on battlefields, and much more.
- Each sensor node is a real computer, with a CPU, RAM, ROM and one or more environmental sensors.
- It runs a small but real operating system, usually one that is event driven, responding to external events or making measurements periodically based on internal clock.
- All the programs are loaded in advance which makes the design much simpler.
- TinyOS is a well-known operating system for a sensor node.

Real Time Operating Systems

- These systems are characterized by having time as a key parameter.
- Real time operating system has well defined, fixed time constraints. Processing must be done within define constraints or the system will fail.
- Types of Real Time Operating System:
  - ✓ Hard real time system
    - o Many of these are found in industrial process control, avionics, and military and similar application areas.
    - o These systems must provide absolute guarantees that a certain action will occur be a certain time.
  - ✓ Soft real time system
    - o Missing an occasional deadline, while not desirable is acceptable and does not cause any permanent damage.
    - o Digital audio, digital telephone and multimedia systems fall into this category.
- An example of real time system is e-Cos.

Smart Card Operating Systems

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

others can handle multiple functions on the same card.

- Often these are proprietary systems.

(9) Explain different types of operating system structure. OR Explain architectures of different operating system structure.

Monolithic system

- In this approach the entire operating system runs as a single program in kernel mode.
- The operating system is written as a collection of procedures, linked together into a single large executable binary program.
- When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.
- To construct the actual object program of the operating system, when this approach is used, one first compiles all the individual procedure and then binds (group) them all together into a single executable file using the system linker.
- The services (system calls) provided by the operating system are requested by putting the parameters in a well-defined place (e.g., on the stack) and then executing a trap instruction.
- This instruction switches the machine from user mode to kernel mode and transfers control to the operating system.
- The operating system then fetches the parameters and determines which system call is to be carried out.
- This organization suggests a basic structure for the operating system.
  - ✓ A main program that invoke (call up) the requested service procedure.
  - ✓ A set of service procedures that carry out the system calls.
  - ✓ A set of utility procedures that help the service procedure.
- In this model, for each system call there is one service procedure that takes care of it and executes it.
- The utility procedures do things that are needed by several services procedure, such as fetching data from user programs.
- This division of the procedure into three layers is shown in figure 1-3

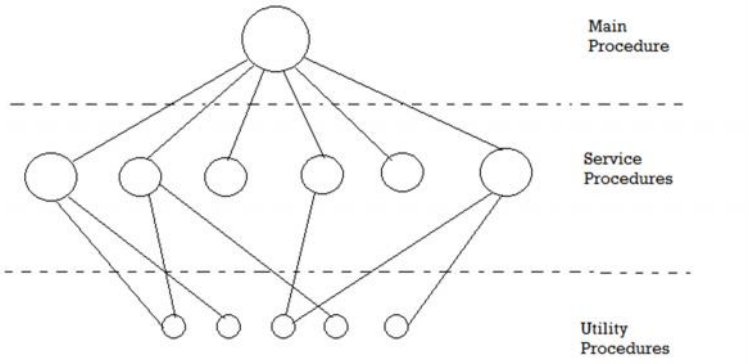


Figure 1-3. A simple structuring model for a monolithic system.

Layered system

- In this system, operating system is organized as a hierarchy of layers, each one constructed upon the one below it as shown below in figure 1-4.

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

Figure 1-4. Structure of THE operating system.

- The first system constructed in this way was the THE system.
- The system had six layers.
- Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired.
- Layer 0 provided the basic multiprogramming of the CPU.
- Layer 1 did the memory management. It allocated space for process in main memory and on a 512K word drum used for holding parts of processes for which there was no room in main memory.
- Layer 2 handled communication between each process and the operator console (i.e. user).
- Layer 3 takes care of managing the I/O devices and buffering the information

streams to and from them.

- Layer 4 was where the user programs were found.
- The system operator process was located in layer 5.
- A further generalization of the layering concept was present in the MULTICS system.
- Instead of layers, 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.
- Although the entire operating system was part of the address space of each user process in MULTICS, the hardware made it possible to designate individual procedures (memory segments, actually) as protected against reading, writing, or executing.

Microkernel

- With the layered approach, the designers have a choice where to draw the kernel user boundary.
- Traditionally, all the layers went in the kernel, but that is not necessary.
- In fact, a strong case can be made for putting as little as possible in kernel mode because bugs in the kernel can bring down the system instantly.
- In contrast, user processes can be set up to have less power so that a bug may not be fatal.
- The basic idea behind the microkernel design is to achieve high reliability by splitting the operating system up to into small, well defined modules, only one of which the microkernel runs in kernels mode and the rest of all are powerless user processes which would run in user mode.
- By running each device driver and file system as separate user processes, a bug in one of these can crash that component but cannot crash the entire system.
- Examples of microkernel are Integrity, K42, L4, PikeOS, QNX, Symbian, and MINIX 3.
- MINIX 3 microkernel is only 3200 lines of C code and 800 lines of assembler for low level functions such as catching interrupts and switching processes.
- The C code manages and schedules processes, handles inter-process communication and offer a set of about 35 systems calls to the rest of OS to do its work.

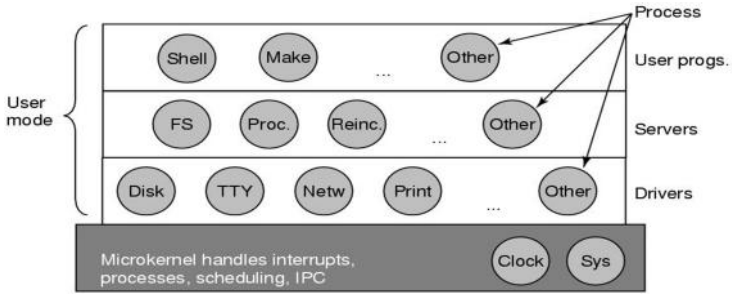


Figure 1-5. Structure of MINIX 3 system.

- The process structure of MINIX 3 is shown in figure 1-5, with kernel call handler labeled as *Sys*.
- The device driver for the clock is also in the kernel because the scheduler interacts closely with it. All the other device drivers run as separate user processes.
- Outside the kernel, the system is structured as three layers of processes all running in user mode.
- The lowest layer contains the device driver. Since they run in user mode they do not have access to the I/O port space and cannot issue I/O commands directly.
- Above driver is another user mode layer containing servers, which do most of the work of an operating system.
- One interesting server is the **reincarnation server**, whose job is to check if the other servers and drivers are functioning correctly. In the event that a faulty one is detected, it is automatically replaced without any user intervention.
- All the user programs lie on the top layer.

Client Server Model

- A slight variation of the microkernel idea is to distinguish classes of processes in two categories.
- First one is the servers, each of which provides some services, and the second one is clients, which use these services.
- This model is known as the Client Server model.
- Communication between clients and servers is done by message passing.
- To obtain a service, a client process constructs a message saying what it wants and sends it to the appropriate services.
- The service then does the work and sends back the answer.
- The generalization of this idea is to have the clients and servers run on different

computers, connected by a local or wide area network.

- Since a client communicates with a server by sending 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.
- A PC sends a request for a Web page to the server and the Web page comes back. This is a typical use of client server model in a network.

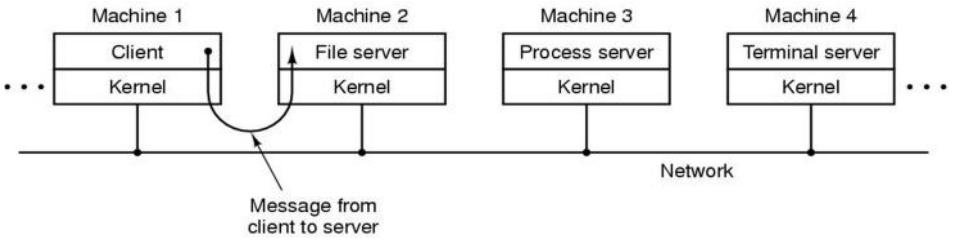


Figure 1-6. The client server model over a network.

Virtual Machine

- The initial releases of OS/360 were strictly batch systems. But many users wanted to be able to work interactively at a terminal, so OS designers decided to write timesharing systems for it.

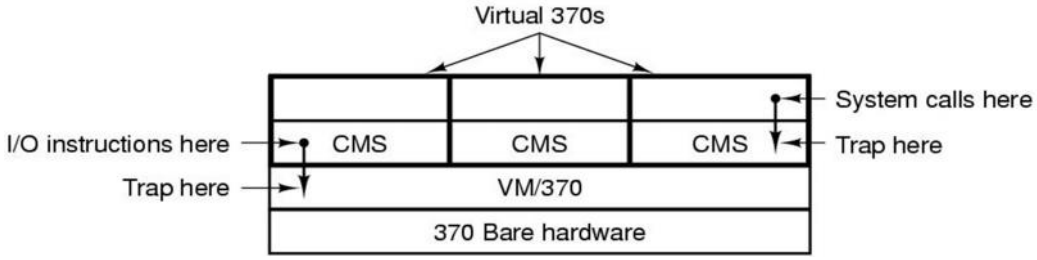


Figure 1-7. The structure of VM/370 with CMS.

- The heart of the system, known as the virtual machine monitor, run on the bare hardware and does the multiprogramming, providing not just one but several virtual machines to the next layer up.
- Each virtual machine is identical to the true hardware; each one can run any OS that will run directly on the bare hardware.
- Different virtual machines can run different operating systems.
- On VM/370, some run OS/360 while the others run single user interactive system

called CMS (Conversational Monitor System) for interactive time sharing users.

- When CMS program executed a system call, a call was trapped to the operating system in its own virtual machine, not on VM/370. CMS then issued the normal hardware I/O instruction for reading its virtual disk or whatever was needed to carry out the call.
- These I/O instructions were trapped by VM/370 which then performs them.
- The idea of a virtual machine is heavily used nowadays in a different context.
- An area where virtual machines are used, but in a somewhat different way, is for running Java programs.
- When Sun Microsystems invented the Java programming language, it also invented a virtual machine (i.e., a computer architecture) called the **JVM (Java Virtual Machine).**
- The Java compiler produces code for JVM, which then typically is executed by a software JVM interpreter.
- 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.

Virtual Machines Rediscovered

- Many huge companies have considered virtualization as a way to run their mail servers, Web servers, FTP servers and other servers on the same machine without having a crash of one server bring down the rest.
- Virtualization is also popular in Web hosting world.
- Web hosting company offers virtual machines for rent, where a single physical machine can run many virtual machines; each one appears to be a complete machine.
- Customers who rent a virtual machine can run any OS or software they want to but at a fraction of the cost of dedicated server.
- Another use of virtualization is for end users who want to be able to run two or more operating systems at the same time, say Windows and Linux, because some of their favorite application packages run on one and some am on the other.
- This situation is illustrated in Fig. 1-8(a), where the term "virtual machine monitor" has been renamed type 1 **hypervisor** in recent years.
- VMware Workstation is a type 2 hypervisor, which is shown in Fig. 1-8(b). In contrast to type 1 hypervisors, which run on the bare metal, type 2 hypervisors run

as application programs on top of Windows, Linux, or some other operating system, known as the **host operating system.**

- After a type 2 hypervisor is started, it reads the installation CD-ROM for the chosen **guest operating system** and installs on a virtual disk, which is just a big file in the host operating system's file system.
- When the guest operating system is booted, it does the same thing it does on the actual hardware, typically starting up some background processes and then a GUI.
- Some hypervisors translate the binary programs of the guest operating system block by block, replacing certain control instructions with hypervisor calls.
- The translated blocks are then executed and cached for subsequent use.
- A different approach to handling control instructions is to modify the operating system to remove them. This approach is not true virtualization, but **para-virtualization.**

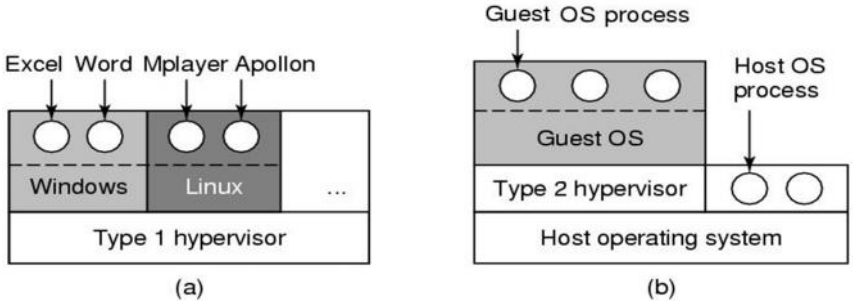


Figure 1-8. Type 1 & 2 Hypervisor.

Exokernels

- Rather than cloning (copying) the actual machine, as is done with virtual machines, another strategy is partitioning it.
- In other words, giving each user a subset of the resource.
- For example one virtual machine might get disk blocks 0 to 1023, the next one might get block 1024 to 2047, and so on.
- Program running at the bottom layer (kernel mode) called the exokernel. Its job is to allocate resources to virtual machines and then check attempt to use them to make sure no machine is trying to use somebody else's resources.
- The advantage of the exokernel scheme is that it saves a layer of mapping.
- In the other designs, each virtual machine thinks it has its own disk, with blocks running from 0 to some maximum, so the virtual machine monitor must maintain tables to remap disk addresses.
- In exokernel remapping is not needed. The exokernel need only keep track of which

virtual machine has been assigned which resource.

(10) What is a system call? How it is handled by an OS? OR Write a short note on system calls.

- The interface between the operating system and the user programs is defined by the set of system calls that the operating system provides.
- The system calls available in the interface vary from operating system to operating system.
- Any single-CPU computer can execute only one instruction at a time.
- If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap or system call instruction to transfer control to the operating system.
- The operating system then figures out what the calling process wants by inspecting the parameters.
- Then it carries out the system call and returns control to the instruction following the system call.

Following steps describe how a system call is handled by an operating system.

- To understand how OS handles system calls, let us take an example of read system call.
- Read system call has three parameters: the first one specifying the file, the second one pointing to the buffer, and the third one giving the number of bytes to read.
- Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: read.
- A call from a C program might look like this:

  count = read(fd, buffer, nbytes);

- The system call return the number of bytes actually read in count.
- This value is normally the same as nbytes, but may be smaller, if, for example, end-of-file is encountered while reading.
- If the system call cannot be carried out, either due to an invalid parameter or a disk error, count is set to -1, and the error number is put in a global variable, errno.
- Programs should always check the results of a system call to see if an error occurred.
- System calls are performed in a series of steps.
- To make this concept clearer, let us examine the read call discussed above.
- In preparation for calling the read library procedure, which actually makes the read system call, the calling program first pushes the parameters onto the stack, as shown in

steps 1-3 in Fig. 1-9.

- The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the address of the buffer (indicated by &) is passed, not the contents of the buffer.
- Then comes the actual call to the library procedure (step 4). This instruction is the normal procedure call instruction used to call all procedures.
- The library procedure, possibly written in assembly language, typically puts the system call number in a place where the operating system expects it, such as a register (step 5).

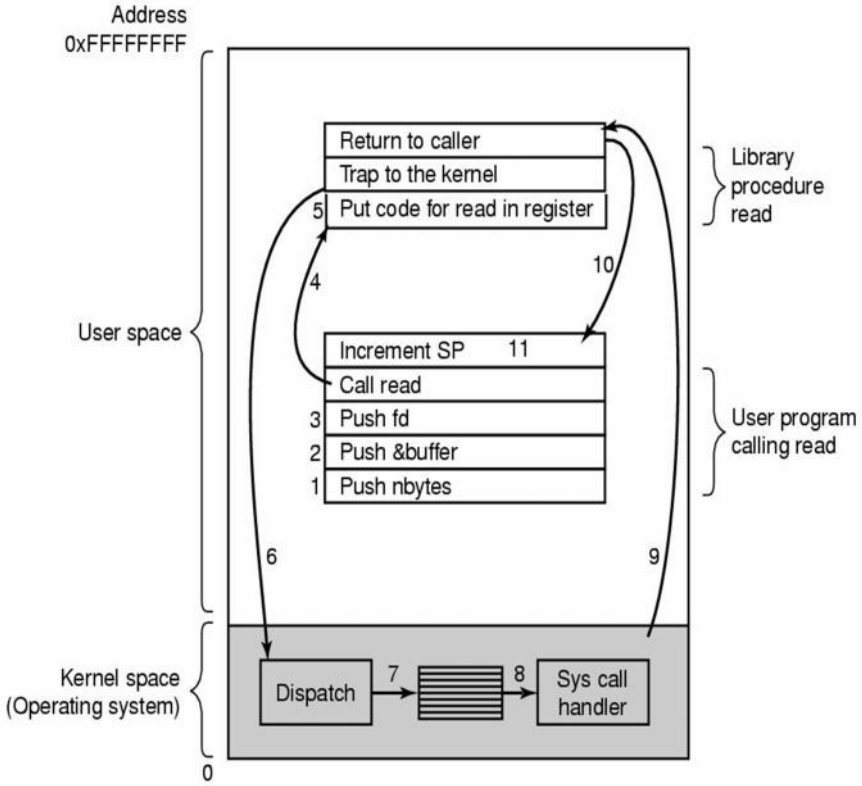


Figure 1-9. The 11 steps in making the system call read(fd, buffer, nbytes).

- Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6).
- The kernel code that starts examines the system call number and then dispatches to the correct system call handler, usually via a table of pointers to system call handlers indexed on system call number (step 7).
- At that point the system call handler runs (step 8).

Process management	
Call	**Description**
pid = for k( )	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
File management	
Call	**Description**
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(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 file's status information
Director 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
Miscellaneous	
Call	**Description**
s = chdir(dir name)	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

Table 1-1. Some of the major POSIX system calls.

- Once the system call handler has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction (step 9).
- This procedure then returns to the user program in the usual way procedure calls return (step 10).
- To finish the job, the user program has to clean up the stack, as it does after any procedure call (step 11).

(1) What is Process? Give the difference between Process and Program.

Process:

- Process is a program under execution.
- It is an instance of an executing program, including the current values of the program counter, registers & variables.
- Process is an abstraction of a running program.

Process	Program
A process is program in execution.	**A program is set of instructions.**
A process is an active/ dynamic entity.	A program is a passive/ static entity.
A process has a limited life span. It is created when execution starts and terminated as execution is finished.	A program has a longer life span. It is stored on disk forever.
A process contains various resources like memory address, disk, printer etc… as per requirements.	A program is stored on disk in some file. It does not contain any other resource.
A process contains memory address which is called address space.	A program requires memory space on disk to store all instructions.

(2) What is multiprogramming?

- A process is just an executing program, including the current values of the program counter, registers, and variables.
- Conceptually, each process has its own virtual CPU.
- In reality, the real CPU switches back and forth from process to process, but to understand the system, it is much easier to think about a collection of processes running in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program.
- This rapid switching back and forth is called multiprogramming and the number of processes loaded simultaneously in memory is called degree of multiprogramming.

(3) What is context switching?

- Switching the CPU to another process requires saving the state of the old process and

loading the saved state for the new process.

- This task is known as a context switch.
- The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state and memory-management information.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure overhead, because the system does no useful work while switching.
- Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions.

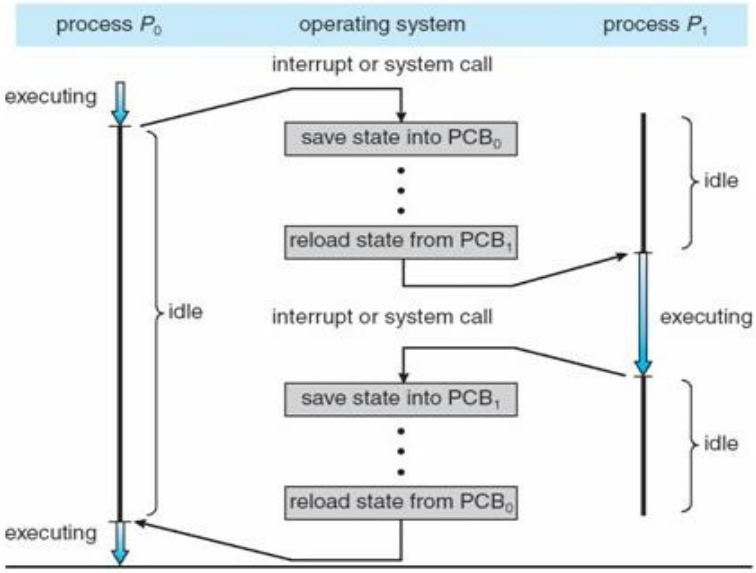


Figure 2-1. Context Switching

(4) Explain Process Model in brief.

- In this model, all the runnable software on the computer, sometimes including the operating system, is organized into a number of sequential processes.
- A process is just an executing program, including the current values of the program counter, registers, and variables.
- Conceptually, each process has its own virtual CPU.
- In reality, of course, the real CPU switches back and forth from process to process, but to understand the system, much easier to think about a collection of processes running

in (pseudo) parallel, than to try to keep track of how the CPU switches from program to program.

- This rapid switching back and forth is called multiprogramming.
- In Fig. 2-2 (a) we see a computer multiprogramming four programs in memory.
- In Fig. 2-2 (b) we see four processes, each with its own flow of control (i.e., its own logical program counter), and each one running independently of the other ones.
- There is only one physical program counter, so when each process runs, its logical program counter is loaded into the real program counter.
- When it is finished for the time being, the physical program counter is saved in the process' logical program counter in memory.
- In Fig. 2-2 (c) we see that over a long period of time interval, all the processes have made progress, but at any given instant only one process is actually running.
- With the CPU switching back and forth among the processes, the rate at which a process performs its computation will not be uniform and probably not even reproducible if the same processes are run again.
- Thus, processes must not be programmed with built-in assumptions about timing.

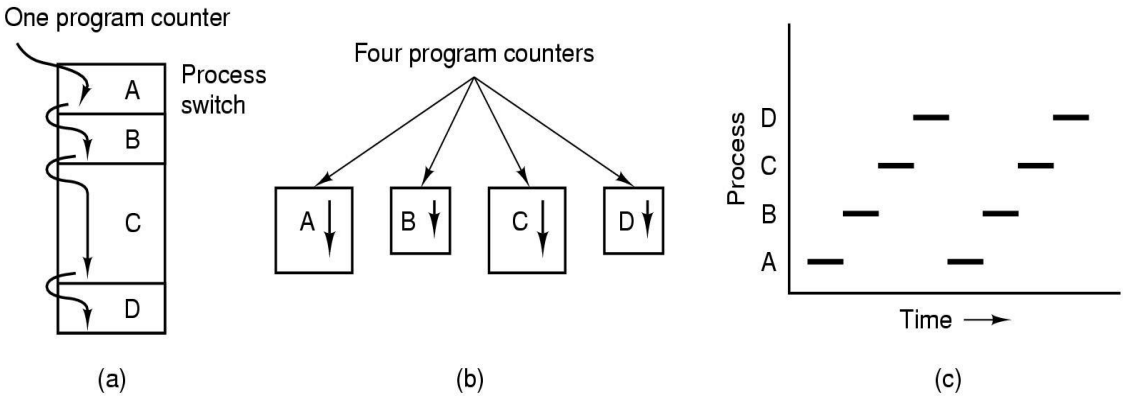


Figure 2-2. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Process Creation

- There are four principal events that cause processes to be created:
  1. System initialization.
  2. Execution of a process creation system call by a running process.
  3. A user request to create a new process.
  4. Initiation of a batch job.

Process Termination

- After a process has been created, it starts running and does whatever its job is.
- However, nothing lasts forever, not even processes. Sooner or later the new process will terminate, usually due to one of the following conditions:
  1. Normal exit (voluntary).
  2. Error exit (voluntary).
  3. Fatal error (involuntary).
  4. Killed by another process (involuntary).

Process Hierarchies

- In some systems, 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.
- An example of process hierarchy is in UNIX, which initializes itself when it is started.
- A special process, called **init**, is present in the boot image.
- When it starts running, it reads a file telling how many terminals there are.
- Then it forks off one new process per terminal. These processes wait for someone to log in.
- If a login is successful, the login process executes a shell to accept commands. These commands may start up more processes, and so forth.
- Thus, all the processes in the whole system belong to a single tree, with **init** at the root.
- In contrast, Windows does not have any concept of a process hierarchy. All processes are equal.
- The only identification for parent child process is that when a process is created, the parent is given a special token (called a **handle)** that it can use to control the child.
- However, it is free to pass this token to some other process, thus invalidating the hierarchy. Processes in UNIX cannot disinherit their children.

(5) What is process state? Explain state transition diagram. OR

What is process state? Explain different states of a process with various queue generated at each stage.

Process state:

- The state of a process is defined by the current activity of that process.
- During execution, process changes its state.

- The process can be in any one of the following three possible states.
    1) **Running** (actually using the CPU at that time and running).
    2) **Ready** (runnable; temporarily stopped to allow another process run).
    3) **Blocked** (unable to run until some external event happens).

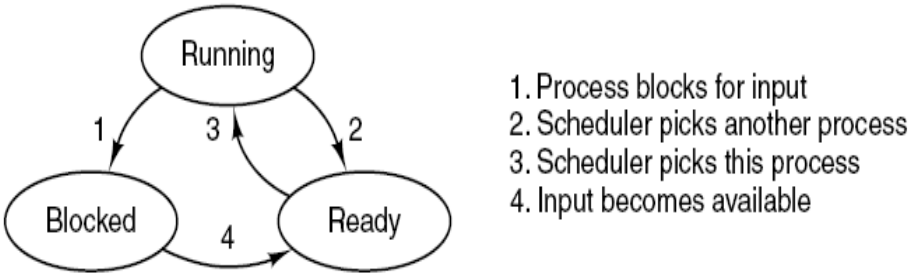


Figure 2-3. Process state transition diagram

- Figure 2-3 shows the state transition diagram.
- Logically, the first two states are similar. In both cases the process is willing to run, but in the ready state there is no CPU temporarily available for it.
- In blocked state, the process cannot run even if the CPU is available to it as the process is waiting for some external event to take place.
- There four possible transitions between these three states.
- Transition 1 occurs when the operating system discovers that a process cannot continue right now due to unavailability of input. In other systems including UNIX, when a process reads from a pipe or special file (e.g. terminal) and there is no input available, the process is automatically blocked.
- Transition 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 had their fair share and it is time for the first process to get the CPU to run again. The subject of scheduling, that is, deciding which process should run when and for how long, is an important one.
- Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that time, transition 3 will be triggered and the process will start running. Otherwise it may have to wait in ready state for a little time until the CPU is available and its turn comes.

Scheduling queues generated at each stage:

- As the process enters the system, it is kept into a job queue.
- The processes that are ready and are residing in main memory, waiting to be executed are kept in a ready queue.
- This queue is generally stored as a linked list.
- The list of processes waiting for a particular I/O device is called a device queue.
- Each device has its own queue. Figure 2-4 shows the queuing diagram of processes.
- In the figure 2-4 each rectangle represents a queue.
- There are following types of queues
  - ✓ **Job queue** : set of all processes in the system
  - ✓ **Ready queue** : set of processes ready and waiting for execution
  - ✓ **Set of device queues** : set of processes waiting for an I/O device
- The circle represents the resource which serves the queues and arrow indicates flow of processes.
- A new process is put in ready queue. It waits in the ready queue until it is selected for execution and is given the CPU.

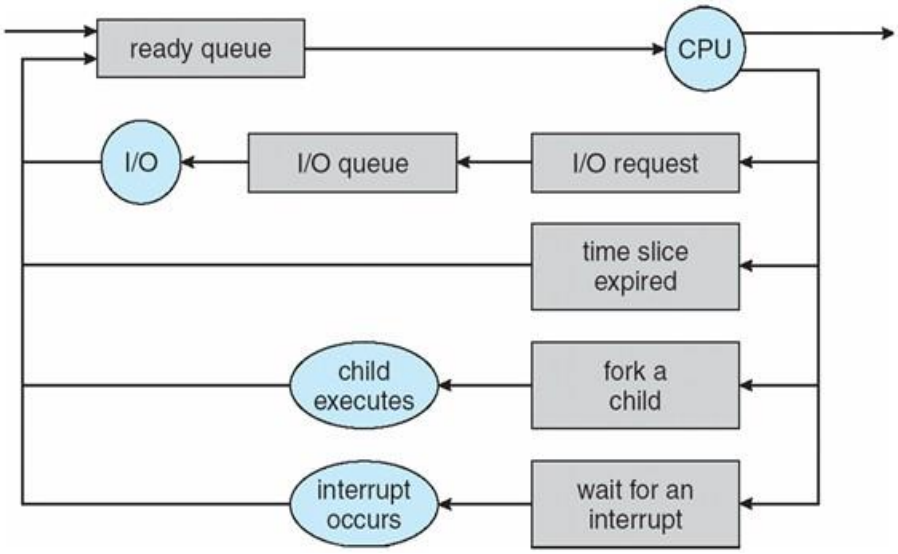


Figure 2-4. Queuing diagram representation of Process Scheduling

- Once the process starts execution one of the following events could occur.
  - ✓ The process issues an I/O request and then be placed in an I/O queue.
  - ✓ The process creates a new sub process and wait for its termination.
  - ✓ The process is removed forcibly from the CPU and is put back in ready queue.

- A process switches from waiting state to ready state and is put back in ready queue.
- The cycle continues unless the process terminates, at which the process is removed from all queues and has its PCB and resources de-allocated.

(6) Explain Process Control Block (PCB).

PCB (Process Control Block):

- To implement the process model, the operating system maintains a table (an array of structures) called the process table, with one entry per process, these entries are known as Process Control Block.
- Process table contains the information what the operating system must know to manage and control process switching, including the process location and process attributes.
- Each process contains some more information other than its address space. This information is stored in PCB as a collection of various fields. Operating system maintains the information to manage process.
- Various fields and information stored in PCB are given as below:
  - **Process Id:** Each process is given Id number at the time of creation.
  - **Process state:** The state may be ready, running, and blocked.
  - **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
  - **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
  - **CPU-scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
  - **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.
  - **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
  - **Status information:** The information includes the list of I/O devices allocated to this process, a list of open files, and so on.

Process management	Memory management	File management
Registers	Pointer to text segment	UMASK mask
Program counter	Pointer to data segment	Root directory
Program status word	Pointer to bss segment	Working directory
Stack pointer	Exit status	File descriptors
Process state	Signal status	Effective uid
Time when process started	Process id	Effective gid
CPU time used	Parent process	System call parameters
Children's CPU time	Process group	Various flag bits
Time of next alarm	Real uid	
Message queue pointers	Effective uid	
Pending signal bits	Real gid	
Process id	Effective gid	
Various flag bits	Bit maps for signals	
	Various flag bits	

Figure 2-5. Some of the fields of a typical process table entry.

- Figure 2-5 shows some of the more important fields in a typical system.
- The fields in the first column relate to process management.
- The other two columns relate to memory management and file management, respectively.

In practice, which fields the process table has is highly system dependent, but this figure gives a general idea of the kinds of information needed.

(7) What is interrupt? How it is handled by an OS?

- A software interrupt is caused either by an exceptional condition in the processor itself, or a special instruction in the instruction set which causes an interrupt when it is executed.
- The former is often called a trap or exception and is used for errors or events occurring during program execution that is exceptional enough that they cannot be handled within the program itself.
- Interrupts are a commonly used technique for process switching.
- Associated with each I/O device class (e.g., floppy disks, hard disks etc…) there 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.
- That is all the hardware does.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler marks waiting task as ready.
7. Scheduler decides which process is to run next.
8. C procedure returns to the assembly code.
9. Assembly language procedure starts up new current process.

Figure 2-6. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

- From here on, it is up to the software, in particular, the interrupt service procedure.
- All interrupts start by saving the registers, often in the process table entry for the current process.
- Then the information pushed onto the stack by the interrupt is removed and the stack pointer is set to point to a temporary stack used by the process handler.
- When this routine is finished, it calls a C procedure to do the rest of the work for this specific interrupt type.
- When it has done its job, possibly making some process now ready, the scheduler is called to see who to run next.
- After that, control is passed back to the assembly language code to load up the registers and memory map for the now-current process and start it running.
- Interrupt handling and scheduling are summarized in Figure 2-6.

(8) What is thread? Explain thread structure. Explain different types of thread. OR Explain thread in brief.

Thread

- A program has one or more locus of execution. Each execution is called a thread of execution.
- In traditional operating systems, each process has an address space and a single thread of execution.
- It is the smallest unit of processing that can be scheduled by an operating system.
- A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams.

Thread Structure

- Process is used to group resources together and threads are the entities scheduled for execution on the CPU.
- The thread has a program counter that keeps track of which instruction to execute next.
- It has registers, which holds 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.
- Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately.
- What threads add to the process model is to allow multiple executions to take place in the same process environment, to a large degree independent of one another.
- Having multiple threads running in parallel in one process is similar to having multiple processes running in parallel in one computer.

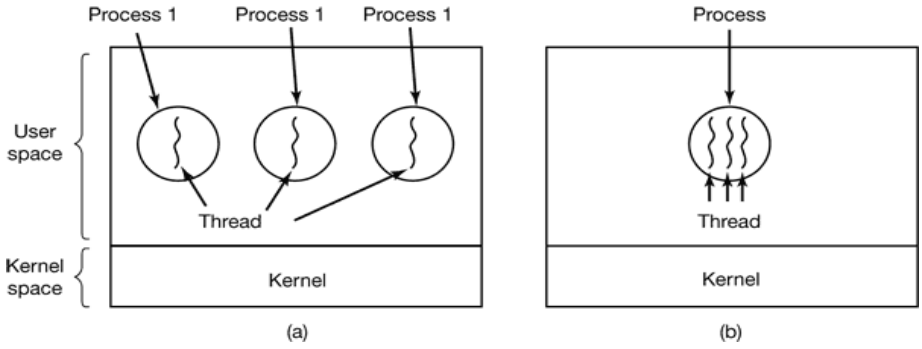


Figure 2-7. (a) Three processes each with one thread. (b) One process with three threads.

- In former case, the threads share an address space, open files, and other resources.
- In the latter case, process share physical memory, disks, printers and other resources.
- In Fig. 2-7 (a) we see three traditional processes. Each process has its own address space and a single thread of control.
- In contrast, in Fig. 2-7 (b) we see a single process with three threads of control.
- Although in both cases we have three threads, in Fig. 2-7 (a) each of them operates in a different address space, whereas in Fig. 2-7 (b) all three of them share the same address space.
- Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: running, blocked, ready, or terminated.

- When multithreading is present, processes normally start with a single thread present. This thread has the ability to create new threads by calling a library procedure ***thread_create***.
- When a thread has finished its work, it can exit by calling a library procedure ***thread_exit***.
- One thread can wait for a (specific) thread to exit by calling a procedure ***thread_join***. This procedure blocks the calling thread until a (specific) thread has exited.
- Another common thread call is ***thread_yield***, which allows a thread to voluntarily give up the CPU to let another thread run.

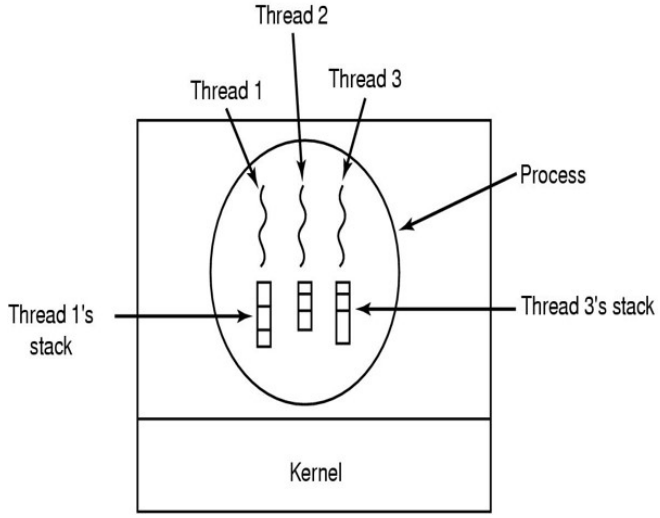


Figure 2-8. Each thread has its own stack.

Types of thread

1. User Level Threads
2. Kernel Level Threads

User Level Threads

- User level threads are implemented in user level libraries, rather than via systems calls.
- So thread switching does not need to call operating system and to cause interrupt to the kernel.
- The kernel knows nothing about user level threads and manages them as if they were single threaded processes.

- When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process.
- This table keeps track only of the per-thread properties, such as each thread's program counter, stack pointer, registers, state, and so forth.
- The thread table is managed by the run-time system.
- Advantages
  - It can be implemented on an Operating System that does not support threads.
  - A user level thread does not require modification to operating systems.
  - **Simple Representation**: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
  - **Simple Management**: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
  - **Fast and Efficient**: Thread switching is not much more expensive than a procedure call.
  - User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm.
- Disadvantages
  - There is a lack of coordination between threads and operating system kernel. Therefore, process as a whole gets one time slice irrespective of whether process has one thread or 1000 threads within. It is up to each thread to give up control to other threads.
  - Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
  - A user level thread requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will be blocked in the kernel, even if a single thread is blocked but other runnable threads are present. For example, if one thread causes a page fault, the whole process will be blocked.

Kernel Level Threads

- In this method, the kernel knows about threads and manages the threads.
- No runtime system is needed in this case.

- Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.
- Advantages
  - Because kernel has full knowledge of all threads, scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
  - Kernel threads do not require any new, non-blocking system calls. Blocking of one thread in a process will not affect the other threads in the same process as Kernel knows about multiple threads present so it will schedule other runnable thread.
- Disadvantages
  - The kernel level threads are slow and inefficient. As thread are managed by system calls, at considerably greater cost.
  - Since kernel must manage and schedule threads as well as processes. It requires a full thread control block (TCB) for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

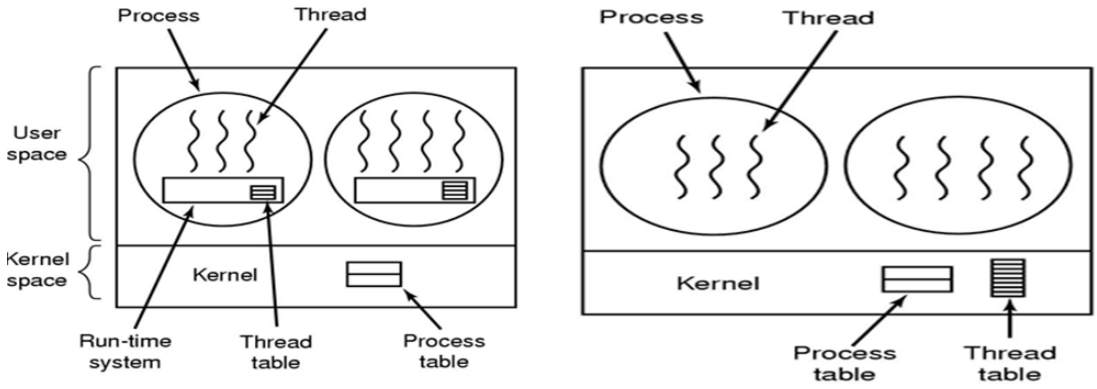


Figure 2-9. (a) A user-level threads package. (b) A threads package managed by the kernel.

Hybrid Implementations

- To combine the advantages of user-level threads with kernel-level threads, one way is to use the kernel-level threads and then multiplex user-level threads onto some

or all of the kernel threads, as shown in Fig. 2-10.

- In this design, the kernel is aware of only the kernel-level threads and schedules those.
- Some of those threads may have multiple user-level threads multiplexed on top of them.
- These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capability.
- In this model, each kernel-level thread has some set of user-level threads that take turns using it.
- This model gives the ultimate in flexibility as when this approach is used, the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one.

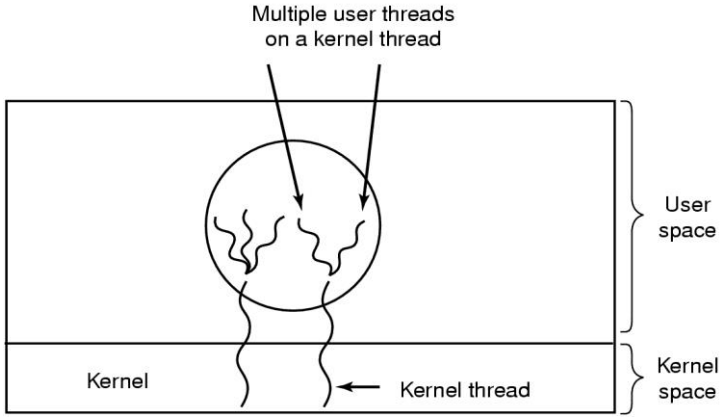


Figure 2-10. Multiplexing user-level threads onto kernel-level threads.

(9) Write the short note on Multithreading and Multitasking.

Multithreading

- The ability of an operating system to execute different parts of a program, called *threads* simultaneously, is called multithreading.
- The programmer must carefully design the program in such a way that all the threads can run at the same time without interfering with each other.
- On a single processor, multithreading generally occurs by time division multiplexing (as in multitasking) the processor switches between different threads.
- This context switching generally happens so speedy that the user perceives the threads or tasks as running at the same time.

Multitasking

- The ability to execute more than one *task* at the same time is called multitasking.
- In multitasking, only one CPU is involved, but it switches from one program to another so quickly that it gives the appearance of executing all of the programs at the same time.
- There are two basic types of multitasking.
    1) *Preemptive:* In preemptive multitasking, the operating system assign CPU *time slices* to each program.
    2) *Cooperative:* In cooperative multitasking, each program can control the CPU for as long as it needs CPU. If a program is not using the CPU, however, it can allow another program to use it.

(10) Write the similarities and dissimilarities (difference) between process and thread.

Similarities

- Like processes threads share CPU and only one thread is active (running) at a time.
- Like processes threads within a process execute sequentially.
- Like processes thread can create children.
- Like a traditional process, a thread can be in any one of several states: running, blocked, ready, or terminated.
- Like process threads have Program Counter, stack, Registers and state.

Dissimilarities

- Unlike processes threads are not independent of one another, threads within the same process share an address space.
- Unlike processes all threads can access every address in the task.
- Unlike processes threads are design to assist one other. Note that processes might or might not assist one another because processes may be originated from different users.

(11)	*Explain FCFS, Round Robin, Shortest Job First, Shortest Remaining Job First and Priority Scheduling algorithms with illustration.*
I	FCFS (First Come First Serve): • **Selection criteria :** The process that request first is served first. It means that processes are served in the exact order of their arrival. • **Decision Mode :** Non preemptive: Once a process is selected, it runs until it is blocked for an I/O or some event, or it is terminated. • **Implementation:** This strategy can be easily implemented by using FIFO queue, FIFO means First In First Out. When CPU becomes free, a process from the first position in a queue is selected to run. • **Example :** Consider the following set of four processes. Their arrival time and time required to complete the execution are given in following table. Consider all time values in milliseconds.

Process	Arrival Time (T0)	Time required for completion (ΔT) (CPU Burst Time)
P0	0	10
P1	1	6
P2	3	2
P3	5	4

- **Gantt Chart :**

P0	P1	P2	P3
0	10	16	18 22

- Initially only process P0 is present and it is allowed to run. But, when P0 completes, all other processes are present. So, next process P1 from ready queue is selected and

allowed to run till it completes. This procedure is repeated till all processes completed their execution.

- **Statistics :**

Process	Arrival Time (T0)	CPU Burst Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (WT=TAT-ΔT)
P0	0	10	10	10	0
P1	1	6	16	15	9
P2	3	2	18	15	13
P3	5	4	22	17	13

Average Turnaround Time:	(10+15+15+17)/4	= 57/4	**= 14.25 ms.**
Average Waiting Time:	(0+9+13+13)/4	= 35/4	**= 8.75 ms.**

- **Advantages:**
  - Simple, fair, no starvation.
  - Easy to understand, easy to implement.
- **Disadvantages :**
  - Not efficient. Average waiting time is too high.
  - Convoy effect is possible. All small I/O bound processes wait for one big CPU bound process to acquire CPU.
  - CPU utilization may be less efficient especially when a CPU bound process is running with many I/O bound processes.

II Shortest Job First (SJF):

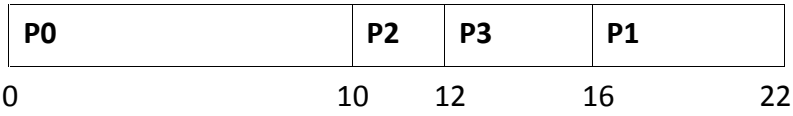
- **Selection Criteria :**
  The process, that requires shortest time to complete execution, is served first.
- **Decision Mode :**
  Non preemptive: Once a process is selected, it runs until either it is blocked for an I/O or some event, or it is terminated.
- **Implementation :**

This strategy can be implemented by using sorted FIFO queue. All processes in a queue are sorted in ascending order based on their required CPU bursts. When CPU becomes free, a process from the first position in a queue is selected to run.

- **Example :**
  Consider the following set of four processes. Their arrival time and time required to complete the execution are given in following table. Consider all time values in milliseconds.

Process	Arrival Time (T0)	Time required for completion (ΔT) (CPU Burst Time)
P0	0	10
P1	1	6
P2	3	2
P3	5	4

- **Gantt Chart :**



- Initially only process P0 is present and it is allowed to run. But, when P0 completes, all other processes are present. So, process with shortest CPU burst P2 is selected and allowed to run till it completes. Whenever more than one process is available, such type of decision is taken. This procedure us repeated till all process complete their execution.

- **Statistics :**

Process	Arrival Time (T0)	CPU Burst Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (Wt=TAT-ΔT)
P0	0	10	10	10	0
P1	1	6	22	21	15
P2	3	2	12	9	7

	P3	5	4	16	11	7

Average Turnaround Time:	(10+21+9+11)/4	= 51/4	**= 12.75 ms.**
Average Waiting Time:	(0+15+7+7) / 4	= 29 / 4	**= 7.25 ms.**

- **Advantages:**
  - Less waiting time.
  - Good response for short processes.
- **Disadvantages :**
  - It is difficult to estimate time required to complete execution.
  - Starvation is possible for long process. Long process may wait forever.

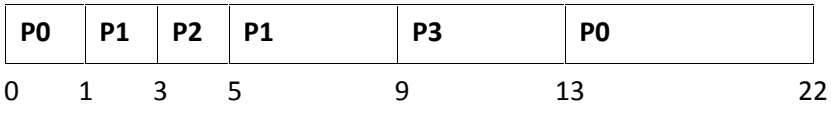
III Shortest Remaining Time Next (SRTN):

- **Selection criteria :**
  The process, whose remaining run time is shortest, is served first. This is a preemptive version of SJF scheduling.
- **Decision Mode:**
  Preemptive: When a new process arrives, its total time is compared to the current process remaining run time. If the new job needs less time to finish than the current process, the current process is suspended and the new job is started.
- **Implementation :**
  This strategy can also be implemented by using sorted FIFO queue. All processes in a queue are sorted in ascending order on their remaining run time. When CPU becomes free, a process from the first position in a queue is selected to run.
- **Example :**
  Consider the following set of four processes. Their arrival time and time required to complete the execution are given in following table. Consider all time values in milliseconds.

Process	Arrival Time (T0)	Time required for completion (ΔT) (CPU Burst Time)
P0	0	10
P1	1	6

P2	3	2
P3	5	4

- **Gantt Chart :**



- Initially only process P0 is present and it is allowed to run. But, when P1 comes, it has shortest remaining run time. So, P0 is preempted and P1 is allowed to run. Whenever new process comes or current process blocks, such type of decision is taken. This procedure is repeated till all processes complete their execution.

- **Statistics :**

Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (WT=TAT-ΔT)
P0	0	10	22	22	12
P1	1	6	9	8	2
P2	3	2	5	2	0
P3	5	4	13	8	4

Average Turnaround Time:	(22+8+2+8) / 4	= 40/4	**= 10 ms.**
Average Waiting Time:	(12+2+0+4)/4	= 18 / 4	**= 4.5 ms.**

- **Advantages :**
  - Less waiting time.
  - Quite good response for short processes.
- **Disadvantages :**
  - Again it is difficult to estimate remaining time necessary to complete execution.
  - Starvation is possible for long process. Long process may wait forever.
  - Context switch overhead is there.

IV Round Robin:

- **Selection Criteria:**
- Each selected process is assigned a time interval, called time quantum or time slice. Process is allowed to run only for this time interval. Here, two things are possible: First, Process is either blocked or terminated before the quantum has elapsed. In this case the CPU switching is done and another process is scheduled to run. Second, Process needs CPU burst longer than time quantum. In this case, process is running at the end of the time quantum. Now, it will be preempted and moved to the end of the queue. CPU will be allocated to another process. Here, length of time quantum is critical to determine.
- **Decision Mode:**

  Preemptive:
- **Implementation :**

  This strategy can be implemented by using circular FIFO queue. If any process comes, or process releases CPU, or process is preempted. It is moved to the end of the queue. When CPU becomes free, a process from the first position in a queue is selected to run.

- **Example :**

  Consider the following set of four processes. Their arrival time and time required to complete the execution are given in the following table. All time values are in milliseconds. Consider that time quantum is of 4 ms, and context switch overhead is of 1 ms.

Process	Arrival Time (T0)	Time required for completion (ΔT)
P0	0	10
P1	1	6
P2	3	2
P3	5	4

- **Gantt Chart :**

P0		P1		P2		P0		P3		P1		P0

0 4 5 9 10 12 13 17 18 22 23 25 26 28

- At 4ms, process P0 completes its time quantum. So it preempted and another process P1 is allowed to run. At 12 ms, process P2 voluntarily releases CPU, and another process is selected to run. 1 ms is wasted on each context switch as overhead. This procedure is repeated till all process completes their execution.

- **Statistics:**

Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (WT=TAT-ΔT)
P0	0	10	28	28	18
P1	1	6	25	24	18
P2	3	2	12	9	7
P3	5	4	22	17	13

Average Turnaround Time:	(28+24+9+17)/4	= 78 / 4	= 19.5 ms
Average Waiting Time:	(18+18+7+13)/4	= 56 / 4	= 14 ms

- **Advantages:**
  - One of the oldest, simplest, fairest and most widely used algorithms.
- **Disadvantages:**
  - Context switch overhead is there.
  - Determination of time quantum is too critical. If it is too short, it causes frequent context switches and lowers CPU efficiency. If it is too long, it causes poor response for short interactive process.

V Non Preemptive Priority Scheduling:

- **Selection criteria :**
  The process, that has highest priority, is served first.
- **Decision Mode:**
  Non Preemptive: Once a process is selected, it runs until it blocks for an I/O or some event, or it terminates.

- **Implementation :**

  This strategy can be implemented by using sorted FIFO queue. All processes in a queue are sorted based on their priority with highest priority process at front end. When CPU becomes free, a process from the first position in a queue is selected to run.

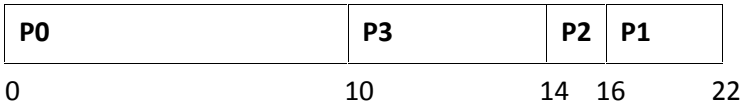
- **Example :**

  Consider the following set of four processes. Their arrival time, total time required completing the execution and priorities are given in following table. Consider all time values in millisecond and small values for priority means higher priority of a process.

Process	Arrival Time (T0)	Time required for completion (ΔT)	Priority
P0	0	10	5
P1	1	6	4
P2	3	2	2
P3	5	4	0

Here, process priorities are in this order: P3 > P2 > P1 > P0.

- **Gantt Chart :**



- Initially only process P0 is present and it is allowed to run. But, when P0 completes, all other processes are present. So, process with highest priority P3 is selected and allowed to run till it completes. This procedure is repeated till all processes complete their execution.

- **Statistics :**

Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (TAT-ΔT)
P0	0	10	10	10	0
P1	1	6	22	21	15
P2	3	2	16	13	11
P3	5	4	14	9	5

Average Turnaround Time:	(10+21+13+9) / 4	= 53 / 4	= 13.25 ms

Average Waiting Time:	(0+15+11+5) / 4	= 31 / 4	= 7.75 ms

- **Advantages:**
  - Priority is considered. Critical processes can get even better response time.
- **Disadvantages:**
  - Starvation is possible for low priority processes. It can be overcome by using technique called 'Aging'.
  - Aging*:* gradually increases the priority of processes that wait in the system for a long time.

Preemptive Priority Scheduling:

- **Selection criteria :**
  The process, that has highest priority, is served first.
- **Decision Mode:**
  Preemptive: When a new process arrives, its priority is compared with current process priority. If the new job has higher priority than the current, the current process is suspended and new job is started.
- **Implementation :**
  This strategy can be implemented by using sorted FIFO queue. All processes in a queue are sorted based on priority with highest priority process at front end. When CPU becomes free, a process from the first position in a queue is selected to run.
- **Example :**
  Consider the following set of four processes. Their arrival time, time required completing the execution and priorities are given in following table. Consider all time values in milliseconds and small value of priority means higher priority of the process.

Process	Arrival Time (T0)	Time required for completion (ΔT)	Priority
P0	0	10	5
P1	1	6	4
P2	3	2	2
P3	5	4	0

Here process priorities are in this order: P3>P2>P1>P0

- **Gantt chart:**

P0	P1	P2	P3	P1	P0

0 1 3 5 9 13 22

- Initially only process P0 is present and it is allowed to run. But when P1 comes, it has higher priority. So, P0 is preempted and P1 is allowed to run. This process is repeated till all processes complete their execution.

- **Statistics:**

Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (TAT-ΔT)
P0	0	10	22	22	12
P1	1	6	13	12	6
P2	3	2	5	2	0
P3	5	4	9	4	0

Average Turnaround Time:	(22+12+2+4) / 4	= 40 / 4	= 10 ms
Average Waiting Time:	(12+6+0+0) / 4	= 18 / 4	= 4.5 ms

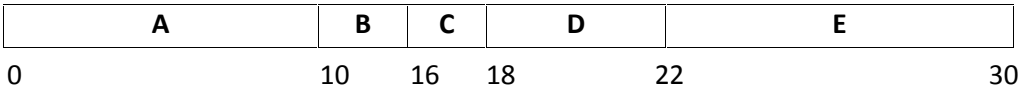
- **Advantages:**
  - Priority is considered. Critical processes can get even better response time.
- **Disadvantages:**
  - Starvation is possible for low priority processes. It can be overcome by using technique called 'Aging'.
  - Aging: gradually increases the priority of processes that wait in the system for a long time.
  - Context switch overhead is there.

(2) Five batch jobs A to E arrive at same time. They have estimated running times 10,6,2,4 and 8 minutes. Their priorities are 3,5,2,1 and 4 respectively with 5 being highest priority. For each of the following algorithm determine mean process turnaround time. Ignore process swapping overhead.

Round Robin, Priority Scheduling, FCFS, SJF.

Process	Running Time Time required for completion (ΔT)	Priority
A	10	3
B	6	5
C	2	2
D	4	1
E	8	4

First Come First Served:



Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (TAT-ΔT)
A	0	10	10	10	0
B	0	6	16	16	10
C	0	2	18	18	16
D	0	4	22	22	18
E	0	8	30	30	22

Average Turnaround Time:	(10+16+18+22+30) / 5	= 96 / 5	= 19.2 ms
Average Waiting Time:	(0+10+16+18+22) / 5	= 56 / 5	= 13.2 ms

Shortest Job First:

C	D	B	E	A

0 2 6 12 20 30

Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (TAT-ΔT)
A	0	10	30	30	20
B	0	6	12	12	6
C	0	2	2	2	0
D	0	4	6	6	2
E	0	8	20	20	12

Average Turnaround Time:	(30+12+2+6+20) / 5	= 70 / 5	= 14 ms
Average Waiting Time:	(20+6+0+2+12) / 5	= 40 / 5	= 8 ms

Priority:

B	E	A	C	D

0 6 14 24 26 30

Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (TAT-ΔT)
A	0	10	24	24	14
B	0	6	6	6	0
C	0	2	26	26	24
D	0	4	30	30	26
E	0	8	14	14	6

Average Turnaround Time:	(24+6+26+30+14) / 5	= 100 / 5	= 20 ms

	Average Waiting Time:	(14+0+24+26+6) / 5	= 70 / 5	= 14 ms

Round Robin:

Time slice OR Quantum time= 2min.

A	B	C	D	E	A	B	D	E	A	B	E	A	E	A

0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (TAT-ΔT)
A	0	10	30	30	20
B	0	6	22	22	16
C	0	2	6	6	4
D	0	4	16	16	12
E	0	8	28	28	20

Average Turnaround Time:	(30+22+6+16+28) / 5	= 102 / 5	= 20.4 ms
Average Waiting Time:	(20+16+4+12+20) / 5	= 72 / 5	= 14.4 ms

(3) Suppose that the following processes arrive for the execution at the times indicated. Each process will run the listed amount of time. Assume preemptive scheduling.

Process	Arrival Time (ms)	Burst Time (ms)
P1	0.0	8
P2	0.4	4
P3	1.0	1

What is the turnaround time for these processes with Shortest Job First scheduling algorithm?

(4)	Consider the following set of processes with length of CPU burst time given in milliseconds. Process Burst Time Priority P1 10 3 P2 1 1 P3 2 3 P4 1 4 P5 5 2 Assume arrival order is: P1, P2, P3, P4, P5 all at time 0 and a smaller priority number implies a higher priority. Draw the Gantt charts illustrating the execution of these processes using preemptive priority scheduling.

(1) Define following terms.

Race Condition

- Race condition can be defined as situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when (their relative execution order).

Mutual Exclusion

- It is a way of making sure that if one process is using a shared variable or file; the other process will be excluded (stopped) from doing the same thing.

Turnaround Time

- Time required to complete execution of process is known as turnaround time.
- Turnaround time = Process finish time – Process arrival time.

Throughput

- Number of processes completed per time unit is called throughput.

Critical Section

- The part of program or code of segment of a process where the shared resource is accessed is called critical section.

Waiting time

- It is total time duration spent by a process waiting in ready queue.
- Waiting time = Turnaround time – Actual execution time.

Response Time

- It is the time between issuing a command/request and getting output/result.

(2) Explain different mechanisms for achieving mutual exclusion with busy waiting.

Disabling interrupts

- In this mechanism a process disables interrupts before entering the critical section and enables the interrupt immediately after exiting the critical section.

```
while( true )
{
        < disable interrupts >;
        < critical section >;
        < enable interrupts >;
        < remainder section>;    }
```

- Mutual exclusion is achieved because with interrupts disabled, no clock interrupts can occur.
- The CPU is only switched from process to process as a result of clock or other interrupts, and with interrupts turned off the CPU will not be switched to another process.
- Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.
- This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts.
- Suppose that one of them did it and never turned them on again? That could be the end of the system.
- Furthermore if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.
- On the other hand, it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists.
- If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.
- The conclusion is: disabling interrupts is often a useful technique within the operating system itself but is not appropriate as a general mutual exclusion mechanism for user processes.

Shared lock variable:

- In this mechanism there is a shared variable lock having value 0 or 1.
- Before entering in to critical region a process check a shared variable lock's value.
- If the value of lock is 0 then set it to 1 before entering the critical section and enters into critical section and set it to 0 immediately after leaving the critical section.

```
While (true)
{
        < set shared variable to 1>;
        < critical section >;
        < set shared variable to 0>;
        < remainder section>;
}
```

- If it is 1, means some other process is inside the critical section so, the process requesting to enter the critical region will have to wait until it becomes zero.
- This mechanism suffers the same problem (race condition). If process P0 sees the value of lock variable 0 and before it can set it to 1, context switching occurs.
- Now process P1 runs and finds value of lock variable 0, so it sets value to 1, enters critical region.
- At the same point of time P0 resumes, sets the value of lock variable to 1, enters critical region.
- Now two processes are in their critical regions accessing the same shared memory, which violates the mutual exclusion condition.

Strict Alteration:

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

(a)

Process 1

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

(b)

Process 2

- In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section.
- Initially turn=0. Process 1 inspects turn, finds it to be 0, and enters in its critical section. Process 2 also finds it to be 0 and therefore sits in a loop continually testing 'turn' to see when it becomes 1.
- Continuously testing a variable waiting for some value to appear is called the **busy waiting**.
- When process 1 exit from critical region it set turn to 1 and now process 2 can find it to be 1 and enters in to critical region.
- In this way both the process get alternate turn to enter in critical region.
- Taking turns is not a good idea when one of the processes is much slower than the other.
- Consider the following situation for two processes P0 and P1

- P0 leaves its critical region, set turn to 1, enters non critical region.
- P1 enters and finishes its critical region, set turn to 0.
- Now both P0 and P1 in non-critical region.
- P0 finishes non critical region, enters critical region again, and leaves this region, set turn to 1.
- P0 and P1 are now in non-critical region.
- P0 finishes non critical region but cannot enter its critical region because turn = 1 and it is turn of P1 to enter the critical section.
- Hence, P0 will be blocked by a process P1 which is not in critical region. This violates one of the conditions of mutual exclusion.

- It wastes CPU time, so we should avoid busy waiting as much as we can.
- Can be used only when the waiting period is expected to be short.

TSL (Test and Set Lock) Instruction

```
enter_region:
    TSL REGISTER,LOCK        | copy lock to register and set lock to 1
    CMP REGISTER,#0          | was lock zero?
    JNE enter_region         | if it was nonzero, 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
```

- Test and Set Lock that works as follows. It reads the contents of the memory word lock into register RX and then stores a nonzero value at the memory address lock.
- The operations of reading the word and storing into it are guaranteed to be indivisible—no other processor can access the memory word until the instruction is finished.
- The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.
- The solution using TSL is given above.
- There is a four-instruction subroutine in a typical assembly language code.
- The first instruction copies the old value of lock to the register and then sets lock to 1.

- Then the old value is compared with 0. If it is nonzero, the lock was already set, so the program just goes back to the beginning and tests it again.
- Sooner or later it will become 0 (when the process currently in its critical region is done with its critical region), and the subroutine returns, with the lock set.
- Clearing the lock is simple. The program just stores a 0 in lock. No special instructions are needed.
- One solution to the critical region problem is now straightforward.
- Before entering its critical region, a process calls enter_region, which does busy waiting until the lock is free; then it acquires the lock and returns.
- Whenever the process wants to leave critical region, it calls leave_region, which stores a 0 in lock.
- As with all solutions based on critical regions, the processes must call enter_region and leave_region at the correct times for the method to work.

Exchange Instruction

```
enter_region:
    MOVE REGISTER,#1          | put a 1 in the register
    XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
    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
```

- An alternative instruction to TSL is XCHG.
- Move value 1 to CPU register A (A = lock)
- Exchange the value of CPU register and lock variable.
- How does XCHG solve problem?
- If any process wants to enter critical region, it calls the procedure enter_region.
- Process executes XCHG RX, lock in order to gain access. If it finds lock = 0 then it proceeds and enters in critical region (and resets lock to 1), but if it finds lock = 1 then it loops back and will keep doing so until lock = 0. Other processes will see lock = 1 and similarly loop.
- When process leaves it sets lock = 0 by calling leave_region, thereby allowing another process to enter.

Peterson's Solution

- By combining the idea of taking turns with the idea of lock variables and warning variables, Peterson discovered a much simpler way to achieve mutual exclusion.
- This algorithm consists of two procedures written in ANSI C as shown below.
- Before using the shared variables (i.e., before entering its critical region), each process calls procedure enter_region with its own process number, 0 or 1, as parameter.
- This call will cause it to wait, if needed, until it is safe to enter critical region.
- After it has finished with the shared variables, the process calls leave_region to indicate that it is done and to allow the other process to enter, if it so desires.

```
#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 region */
}
```

- Let us see how this solution works. Initially neither process is in its critical region.
- Now process 0 calls enter_region. It indicates its interest by setting its array element and sets turn to 0.
- Since process 1 is not interested, enter_region returns immediately. If process 1 now calls enter_region, it will be blocked there until interested [0] goes to FALSE, an event that only happens when process 0 calls leave_region to exit the critical region.
- Now consider the case that both processes call enter_region almost simultaneously. Both will store their process number in turn. Whichever store is done last is the one that counts; the first one is overwritten and lost.
- Suppose that process 1 stores last, so turn is 1. When both processes come to the

while statement, process 0 executes it zero times and enters its critical region.

- Process 1 loops and does not enter its critical region until process 0 exits its critical region.

(3) What is priority inversion problem in inter-process communication? How to solve it?

Priority inversion problem:

- Priority inversion means the execution of a high priority process/thread is blocked by a lower priority process/thread.
- Consider a computer with two processes, H having high priority and L having low priority.
- The scheduling rules are such that H runs first then L will run.
- At a certain moment, L is in critical region and H becomes ready to run (e.g. I/O operation complete).
- H now begins busy waiting and waits until L will exit from critical region.
- But H has highest priority than L so CPU is switched from L to H.
- Now L will never be scheduled (get CPU) until H is running so L will never get chance to leave the critical region so H loops forever. This situation is called priority inversion problem.

Solving priority inversion problem solution:

- Execute the critical region with masked (disable) interrupt. Thus the current process cannot be preempted.
- Determine the maximum priority among the process that can execute this critical region, and then make sure that all processes execute the critical region at this priority. Since now competing process have the same priority, they can't preempt other.
- **A priority ceiling** : With priority ceilings, the shared Mutex process (that runs the operating system code) has a characteristic (high) priority of its own, which is assigned to the task locking the Mutex. This works well, provided the other high priority task(s) that tries to access the Mutex does not have a priority higher than the ceiling priority.
- **Priority inheritance:** Under the policy of priority inheritance, whenever a high priority task has to wait for some resource shared with an executing low priority

task, the low priority task is temporarily assigned the priority of the highest waiting priority task for the duration of its own use of the shared resource, thus keeping medium priority tasks from pre-empting the (originally) low priority task, and thereby affecting the waiting high priority task as well. Once the resource is released, the low priority task continues at its original priority level.

(4) What is Producer Consumer problem? Explain how sleep and wakeup system calls work.

- Some inter-process communication primitives cause the calling process to be blocked instead of wasting CPU time when they are not allowed to enter their critical regions.
- One of the simplest pair is the sleep and wakeup.
- Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
- The wakeup call has one parameter, the process to be awakened.
- Alternatively, both sleep and wakeup each have one parameter, a memory address used to match up sleep with wakeup.
- As an example of how these primitives can be used, let us consider 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.
- Trouble arises when the producer wants to put a new item in the buffer, but buffer is already full.
- The solution for the producer is to go to sleep whenever the buffer is full.
- When the consumer removes one or more items from the buffer and vacant slots will be available, consumer awakens the producer.
- Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.
- This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory.
- To keep track of the number of items in the buffer, we will need a variable, count.
- Buffer can hold maximum N data items. The producer first check the value of count, if it is N then producer will go to sleep.

- If count is less than N, then 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.
- The code for both producer and consumer is shown in figure 3.1.

```
#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 3-1. The producer-consumer problem using sleep & wake up system calls.

- The race condition can occur in producer consumer example because access to count is unconstrained.
- The following situation could possibly occur.
- The buffer is empty and the consumer has just count to see if it is 0.

- At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer.
- The producer inserts an item in the buffer, increments count, and notices that it is now 1. Reasoning that count was just 0, and thus the consumer must be sleeping, the producer calls wakeup to wake the consumer up.
- Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost.
- When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.
- The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost.
- The simpler solution to lost wake up signal is to add **wakeup waiting bit**.
- When a wakeup is sent to a process that is still awake, this bit is set.
- Later, when the process tries to go to sleep, if the wakeup waiting bit is on, it will be turned off, but the process will stay awake.
- The wakeup waiting bit is a piggy bank for wakeup signals.
- But with three or more processes one wakeup waiting bit is insufficient. So, the problem of race condition will be still there.

(5) What is Semaphore? Give the implementation of Bounded Buffer Producer Consumer problem using Semaphore.

Semaphore:

- A semaphore is a variable that provides an abstraction for controlling access of a shared resource by multiple processes in a parallel programming environment.
- There are 2 types of semaphores:
    1. Binary semaphores: - Binary semaphores have 2 methods associated with it (up, down / lock, unlock). Binary semaphores can take only 2 values (0/1). They are used to acquire locks.
    2. Counting semaphores: - Counting semaphore can have possible values more than two.

Bounded Buffer Producer Consumer problem:

- A buffer of size N is shared by several processes. We are given sequential code
    - insert_item - adds an item to the buffer.
    - remove_item - removes an item from the buffer.

- We want functions insert _item and remove_item such that the following hold:
    1. Mutually exclusive access to buffer: At any time only one process should be executing insert_item or remove_item.
    2. No buffer overflow: A process executes insert_item only when the buffer is not full (i.e., the process is blocked if the buffer is full).
    3. No buffer underflow: A process executes remove_item only when the buffer is not empty (i.e., the process is blocked if the buffer is empty).
    4. No busy waiting.
    5. No producer starvation: A process does not wait forever at insert_item() provided the buffer repeatedly becomes full.
    6. No consumer starvation: A process does not wait forever at remove_item() provided the buffer repeatedly becomes empty.

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
      int item;

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


void consumer(void)
{
      int item;

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

Figure 3-2 The producer-consumer problem using semaphores.

Implementation of Bounded Buffer Producer Consumer problem using Semaphore:

- In the above algorithm we have N slots in buffer, into which producer can insert item and from which consumer can consume item only.
- Semaphore is special kind of integer.
- Empty is the number of empty buffer slots and full is the number of full buffer slots.
- Semaphore variable Mutex is used for mutual exclusion.
- Initially there no item in buffer means buffer is empty.
- Whenever Producer wants to insert some item in buffer, it calls void producer (void) and follows the steps given below:
    1) Generate something to put in buffer
    2) As producer will be inserting item in the buffer, it will decrement empty count.
    3) Then it will perform down operation on the Mutex to get the exclusive access of shared buffer so, consumer cannot access buffer until producer finishes its work.
    4) Now producer enters in the critical region i.e. it inserts the item into buffer.
    5) When it leaves the critical region, it does up on Mutex.
    6) Finally, it Increments the full count as one item is inserted into the buffer.
- Consumer will consume some item from buffer and call void consumer (void) and follows the steps given below:
    1) Decrement the value of full count as consumer will remove an item from buffer.
    2) Then it will perform down the on Mutex so that, producer cannot access buffer until consumer will finish its work.
    3) Enter in critical section.
    4) Take item from buffer.
    5) Up Mutex and exit from critical section.
    6) Increment empty count.
    7) Consume that item.

(6) Give the implementation of Readers Writer problem using Semaphore.

- In the readers and writers problem, many competing processes are wishing to perform reading and writing operations in a database.
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have

access to the database, not even readers.

- In this solution, the first reader to get access to the database does a down on the semaphore db.

```
typedef int Semaphore;
Semaphore mutex = 1;                    // Controls access to the reader count
Semaphore db = 1;                       // Controls access to the database
int reader_count = 0;                   // The number of processes reading the data base


void Reader(void)
{
while (TRUE) {                          // loop forever
  down(&mutex);                         // gain access to reader_count
  reader_count = reader_count + 1;      // increment the reader_count
   if (reader_count == 1)               /* if this is the first process to read the database,  a down on
         down(&db);                     db is executed to gain the  access of the Database and to
                                        prevent writer process from accessing database */
   up(&mutex);                          // allow other processes to access reader_count

   read_database();
  down(&mutex);                         // gain access to reader_count
  reader_count = reader_count – 1;      // decrement reader_count
  if (reader_count == 0)                /* if there are no more processes reading from the  Database,
         up(&db);                       leave the control of database & allow writing process to
                                        access the data */
   up(&mutex);                          // allow other processes to access reader_count
   use_read_data(); }                   // use the data read from the database (non-critical)
}


void Writer(void)
{
    while (TRUE) {                      // loop forever
    create_data();                      // create data to enter into database (non-critical)
    down(&db);                          // gain access to the database
    write_db();                         // write information to the database
     up(&db);     }                     // release exclusive access to the database
}
```

Figure 3-3 A solution to the readers and writers problem.

- Subsequent readers only increment a counter, rc. As readers leave, they decrement the counter and the last one out does an up on the semaphore, allowing a blocked

writer, if there is one, to get in.Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted.

- A third and subsequent readers can also be admitted if they come along.
- Now suppose that a writer comes along. The writer cannot be admitted to the database, since writers must have exclusive access, so the writer is suspended.
- As long as at least one reader is still active, subsequent readers are admitted.
- As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive.
- The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never allow accessing the database.
- To prevent this situation, the solution is given as: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately.
- In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance.

(7) What is Monitor? Give the implementation of Bounded Buffer Producer Consumer problem using Monitor.

Monitor:

- Monitors were developed in the 1970s to make it easier to avoid race conditions and implement mutual exclusion.
- A monitor is a collection of procedures, variables, and data structures grouped together in a single module or package.
- Processes can call the monitor procedures but cannot access the internal data structures.
- 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.
- The compiler usually enforces mutual exclusion.
- Typically when a process calls monitor procedure, the first few instruction of procedure will check to see if any other process is currently active in monitor. If so, calling process will be suspended until the other process left the monitor.

- If no other process is using monitor, the calling process may enter.
- The solution uses condition variables, along with two operations on them, wait and signal. When a monitor procedure discovers that it cannot continue (e.g., the producer finds the buffer full), it does a wait on some condition variable, full.

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Figure 3-4. An outline of the producer-consumer problem with monitors.

- This action causes the calling process to block. It also allows another process that had been previously prohibited from entering the monitor to enter now.

- This other process, for example, the consumer, can wake up its sleeping partner by doing a signal on the condition variable that its partner is waiting on.
- If a signal is done on a condition variable on which several processes are waiting, only one of them, determined by the system scheduler, is revived.

(8) What is Mutex? How it can solve Producer Consumer problem?

Mutex

- It is method of making sure that no two processes or threads can be in their critical section at the same time.
- Mutex is the short form for 'Mutual Exclusion Object'. A Mutex and the binary semaphore are essentially the same. Both can take values: 0 or 1.

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy Mutex to register and set Mutex to 1
        CMP REGISTERS, #0       | was Mutex zero?
        JZE ok                  | if it was zero, Mutex was unlocked, so return
        CALL thread_yield       | Mutex is busy; schedule another thread
        JMP mutex_lock          | try again later
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in Mutex
        RET                     | return to caller
```

Figure 3.5. Implementation of Mutex lock and Mutex Unlock

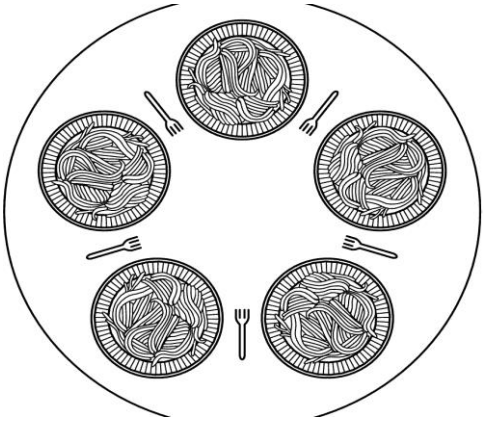
Solution of Producer Consumer Problem with Mutex

- Consider the standard producer consumer problem.
- We have a buffer of N slots. A producer will produce the data and writes it to the buffer. A consumer will remove the data from the buffer.
- Our objective is to achieve the mutual exclusion and at the same time to avoid race conditions.
- **Mutex** is a variable that can be in one of two states: unlocked or locked.
- With 0 meaning unlocked and all other values meaning locked.

- Two procedures are used with Mutex.
- When a thread (or process) needs access to a critical region, it calls mutex_lock . If the mutex is current unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.
- 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 . If multiple threads are blocked on the mutex, one of them is chosen at random and allowed to acquire the lock.
- Because mutexes are so simple, they can easily be implemented in user space if a TSL instruction is available. The code for mutex_lock and mutex_unlock for use with a user-level threads package are shown in figure 3.5.

(9) Explain Dinning Philosopher Problem.

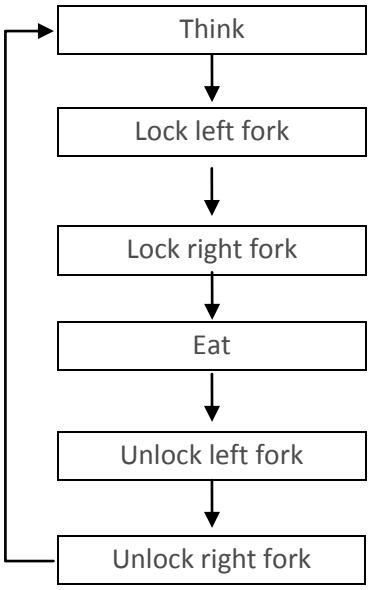
- Dining philosopher's problem is a classic multiprocess synchronization problem.
- Problem is summarized as 5 philosophers sitting at a round table doing 2 things, eating and thinking.
- While eating they are not thinking and while thinking they are not eating.
- Each philosopher has plates that are 5 plates. And there is a fork place between each pair of adjacent philosophers that is total of 5 forks.
- Each philosopher needs 2 forks to eat. Each philosopher can only use the forks on his immediate left and immediate right.



```
#define N 5                     /* number of philosophers */
#define LEFT(i+N-1)%N           /* number of i's left neighbor */
#define RIGHT(i+1)%N            /* number of i's right neighbor */
#define THINKING 0              /* philosopher is thinking */
#define HUNGRY   1              /* philosopher is trying to get forks */
#define EATING   2              /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */
void philosopher (int i)        /* i: philosopher number, from 0 to N-1 */
{
while (TRUE) {                  /* repeat forever */
  think();                      /* philosopher is thinking */
  take_forks(i);                /* acquire two forks or block */
  eat();                        /* eating spaghetti */
  put_forks(i);}                /* put both forks back on table */
}

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

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

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

Figure 3-6. A solution to the dining philosopher's problem.

- The "pick up forks" part is the key point. How does a philosopher pick up forks?
- The problem is each fork is shared by two philosophers and hence a shared resource. We certainly do not want a philosopher to pick up a fork that has already been picked up by his neighbor. This is a race condition.
- To address this problem, we may consider each fork as a shared item protected by a mutex lock.
- Each philosopher, before he can eat, locks his left fork and locks his right fork. If the acquisitions of both locks are successful, this philosopher now owns two locks (hence two forks), and can eat.
- After finishes easting, this philosopher releases both forks, and thinks. This execution flow is shown below.



(10) Explain Message Passing. Give the implementation of Producer Consumer problem using message passing.

Message Passing

- This method will use two primitives
  1) Send: It is used to send message.

     Send (destination, &message)

     In above syntax destination is the process to which sender want to send

message and message is what the sender wants to send.

2) Receive: It is used to receive message.

Receive (source, &message)

In above syntax source is the process that has send message and message is what the sender has sent.

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

void producer(void)
{
      int item;
      message m;                                /* message buffer */

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

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

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

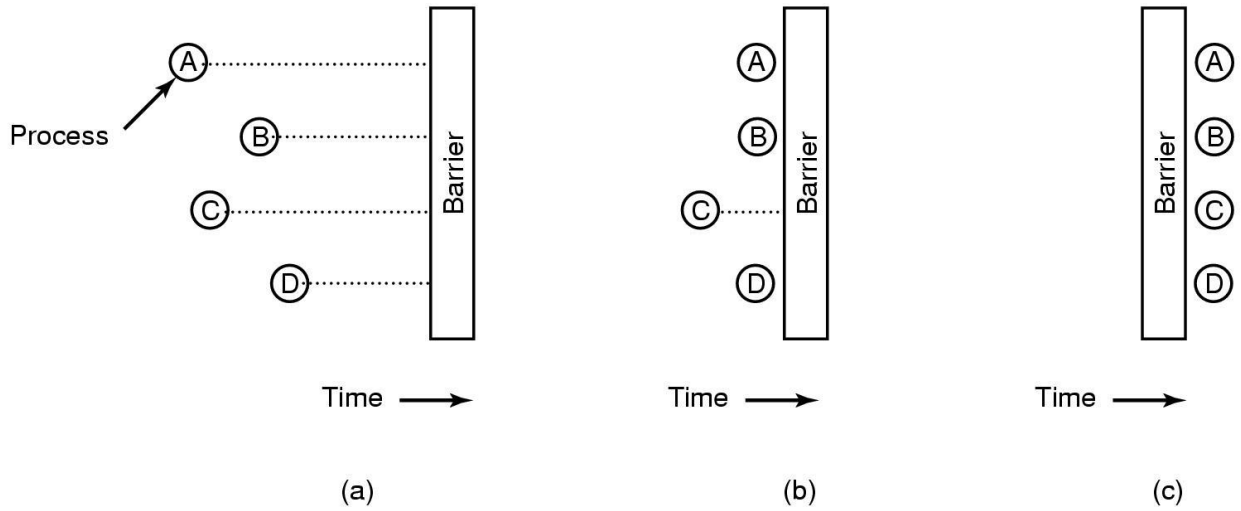
Figure 3-7. The producer-consumer problem with *N* messages.

- In this solution, a total of N messages are used, analogous to the N slots in a shared memory buffer.
- The consumer starts out by sending N empty messages to the producer.
- Whenever the producer has an item to give to the consumer, it takes and empty

message and sends back a full one and receiver receives that filled message and consume or use that message.

- In this way, the total number of messages in the system remains constant in time.
- If the producer works faster, all the messages will end up full, waiting for the consumer; the producer will be blocked, waiting for an empty to come back.
- If the consumer works faster, then the reverse happens, all the messages will be empties waiting for the producer to fill them up; the consumer will be blocked, waiting for a full message.
- Two of the common variants of Message Passing are as follows; one way is to assign each process a unique address and have messages be addressed to processes. The other way is to use a data structure, called a mailbox, a place to buffer a certain number of massages, and typically specified when the mailbox is created. For the Producer Consumer problem, both the producer and the consumer would create mailboxes large enough to hold N messages.

(11) Explain Barrier.



Barrier

- In this mechanism some application are divided into phases and have that rule that no process may proceed into the next phase until all process completed in this phase and are ready to proceed to the next phase.
- This behavior is achieved by placing barrier at the end of each phase.
- When a process reaches the barrier, it is blocked until all processes have reach at

the barrier.

- In the above fig. (a) There are four processes with a barrier.
- In fig. (b) Processes A, B and D are completed and process C is still not completed so until process C will complete, process A,B and D will not start in next phase.
- In fig. (c) Process C completes all the process start in next phase.

(12) What is scheduler? Explain queuing diagram representation of process scheduler with figure.

Scheduler:

- A program or a part of operating system that arranges jobs or a computer's operations into an appropriate sequence is called scheduler.

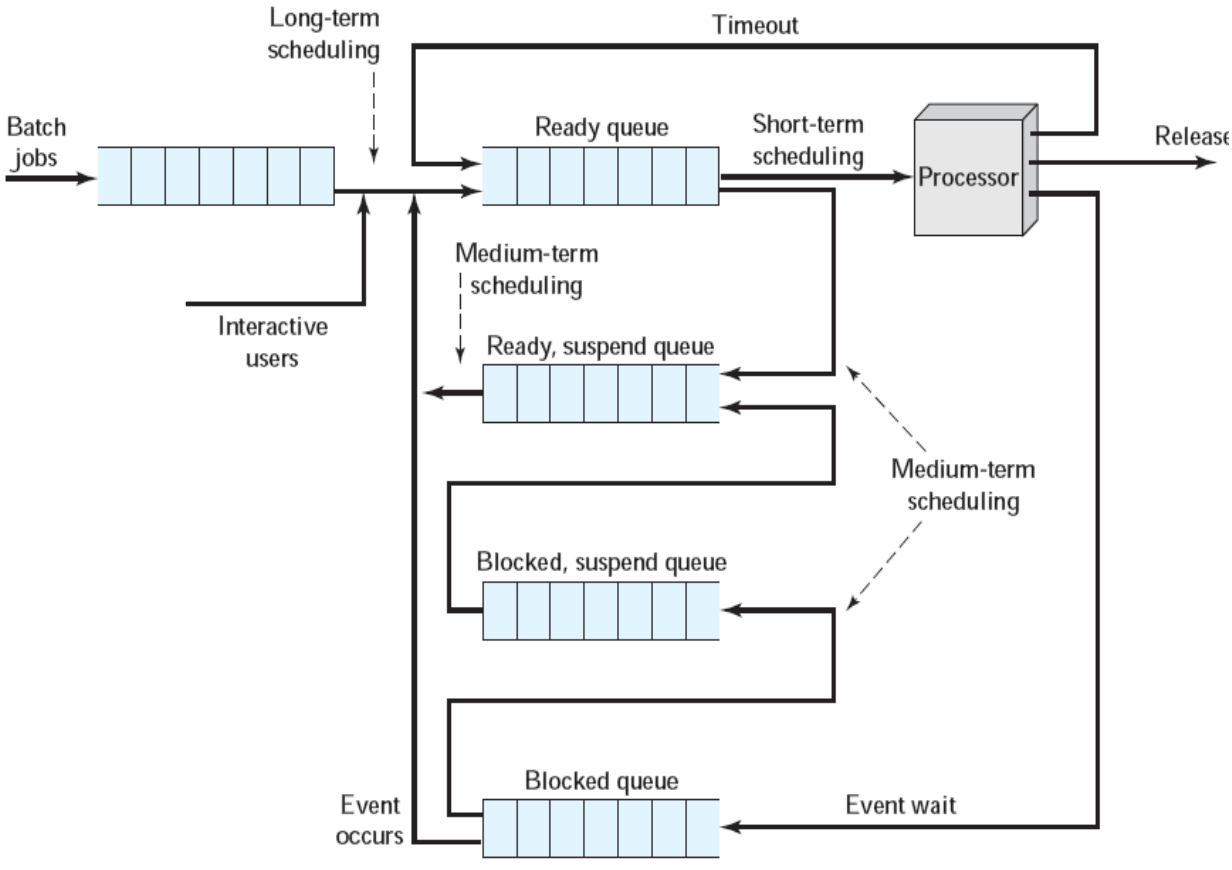


Figure 3-6 A Queuing diagram representation of process scheduler:

- As processes enter the system, they are put into a **job queue (batch jobs),** which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue** and wait there until it is selected for execution or dispatched.
- The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue.
- Once the process is allocated the CPU and is executing, one of several events could occur:
  - ✓ The process could issue an I/O request and then be placed in an I/O queue.
  - ✓ The process could create a new sub process and wait for the sub process's termination.
  - ✓ The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Scheduler

- Schedulers are special system software which handles process scheduling in various ways.
- Their main task is to select the jobs to be submitted into the system and to decide which process to run.
- Schedulers are of three types:
  1. Long Term Scheduler
  2. Short Term Scheduler
  3. Medium Term Scheduler

Long Term Scheduler

- It is also called job scheduler.
- Long term scheduler determines which programs are admitted to the system for processing.
- Job scheduler selects processes from the queue and loads them into memory for execution.
- Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound.
- It also controls the degree of multiprogramming.

- If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.
- On some systems, the long term scheduler may not be available or minimal.
- Time-sharing operating systems have no long term scheduler.
- When process changes the state from new to ready, then there is use of long term scheduler.

Short Term Scheduler

- It is also called CPU scheduler.
- Main objective is increasing system performance in accordance with the chosen set of criteria.
- It is the change of ready state to running state of the process.
- CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.
- Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next.
- Short term scheduler is faster than long term scheduler.

Medium Term Scheduler

- Medium term scheduling is part of the swapping.
- It removes the processes from the memory.
- It reduces the degree of multiprogramming.
- Running process may become suspended if it makes an I/O request.
- Suspended processes cannot make any progress towards completion.
- In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage.
- This process is called swapping, and the process is said to be swapped out or rolled out.
- Swapping may be necessary to improve the process mix. e of handling the swapped out-processes.

(1) What is Deadlock?

Deadlock

- 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.
- Suppose process D holds resource T and process C holds resource U.
- Now process D requests resource U and process C requests resource T but none of process will get this resource because these resources are already hold by other process so both can be blocked, with neither one be able to proceed, this situation is called deadlock.

Example:

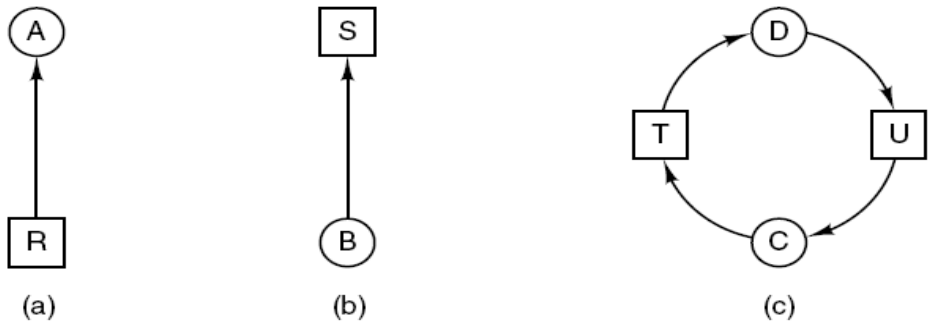


Figure 4-1 Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- As shown in figure 4-1, resource T assigned to process D and resource U is assigned to process C.
- Process D is requesting / waiting for resource U and process C is requesting / waiting for resource T.
- Processes C and D are in deadlock over resources T and U.

(2) Explain preemptable and non-preemptable resource with example.

- Resources come in two types: preemptable and non-preemptable.
- A preemptable resource is one that can be taken away from the process holding it with no ill effects.
- Consider, for example, a system with 32 MB of user memory, one printer, and two 32-

MB processes such that each process wants 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.
- Process B now runs and tries, unsuccessfully, to acquire the printer.
- Potentially, we now have a deadlock situation, because A has the printer and B has the memory, and neither can proceed without the resource held by the other.
- Fortunately, it is possible to preempt (take away) the memory from B by swapping it out and swapping A in. Now A can run, do its printing, and then release the printer. No deadlock occurs.
- A **non-preemptable resource**, in contrast, is one that cannot be taken away from its current owner without causing the computation to fail.
- 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, CD recorders are not preemptable at an arbitrary moment.
- In general, deadlocks involve non-preemptable resources.

(3) List the conditions that lead to deadlock.

Conditions that lead to deadlock

- There are four conditions that must hold for deadlock:
  1) Mutual exclusion condition
     - Each resource is either currently assigned to exactly one process or is available.
  2) Hold and wait condition
     - Process currently holding resources granted earlier can request more resources.
  3) No preemption condition
     - Previously granted resources cannot be forcibly taken away from process.
  4) Circular wait condition
     - There must be a circular chain of 2 or more processes. Each process is waiting for resource that is held by next member of the chain.
- **All four of these conditions must be present for a deadlock to occur.**

(4) Explain deadlock ignorance. OR Explain Ostrich Algorithm.

- Pretend (imagine) that there's no problem.
- This is the easiest way to deal with problem.
- This algorithm says that stick your head in the sand and pretend (imagine) that there is no problem at all.
- This strategy suggests to ignore the deadlock because deadlocks occur rarely, but system crashes due to hardware failures, compiler errors, and operating system bugs frequently, then not to pay a large penalty in performance or convenience to eliminate deadlocks.
- This method is reasonable if
  - Deadlocks occur very rarely
  - Cost of prevention is high
- UNIX and Windows take this approach
  - Resources (memory, CPU, disk space) are plentiful
  - Deadlocks over such resources rarely occur
  - Deadlocks typically handled by rebooting
- Trade off between convenience and correctness

(5) Explain deadlock detection and recovery.

Deadlock detection with single resource of each type.

- Algorithm for detecting deadlock for single resource
  1. For each node, N in the graph, perform the following five steps with N as the starting node.
  2. Initialize L to the empty list, designate all arcs as unmarked.
  3. Add current node to end of L, check to see if node now appears in L two times. If it does, graph contains a cycle (listed in L), algorithm terminates.
  4. From given node, see if 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 step 3.

6. If this is initial node, graph does not contain any cycles, algorithm terminates. Otherwise, dead end. Remove it, go back to previous node, make that one current node, go to step 3.

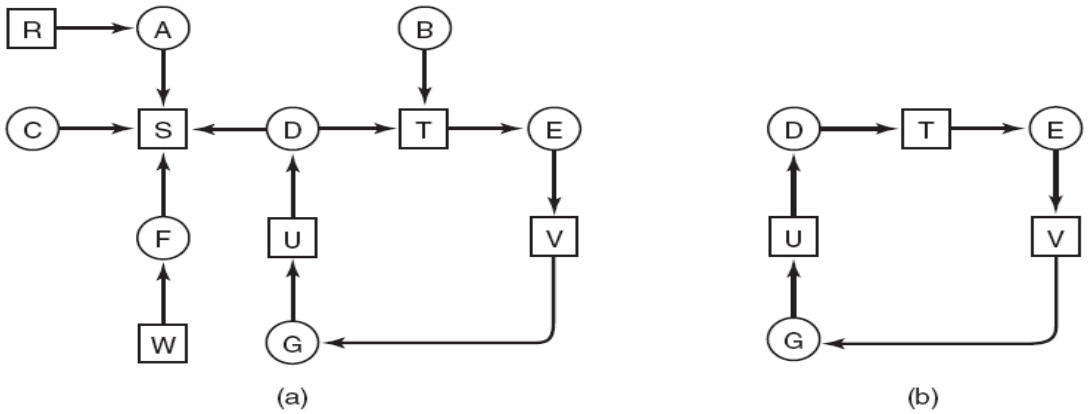


Figure 4-2 (a) A resource graph. (b) A cycle extracted from (a).

- For example as shown in figure 4-2 (a),
  - We are starting from node D.
  - Empty list L = ()
  - Add current node so Empty list = (D).
  - From this node there is one outgoing arc to T so add T to empty list.
  - So Empty list become L = (D, T).
  - Continue this step….so we get empty list as below
  - L = (D, T, E)…… L = (D, T, E, V, G, U, D)
  - In the above step in empty list the node D appears twice, so deadlock.

Deadlock detection for multiple resource

- When multiple copies of some of the resources exist, a matrix-based algorithm can be used for detecting deadlock among n processes.
- Let the number of resource classes be m, with E1 resources of class 1, E2 resources of class 2, and generally, Ei resources of class i (1 < i < m).
- E is the existing resource vector. It gives the total number of instances of each resource in existence.

- Let A be the available resource vector, with Ai giving the number of instances of resource i that are currently available (i.e., unassigned).
- There are two matrices used in the algorithm. C is a current allocation matrix, and R, the request matrix.

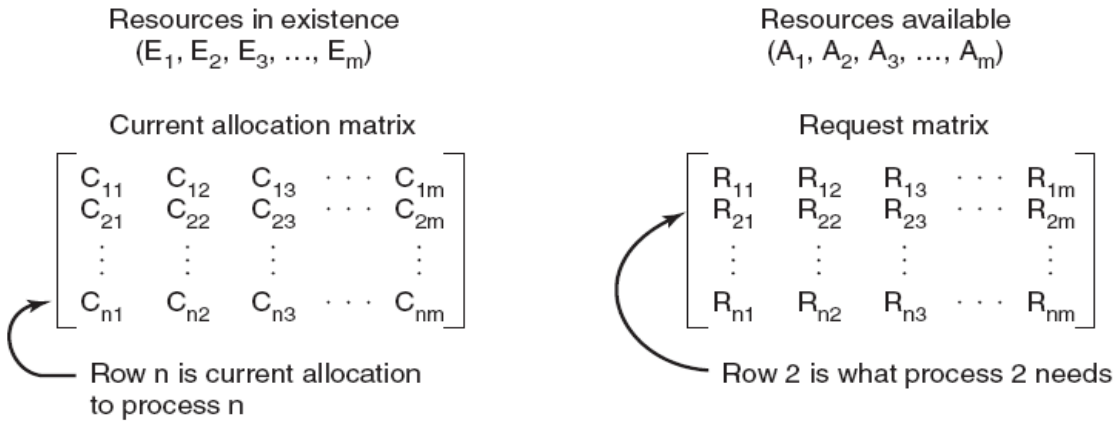


Figure 4-3. The four data structures needed by the deadlock detection algorithm.

- The i-th row of C tells how many instances of each resource class Pi currently holds.
- Thus Cij is the number of instances of resource j that are held by process i.
- Similarly, Rij is the number of instances of resource j that Pi wants.
- These four data structures are shown in Fig. 4-3.
- The deadlock detection algorithm can now be given, as follows.
    1. Look for an unmarked process, Pi, for which the i-th row of R is less than or equal to A.
    2. If such a process is found, add the i-th row of C to A, mark the process, and go back to step 1.
    3. If no such process exists, the algorithm terminates.

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

Example of deadlock detection in multiple resource

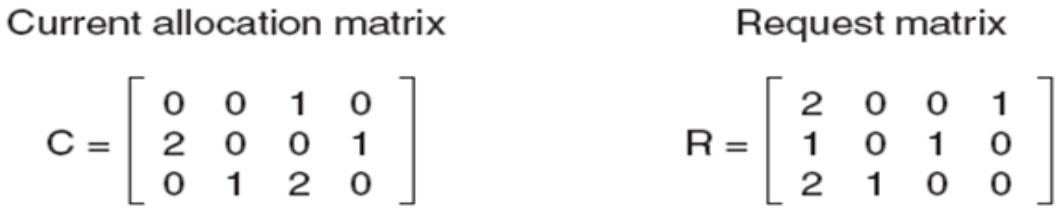


Figure 4-4. An example for the deadlock detection algorithm.

- E is the total no of each resource, C is the number of resources held by each process, A is the number of resources that are available (free), and R is the number of resources still needed by each process to proceed.
- At this moment, we run the algorithm:
- P3 can be satisfied, so P3 completes, returns resources A = (2 2 2 0)
- P2 can be satisfied, so P2 completes, returns resources A = (4 2 2 1)
- P1 can be satisfied, so P1 completes.
- Now all processes are marked, so no deadlock.

Deadlock recovery

- **Recovery through preemption**
  - In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
  - The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource.
  - Recovering this way is frequently difficult or impossible.
  - Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

- **Recovery through rollback**
  - Create a checkpoint.
  - Checkpoint a process periodically.
  - Checkpointing 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, that is, which resources are currently assigned to the process.
- 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 some other resource by starting one of its earlier checkpoints.
- In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes.
- If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

- **Recovery through killing processes**
  - The crudest, but simplest way to break a deadlock is to kill one or more processes.
  - One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue.
  - If this does not help, it can be repeated until the cycle is broken.
  - Alternatively, 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 resources that some process in the cycle needs.

(6) How Resource Trajectories can be helpful in avoiding the deadlock?

- Following example explains how Resource Trajectories can be helpful in avoiding the deadlock.
- Consider 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 A.
- The vertical axis represents the number of instructions executed by process B.
- At I1 process A requests a printer; at I2 it needs a plotter.
- The printer and plotter are released at I3 and I4, respectively.
- Process B needs the plotter from I5 to I7 and the printer from I6 to I8.

- Every point in the diagram represents a joint state of the two processes.
- Initially, the state is at **p**, with neither process having executed any instructions.
- If the scheduler chooses to run process A first, we get to the point **q**, in which process A has executed some number of instructions, but process B has executed none.
- At point **q** the trajectory becomes vertical, indicating that the scheduler has chosen to run process B.
- When process A crosses the I1 line on the path from **r** to **s**, it requests and is granted the printer.

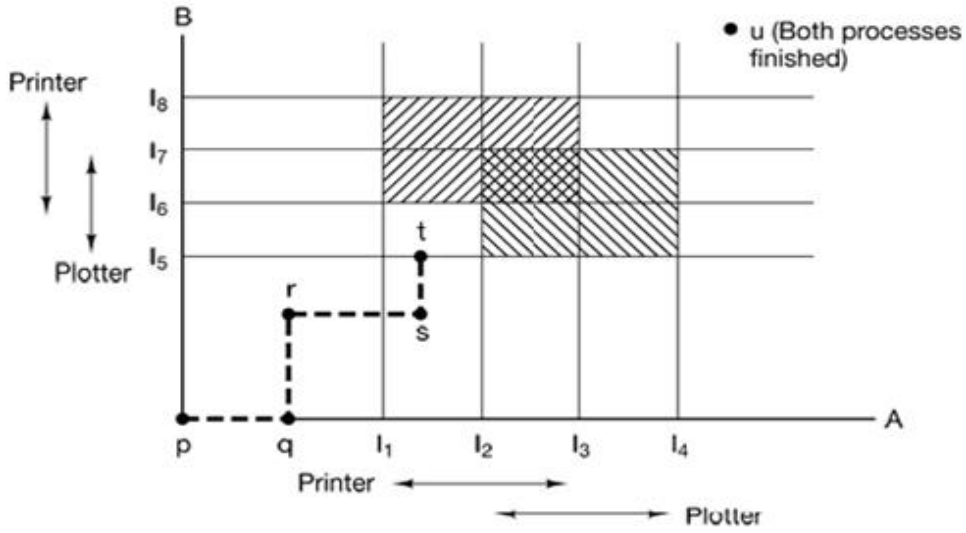


Figure 4-5. Two process resource trajectories.

- When process B reaches point **t**, it requests the plotter.
- The regions that are shaded are especially interesting. The region with lines slanting from southwest to northeast represents both processes having the printer. The mutual exclusion rule makes it impossible to enter this region.
- Similarly, the region shaded the other way represents both processes having the plotter, and is equally impossible.
- If the system ever enters the box bounded by I1 and I2 on the sides and I5 and I6 top and bottom, it will eventually deadlock when it gets to the intersection of I2 and I6.
- At this point process A is requesting the plotter and process B is requesting the printer, and both are already assigned.
- The entire box is unsafe and must not be entered.

- At point **t** the only safe thing to do is run process A until it gets to I4.
- The important thing to see here is at point **t** process B is requesting a resource.
- The system must decide whether to grant it or not.
- If the grant is made, the system will enter an unsafe region and eventually deadlock.
- To avoid the deadlock, B should be suspended until A has requested and released the plotter.

(7) Explain safe and unsafe states with example.

- At any instant of time, there is a current state consisting of existing resource vector *E*, available resource vector *A*, current allocation matrix *C*, and request matrix *R*.
- A state is said to be safe if it is not deadlocked and there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately.
- It is easiest to illustrate this concept by an example using one resource.
- In Fig. 4-6(a) we have a state in which process *A* has 3 instances of the resource but may need as many as 9 eventually.
- Process *B* currently has 2 and may need 4 altogether, later.
- Similarly, process *C* also has 2 but may need an additional 5.

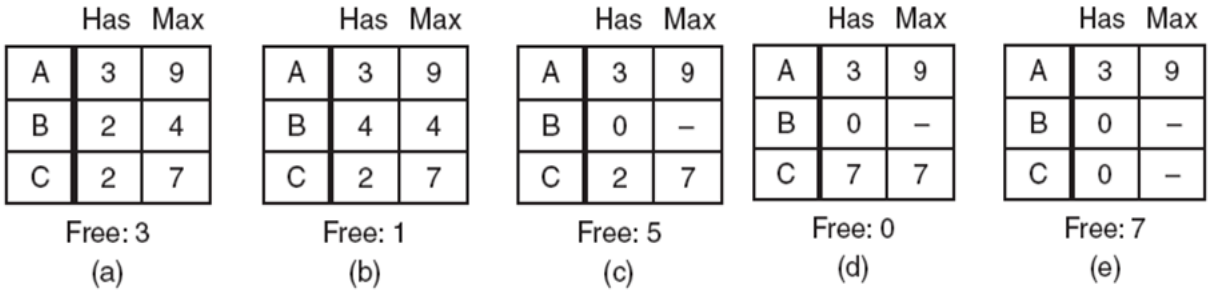


Figure 4-6. Demonstration that the state in (a) is safe.

- A total of 10 instances of the resource exist, so with 7 resources already allocated, there are 3 still free.
- The state of Fig. 4-6 (a) is safe because there exists a sequence of allocations that allows all processes to complete.
- Namely, the scheduler could simply run *B* exclusively, until it asked for and got two

more instances of the resource, leading to the state of Fig. 4-6 (b).

- When *B* completes, we get the state of Fig. 4-6 (c).
- Then the scheduler can run *C* leading eventually to Fig. 4-6 (d). When *C* completes, we get Fig. 4-6 (e).
- Now *A* can get the six instances of the resource it needs and also complete.
- Thus the state of Fig. 4-6 (a) is safe because the system, by careful scheduling, can avoid deadlock.
- Now suppose we have the initial state shown in Fig. 4-7 (a), but this time *A* requests and gets another resource, giving Fig. 4-7 (b).
- Can we find a sequence that is guaranteed to work? Let us try. The scheduler could run *B* until it asked for all its resources, as shown in Fig. 4-7 (c).
- Eventually, *B* completes and we get the situation of Fig. 4-7 (d).
- At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion.
- Thus the allocation decision that moved the system from Fig. 4-7 (a) to Fig. 4-7 (b) results into an unsafe state.
- It is worth noting that an unsafe state is not a deadlocked state.
- The difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

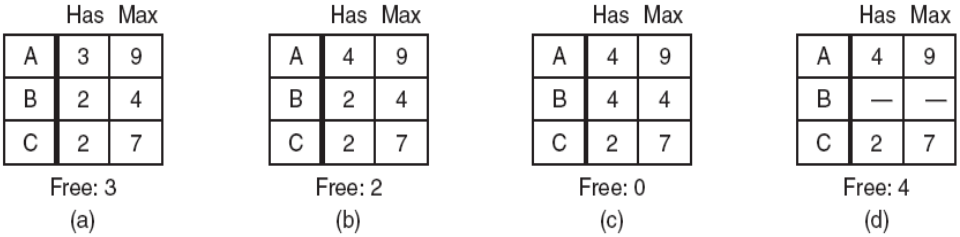


Figure 4-7. Demonstration that the state in (b) is not safe.

(8) Explain the use of Banker's Algorithm for Deadlock Avoidance with illustration.

Deadlock avoidance and bankers algorithm for deadlock avoidance

- Deadlock can be avoided by allocating resources carefully.

- Carefully analyze each resource request to see if it can be safely granted.
- Need an algorithm that can always avoid deadlock by making right choice all the time.

Banker's algorithm for 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. If it does, the request is denied.
- If granting the request leads to a safe state, it is carried out.
- In fig. 4-8 we see four customers, A, B, C, and D, each of whom has been granted a certain number of credit units.
- The banker knows that not all customers will need their maximum credit at a time, so he has reserved only 10 units rather than 22 to service them.

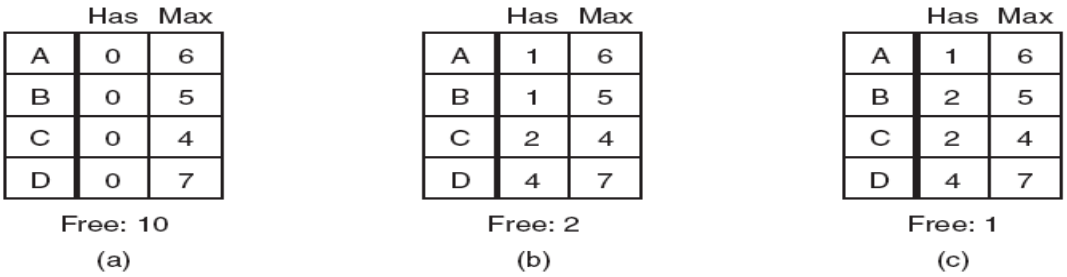


Figure 4-8. Three resource allocation states :(a) Safe. (b) Safe. (c) Unsafe.

- The customers go about their respective businesses, making loan requests from time to time (i.e. asking for resources).
- First if we have situation as per fig 4-8 (a) then it is safe state because with 10 free units one by one all customers can be served.
- Second situation is as shown in fig. 4-8 (b) This state is safe because with two units left (free units), the banker can allocate units to C, thus letting C finish and release all four of his resources.
- With those four free units, the banker can let either D or B have the necessary units,

and so on.

- Consider the third situation, what would happen if a request from B for one more unit were granted as shown in fig. 4-8 (c) then it becomes unsafe state.
- In this situation we have only one free unit and minimum 2 units are required by C. No one can get all resources to complete their work so it is unsafe state.

Banker's algorithm for multiple resource

- The banker's algorithm can be generalized to handle multiple resources.
- In fig. 4-9 we see two matrices. 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 resource each process still needs in order to complete the operation.
- These matrices are labeled as C and R respectively.

Process	Tape drives	Plotters	Printers	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	Printers	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 4-9. The banker's algorithm with multiple resources.

- The three vectors at the right of the figure show the existing (total) resources E, the hold resources P, and the available resources A, respectively.
- From E we see that the system has six tape drives, three plotters, four printers, and two CD ROM drives. Of these, five tape drives, three plotters, two printers, and two CD ROM drives are currently assigned.
- This fact can be seen by adding up the four resource columns in the left hand matrix.
- The available resource vector is simply the difference between what the system has and what is currently in use, i.e. A = E – P.
- The algorithm for checking to see if a state is safe can now be stated.

1. Look for each row in R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion (assuming processes keep all resources until they exit).
2. Assume the process of the row chosen requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all its resources to the vector A.
3. Repeat step 1 and 2 until either all processes are marked terminated (in which case the initial state was safe) or no process is left whose resources needs can be met (in which case there is a deadlock).

- If several processes are eligible to be chosen in step 1, it does not matter which one is selected: the pool of available resources either gets larger, or at worst, stays the same.
- Now let us get back to the example of figure 4-9. The current state is safe. Suppose that process B now makes a request for the printer. This request can be granted because the resulting state is still safe (process D can finish, and then processes A or E, followed by the rest).
- Now imagine that if process D request for 1 printer and 1 CD ROM then there is deadlock.

(9) How deadlock can be prevented? OR Explain deadlock prevention.

Deadlock Prevention

- Deadlock can be prevented by attacking the one of the four conditions that leads to deadlock.

1) Attacking the Mutual Exclusion Condition

- No deadlock if no resource is ever assigned exclusively to a single process.
- Some devices can be spooled such as printer, by spooling printer output; several processes can generate output at the same time.
- Only the printer daemon process uses physical printer.
- Thus deadlock for printer can be eliminated.
- Not all devices can be spooled.
- Principle:

- Avoid assigning a resource when that is not absolutely necessary.
- Try to make sure that as few processes as possible actually claim the resource.

2) Attacking the Hold and Wait Condition

- Require processes to request all their resources before starting execution.
- A process is allowed to run if all resources it needed is available. Otherwise nothing will be allocated and it will just wait.
- Problem with this strategy is that a process may not know required resources at start of run.
- Resource will not be used optimally.
- It also ties up resources other processes could be using.
- Variation: A process must give up all resources before making a new request. Process is then granted all prior resources as well as the new ones only if all required resources are available.
- Problem: what if someone grabs the resources in the meantime how can the processes save its state?

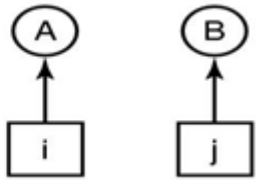
3) Attacking the No Preemption Condition

- This is not a possible option.
- When a process P0 request some resource R which is held by another process P1 then resource R is forcibly taken away from the process P1 and allocated to P0.
- Consider a process holds the printer, halfway through its job; taking the printer away from this process without having any ill effect is not possible.

4) Attacking the Circular Wait Condition

- To provide a global numbering of all the resources.
- Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order.
- A process need not acquire them all at once.
- Circular wait is prevented if a process holding resource n cannot wait for resource m, if m > n.
- No way to complete a cycle.

1. Imagesetter
2. Scanner
3. Plotter
4. Tape drive
5. CD Rom drive



- A process may request 1st a CD ROM drive, then tape drive. But it may not request 1st a plotter, then a Tape drive.
- Resource graph can never have cycle.

(1) Compare multiprogramming with fixed partition & multiprogramming with variable partition with diagram. <u>OR</u>

Explain swapping in memory management.

Swapping

- In practice total amount of memory needed by all the processes is often much more than the available memory.
- Swapping is used to deal with memory overhead.
- **Swapping** consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk.
- The event of copying process from hard disk to main memory is called as **Swapped-in**.
- The event of copying process from main memory to hard disk is called as **Swapped-out**.
- When swapping creates multiple holes in memory, it is possible to combine them all into one big one by moving all the processes downward as far as possible. This technique is called as **memory compaction.**
- Two ways to implement Swapping System
    - Multiprogramming with Fixed partitions.
    - Multiprogramming with dynamic partitions.

Multiprogramming with Fixed partitions

- This method allows multiple processes to execute simultaneously.
- <u>Here memory is divided into fixed sized partitions. Size can be equal or unequal for different partitions.</u>
- Generally unequal partitions are used for better utilizations.
- Each partition can accommodate exactly one process, means only single process can be placed in one partition.
- The partition boundaries are not movable.
- Whenever any program needs to be loaded in memory, a free partition big enough to hold the program is found. This partition will be allocated to that program or process.
- If there is no free partition available of required size, then the process needs to wait. Such process will be put in a queue.
- There are two ways to maintain queue
    - Using multiple Input Queues.
    - Using single Input Queue.

- The disadvantage of sorting the incoming jobs into separate queues becomes apparent when the queue for a large partition is empty but the queue for a small partition is full, as is the case for partitions 1 and 3 in Fig. 5-1(a).
- Here small jobs have to wait to get into memory, even though plenty of memory is free. An alternative organization is to maintain a single queue as in Fig. 5-1(b). Whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run.

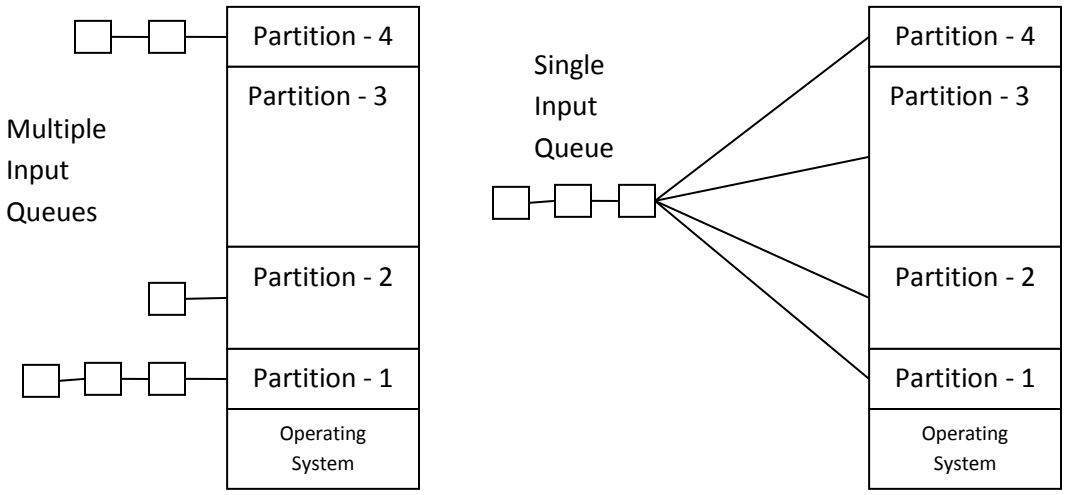


Figure 5-1. (a) Fixed memory partitions with separate input queues for each partition. (b) Fixed memory partitions with a single input queue.

Multiprogramming with dynamic partitions

- This method also allows multiple processes to execute simultaneously.
- Here, memory is shared among operating system and various simultaneously running processes.
- Memory is not divided into any fixed partitions. Also the number of partitions is not fixed. Process is allocated exactly as much memory as it requires.
- Initially, the entire available memory is treated as a single free partition.
- Whenever any process enters in a system, a chunk of free memory big enough to fit the process is found and allocated. The remaining unoccupied space is treated as another free partition.
- If enough free memory is not available to fit the process, process needs to wait until required memory becomes available.

- Whenever any process gets terminate, it releases the space occupied. If the released free space is contiguous to another free partition, both the free partitions are merged together in to single free partition.
- Better utilization of memory than fixed sized size partition.
- This method suffers from **External fragmentation**.

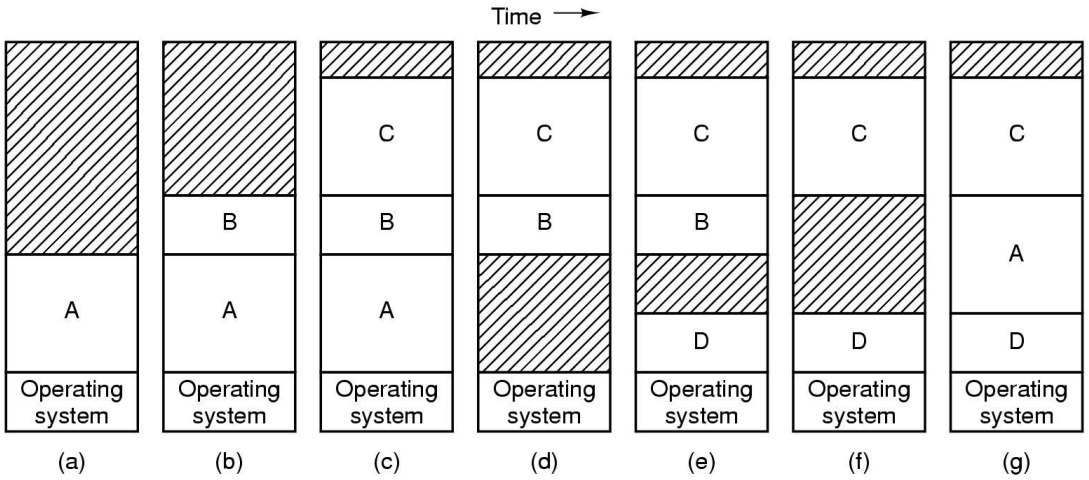


Figure 5-2. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

(2) Explain memory management with Linked list and Bit maps.

- When memory is assigned dynamically, the operating system must manage it.
- In general there are two ways to keep track of memory usage.
  - Memory Management with Bitmap.
  - Memory Management with Linked List.

Memory Management with Bitmap

- With bitmap, memory is divided into allocation units.
- Corresponding to each allocation unit there is a bit in a bitmap.
- Bit is 0 if the unit is free.
- Bit is 1 if unit is occupied.
- The size of allocation unit is an important design issue, the smaller the size, the larger the bitmap.
- The main problem is that when it has been decided to bring a k unit process, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the

map.

- Searching a bitmap for a run of a given length is a slow operation.
- Figure 5-3 shows part of memory and the corresponding bitmap.

Memory Management with Linked List

- Another way to keep track of memory is to maintain a linked list of allocated and free memory segments, where segment either contains a process or is an empty hole between two processes.
- The memory of Fig. 5-3(a) is represented in Fig. 5-3(c) as a linked list of segments.
- Each entry in the list specifies a hole (H) or process (P), the address at which it starts the length and a pointer to the next entry.

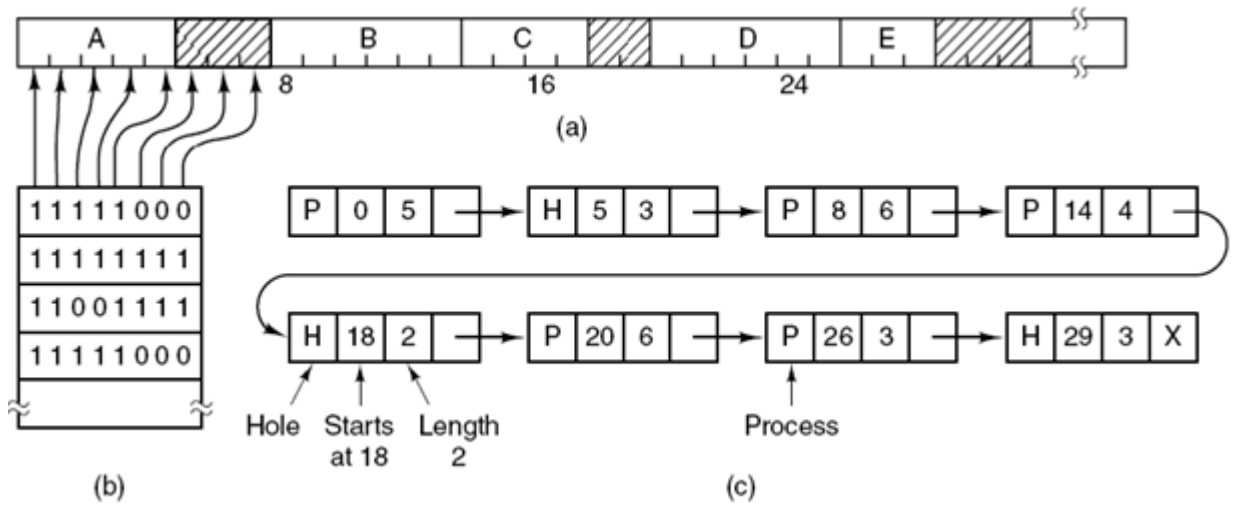


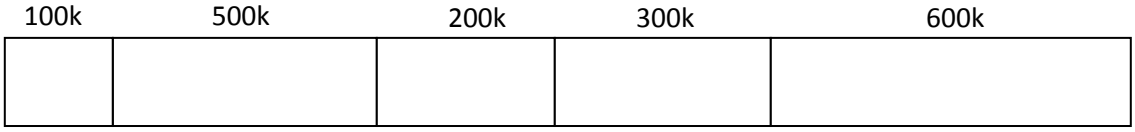
Figure 5-3. (a) A part of memory with five processes and three holes. The tick marks show the memory allocation units. The shaded regions (0 in the bitmap) are free. (b) The corresponding bitmap. (c) The same information as a list.

- 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 terminating process normally has two neighbors (except when it is at the very top or bottom of memory).
- These may be either processes or holes, leading to the four combinations of Fig. 5-4. In Fig. 5-4(a) updating the list requires replacing a P by an H. In Fig. 5-4(b) and Fig. 5-4(c) two entries are merged into one, and the list becomes one entry shorter.
- In Fig. 5-4(d), three entries are merged and two items are removed from the list.

Figure 5-4. Four neighbor combinations for the terminating process *X*.

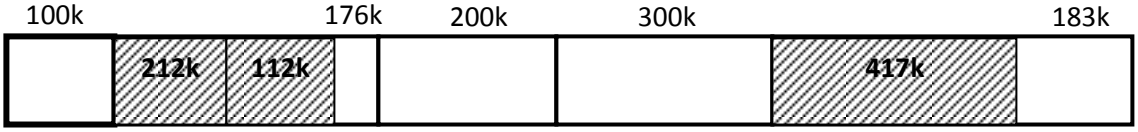
(3) Given memory partition of 100K, 500K, 200K, 300K, and 600K in order, how would each of the First-fit, Next fit, Best-fit and Worst-fit algorithms place the processes of 212K, 417K, 112K and 426K in order? Which algorithm makes the most efficient use of memory? Show the diagram of memory status in each case.

Given memory Partition is as shown below.



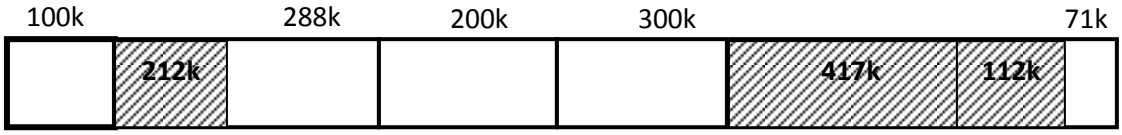
First Fit:

- Search Starts from the starting location of the memory.
- First available hole that is large enough to hold the process is selected for allocation.
- The hole is then broken up into two pieces, one for process and another for unused memory.
- Search time is smaller here.
- Memory loss is higher, as very large hole may be selected for small process.
- Here process of size 426k will not get any partition for allocation.



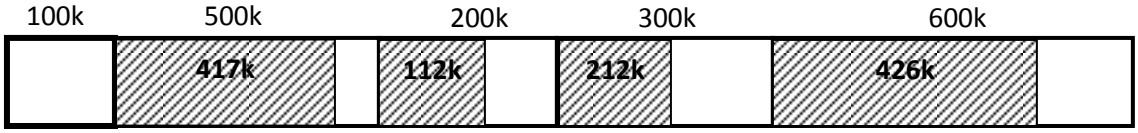
Next Fit:

- Next fit is minor variation of first fit algorithm.
- It works in the same way as first fit, except that it keeps the track of where it is whenever it finds a suitable hole.
- The next time when it is called to find a hole, it starts searching the list from the place where it left off last time.
- Search time is smaller here.
- Here process of size 426k will not get any partition for allocation.



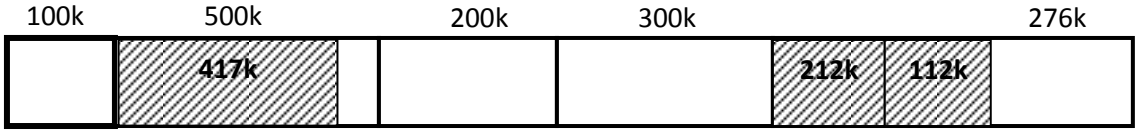
Best Fit

- Entire memory is searched here.
- The smallest hole, which is large enough to hold the process, is selected for allocation.
- Search time is high, as it searches entire memory.
- Memory loss is less. More sensitive to external fragmentation, as it leaves tiny holes into which no process can fit.



Worst Fit

- Entire memory is searched here also. The largest hole, which is largest enough to hold the process, is selected for allocation.
- This algorithm can be used only with dynamic partitioning.
- Here process of size 426k will not get any partition for allocation.



Best fit algorithm gives the efficient allocation of memory.

(4) What is Virtual Memory? Explain.

- The basic idea behind virtual memory is that each program has its own address space, which is broken up into pages.
- Each page is a contiguous range of addresses.
- These pages are mapped onto the physical memory but, to run the program, all pages are not required to be present in the physical memory.
- The operating system keeps those parts of the program currently in use in main memory, and the rest on the disk.
- Virtual memory works fine in a multiprogramming system, with bits and pieces of many programs in memory at once.
- While a program is waiting for part of itself to be brought in, it is waiting for I/O and cannot run, so the CPU can be given to another process.
- In a system using virtual memory, the physical memory is divided into page frames and the virtual address space is divided in into equally-sized partitions called pages.

(5) What is paging? Explain the conversion of virtual address in paging with example.

Paging

- The program generated address is called as **Virtual Addresses** and form the **Virtual Address Space**.
- Most virtual memory systems use a technique called **paging**.
- Virtual address space is divided into fixed-size partitions called **pages**.
- The corresponding units in the physical memory are called as **page frames**.
- The pages and page frames are always of the same size.
- Size of Virtual Address Space is greater than that of Main memory, so instead of loading entire address space in to memory to run the process, MMU copies only required pages into main memory.
- In order to keep the track of pages and page frames, OS maintains a data structure called page table.

The conversion of virtual address to physical address

- When virtual memory is used, the virtual address is presented to an **MMU (Memory Management Unit)** that maps the virtual addresses onto the physical memory addresses.
- A very simple example of how this mapping works is shown in Fig. 5-6.
- In this example, we have a computer generated 16-bit addresses, from 0 up to 64K. These are the virtual addresses.

- However, only 32 KB of physical memory is available, so although 64-KB programs can be written, they cannot be loaded in to memory in their entirety and run.
- A complete copy of a program's core image, up to 64 KB, must be present on the disk. Only required pages are loaded in the physical memory.

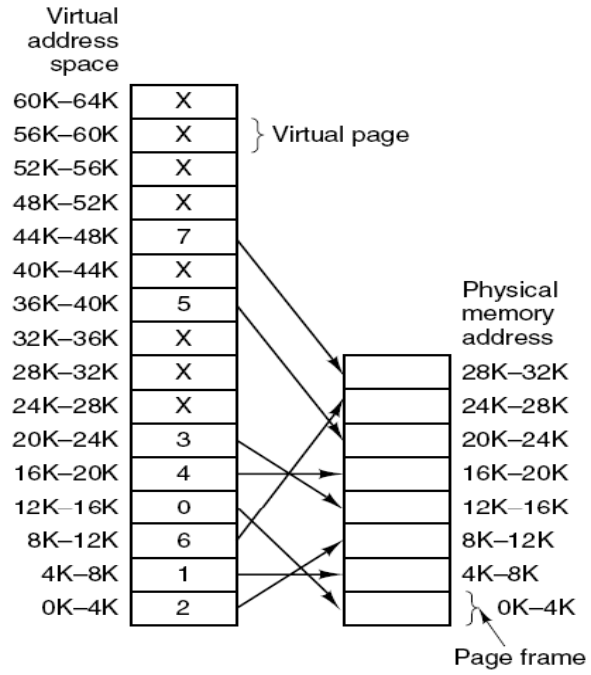


Figure 5-5. The relation between virtual addresses and physical memory addresses is given by the page table.

- With 64 KB of virtual address space and 32 KB of physical memory, we get 16 virtual pages and 8 page frames.
- Transfers between RAM and disk are always in units of a page.
- The virtual address is split into a virtual page number (high order bits) and an offset (low-order bits).
- With a 16-bit address and a 4 - KB page size, the upper 4 bits could specify one of the 16 virtual pages and the lower 12 bits would then specify the byte offset (0 to 4095) within the selected page.
- The virtual page number is used as an index into the Page table.
- If the present/absent bit is 0, it is page-fault; a trap to the operating system is caused to bring required page into main memory.
- If the present/absent bit is 1, required page is there with main memory and page

frame number found in the page table is copied to the higher order bit of the output register along with the offset.

- Together Page frame number and offset creates physical address. Physical Address = Page frame Number + offset of virtual address.

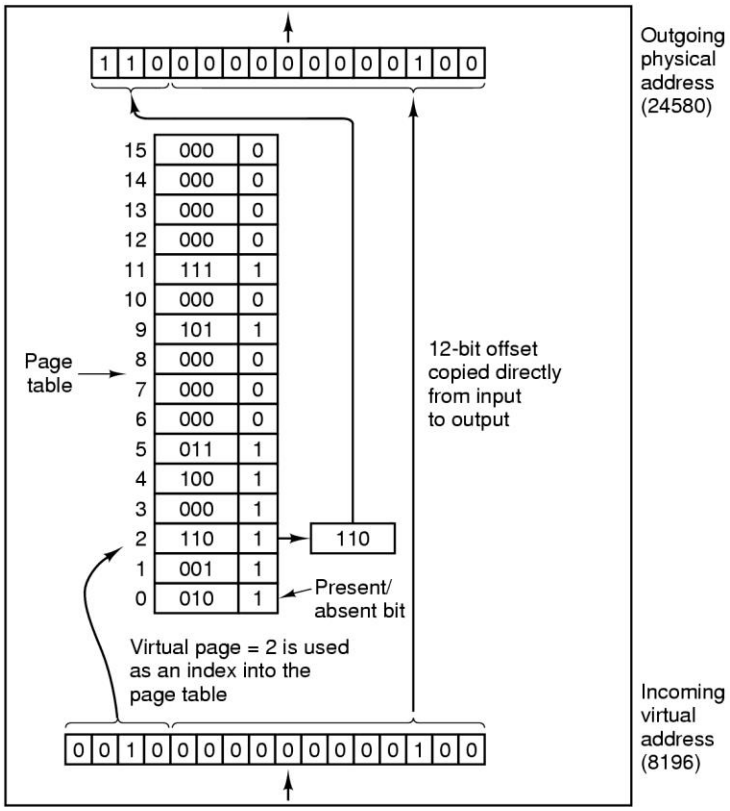


Figure 5-6. The internal operation of the MMU with 16 4-KB pages.

(6) Explain page table in brief. OR
Explain Structure of Page Table.

Page Table

- **Page table** is a data structure which translates virtual address into equivalent physical address.
- The virtual page number is used as an index into the Page table to find the entry for that virtual page and from the Page table physical page frame number is found.
- Thus the purpose of page table is to map virtual pages onto page frames.
- Typical fields of page table are Page frame number, Present / absent bit, Protection

bits, Modified field, Referenced field, Cashing disabled field etc…

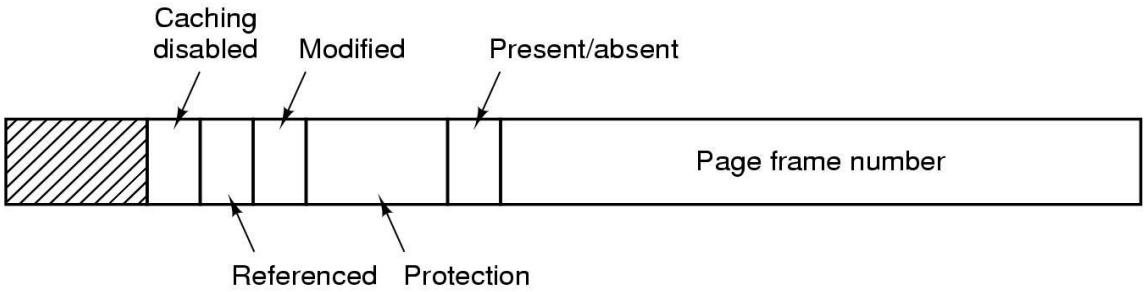


Figure 5-7. A typical page table entry

- Different fields of typical page table are as below
- **Page frame Number:** This field shows the corresponding physical page frame number for a particular virtual page.
- **Present/Absent bit:** If this bit is 1, the entry is valid and can be used, if it is 0 the virtual page to which the entry belongs is not currently in memory. Accessing page table entry with this bit set to 0 causes a page fault.
- **The Protection bits:** This tells what kind of access is permitted. In the simplest form, this field contains 1 bit, with 0 for read/write and 1 for read only.
- **Modified bit:** When a page is written to, the hardware sets the modified bit. If the page in memory has been modified, it must be written back to disk. This bit is also called as dirty bit as it reflects the page's state.
- **Referenced bit:** A references bit is set whenever a page is referenced, either for reading or writing. Its value helps operating system in page replacement algorithm.
- **Cashing Disabled bit:** This feature is important for pages that maps onto device registers rather than memory. With this bit cashing can be turned off.

(7) Explain Following Terms.

Demand Paging

- In paging system, processes are started up with none of their pages in memory.
- When CPU tries to fetch the first instruction, it gets page fault, other page faults for global variables and stack usually follow quickly.
- After a while, the process has most of the pages it needs in main memory and it has few page faults.

- This strategy is called **demand paging** because pages are loaded only on demand, not in advance.

Working Set

- The set of pages that a process is currently using is known as **working set.**

Thrashing

- A program causing page faults every few instructions is said to be **thrashing.**

Pre-paging

- 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.
- Loading pages before allowing processes to run is called **pre-paging.**

(8) Explain the various page replacement strategies.

Various page replacement strategies are explained below.

Optimal Page Replacement Algorithm

- The moment a page fault occurs, some set of pages will be in the memory.
- One of these pages will be referenced on the very next instruction.
- Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later.
- Each page can be labeled with the number of instructions that will be executed before that page is first referenced.
- The optimal page algorithm simply says that the page with the highest label should be removed.
- The only problem with this algorithm is that it is unrealizable.
- At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next.

FIFO Page replacement Algorithm

- The first in first out page replacement algorithm is the simplest page replacement algorithm.
- The operating system maintains a list of all pages currently in memory, with the most recently arrived page at the tail and least recent at the head.
- On a page fault, the page at head is removed and the new page is added to the tail.
- When a page replacement is required the oldest page in memory needs to be replaced.
- The performance of the FIFO algorithm is not always good because it may happen

that the page which is the oldest is frequently referred by OS.

- Hence removing the oldest page may create page fault again.

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 page.
- 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 is called **second chance.**
- What second chance is doing is looking for an old page that has not been referenced in the previous clock interval. If all the pages have been referenced, second chance degenerates into pure FIFO.

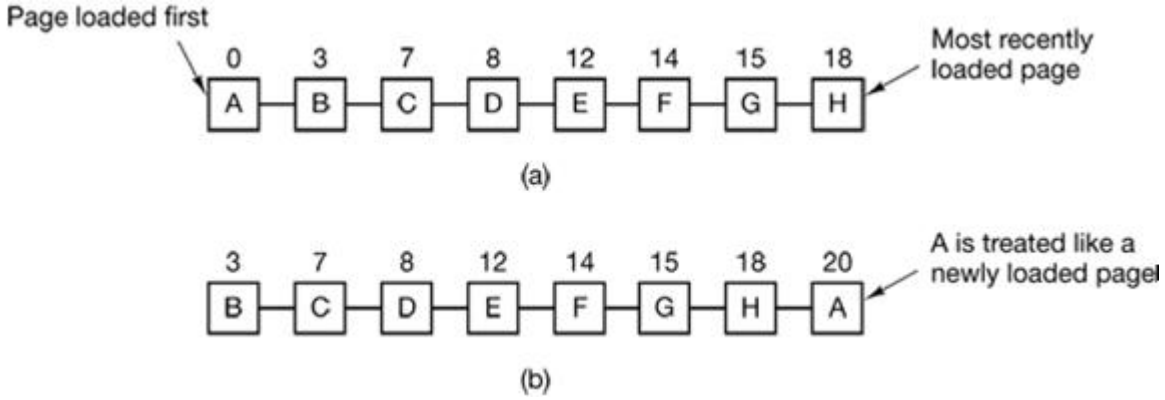


Figure 5-8. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their loading times.

The Clock Page Replacement Algorithm

- Although second chance is a reasonable algorithm, it is unnecessarily inefficient because it is constantly moving pages around on its list.
- A better approach is to keep all the page frames on a circular list in the form of a clock, as shown in Fig. 5-9.
- A hand points to the oldest page.
- When a page fault occurs, the page being pointed to by the hand is inspected.

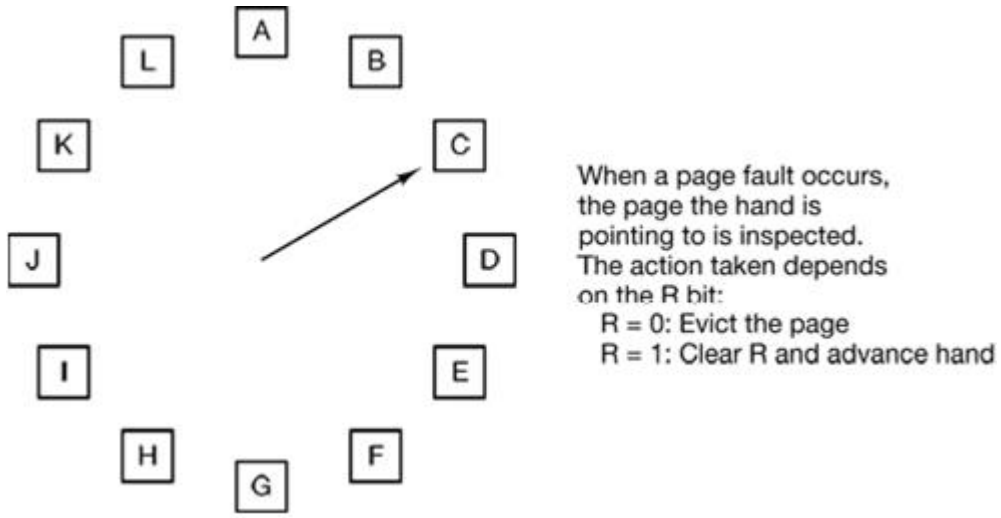


Figure 5-9. The clock page replacement algorithm.

- If its *R* bit is 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. Not surprisingly, this algorithm is called **clock.**
- It differs from second chance only in the implementation.

LRU (Least Recently Used) Page Replacement Algorithm

- A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in last few instructions will probably be heavily used again in next few instructions.
- When page fault occurs, throw out the page that has been used for the longest time. This strategy is called LRU (Least Recently Used) paging.
- To fully implement LRU, 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.
- The 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 operations.
- There are other ways to implement LRU with special hardware.
- For a machine with *n* page frames, the LRU hardware can maintain a matrix of $n \times n$ 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 *k* to 0.
- At any instant, 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. 5-11 for four page frames and page references in the order 0 1 2 3 2 1 0 3 2 3.
- After page 0 is referenced, we have the situation of Fig. 5-11(a).
- After page 1 is reference, we have the situation of Fig. 5-11(b), and so forth.
- As shown in Fig. 5-11(j), if page fault occurs at that moment, the page from the page frame 1 with lowest binary value will be replaced.

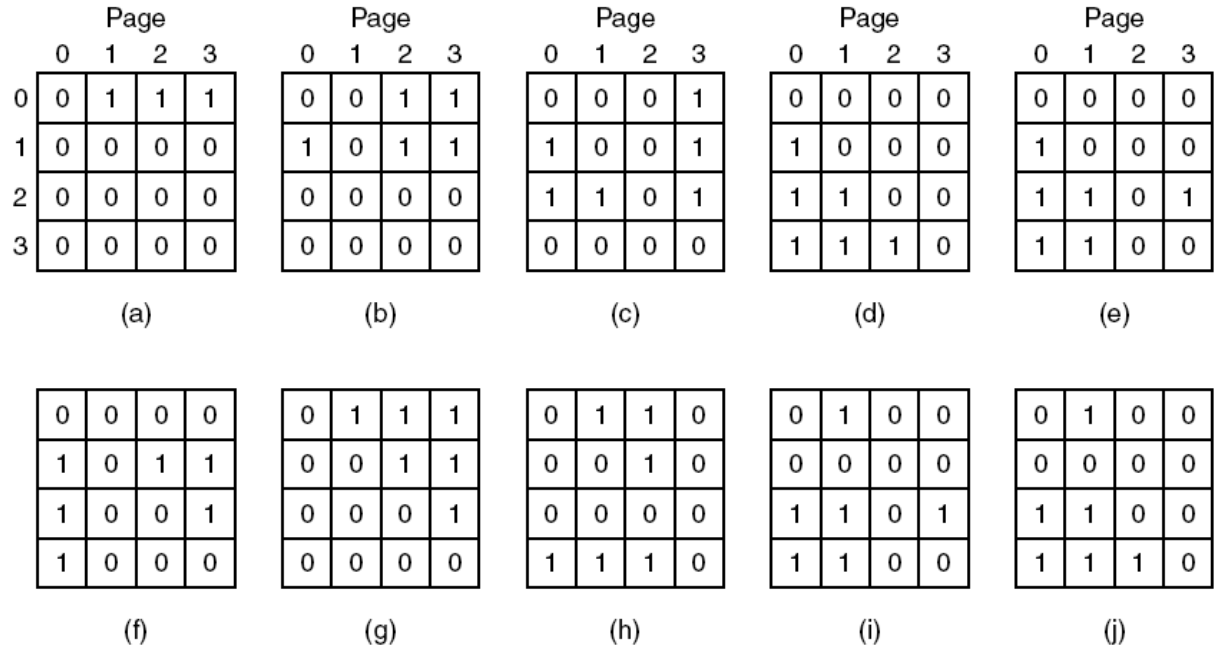


Figure 5-11. LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

NRU (Not Recently Used)

- NRU makes approximation to replace the page based on R and M bits.
- When a process is started up, both page bits for all pages are set to 0 by operating system.
- Periodically, the R bit is cleared, to distinguish pages that have not been referenced recently from those that have been.

- When page fault occurs, the operating system inspects all the pages and divide them into 4 categories based on current values of their R and M bits
  - Case 0 : not referenced, not modified
  - Case 1 : not referenced, modified
  - Case 2 : referenced, not modified
  - Case 3 : referenced, modified
- The NRU (Not Recently Used) algorithm removes a page at random from the lowest numbered nonempty class.
- In given example,
  - Page-0 is of class-2 (referenced, not modified)
  - Page-1 is of class-1 (not referenced, modified)
  - Page-2 is of class-0 (not referenced, not modified)
  - Page-3 is of class-3 (referenced, modified)

So lowest class page-2 needs to be replaced by NRU.

(9) Compare optimal, LRU and FIFO page replacement algorithms with illustration. OR

For the Page Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 Calculate the Page Faults applying (i) Optimal (ii) LRU and (iii) FIFO, Page Replacement Algorithms for a Memory with three frames.

Optimal Page Replacement Algorithm

- The moment a page fault occurs, some set of pages will be in the memory.
- One of these pages will be referenced on the very next instruction (the page containing that instruction).
- Other pages may not be referenced until 10, 100, or perhaps 1000 instructions later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced.
- The optimal page algorithm simply 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 not be used for 6 million instructions, removing the former pushes the page fault that will fetch it back as far into the future as possible.
- The only problem with this algorithm is that it is unrealizable.
- At the time of the page fault, the operating system has no way of knowing when each of the pages will be referenced next.

FIFO Page replacement Algorithm

- The first in first out page replacement algorithm is the simplest page replacement algorithm.
- The operating system maintains a list of all pages currently in memory, with the most recent arrived page at the tail and least recent at the head.
- On a page fault, the page at head is removed and the new page is added to the tail.
- When a page replacement is required the oldest page in memory needs to be replaced.
- The performance of the FIFO algorithm is not always good because it may happen that the page which is the oldest is frequently referred by OS.
- Hence removing the oldest page may create page fault again.

Page Requests	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
Frame 1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
Frame 2		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
Frame 3			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
Page Faults (15)	F	F	F	F		F	F	F	F	F	F			F	F			F	F	F

LRU (Least Recently Used) Page Replacement Algorithm

- A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in last few instructions will probably be heavily used again in next few instructions.
- When page fault occurs, throw out the page that has been used for the longest time. This strategy is called LRU (Least Recently Used) paging.

Page Requests	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**4**	**2**	**3**	**0**	**3**	**2**	**1**	**2**	**0**	**1**	**7**	**0**	**1**
Frame 1	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
Frame 2		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
Frame 3			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
Page Faults (12)	F	F	F	F		F		F	F	F	F			F		F		F		

(10) A computer has four page frames. The time of loading, time of last access and the

R and M bit for each page given below. Which page NRU, FIFO, LRU will replace.

Page	Loaded	Last Ref.	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

FIFO

- Page which is arrived first needs to be removed first, so here **Page – 3** having minimum loading time so as per FIFO page-3 needs to replace.

LRU (Least Recently Used)

- When page fault occurs, throw out the page that has been used for the longest time.
- Page page-1 is not used for the long time from all four, so LRU suggest replacing **page-1**.

NRU (Not Recently Used)

- NRU make approximation to replace based on R and M bits.
- When a process is started up, both page bits for all pages are set to 0 by operating system.
- Periodically, the R bit is cleared, to distinguish pages that have not been referenced recently from those that have been
- When page fault occurs, the operating system inspects all the pages and divide them into 4 categories based on current values of their R and M bits
  - Case 0 : not referenced, not modified
  - Case 1 : not referenced, modified
  - Case 2 : referenced, not modified
  - Case 3 : referenced, modified
- The NRU (Not Recently Used) algorithm removed a page at random from the lowest nonempty class.
- In given example
  - Page-0 is of class-2 (referenced, not modified)
  - Page-1 is of class-1 (not referenced, modified)
  - Page-2 is of class-0 (not referenced, not modified)
  - Page-3 is of class-3 (referenced, modified)
  - So lowest class **page-2** needs to be replaced by NRU

(11) What is page fault handling? How OS handles page fault handling?

The sequence of events is as follows:

1. The hardware traps to the kernel, saving the program counter on the stack.

2. An assembly code routine is started to save the general registers and other volatile information, to keep the operating system from destroying it.

3. The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed. Often one of the hardware registers contains this information. If not, the operating system must retrieve the program counter, fetch the instruction, and parse it in software to figure out what it was doing when the fault hit.

4. Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection consistent with the access. If not, the process is sent a signal or killed. If the address is valid and no protection fault has occurred, the system checks to see if a page frame is free. It no frames are free, the page replacement algorithm is run to select a victim.

5. If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place, suspending the faulting process and letting another one run until the disk transfer has completed. In any event, the frame is marked as busy to prevent it from being used for another purpose.

6. As soon as the page frame is clean (either immediately or after it is written to disk), the operating system looks up the disk address where the needed page is, and schedules a disk operation to bring it in. While the page is being loaded, the faulting process is still suspended and another user process is run, if one is available.

7. When the disk interrupt indicates that the page has arrived, the page tables are updated to reflect its position, and the frame is marked as being in normal state.

8. The faulting instruction is backed up to the state it had when it began and the program counter is reset to point to that instruction.

9. The faulting process is scheduled, and the operating system returns to the assembly language routine that called it.

10. This routine reloads the registers and other state information and returns to user space to continue execution, as if no fault had occurred.

(12) What is Segmentation? Explain.

- Virtual memory is one-dimensional in nature because the virtual address space go from 0 to some maximum address, one address after another.
- There are many problems where separate virtual address space for different entities of process is much better.

Segment

- A general solution is to provide the machine with many independent address spaces called **segments**.
- Each segment consists of a linear sequence of addresses, from 0 to the maximum allowed.
- Different Segments may have different lengths.
- Segment lengths may change during execution.
- Different Segments can grow or shrink independently without affecting each other.
- To specify address in these segments or two dimensional memories, the program must supply a two part address, a segment number and address within the segment.

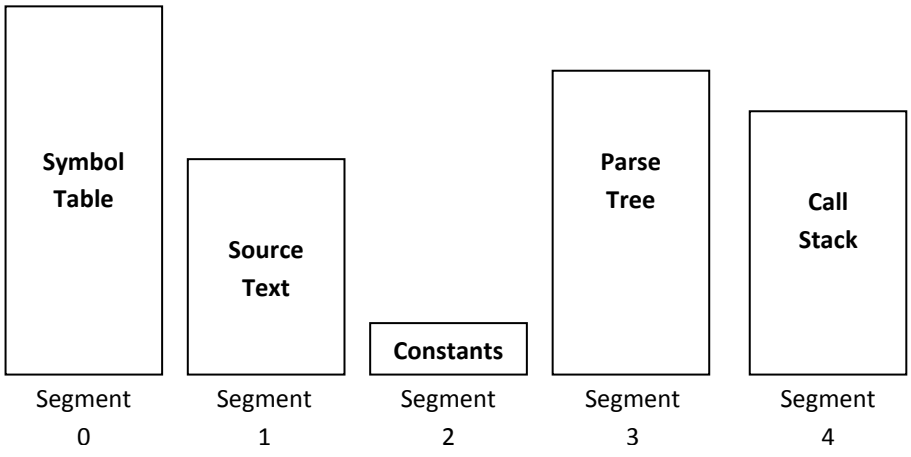
Advantages of Segmentation

- Segmentation provides simplified data structure to handle growing and shrinking of data.
- If each procedure occupies a separate segment, with address 0 as its starting address, the linking of procedures compiled separately is simplified.
- If the procedure in segment n is modified and recompiled, no other procedures need to be changed.
- Segmentation also facilitates sharing procedures or data between several processes. A common example is shared library.
- Each segment forms a logical entity of which the programmer is aware, such as procedure, or an array, or a stack, different segments can have different kind of protection.

- In paging system programmer is unaware of the content of pages, but in segmentation user knows in advance the content of the segments.

Paged Segmentation

- Generally in segmented memory swapping done entirely in segment wise. Entire segment get swapped-out or swapped-in.
- If segments are large, it may be inconvenient or even impossible to keep them in main memory in their entirely.
- Idea of paging the segments is practical one, so only those pages from segments that are actually needed have to be around, these concept dividing segments into pages is called **paged segmentation.**



Different Logical Segments of Compiler having five tables

Figure 5-12. A segmented memory allows each table to grow or shrink independently

- Here, an example of MULTICS is considered to understand segmentation with paging.
- Segmented memory Virtual Address consists of two parts: the segment number and address within segment, the address within segment is further divided into a page number and a word within a page.
- Program has a segment table, with one entry per segment, its entry contains pointer to its page table.
- When memory reference occurs, the following steps occur…
    - The segment number used to find segment descriptor.
    - Check is made to see if the segment's page table is in memory.
        - If not, segment fault occurs.

- If there is a protection violation, a fault (trap) occurs.

- Page table entry for the requested virtual page examined.
  - If the page itself is not in memory, a page fault is triggered.
  - If it is in memory, the main memory address of the start of the page is extracted from the page table entry
- The offset is added to the page origin to give the main memory address where the word is located.
- The read or store finally takes place.

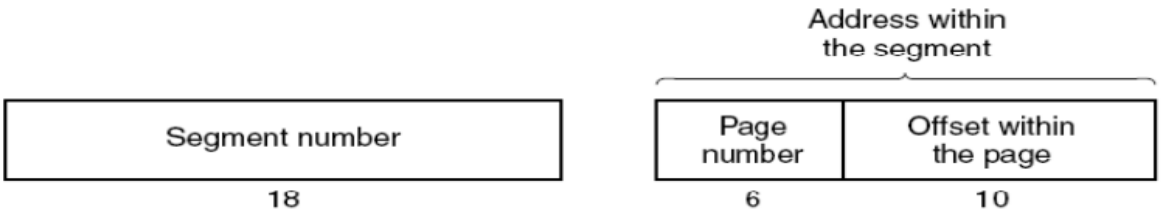


Figure 5-13. A 34-bit MULTICS virtual address.

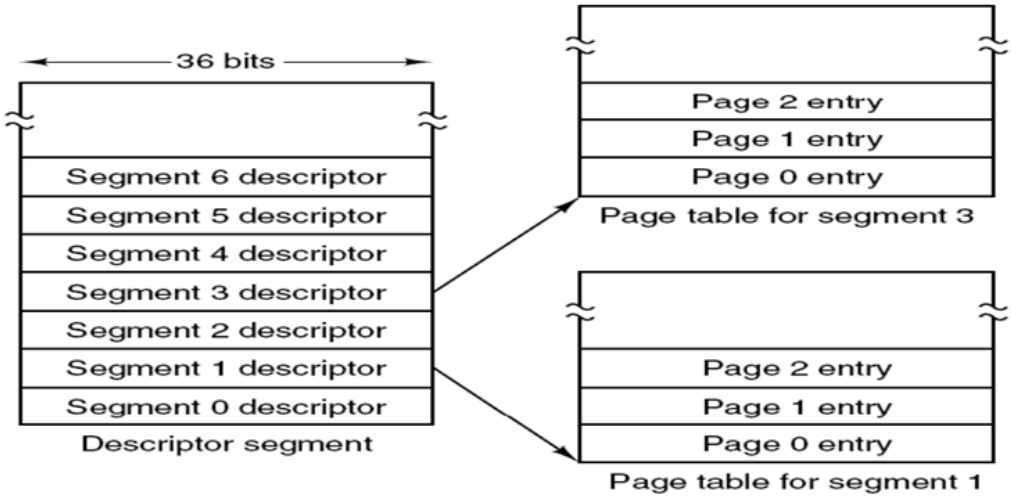


Figure 5-14 (a). The MULTICS virtual memory. The descriptor segment points to the Page tables.

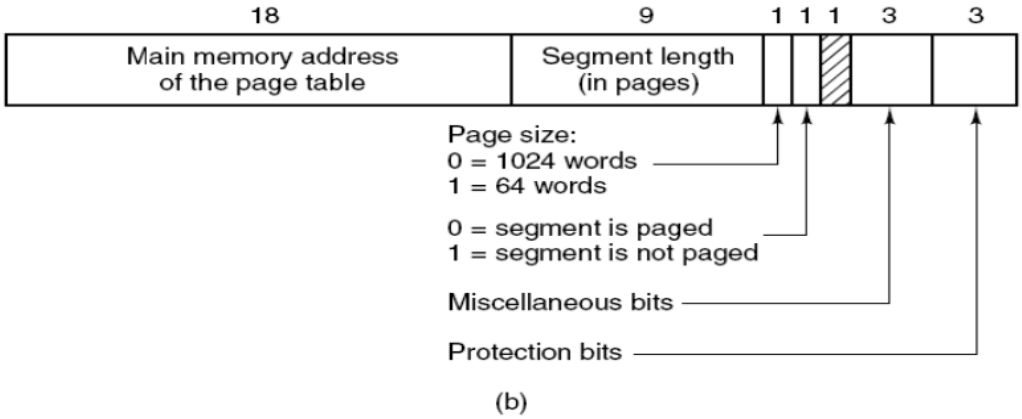


Figure 5-14. The MULTICS virtual memory. (b) A segment descriptor. The numbers are the field lengths.

(13) Discuss the issues associated with paging and solution of it.

- 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 mapping from virtual address to physical address must be fast.

- In the absence of paging, to fetch an instruction requires only one memory reference.
- With paging, one more memory reference is required to access the page table.
- Since execution speed is generally limited by the rate at which the CPU can get instructions and data out of the memory, having to make two memory references per instruction reduce performance by half.
- Most programs access a small number of pages frequently.
- The solution that has been devised 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 TLB (Translation Lookaside Buffer) or sometimes an associative memory, is illustrated in Fig. 5-15.
- It is usually inside the MMU and consists of a small number of entries, eight in this example, but rarely more than 256.

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

Figure 5-15. A TLB to speed up paging.

- Each entry contains information about one page, including the virtual page number, a bit that is set when the page is modified, the protection code (read/write/execute permissions), and the physical page frame in which the page is located.
- These fields have a one-to-one correspondence with the fields in the page table, except for the virtual page number, which is not needed in the page table. Another bit indicates whether the entry is valid (i.e., in use) or not.
- Let us now see how the TLB functions.
- When a virtual address is presented to the MMU for translation, the hardware first checks to see if its virtual page number is present in the TLB by comparing it to all the entries simultaneously (i.e., in parallel).
- Doing so requires special hardware, which all MMUs with TLBs have.
- If a valid match is found and the access does not violate 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 TLB but the instruction is trying to write on a read-only page, a protection fault is generated.
- When the virtual page number is not in the TLB, the MMU detects the miss and does an ordinary page table lookup.
- It then evicts one of the entries from the TLB and replaces it with the page table entry just looked up.
- Thus if that page is used again soon, the second time it will result in a TLB hit rather than a miss.

- When an entry is purged from the TLB, the modified bit is copied back into the page table entry in memory.
- The other values are already there, except the reference bit. When the TLB is loaded from the page table, all the fields are taken from memory.

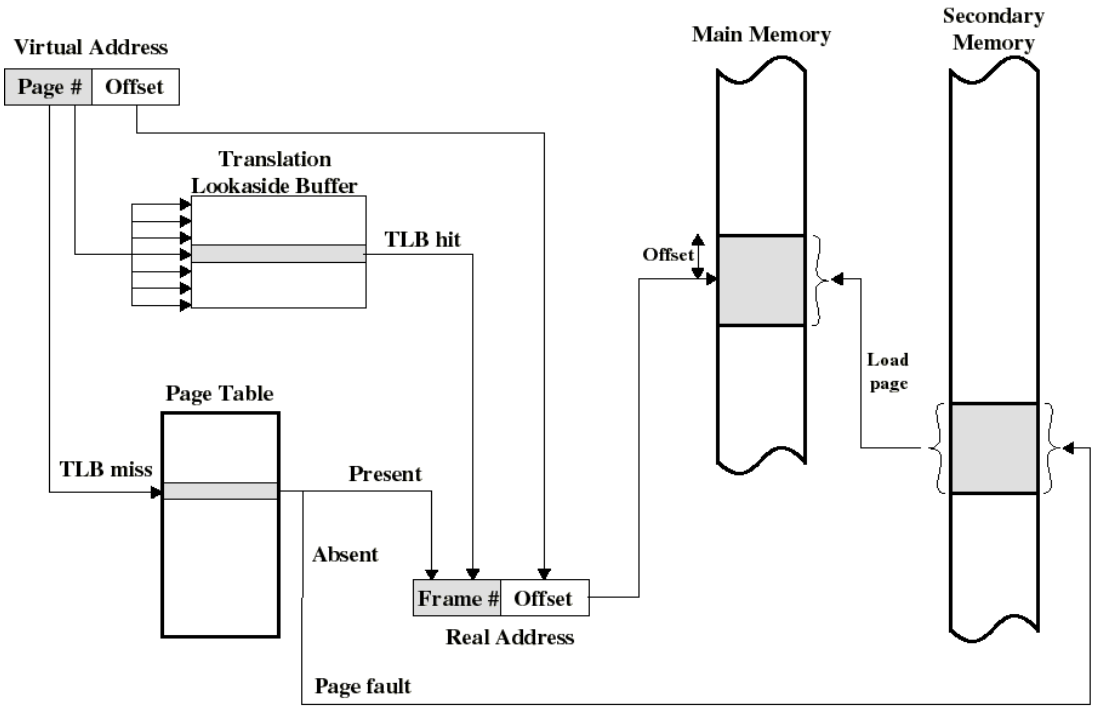


Figure 5-16. Translation Lookaside buffers Example

If the virtual address space is large, the page table will be large.

There are two ways of dealing with very large virtual address spaces.

1. **Multilevel Page Tables**
   - 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.
   - In Fig. 5-17(b) we see how the two-level page table works.
   - On the left we see the top-level page table, with 1024 entries, corresponding to the 10-bit *PT1* field.
   - When a virtual address is presented to the MMU, it first extracts the *PT1* field and uses this value as an index into the top-level page table.

- Each of these 1024 entries in the top-level page table represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 4096 bytes.
- 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 points to the page table for the program text, entry 1 points to the page table for the data, and entry 1023 points to the page table for the stack.
- The other (shaded) entries are not used. The *PT2* field is now used as an index into the selected second-level page table to find the page frame number for the page itself.

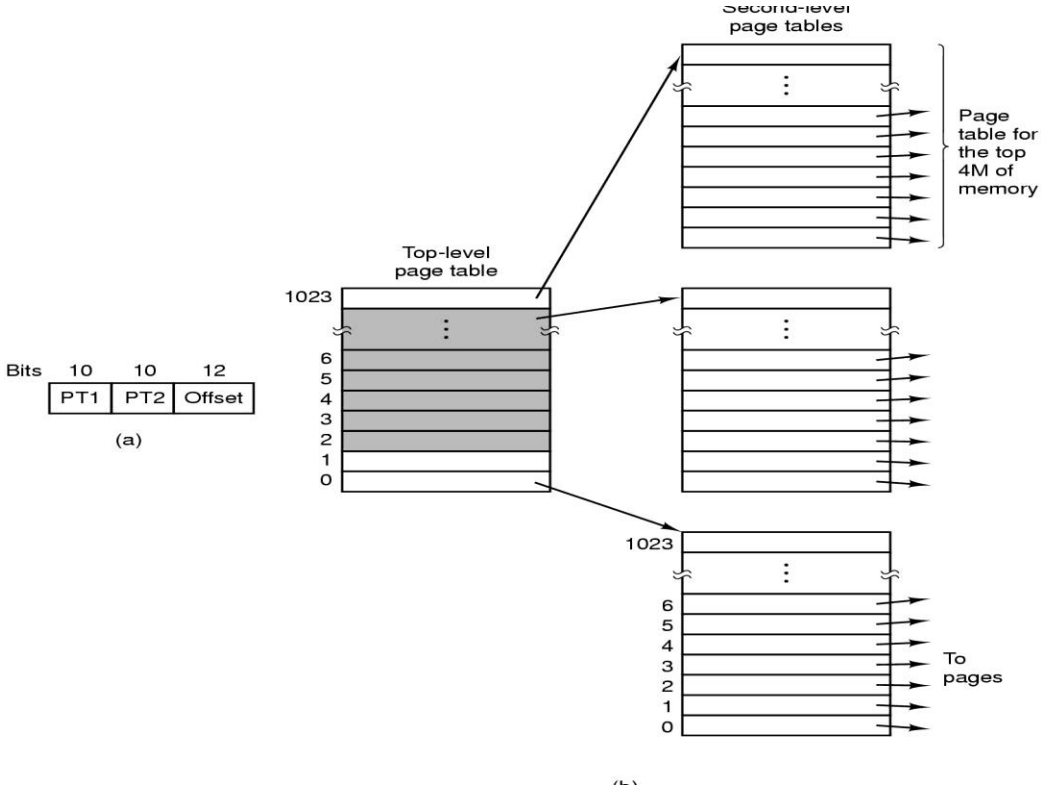


Figure 5-17. (a) A 32-bit address with two page table fields. (b) Two-level page tables.

2. Inverted Page Tables

- An alternative to ever-increasing levels in a paging hierarchy is known as **inverted page tables**.
- In this design, there is one entry per page frame in real memory, rather than one

entry per page of virtual address space.

- For example, with 64-bit virtual addresses, a 4-KB page size, and 4 GB of RAM, an inverted page table requires only 1,048,576 entries.
- The entry keeps track of which (process, virtual page) is located in the page frame.
- Although inverted page tables save lots of space, at least when the virtual address space is much larger than the physical memory, they have a serious down-side: virtual-to-physical translation becomes much harder**.**
- The solution is to make use of the TLB.
- If the TLB can hold all of the heavily used pages, translation can happen just as fast as with regular page tables.
- On a TLB miss, however, the inverted page table has to be searched in software.
- One feasible way to accomplish this search is to have a hash table hashed on the virtual address.

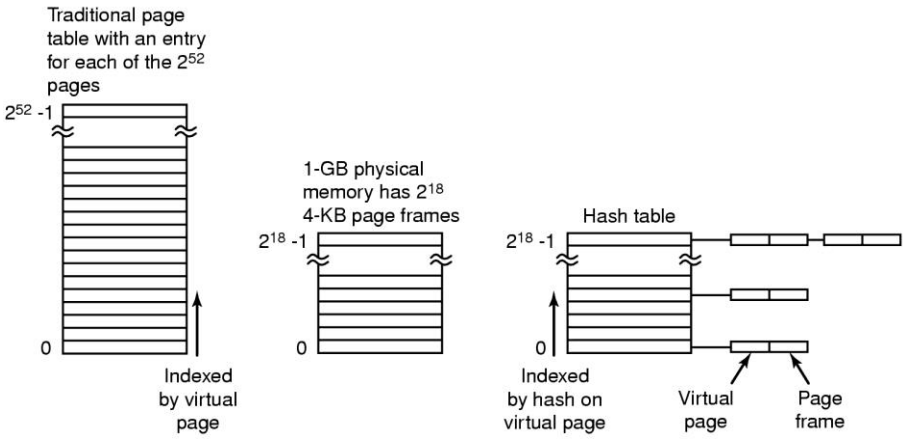


Figure 5-18. Comparison of a traditional page table with an inverted page table.

- All the virtual pages currently in memory that have the same hash value are chained together, as shown in Fig. 5-18.
- 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.
- Once the page frame number has been found, the new (virtual, physical) pair is entered into the TLB.

(14) Compare paging and segmentation.

Paging	Segmentation
Paging was invented to get large address apace without having to buy more physical memory.	Segmentation was invented to allow programs and data to be broken up into logically independent address space and to add sharing and protection.
The programmer does not aware that paging is used.	The programmer is aware that segmentation is used.
Address space is one dimensional.	Many independent address spaces are there.
Procedure and data cannot be distinguished and protected separately.	Procedure and data be distinguished and protected separately.
Change in data or procedure requires compiling entire program.	Change in data or procedure requires compiling only affected segment not entire program.
Sharing of different procedures not available.	Sharing of different procedures available.

Write a short note on Device Controller. OR

(1) Explain Device Controller in brief.

- I/O unit consist of a mechanical component and an electronic component. Electronic component of I/O devices is called the **Device Controller**.
- The mechanical component is device itself.
- The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged.
- Some devices have their own built in controller. Many controllers have two, four or even eight identical devices.
- If the interface between the controller and device is a standard interfacing (ISO, ANSI or IEEE), then companies can make device or controller that fit that interface.
- The controllers are used to convert the serial bit stream into a block of bytes and perform any error correction if necessary.
- The block of bytes is typically first assembled bit by bit in buffer inside controller.
- After verification, the block has been declared to be error free, and then it can be copied to main memory.

(2) Explain Memory-Mapped I/O

- Each device controller has a few registers that are used for communicating with the CPU.
- By writing into these registers, the OS can command the device to deliver data, accept data, switch itself on or off, or perform some action.
- By reading from these registers OS can learn what the device's status is, whether it is prepared to accept a new command and so on.
- There are two ways to communicate with control registers and the device buffers
    - I/O Port.
    - Memory mapped I/O.

I/O Port

- 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 program cannot access it.

Memory-Mapped I/O

- Memory mapped I/O is an approach to map all the control registers into memory space.
- Each control register is assigned a unique memory address to which no memory is assigned, this system is called memory mapped I/O.
- Generally assigned addresses are at the top of the address space.
- When CPU wants to read a word, either from memory or an I/O port, it puts the address it needs on the bus' address lines and then issues a READ signal on bus' control line.
- A second signal line is used to tell whether I/O space or memory space is needed.
- If it is memory space, the memory responds to the request. If it is I/O space, the I/O device responds to the request.

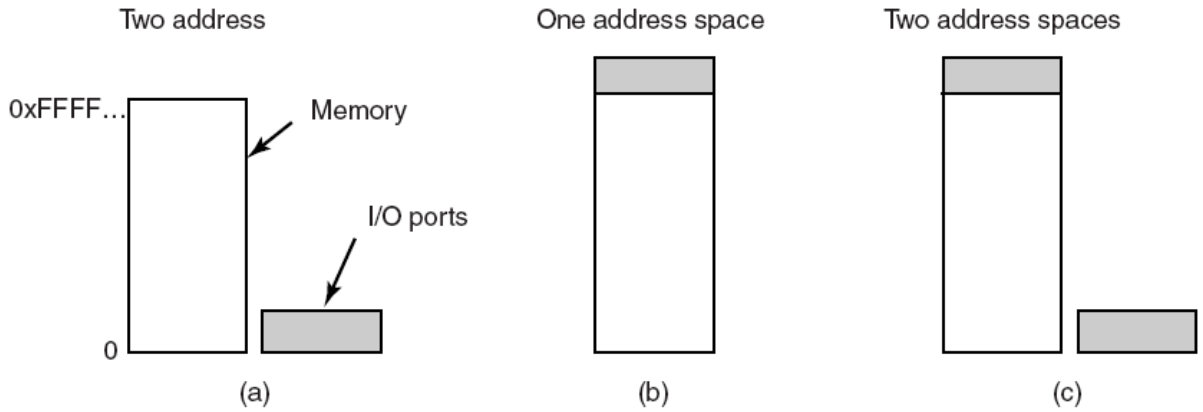


Figure 6-1. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

- **Advantages:**
  - With memory mapped I/O, device control registers are just variables in memory. So with memory mapped I/O device drivers can easily be written in 'C' like languages, no assembly code is required to access registers.
  - No special protection mechanism is needed to keep user processes from performing I/O.
  - With memory-mapped I/O, every instruction that can reference memory can also reference control registers.
- **Disadvantages:**
  - Most computers nowadays have some form of cashing of memory words. Cashing a device control registers would create problems, **cashing needs to be**

disabled in paging system.

- If there is only one address space, then all memory modules and all I/O devices must examine all memory reference to see which ones to respond to.

(3) Explain the Direct Memory Access.

- CPU needs to address the device controllers to exchange data with them.
- CPU can request data from an I/O controller one byte at a time, which is wastage of time.
- So a different scheme called DMA (Direct Memory Access) is used. The operating system can only use DMA if the hardware has DMA controller.
- A DMA controller is available for regulating transfers to multiple devices.
- The DMA controller has separate access to the system bus independent to CPU as shown in figure 6-2. It contains several registers that can be written and read by CPU.
- These registers includes memory address register, a byte count register, one or more control registers.

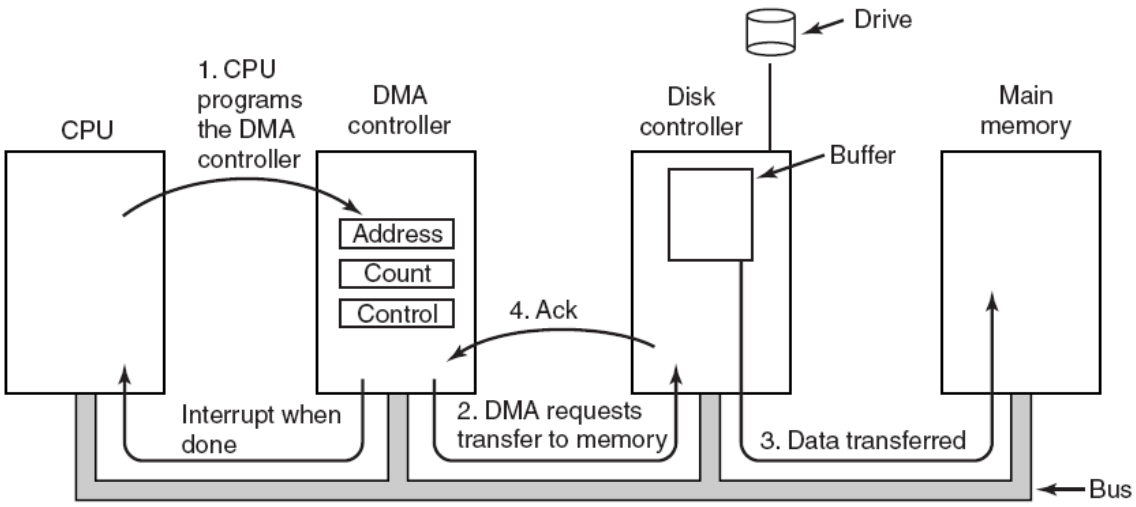


Figure 6-2. Operation of a DMA transfer

Disk read-write without a DMA

- The disk controller reads the block from the drive serially, bit by bit, until the entire block is in the controller's buffer.
- Next, it computes the checksum to verify that no read errors have occurred.

- Then the controller causes an interrupt, so that operating system can read the block from controller's buffer (a byte or a word at a time) by executing a loop.
- After reading every single part of the block from controller device register, the operating system will store them into the main memory.

Disk read-write with DMA

- First the CPU programs the controller by setting its register so DMA knows what to transfer where.
- Simultaneously it issues a command to the disk controller telling it to read data from the disk into its internal buffer and perform the checksum.
- Data transfer is initiated by DMA through a read request from the bus to the disk controller. This read request looks like any other read request, and the disk controller does not know or care whether it came from the CPU or from a DMA controller.
- When writing is completed the disk controller will send an acknowledgement signal to DMA. DMA will then increment the memory address and decrement the byte count.
- Above process will be repeated until the byte count reaches to zero and after that DAM will interrupt the CPU to let it know that the transfer is now completed, so when the operating system starts up it does not have to wait for disk block as it is already there.

The buses can be operated in two modes:

- **Word-at-a-time mode:** Here the DMA requests for the transfer of one word and gets it. If CPU wants the bus at same time then it has to wait. This mechanism is known as Cycle Stealing as the device controller sneaks in and steals an occasional bus cycle from CPU, delaying it slightly.
- **Block mod:** Here the DMA controller tells the device to acquire the bus, issues a series of transfer and then releases the bus. This form of the operation is called Burst mode. It is more efficient then cycle stealing.

Disadvantages of DMA:

- Generally the CPU is much faster than the DMA controller and can do the job much faster so if there is no other work for it to do then CPU needs to wait for the slower DMA.

(4) Explain Goals of I/O Software

- **Device Independence:**
  - It should be possible to write programs that can access any I/O devices without having to specify device in advance.
  - For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.

- **Uniform naming:**
  - Name of file or device should be some specific string or number. It must not depend upon device in any way.
  - In UNIX, all disks can be integrated in file system hierarchy in arbitrary way so user need not be aware of which name corresponds to which device.
  - All files and devices are addressed the same way: by a path name.
- **Error handling:**
  - Error should be handled as close to hardware as possible. If any controller generates error then it tries to solve that error itself. If controller can't solve that error then device driver should handle that error, perhaps by reading all blocks again.
  - Many times when error occur, error solve in lower layer. If lower layer are not able to handle error problem should be told to upper layer.
  - In many cases error recovery can be done at a lower layer without the upper layers even knowing about error.
- **Synchronous versus Asynchronous:**
  - Most of devices are asynchronous device. CPU starts transfer and goes off to do something else until interrupt occurs. I/O Software needs to support both the types of devices.
  - User programs are much easier to write if the I/O operations are blocking.
  - It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs.
- **Buffering:**
  - Data comes in main memory cannot be stored directly. For example data packets come from the network cannot be directly stored in physical memory. Packets have to be put into output buffer for examining them.
  - Some devices have several real-time constraints, so data must be put into output buffer in advance to decouple the rate at which buffer is filled and the

rate at which it is emptied, in order to avoid buffer under runs.

- Buffering involved considerable copying and often has major impact on I/O performance.

(5) Explain device driver in OS.

- Each Controller has some device registers used to give it commands or read out its statues or both.
- For example mouse driver has to accept information about position of pointer and which buttons are currently depressed.
- I/O devices which are plugged with computer have some specific code for controlling them. This code is called the **device driver**.
- Each device driver normally handles one device type, or at most one class of closely related devices.
- Generally device driver is delivered along with the device by device manufacturer. Each driver handles one type of closely related devices.
- In order to access the device's hardware, meaning the controller's registers, device driver should be a part of operation system kernel.
- Device drivers are normally positioned below the rest of Operating System.
- **Functions of device drivers:**
  - Device driver accept abstract read and write requests from device independent software.
  - Device driver must initialize the device if needed. It also controls power requirement and log event.
  - It also checks statues of devices. If it is currently in use then queue the request for latter processing. If device is in idle state then request can be handled now.
  - Controlling device means issuing a sequence of command to it. Device driver is a place where command sequence is determined, depending upon what has to be done.

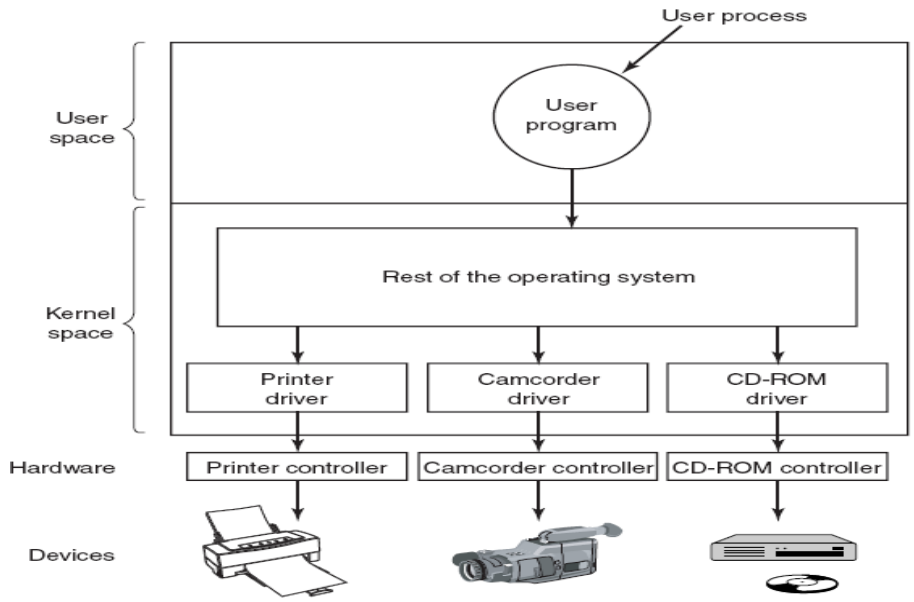


Figure 6-3. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

- Pluggable device can be added or removed while the computer is running. At that time the device driver inform CPU that the user has suddenly removed the device from system.

(6) Explain Device Independent I/O Software.

- Exact boundary between the drivers and the device independent I/O software is system dependent.
- Function of device independent I/O Software
  - Uniform interfacing for device drivers.
  - Buffering.
  - Error Reporting.
  - Allocating and releasing dedicated devices.
  - Providing a device-independent block size.

- **Uniform interfacing for device drivers**
  - A major issue of an OS is how to make all I/O devices and drivers look more or less the same.
  - One aspect of this issue is the interface between the device drivers and the rest of the OS.

- Fig 6-4(a) shows situation in which each device driver has a different interface to OS, it means that interfacing each new driver requires a lot of new programming effort.
- Fig 6-4(b) shows a different design in which all drivers have the same interface.
- Now it becomes much easier to plug in a new driver.
-

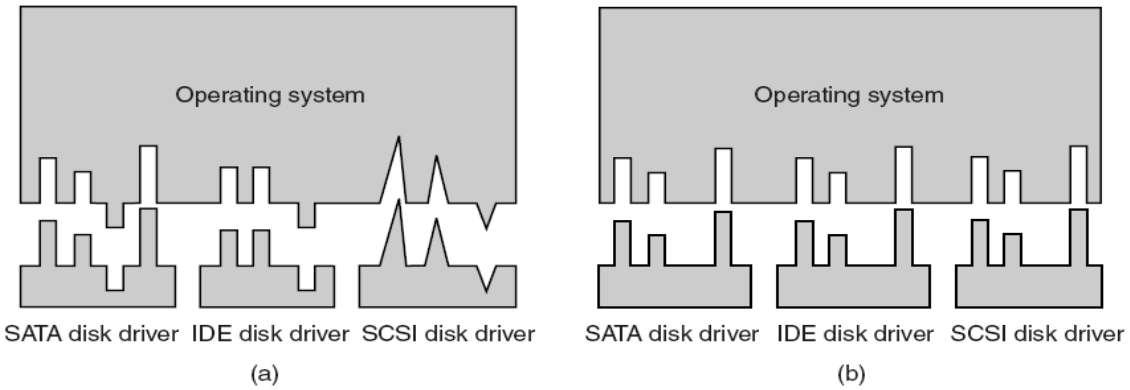


Figure 6-4. (a) Without a standard driver interface. (b) With a standard driver interface.

- Another aspect of having a uniform interface is how I/O devices are named.
- The device independent software takes care of mapping symbolic device names onto the proper driver.
- Protection is one more aspect which is closely related to naming.
- In UNIX and Windows, devices appear in the file system as named objects, so that the usual protection rules for the file are also applied to I/O devices.

- **Buffering**
  - Buffering is also issue, both for block and character devices.
  - In case of a process, which reads the data from the modem, without buffering the user process has to be started up for every incoming character.
  - Allowing a process to run many times for short runs is inefficient, so this design is not a good one. (figure 6-4(a))
  - Buffer in users pace: here user process provides an n-character buffer in user space and does a read of n-characters. (figure 6-4(b))
  - Buffer inside kernel: to create the buffer inside the kernel and interrupt handler is responsible to put the character there. (figure 6-4(c))
  - Two buffers in kernel: the first buffer is used to store characters. When it is full,

it is being copied to user space. During that time the second buffer is used.

- In this way, two buffers take turns.
- This is called double buffering scheme. (figure 6-4(d))

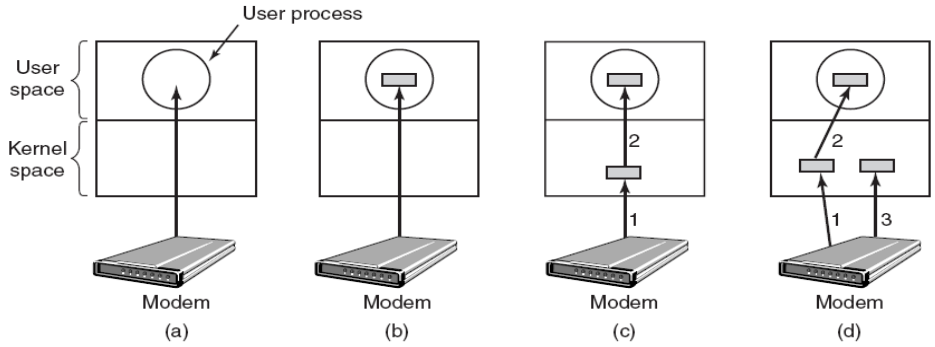


Figure 6-5. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

- **Error reporting**
  - Errors are far more common in the context of I/O than in other context. When they occur, the OS must handle them as best it can.
  - One class of I/O errors is programming errors. These occur when a process asks for something impossible, such as writing to an input device or reading from an output device.
  - The action taken for these errors is, to report an error code back to the caller.
  - Another class of error is the class of actual I/O errors, for example trying to write a disk block that has been damaged.
  - In this case, driver determines what to do and if it does not know the solution then the problem may be passed to the device independent software.

- **Allocating and releasing dedicated devices**
  - Some devices such as CD-ROM recorders can be used only by a single process at any given moment.
  - A mechanism for requesting and releasing dedicated devices is required.
  - An attempt to acquire a device that is not available blocks the caller instead of failing.
  - Blocked processes are put on a queue, sooner or later the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution.

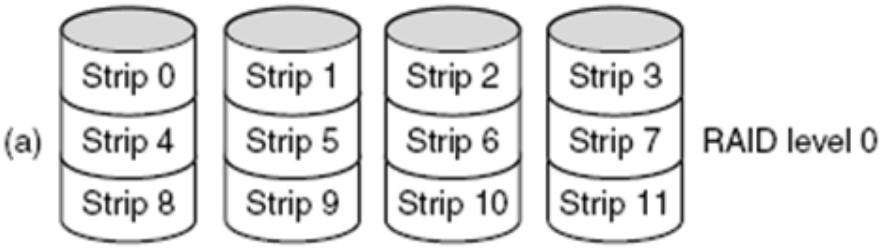
- **Device-independent block size**
  - Different disks may have different sector sizes.
  - It is up to the device independent I/O software to hide this fact and provide a uniform block size to higher layers.

(7) Explain Different RAID levels (RAID – Redundant Array of Independent Disks)

- Parallel processing is being used more and more to speed up CPU performance, parallel I/O can be a good idea.
- The basic idea behind RAID is to install a box full of disks next to the computer, replace the disk controller card with RAID controller.
- All RAID have the property that the data are distributed over drives, to allow parallel operation.

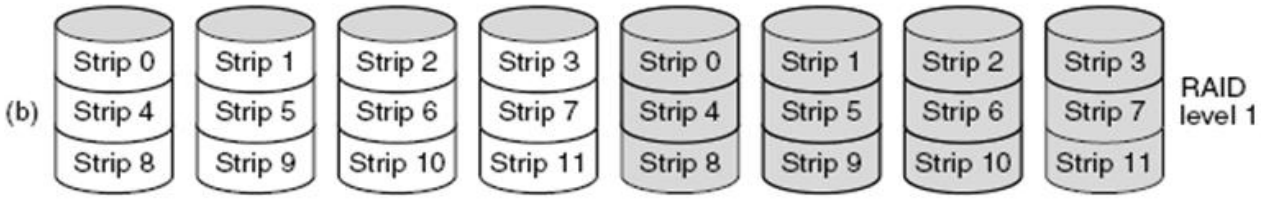
RAID level 0

- It consists of viewing the virtual single disk, as being divided up into strips of k sectors, and distributing data over multiple drives as shown in the figure is called striping.



- The RAID level 0 organization writes consecutive stripes over the drives in round-robin fashion.
- Command to read data block, consisting of four consecutive strips, RAID controller will break, this command up into four separate commands, one for each of the four disks and have them to operate in parallel.
- Thus we have parallel I/O without the software knowing about it.
- One disadvantage of RAID level 0 is that the reliability is potentially worse than having SLED (**S**ingle **L**arge **E**xpensive **D**isk).

RAID level 1



- RAID level 1 is a true RAID.
- It duplicates all the disks, so in figure there are four primary disks and four backup disks.
- On a write operation every strip is written twice.
- On a read operation either copy can be used, distributing the load over more drives.
- Write performance is no better than for a single drive.
- Read performance is twice as good.
- Fault tolerance is excellent: if a drive crashes, a copy is simply used instead.
- Recovery consists of simply installing a new drive and copying the entire backup drive to it.

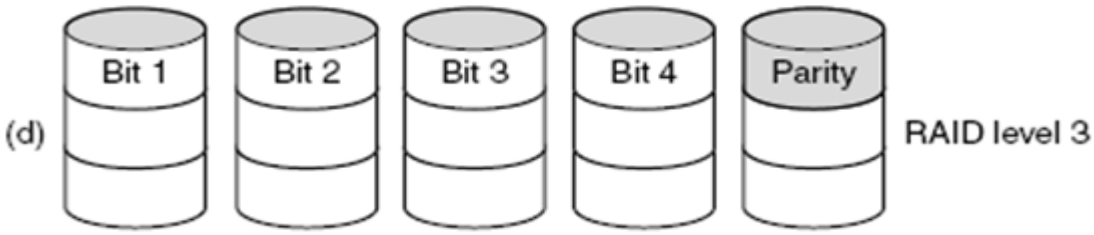
RAID level 2

- RAID level 2 works on word, possibly even a byte basis.
- Imagine splitting each byte into a pair of 4-bit nibbles, then adding Hamming code to each one to form a 7-bit word, of which bit 1, 2 and 4 were parity bits, as shown in figure below.

- In this RAID level 2 each of seven drives needs synchronized in terms of arm position and rotational position, and then it would be possible to write the 7-bit Hamming coded word over the seven drives, one bit per drive.
- Here, losing one drive did not cause problem, which can be handled by Hamming code on the fly.
- But, on the down side this scheme requires all the drives to be rotationally synchronized and it also asks a lot of controller, since it must do a Hamming checksum

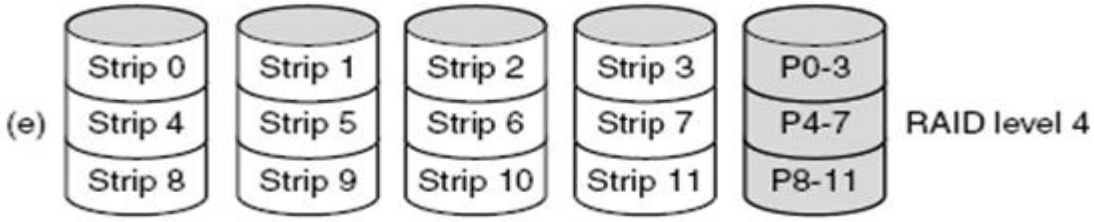
every bit time.

RAID level 3



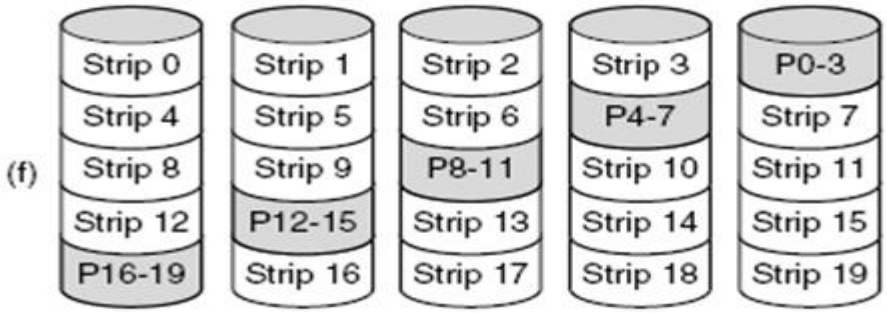
- RAID level 3 is simplified version of RAID level 2, as shown in the above figure.
- Here single parity bit is computed for each data word and written to a parity drive.
- As in RAID level 2 the drives must be exactly synchronized.
- In case of drive crashing, it provides 1-bit error correction since position of each bad bit is known.

RAID level 4



- RAID level 4 works with strips and not individual word with parity.
- They do not require synchronization of drives.
- As shown in the figure RAID level 4 is like RAID level 0, with strip-for-strip parity written onto an extra drive, for example, if each strip is k bytes long, all strips are EXCLUSIVE ORed together, resulting in a parity strip k bytes long.
- If a drive crashes, the lost bytes can be recomputed from the parity drive by reading the entire set of drives.
- This design protects against the loss of a drive but performs poorly for small updates, if one sector is changed, it is necessary to read all the drives in order to recalculate the parity.
- It creates heavy load on parity drive.

RAID level 5



- As with RAID level 4, there is a heavy load in the parity drive, it may become bottleneck.
- This bottleneck can be eliminated in RAID level 5 by distributing the parity bits uniformly over all the drives, round robin fashion, as shown in the above figure.
- In the event of drive crash, reconstructing the contents of the failed drive is complex process.

(8) Disk Arm Scheduling Algorithm.

- The time required to read or write a disk block is determined by three factors:
  1. Seek time (the time to move the arm to the proper cylinder).
  2. Rotational delay (the time for the proper sector to rotate under the head).
  3. Actual data transfer time.
- For most disks, the seek time dominates the other two times, so reducing the mean seek time can improve system performance substantially.
- Various types of disk arm scheduling algorithms are available to decrease mean seek time.
  1. FCSC (First come first serve)
  2. SSTF (Shorted seek time first)
  3. SCAN
  4. C-SCAN
  5. LOOK (Elevator)
  6. C-LOOK

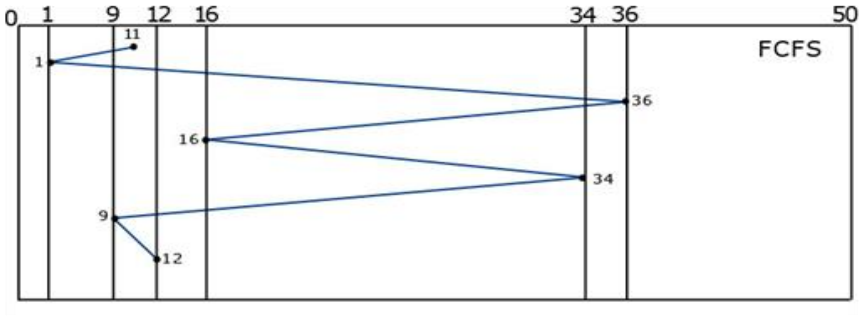
<u>Example</u>: Consider an imaginary disk with 51 cylinders. A request comes in to read a block on cylinder 11. While the seek to cylinder 11 is in progress, new requests come in for cylinders <u>1, 36, 16, 34, 9, and 12</u>, in that order.

Starting from the current head position, what is the total distance (in cylinders)

that the disk arm moves to satisfy all the pending requests, for each of the following disk scheduling Algorithms?

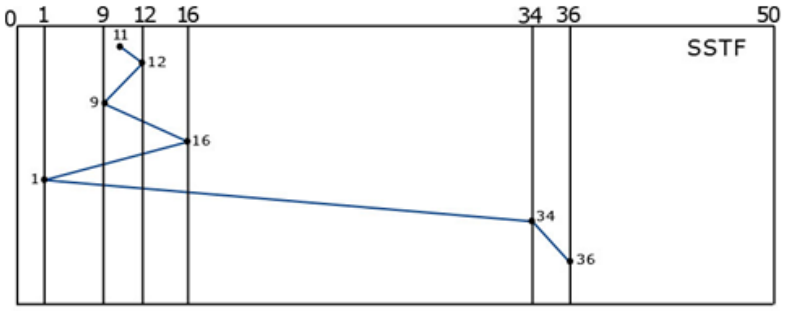
1. FCSC (First come first serve)
2. SSTF (Shorted seek time first)
3. SCAN
4. C-SCAN
5. LOOK (Elevator)
6. C-LOOK

FCSC (First come first serve) Scheduling



- Here requests are served in the order of their arrival.
- In given example disk movement will be **11, 1, 36, 16, 34, 9 and 12** as first come first served.
- Total cylinder movement: (11-1)+ (36-1)+ (36-16) + (34-16) +(34-9)+(9-12) = **111**

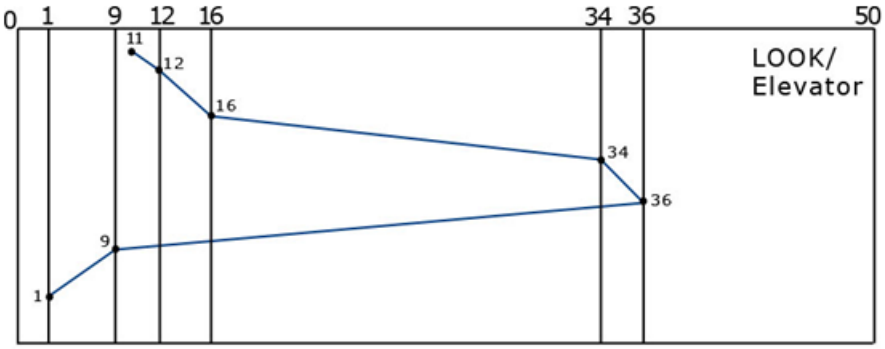
SSTF (Shortest seek time first)



- We can minimize the disk movement by serving the request closest to the current position of the head.
- In given example starting head position is 11, closest to 11 is 12, closest to 12 is 9, and so on.

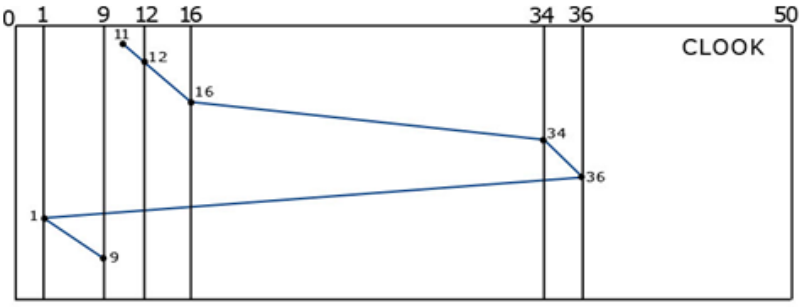
- As per SSTF request will be satisfied in order **11, 12, 9, 16, 1, 34, 36.**
- Total cylinder movement: (12-11) + (12-9) + (16-9) + (16-1) + (34-1) + (36-34) = **61**

LOOK (Elevator) disk arm scheduling



- Keep moving in the same direction until there are no more outstanding requests pending in that direction, then algorithm switches the direction.
- After switching the direction the arm will move to handle any request on the way. Here first go it moves in up direction then goes in down direction.
- This is also called as elevator algorithm.
- In the **elevator algorithm**, the software maintains 1 bit: the current direction bit, which takes the value either *UP* or *DOWN*.
- As per LOOK request will be satisfied in order **11, 12, 16, 34, 36, 9, 1.**
- Total cylinder movement: (12-11) + (16-12) + (34-16) + (36-34) + (36-9) + (9-1)**=60**

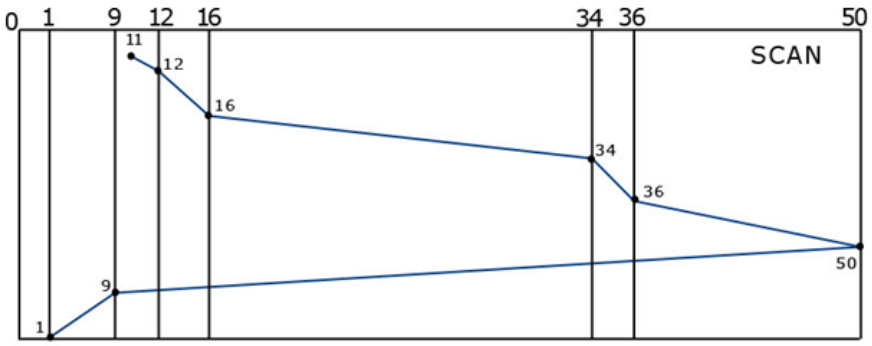
C-LOOK



- Keep moving in the same direction until there are no more outstanding requests pending in that direction, then algorithm switches direction.
- When switching occurs the arm goes to the lowest numbered cylinder with pending requests and from there it continues moving in upward direction again.

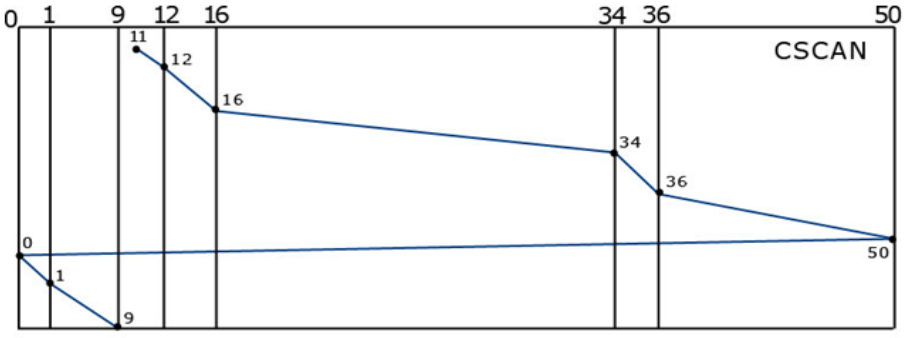
- As per CLOOK requests will be satisfied in order **11, 12, 16, 34, 36, 1, 9**
- Total cylinder movement: (12-11) + (16-12) + (34-16) + (36-34) +(36-1)+(9-1)**=68**

SCAN



- From the current position disk arm starts in up direction and moves towards the end, serving all the pending requests until end.
- At that end arm direction is reversed (down) and moves towards the other end serving the pending requests on the way.
- As per SCAN request will be satisfied in order: **11, 12, 16, 34, 36, 50, 9, 1**
- Total cylinder movement: (12-11) + (16-12) + (34-16) +(36-34) +(50-36) + (50-9) + (9-1) = **88**

CSCAN



- From the current position disk arm starts in up direction and moves towards the end, serving request until end.
- At the end the arm direction is reversed (down), and arm directly goes to other end and again continues moving in upward direction.

- As per SCAN request will be satisfied in order: **11, 12, 16, 34, 36, 50, 0, 1,9**
- Total cylinder movement: (12-11) + (16-12) + (34-16) +(36-34) +(50-36) + (50-0) + (1-0)+ (9-1) = **98**

(1) Explain file attributes and file operations in brief.

- A file is a unit of storing data on a secondary storage device such as a hard disk or other external media.
- Every file has a name and its data. Operating system associates various information with files. For example the date and time of the last modified file and the size of file etc….
- This information is called the file's **attributes or metadata**.
- The attributes varies considerably from system to system. Some of the possibilities of attributes are shown below.

File Attributes

- These attributes relates to the files attributes and tell who may access it and who may not.

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.

- Flags are bits or short fields that control or enable some specific property.

Attribute	Meaning
Read only flag	0 for read/write, 1 for read only.
Hidden flag	0 for normal, 1 for do not display the listings.
System flag	0 for normal, 1 for system file.
Archive flag	0 for has been backed up, 1 for needs to be backed up.
Random access flag	0 for sequential access only, 1 for random access.
Temporary flag	0 for normal, 1 for delete file on process exit.
Lock flag	0 for unlocked, 1 for locked.

- The record length, key position and key length are only present in files whose record can be looked up using the key. They provide the information required to find the keys.

Attributes	Meaning
Record length	Number of bytes in a record.
Key position	Offset of the key within each record.
Key length	Number of bytes in key field.

- The various times keep track of when the file was created, most recently accessed and most recently modified.

Attributes	Meaning
Creation time	Date and time the file was created.
Time of last access	Date and time of file was last accessed.
time of last change	Date and time of file was last changed.

- The current size tells how big the file is at present. And some old mainframe operating system requires the maximum size to be specified.

Attribute	Meaning
Current size	Number of bytes in the file.
Maximum size	Number of bytes the file may grow to.

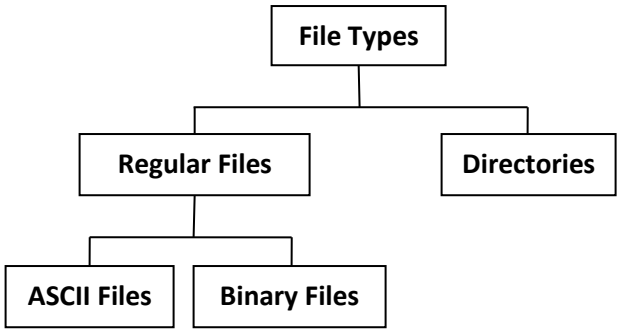
File operations

- File exists to store information and allow it to be retrieved later.
- Different system provides different operations to allow storage and retrieval.
- The most common system calls are shown below.
  - **Create**
    The purpose of the call is to announce that the file is coming and to set some attributes.
  - **Delete**
    When the file is no longer needed, it has to be deleted to free up disk space.
  - **Open**
    The purpose of the open call is to allow the system to fetch the attributes.

- **Close**
  When all accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up table space.
- **Read**
  Data are read from file. Usually the bytes come from the current position.
- **Write**
  Data are written to the file, usually at the current position.
- **Append**
  This call is restricted form of write. It can only add data to the end of file.
- **Seek**
  For a random access files, a method is needed to specify from where to take the data. Seek repositions the file pointer to a specific place in the file.
- **Get attributes**
  Processes often need to read the file attributes to do their work.
- **Set attributes**
  Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible.
- **Rename**
  It frequently happens that a user needs to change the name of an existing file. This system call makes it possible.

(2) Writes short notes on file types & file access.

File Types



Regular File

- Regular files are the ones that contain user information.

- Regular file, as a randomly accessible sequence of bytes, has no other predefined internal structure.
- Application programs are responsible for understanding the structure and content of any specific regular file.

Directories

- Directories are system files for maintaining the structure of the file system.
- To keep track of files, file systems normally have directories or folder.

ASCII files

- ASCII file consists of line of text.
- Advantage of ASCII files is that they can be displayed & printed as it is & they can be edited with ordinary text editor.
- If number of programs use ASCII files for input and output, it is easy to connect the output of one program to the input of another.
- C/C++/Perl/HTML files are all examples of ASCII files.

Binary Files

- Binary files contain formatted information that only certain applications or processors can understand.
- Binary files must be run on the appropriate software or processor before humans can read them.
- Executable files, compiled programs, spreadsheets, compressed files, and graphic (image) files are all examples of binary files.

Device Files

- Under Linux and UNIX each and every hardware device is treated as a file. A device file allows to accesses hardware devices so that end users do not need to get technical details about hardware.
- In short, a device file (also called as a special file) is an interface for a device driver that appears in a file system as if it were an ordinary file.
- This allows software to interact with the device driver using standard input/output system calls, which simplifies many tasks.

Character Special Files

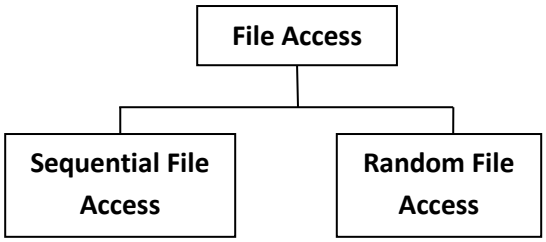
- It is a type of device file which talks to devices in a character by character (1 byte at a time).
- Character Special files are related to input/output and use to model serial I/O devices,

such as terminals, printers, and networks.

Block Special Files

- It is a type of device file which talks to devices 1 block at a time (1 block = 512 bytes to 32KB).
- Block special files are used to model disks, DVD/CD ROM, and memory regions etc.

File Access



Sequential File Access

- In Sequential access, process could read all the bytes or records from a file in order, starting at the beginning, but could not skip around and read them out of order.
- Sequential files could be rewound, however, so they could be read as often as needed.
- These files were convenient when the storage medium was magnetic tape or CD-ROM.

Random File Access

- Files whose bytes or records can be read in any order are called random access files.
- Random access files are essentials for many applications, for example, data base systems.
- If an airline customer calls up and wants to reserve a seat on a particular flight, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights.

(3) Explain directory structures.

- To keep track of files, file systems normally have directories or folders.
- **Directories** are system files for maintaining the structure of the file system.

Single Level Directory system

- The simplest form of directory system is having one directory containing all the files. This is some time called as root directory.
- An example of a system which has one directory is given in figure. Here directory

contain four files.

- The advantages of this scheme are its simplicity and the ability to locate files quickly there is only one directory to look, after all.
- It is often used on simple embedded devices such as telephones.

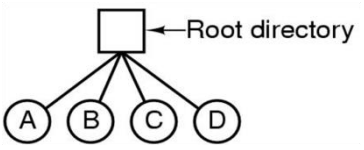


Figure 7-1. A single-level directory system containing four files, owned by three different people, *A*, *B*, and *C*.

Hierarchical Directory system

- The single level is used for simple dedicated application, but for modern user with thousands of files, it would be impossible to find anything if all files were in a single directory.
- The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for user to organize their work.
- When the file system is organized as a directory tree, some way is needed for specifying file names.
- Two methods are used commonly
  - Absolute path name
  - Relative path name
- **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 contain the file mailbox.
- **Relative path name** is used in conjunction with the concept of the working directory.
  - User can designate one directory as the current working directory, in which case, all path names not beginning at the root directory are taken relative to working directory.
  - For example if the current working is /usr/ast, then the file whose absolute path is /usr/ast/mailbox can be referenced simply as mailbox.
- Most operating systems that support a hierarchical directory system have two special entries in every directory,
  - "." Or "dot" refers to current directory

- ".." or "dotdot" refers to parent directory

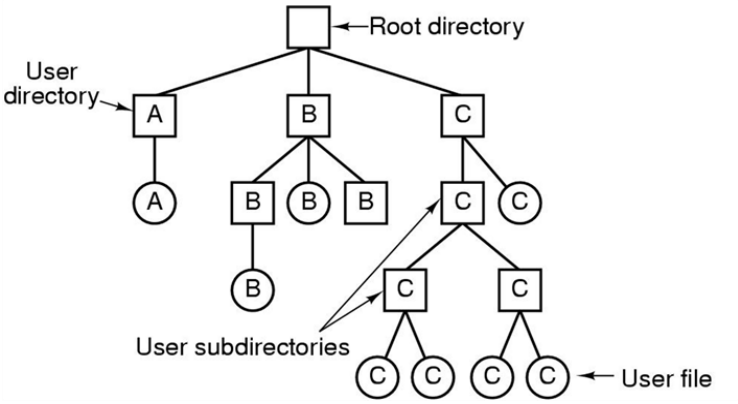


Figure 7-2. A hierarchical directory system.

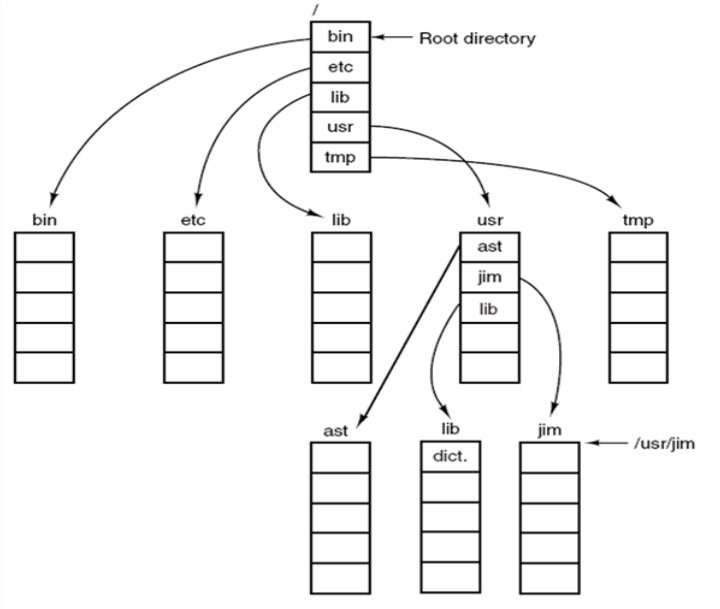


Figure 7-3. A UNIX directory tree.

(4) Explain various directory operations in brief.

- System calls, which are allowed for managing directories exhibit more variation from system to system.
- Following are common directory operations.
  - **Create**

    A directory is created. It is empty except for dot and dot dot, which are put there automatically by the system.

- **Delete**

A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered as an empty directory.

- **Opendir**

Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.

- **Closedir**

When a directory has been read, it should be closed to free up internal table space.

- **Readdir**

This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structure is being used.

- **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 call specifies an existing file and a path name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link**.

- **Unlink**

A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

(5) What is boot block? What is Superblock? How they are used to handle file management system in OS?

- 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 addresses 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 MBR program locates the active partition, reads in its first block, called the **boot block,** and executes it.
- The program in the boot block loads the operating system contained in that partition.
- For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system.
- Other than starting with a boot block, the layout of a disk partition varies strongly from file system to file system.
- Often the file system will contain some of the items shown in Fig. 7-4.
- 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.

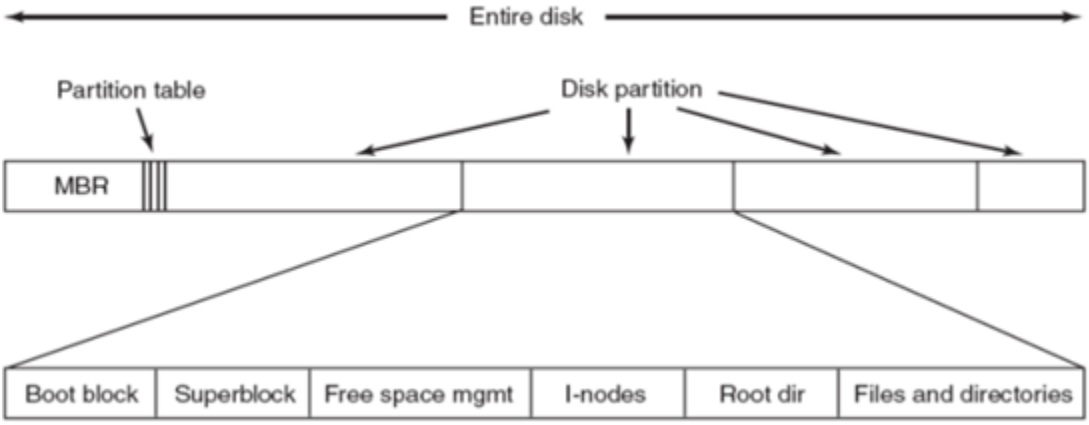


Figure 7-4. A possible file system layout.

- 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 field may contain information 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 directory, which contains the top of the file system tree. Finally, the remainder of the disk typically contains all the other directories and files.

(6) Give different types of file system in brief. OR Explain different types of file implementation.

Probably the most important issue in implementing file storage is keeping track of which blocks go with which file. Various methods to implement files are listed below,

- Contiguous Allocation
- Linked List Allocation
- Linked List Allocation Using A Table In Memory
- I-nodes

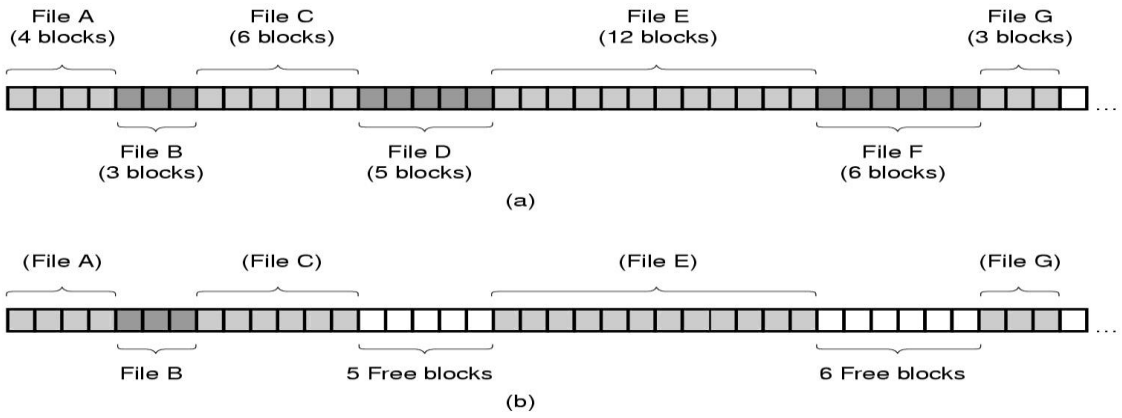


Figure 7-5. (a) Contiguous allocation of disk space for 7 files. (b) The state of the disk after files D and F have been removed.

Contiguous Allocation

- The simplest allocation scheme is to store each file as a contiguous run of disk block.
- We see an example of contiguous storage allocation in fig. 7-5.
- Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the

disk was empty.

- **Advantages**
  - First 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.
  - Second, the read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block), so data comes in at the full bandwidth of the disk.
  - Thus contiguous allocation is simple to implement and has high performance.
- **Drawbacks**
  - Over the course of time, the disk becomes fragmented.
  - Initially, fragmentation is not a problem, since each new file can be written at the end of the disk, following the previous one.
  - However, eventually the disk will fill up and it will become necessary to either compact the disk, which is expensive, or to reuse the free space in the holes.
  - Reusing the space requires maintaining a list of hole.
  - However, when a new file is to be created, 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 continuous allocation is feasible and in fact, widely used: on CD-ROMs. Here all the file sizes are known in advance and will never change during use of CD-ROM file system.

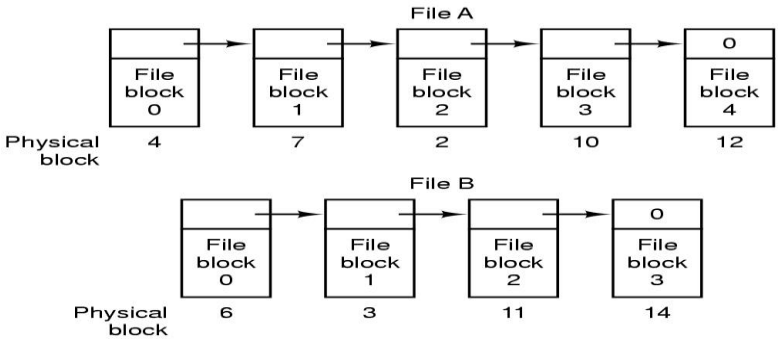


Figure 7-6. Storing a file as a linked list of disk blocks.

Linked List Allocation

- Another method for storing files is to keep each one as a linked list of the disk blocks, as shown in fig. 7-6.
- The first word of each block is used as a pointer to the next one. The rest of the block is

for data.

- Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation.
- It is sufficient for a directory entry to store only disk address of the first block, rest can be found starting there.

Drawbacks

- Although reading a file sequentially is straightforward, random access is **extremely slow**. To get to block n, the operating system has to start at the beginning and read the n-1 blocks prior to it, one at a time.
- The amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While having an unusual size is less efficient because many programs read and write in blocks whose size is a power of two.
- 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, **which generates extra overhead due to the copying**.

Linked List Allocation Using a Table in Memory / FAT (File Allocation Table)

- Both disadvantage 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.
- Figure 7-7 shows what the table looks like; in example we have two files.
- File A uses disk blocks 4, 7, 2, 10, and 12 in that order, and file B uses disk blocks 6, 3, 11, and 14 in that order.
- The same can be done starting with block 6. Both chains are terminated with a special marker (eg. -1) that is not a valid block number. Such a table in main memory is called a FAT (File Allocation Table).
- Using this organization, the entire block is available for the data.
  The primary disadvantage of this method is that the entire table must be in memory all the time to make it work.

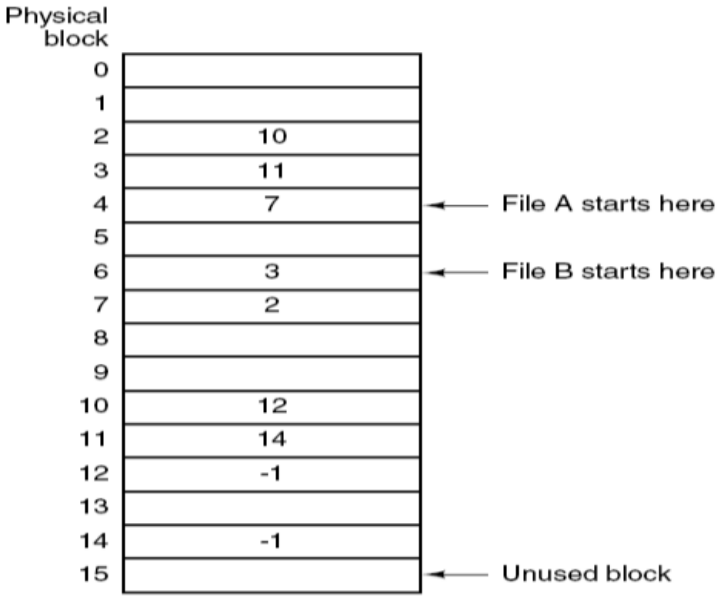


Figure 7-7. Linked list allocation using a file allocation table in main memory.

I-nodes

- A method for 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.
- A simple example is given in figure 7-8.
- Given the i-node, it is then possible to find all the blocks of the file.
- The big advantage of this scheme over linked files using an in-memory table is that 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 by the array holding the i-nodes for the open files is only *kn* bytes. Only this much space needs to be reserved in advance.
- One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit?
- One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk block addresses, as shown in Figure 7-8.

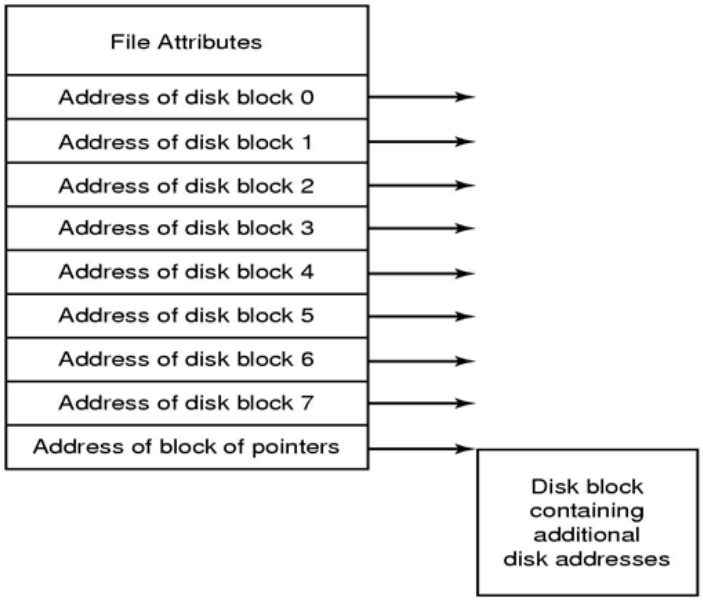


Figure 7-8. An example i-node.

(7) Explain Disk files system management and optimization.

Disk Space Management

- Files are normally stored on disk, so management of disk space is a major concern to file system designers.
- Two general strategies are possible for storing an n byte file:
  - n consecutive bytes of disk space are allocated, or
  - The file is split up into number of contiguous blocks.
- Storing a file as a contiguous sequence of bytes has the problem that if a file grows, it will probably have to be moved on the disk, for this reason all file systems divide files into fixed-sized blocks that need not be adjacent.

Block Size

- Having a large block size means that every file, even a 1 byte file, ties up an entire cylinder. It also means that small files waste a large amount of disk space.
- A small block size means that most files will span multiple seeks.
- Thus, if the allocation is too large, we waste space; if it is too small, we waste time.

Keeping track of free blocks

- Once a block size has been chosen, the next issue is how to keep track of free blocks.
- Two methods are widely used
  - First: Using a linked list of disk blocks, with each block holding as many free disks block numbers as will fit.
  - Second: Bitmap - A disk with n blocks requires a bitmap with n bits. Free blocks are represented by 1s in the map, allocated blocks by 0s or vice versa.
- If free blocks tend to come in long runs of consecutive blocks, the free-list system can be modified to keep track of runs of blocks rather than single blocks.

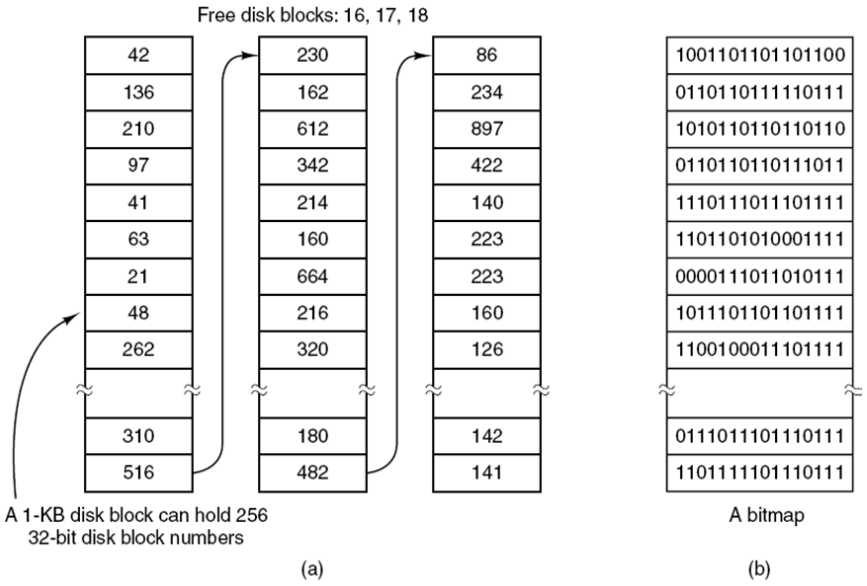


Figure 7-9. (a) Storing the free list on a linked list. (b) A bitmap.

Disk Quotas

- The system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas.
- Any increases in the file's size will be charged to the owner's quota.
- A table contains the quota record for every user with a currently open file, even if the file was opened by someone else.
- When all file are closed, the record is written back to the quota file.
- When a new entry is made in the open file table, a pointer to the owner's quota record is entered into it, it make it easy to find the various limits.
- Every time a block is added to a file, the total number of blocks charged to the owner is

incremented, and a check is made against both the hard and soft limits.

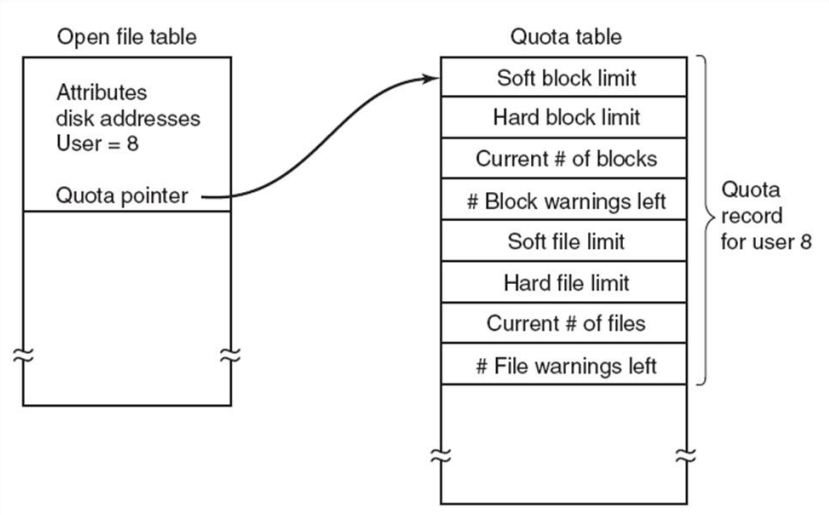


Figure 7-10. Quotas are kept track of on a per-user basis in a quota table.

(8) Explain the file system backups.

- If a computer's file system is irrevocably lost, where due to hardware or software, restoring all the information will be difficult, time consuming, and in many cases, impossible.
- Backups to tape are generally made to handle one of two potential problems:
  - Recover from disaster, Recover from stupidity.
- Making a backup takes a long time and occupies a large amount of space, these consideration raises the following issues (**Issues of Backup**):
  - **Issue: 1:** First should the entire file system be backup or only part of it? It is usually desirable to back up only specific directories and everything in them rather than the entire file system.
  - **Issue: 2:** It is wasteful to back up files that have no change since the previous backup which leads to the idea of incremental dumps.
  - **Issue: 3:** Third since immense amount of data are typically dumped, it may be desirable to compress the data before writing them to tape.
  - **Issue: 4:** It is difficult to perform a backup on an active file system.
  - **Issue: 5:** Making backups introduces many nontechnical problems into an organization.
- Two strategies can be used for dumping a disk to tape: **Physical dump, Logical dump**

- **Physical dump:**
  - It is started at block 0 of the disk, writes all the disk blocks onto the output tape in order, and stops when it has copied the last one.
  - Such program is so simple that it can probably be made 100% bug free, something that can probably not be said about any other useful program.
  - Main advantages of physical dumping are simplicity and great speed.
- **Logical dump:**
  - It is started at one or more specified directories and recursively dumps all files and directories found there that have changed since some given base date.
  - In logical dump, the dump tape gets a series of identifed directories and file.

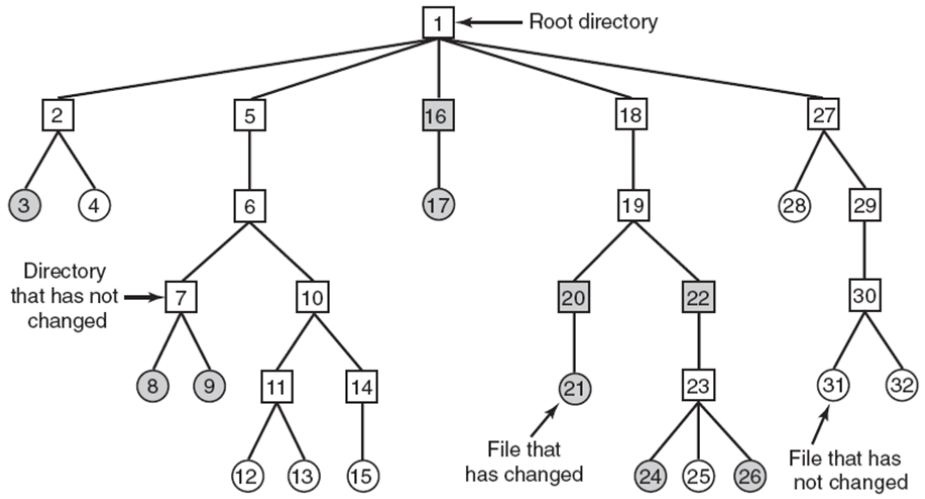


Figure 7-11. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

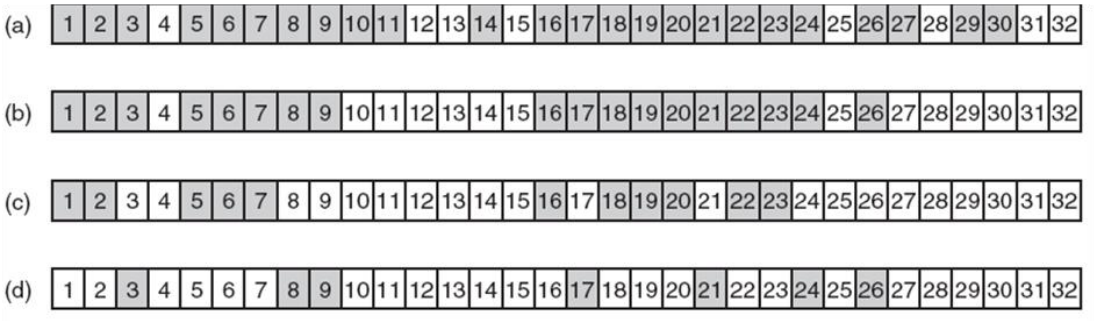


Figure 7-12. Bit maps used by the logical dumping algorithm.

- Restoring a file system from the dump tapes is straightforward. To start with, an empty file system is created on the disk. Then the most recent full dump is restored.
- Since the directories appear first on the tape, they are all restored first, giving a skeleton of the file system.
- Then the files themselves are restored. This process is then repeated with the first incremental dump made after the full dump, then the next one, and so on.

(9) Explain File System Consistency.

- To deal with the problem of inconsistent file systems, most computers have a utility program that checks file system consistency. For example, UNIX has *fsck* and Windows has *scandisk*.
- This utility can be run whenever the system is booted, especially after a crash.
- The description below tells how *fsck* works.
- Two kinds of consistency checks can be made: blocks and files.
- To check for block consistency, the program builds two tables, each one containing a counter for each block, initially set to 0.
- The counters in the first table keep track of how many times each block is present in a file; the counters in the second table record how often each block is present in the free list (or the bitmap of free blocks).
- The program then reads all the i-nodes.
- As each block number is read, its counter in the first table is incremented.
- The program then examines the free list or bitmap, to find all the blocks that are not in use. Each occurrence of a block in the free list results in its counter in the second table being incremented.
- If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 7-13(a).
- If the file system is not consistent, following types of inconsistency may occur.
- The tables might look like Fig. 7-13(b), in which block 2 do not occur in either table.
- It will be reported as being a missing block.
- The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

- Another situation that might occur is that of Fig. 7-13(c).
- Here, number 4 occurs twice in the free list. (Duplicates can occur only if the free list is really a list; with a bitmap it is impossible.) The solution here is also simple: rebuild the free list.
- The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 7-13(d) with block 5.
- The appropriate action for the file system checker to take is to allocate a free block, copy the contents of block 5 into it, and insert the copy into one of the files.
- In addition to checking to see that each block is properly accounted for, the file system checker also checks the directory system. It uses a table of counters per file.
- It starts at the root directory and recursively descends the tree, inspecting each directory in the file system. For every file in every directory, it increments a counter for that file's usage count.
- When it is all done, it has a list, indexed by i-node number, telling how many directories contain each file. It then compares these numbers with the link counts stored in the i-nodes themselves.
- In a consistent file system, both counts will agree.
- However, two kinds of errors can occur:
- The link count in the i-node can be too high or it can be too low.
- If the link count is higher than the number of directory entries, it should be fixed by setting the link count in the i-node to the correct value.
- The other error is potentially catastrophic. If two directory entries are linked to a file, but the i-node says that there is only one, when either directory entry is removed, the i-node count will go to zero.
- When an i-node count goes to zero, the file system marks it as unused and releases all of its blocks.
- This action will result in one of the directories now pointing to an unused i-node, whose blocks may soon be assigned to other files. Again, the solution is just to force the link count in the i-node to the actual number of directory entries.

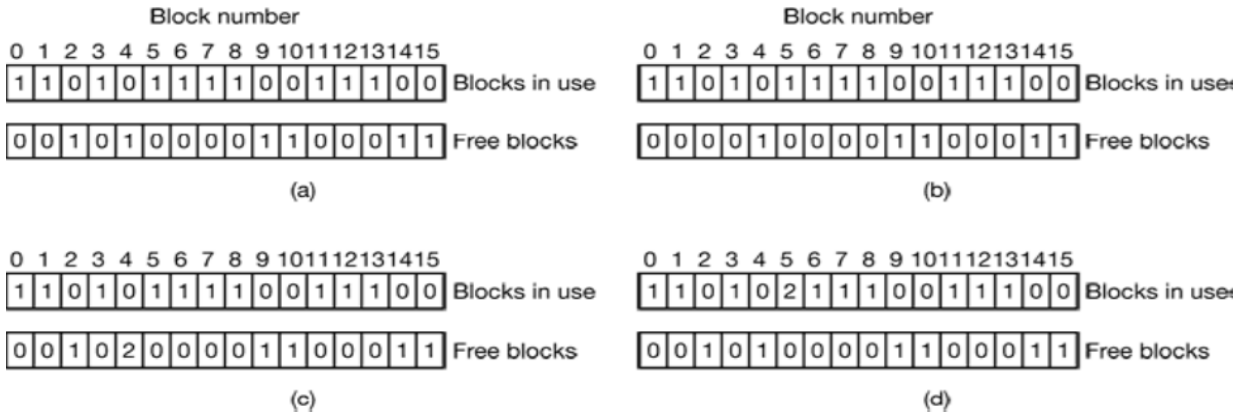


Figure 7-13. The system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

(1) Write short note on Design Principles of Security.

Design Principles of Security

- Principles of Least Privileges:
  - ✓ It restricts how privileges are granted.
  - ✓ This principle states how the privileges are to be granted to a subject. A subject should be given only those privileges that it requires for completing a task.
  - ✓ If a subject does not need a specific right it should not be granted that right.
  - ✓ For example, if a subject requires append rights to an object then it must be given only the append rights and not the write rights.
- Principe of Fail Safe Defaults:
  - ✓ It restricts how privileges are initialized when a subject or object are created.
  - ✓ This principle states that unless the subject is given explicit access to the object it should be denied access to that object.
  - ✓ This means that the default access to object is none.
  - ✓ All the access rights should be given explicitly granted.
- Principle of Economy of Mechanisms:
  - ✓ This principle simplifies the design and implementation of security mechanisms.
  - ✓ This principle states that security mechanism should be as simple as possible.
  - ✓ If design is simple there are fewer chances for errors.
  - ✓ The checking and testing procedure becomes simpler.
- Principles of Complete Mediation:
  - ✓ This principle states that all the accesses to object be checked in order to ensure that they are allowed.
  - ✓ Whenever a subject attempts to read an object the OS mediate the action.
  - ✓ First it determines if the subject is allowed to access the object.
  - ✓ If so it provides resources for reading the object.
  - ✓ If the subject reattempts the read operation then it checks if the subject is still allowed to read the object and then allows for reading.
- Principle of Open Design:
  - ✓ This principle suggests that complexity doesn't add security.
  - ✓ This principle states that the security of mechanism should not depend on the secrecy of its design or implementation.
- Principles of Separation of Privileges:
  - ✓ This principle states that the access of an object should not depend only on fulfilling a single condition.

- ✓ There should be multiple conditions required to grant privilege and two or more system components work together to enforce security.

- Principles of Least Common Mechanism
  - ✓ This principle states that the amount of mechanism common to and depending on multiple users should be kept to the minimum possible.
- Principles of user Acceptability
  - ✓ This principle states that the mechanism used for protection should be acceptable to the users and should be easy to use.
  - ✓ Otherwise, the user may feel a burden to follow the protection mechanism.

(2) Explain Domain Protection mechanism in detail.

- A computer can be viewed as a collection of processes and objects (both H/W & S/W).
- Means a computer system is collections of objects and processes and these objects and processes are needed to be protected.
- Each object has unique name by which it is referred and finite set of operations that processes are allowed to carry out on it.
- There should be some way to prohibit processes from accessing objects that they are not authorized to.
- Operations that are possible depend on the object.

Object	Operation
CPU	Execution
File	Read, Write
Semaphore	Up, Down
Tape Drives	Read, Write, Rewound

- It is also possible to restrict processes to carry out a subset of legal operations.
- In other words we can say that process should be able to access only those resources that it currently requires to complete its task.
- This requirement is known as need to know principle.

Domain Structure

- A protection domain specifies the resources that a process may access.
- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- A domain is defined as a set of < object, {access right set} > pairs. Note that some domains may be disjoint while others overlap.
- Domains need not be disjoint; they may share access rights i.e. it is possible for the same object to be in multiple domains, with different access rights each.
- An access right is the ability to execute an operation on an object.

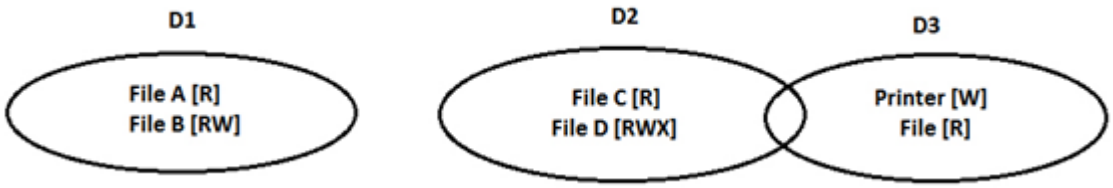


Figure 9-1. Three protection domains.

- Association between a process and a domain may be either static i.e. fixed set of resources available to a process throughout its life time or dynamic.
- In order to understand protection domain more thoroughly let us take an example of UNIX.
  - ✓ In UNIX, domain of process is defined by its uid and gid. Given any (uid, gid) combination it is possible to make complete list of all objects that can be accessed and whether they can be accessed for reading, writing or executing.
- Processes with same (uid, gid) combination will have exactly same set of objects.
- One more example is that each process in UNIX has 2 halves.
  a) Kernel part
  b) User part.
- When a process does a system call, it switches from user part to kernel part.
- The kernel part has access to a different set of objects from the user part. Thus a system call causes a domain switch.
- An important question is how the system keeps track with object belongs to which domain.

Object / Domain	File 1	File 2	File 3	Printer
1	READ	READ WRITE		
2			READ WRITE EXECUTE	WRITE
3				WRITE

- One simple way to accomplish this is by preparing a matrix with rows being domains and columns being objects.
- Now from this Matrix and current domain number the system can tell if an access to a given object in a particular way from specified domain is allowed or not.
- Domain switching can be included in matrix model by realizing that the domain itself is an object with the operation ENTER.

Object / Domain	File 1	File 2	File 3	Printer	D1	D2
1	READ	READ WRITE				ENTER
2			READ	WRITE		
3				WRITE		

- Processes in domain 1 can switch to domain 2 but once there, they can't go back.

(3) Explain Access metrics mechanism.

- Model of protection can be viewed abstractly as a matrix called an access matrix.
- Row of access matrix represents domains and a column represents objects.
- Each entry in the matrix consists of set of access rights.
- The entry access (i, j) defines set of operations that a process executing in domain Di, can invite an object Oj.
- The access matrix scheme provides us with the mechanism for specifying a variety of policies.

- We must ensure that process executing in domain D can access only those objects specified in row P as allows by the access matrix.
- Access matrix also provides the mechanism of defining and implementing control for both static and dynamic association between process and domains.
- We are executing an operation on an object.
- We can control these changes by including the access matrix as an object.
- Operations on domains and the access matrix are not in themselves important.
- Important is that they illustrate the ability of the access matrix model to allow the implementation and control of dynamic protection requirements.
- New objects and new domains can be created dynamically and included in access matrix model.
- We need to consider only the operations that are possible on these new objects (domains and access matrix) and decide how we want processes to be able to execute these operations.
- Allowing controlled change to the contents of the access matrix entries requires three additional operations: copy, owner and control.
- The ability to copy an access right from one domain (row) of the access matrix to another is denoted by asterisk (*) appended to the access right.

Object/ Domain	F1	F2	F3
D1	EXECUTE		WRITE*
D2	EXECUTE	READ *	EXECUTE
D3	EXECUTE		

(A)

Object/ Domain	F1	F2	F3
D1	EXECUTE		WRITE*
D2	EXECUTE	READ *	EXECUTE
D3	EXECUTE	READ	

(B)

- For example as shown in figure (A) a process executing in domain D2 can copy the read operation into any entry associated with file F2.
- Hence the access matrix of figure (A) can be modified into figure (B).

- We also need mechanism to allow addition of owner rights and removal of some rights.
- The ownership right controls these operations.

Object/Domain	F1	F2	F3
D1	OWNER EXECUTE		WRITE
D2		READ* WRITE	READ* WRITE OWNER
D3	EXECUTE		

(C)

Object/ Domain	F1	F2	F3
D1	OWNER EXECUTE		WRITE
D2		OWNER READ* WRITE	READ* WRITE* OWNER
D3		WRITE	WRITE

(D)

- For example figure (C) domain D1 is the owner of F1 and thus can add and delete any valid right in column F1.
- Similarly Domain D2 is the owner of F2 and F3 and thus can add and remove any valid right within these two columns.
- Thus the access matrix figure (C) can be modified to the access matrix shown in figure (D).
- The copy and owner rights allow a process to change the entries in a column. A mechanism is needed to change the entries in row.
- If access (i, j) includes the control right then a process executing in domain Di can remove any access right from row j.
- These operations on the domains and the access matrix are not in themselves particularly model to allow implementation and control of dynamic protection requirements.

(4) What is Access Control list? Explain in brief.

- Most of domains have no access at all to most objects, so storing a very large, mostly empty, Access matrix is a waste of disk space.
- Two methods are practical; the first is storing the matrix by rows and the second method is storing the matrix by columns, and then storing only the nonempty elements.
- The two approaches are surprisingly different.
- The first technique consists of associating with each object an (ordered) list containing all the domains that may access the object, and how.
- This list is called the **Access Control List or ACL** and is illustrated in Fig. 9-2.
- Here we see three processes A, b, and C, each belonging to a different domain. There are three files F1, F2, and F3.
- For simplicity, we will assume that each domain corresponds to exactly one user, in this case, users are A, B, and C.
- Often in the security literature, the users are called subjects or principals, to contrast them with the things owned, the objects, such us files.
- Each file has an ACL associated with it. File F1 has two entries in its ACL (separated by a semicolon).
- The first entry says that any process owned by user A may read and write the file. The second entry says that any process owned by user B may read the file.
- All other accesses by these users and all accesses by other users are forbidden.
- Note that the rights are granted by user, not by process.
- As far as the protection system goes, any process owned by user A can read and write file F1. It does not matter if there is one such process or 100 of them. It is the owner, not the process ID that matters.
- File F2 has three entries in its ACL: A, B, and C can all read the file, and in addition B can also write it.
- No other accesses are allowed, File F3 is apparently an executable program, since B and C can both read and execute it. B can also write it.
- Many systems support the concept of a group of users. Groups have names and can be included in ACLs.

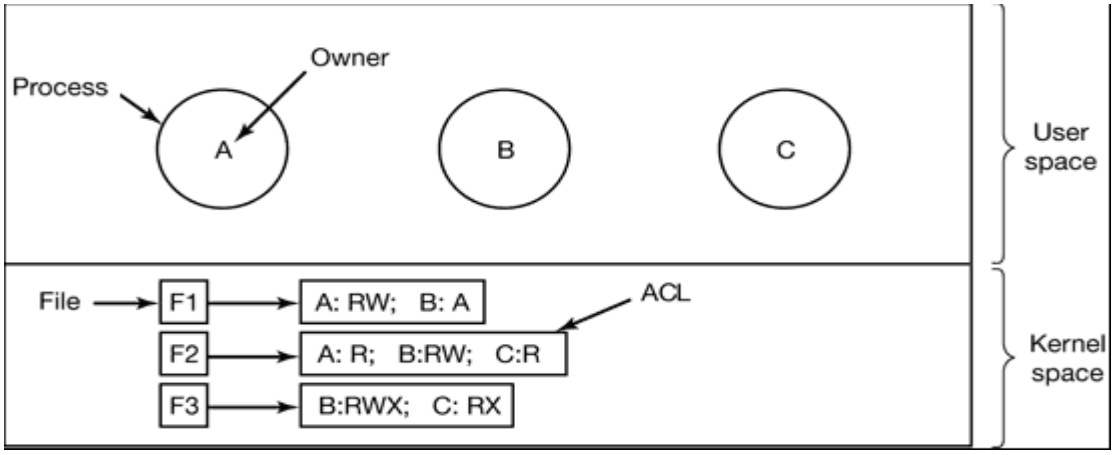


Figure 9-2. Use of access control lists to manage file access.

- The other way of slicing up the matrix of Fig. 9-2 is by rows.
- When this method is used, associated with each process is a list of objects that may be accessed, along with an indication of which operations are permitted on each, in other words, its domain.
- This list is called a capability list or C-list and the individual items on it are called capabilities.
- A set of three processes and their capability lists is shown in Fig. 9-3.

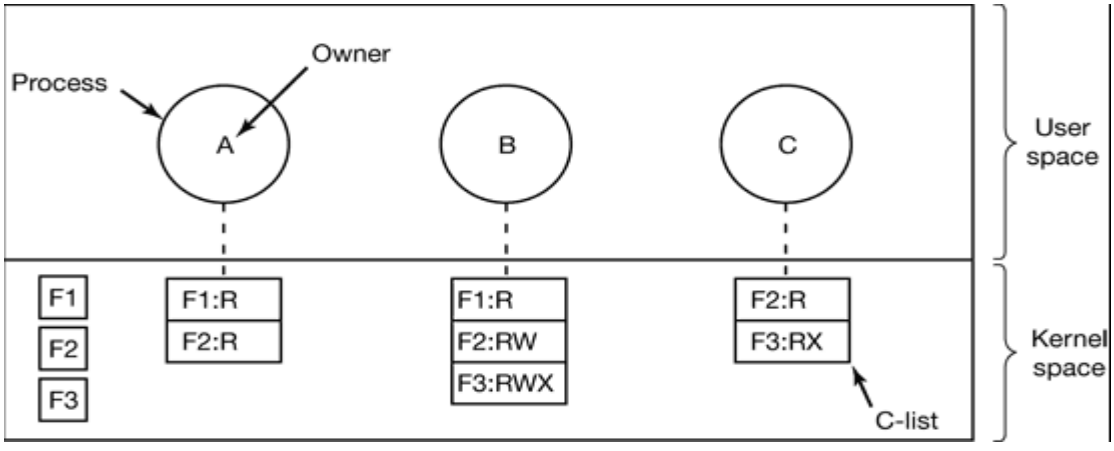


Figure 9-3. When capabilities are used, each process hits a capability list.

- Each capability grants the owner certain rights on a certain object. In Fig. 9-3, the process owned by user A can read files F1 and F2, for example.
- Usually, a capability consists of a file (or more generally, an object) identifier and a bitmap for the various rights. In a UNIX-like system, the file identifier would probably be the i-node number.
- Capability lists are themselves objects and may be pointed to from other capability lists, thus facilitating sharing of subdomains.
- It is fairly obvious that capability lists must be protected from user tampering.
- Three methods of protecting them are known.
- The first way requires a tagged architecture, a hardware design in which each memory word has an extra (or tag) bit that tells whether the word contains a capability or not.
- The tag bit is not used by arithmetic, comparison, or similar ordinary instructions and it can be modified only by programs running in kernel mode (i.e., the operating system).
- The second way is to keep the C-list inside the operating system. Capabilities are then referred to by their position in the capability list.
- The third way is to keep the C-list in user space, but manage the capabilities cryptographically so that users cannot tamper with them. This approach is particularly suited to distributed systems.
- In addition to the specific object-dependent rights, such as read and execute, capabilities (both kernel and cryptographically-protected) usually have generic rights which are applicable to all objects.
- Examples of generic rights are
    1. Copy capability: create a new capability for the same object.
    2. Copy object: create a duplicate object with a new capability.
    3. Remove capability: delete an entry from the C-list; object unaffected.
    4. Destroy object: permanently remove an object and a capability.
- A last remark worth making about capability systems is that revoking access to an object is quite difficult in the kernel-managed version.
- It is hard for the system t find all the outstanding capabilities for any object to take them back, since they may be stored in Clists all over the disk.
- One approach is to have each capability point to an indirect object, rather than to the object itself.
- By having the indirect object point to the real object, the system can always break that

connection, thus invalidating the capabilities.

- ACLs and capabilities have somewhat complementary properties.
- Capabilities are very efficient because if a process says "Open the file pointed to by capability 3," no checking is needed.
- With ACLs, a (potentially long) search of the ACL may be needed.
- If groups are not supported, then granting everyone read access to a file requires enumerating all users in the ACL.
- Capabilities also allow a process to be encapsulated easily, whereas ACLs do not.
- On the other hand, ACLs allow selective revocation of rights, which capabilities do not.
- Finally, if an object is removed and the capabilities are not or the capabilities are removed and an object is not, problems arise. ACLs do not suffer from this problem.

(5) Explain Trojan Horse, Trap Door, Virus and Warms program threats.

Trojan Horse:

- A Trojan horse is a standalone malicious program which may give full control of infected PC to another PC.
- Trojan horses may make copies of them, steal information, or harm the host computer systems.
- The Trojan horse, at first glance will appear to be useful software but will actually do damage once installed or run on your computer.
- Trojans are designed for different purpose like changing your desktop, adding silly active desktop icons or they can cause serious damage by deleting files and destroying information on your system.
- Trojans are also known to create a backdoor on your computer that gives unauthorized users access to your system, possibly allowing confidential or personal information to be compromised.
- Unlike viruses and worms, Trojans do not reproduce by infecting other files nor do they self-replicate. Means Trojan horse viruses differ from other computer viruses in that they are not designed to spread themselves.
- Most popular Trojan horses are Netbus, Subseven, Y3K Remote Administration Tool, Back Orifice, Beast, Zeus, The Blackhole Exploit Kit, Flashback Trojan.

Trap door:

- A trap door is a secret entry point into a program that allows someone that is aware of the trap door to gain access without going through the usual security access

procedures.

- We can also say that it is a method of bypassing normal authentication methods.
- It is also known as back door.
- Trap doors have been used legally for many years by programmers to debug and test programs.
- Trap doors become threats when they are used by dishonest programmers to gain unauthorized access.
- It is difficult to implement operating system controls for trap doors.
- Security measures must focus on the program development and software update activities.

Virus:

- A computer virus is a computer program that can replicate itself and spread from one computer to another.
- Viruses can increase their chances of spreading to other computers by infecting files on a network file system or a file system that is accessed by other computers.
- A computer virus is a program or piece of code that is loaded onto your computer without your knowledge and runs against your wishes.
- Viruses can also replicate themselves.
- All computer viruses are manmade.
- A simple virus that can make a copy of itself over and over again is relatively easy to produce. Even such a simple virus is dangerous because it will quickly use all available memory and bring the system to a halt. An even more dangerous type of virus is one capable of transmitting itself across networks and bypassing security systems.

Worm:

- A worm is a program which spreads usually over network connections.
- Unlike a virus which attaches itself to a host program, worms always need a host program to spread.
- In practice, worms are not normally associated with one person computer systems.
- They are mostly found in multi-user systems such as UNIX environments. A classic example of a worm is Robert Morris is Internet worm 1988.

(1) What is Unix? What is Linux?

- UNIX is a powerful operating system originally developed at AT&T Bell Labs.
- It is very popular among the scientific, engineering, and academic communities due to its multi-user and multi-tasking environment, flexibility and portability, electronic mail and networking capabilities, and the numerous programming, text processing and scientific utilities available.
- The UNIX system is mainly composed of three different parts: the kernel, the file system, and the shell.
- The kernel is that part of the system which manages the resources of whatever computer system it lives on, to keep track of the disks, tapes, printers, terminals, communication lines and any other devices.
- The file system is the organizing structure for data. The file system is perhaps the most important part of the Linux operating system. The file system goes beyond being a simple repository for data, and provides the means of organizing the layout of the data storage in complex ways.
- The shell is the command interpreter. Although the shell is just a utility program, and is not properly a part of the system, it is the part that the user sees. The shell listens to your terminal and translates your requests into actions on the part of the kernel and the many utility programs
- Linux is a freely available, open source, Unix-like operating System. Written originally for the PC by Linus Torvalds (A young student in the University of Helsinki), with the help of many other developers across the internet.
- Linux now runs on multiple hardware platforms, from the smallest to the largest, and serves a wide variety of needs from servers to movie-making to running businesses to user desktops.

(2) Explain the Features of the Unix in detail.

- **The following features are provided by the UNIX operating System.**
- **Open Source**
  - Linux source code is freely available and it is community based development project.
  - Multiple teams are working in collaboration to enhance the capability of Linux operating system and it is continuously evolving.

- **Multiuser capability**
  - In a multi user system the same computer resources hard disk, memory etc are accessible to the many users.
  - Users are given the different terminal to operate, a terminal in turn, is a keyboard and a monitor. All the terminals are connected to the main computer whose resources are available by all users.
  - So the user at any of the terminals cannot use only the computer but also any devices that may be attached like a printer.

- **Multitasking capability**
  - It means that it is capable of carrying out more than one job at the same time it allow you to type in a program in its editor while simultaneously executes some other command you might have given earlier.
  - Say for example you might have given earlier sort and copy a huge file this job will perform in the background while in foreground you use editor.
  - This is managed by dividing the CPU time between all processes being carrying out.

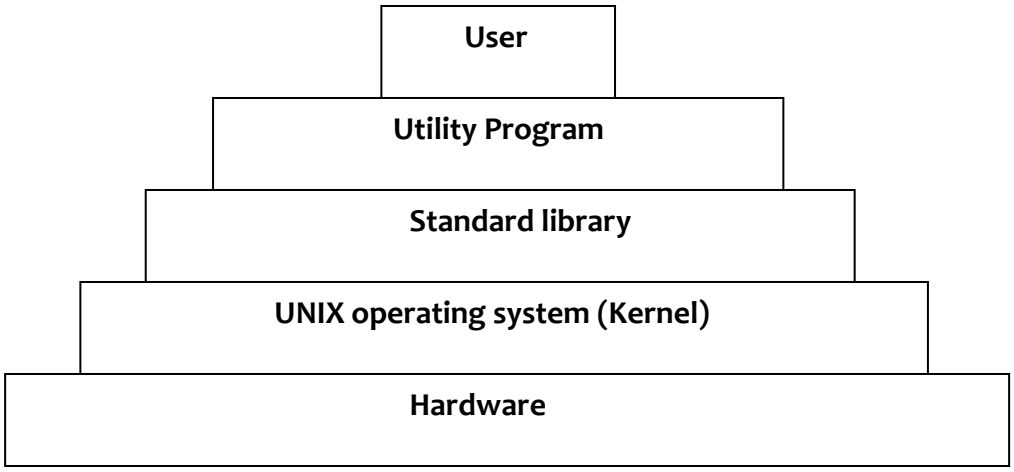
- **Communication**
  - UNIX has the excellent communication with the users. The communication may be within the network of a single main computer or between two or more such computer network.
  - The users can easily exchange mail, data, and programs through such networks.

- **Security**
  - Unix Has the three provisions for protecting the data. The first is provided by assigning the passwords and login names to the individual users and ensuring that nobody has excess to your work.
  - All the five have the read write and execute permissions to each file which decide who can excess a particular file, which can modify and execute it.
  - Lastly there is a file encryption utility which encodes your file into an unreadable format so even if someone succeeds in opening it your secrets are safe.

- **Portability**
  - Portability means software can works on different types of hardware in same way.
  - It is one of the main reasons for the popularity of the Unix. It can be ported to almost any computer system with only the minimum adoptions.

- A portability credit of the UNIX is because of the C language, it written in C language and C language is portable.

(3) Explain Linux Architecture.



- **Hardware**
  - The bottom layer is hardware.
  - It contains physical devices of computer like CPU, Memory, Disk, printer etc.
  - UNIX kernel will interface with this hardware.

- **UNIX Kernel**
  - Kernel is program which provides services of OS like memory management, file management and process management.
  - Kernel will provide interface with hardware and user programs.

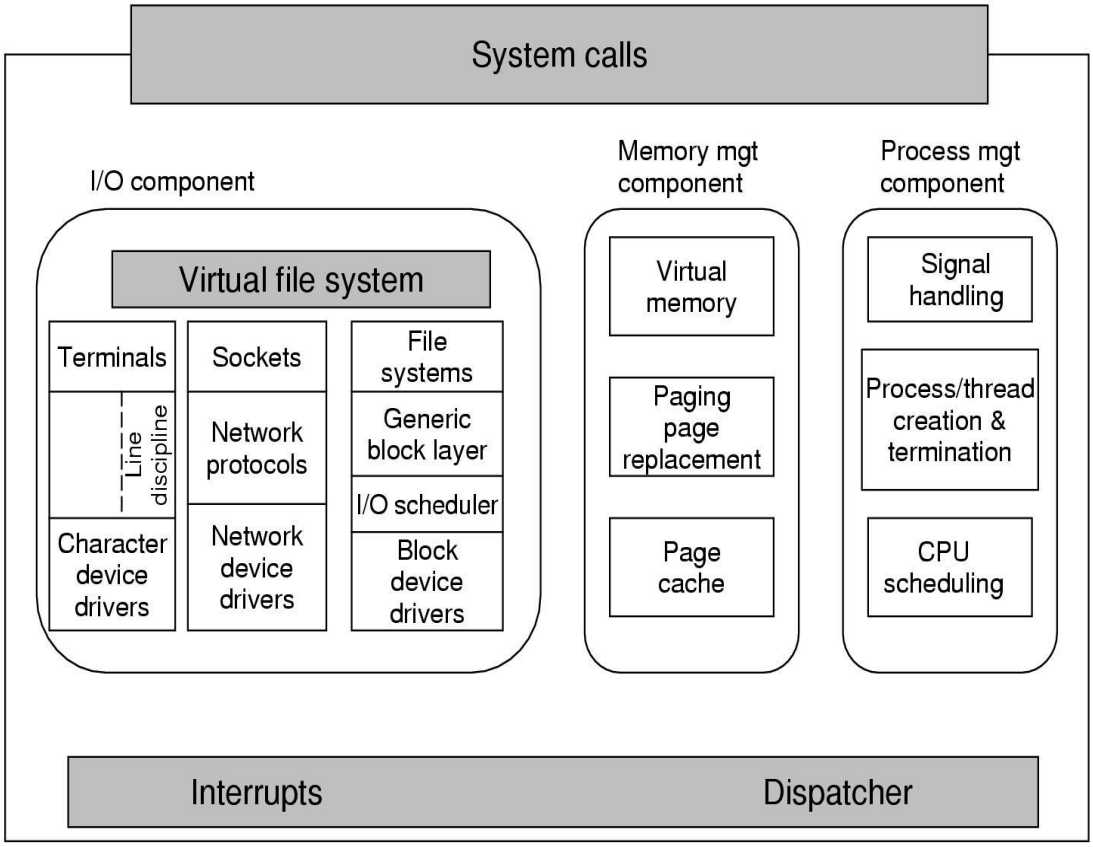
- **Standard library**
  - It contains set of procedures.
  - This is collection of system level files.

- **Utility program:**
  - Utility programs are used to make user programs and make work easier.
  - Utility programs like compilers, assemblers, editors etc.

- **User**
  - User programs are comes for processing and interact with system.

(4) Explain the Linux Kernel Structure.

- The following Figure shows the overall structure of the Linux. The kernel sits directly on the hardware and enables the interactions with I/O devices and memory management.



Structure of the Linux kernel

- At the lowest level it contains interrupt handlers which are the primary way for interacting with the device, and low level dispatching mechanism.
- The dispatching occurs when an interrupt happens and also when the kernel completes some operations and it is time to start up a user process again.
- Now we divide the various kernel subsystems into the three main components. The I/o component contains the kernel pieces and responsible for interacting with the devices and performing the network and storage I/O operations.
- At the highest level the I/O operations are all integrated under a virtual file system and at lowest level, all I/O operations pass through some device driver.

- All Linux drivers are classified as either a character device driver or block device drivers, with the main difference that random accesses are allowed on the block devices and not on the character devices.
- Technically network devices are really character devices, but they are handled somewhat differently, so It is preferable to separate them.
- The layer above the network drivers handle a kind of the routing function and making sure that the right packet goes to the right devices.
- Sockets interface which allows the program to create sockets for the particular network.
- On the top of the disk drivers is the I/O scheduler who is responsible for ordering and issuing disk operation request in a way that tries to converse waste full disk head movement.
- At the very top of the block device column are the file systems Linux may have the multiple file system excising concurrently. In order to hide the architectural differences of the various hardware devices from the file system implementation, a generic block device layer provides an abstraction layer used by all file system.
- At the right side of the fig there are two key components of the Linux kernel these are responsible for the memory and process management tasks.
- Memory management tasks include maintaining the virtual to physical memory mappings, maintaining a cache of the recently accessed pages and implementing a good page replacement policy.
- The key responsibility of the process management component is the creation and termination of the process. It also includes the process scheduler and code for the signal handling.
- Finally at the very top is the system call interface in to the kernel. All system calls come here, causing a trap which switches the execution from the user mode in to the kernel. Mode.

(5) Explain the directory Structure or file system structure of the Unix.

- The initial Linux file system was the MINIX 1 file system. However, infect it limited file names to 14 characters and its maximum file size was 64 MB.
- The first improvement was the ext files system which allow file names of 255 characters and file size of 2 GB, but it was slower than MINIX 1 file system so the ext 2 file system was invented, with long file name, long file, and better performance and it has became

the main file system.

- A Linux file is sequence of 0 or more bytes containing arbitrary information. No distinction is made between ASCII file, binary file or any other kinds of the file.
- File names consists of a based name and an extension, separated by a dot. For example prog.c is c program anprog.o is an object file. Extension may be of any length and file may have multiple extensions, for example prog.java.gz is a compressed java program.
- File can be grouped together in directories. Directories are stored as a files and it can be treated like files.
- The / characters is known as the root directory and also use to separate directory names.

Directory	Contents
bin	Binary (executable) programs
dev	Specials files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Some important directory found in most Linux systems

- There are 2 ways to specify file names in Linux.

1) Absolute path :-

- It tells us how to get the file starting at the root directory. For example / usr/ ast/ books/ mos3/ chap-10. This tells system to look in the root directory for an usr directory. Then looks for the directory ast. The ast directory contains book directory which contains the mos3 directory which contains the file chap-10.

2) Relative path :-

- Absolute path name are often long and inconvenient for this reason Linux allows the users to designate the directory in which they are currently working as the working directory. A path name specified relative to the working directory is a relative path. For example if / usr/ ast /books/mos3 is the working directory then the command.

cp chap-10 backup -10

- It has exactly the same effect as the following longer command.

cp /usr/ast/books/mos3/chap-10

- For example if two user are sharing a file it will be located in directory belonging to

one of them so the other will have to use an absolute path name to refer to it.

- If this is long enough it may become irritating to have to keep typing it so the Linux provides a solution to the problem by allowing the users to make a new directory entry that points to an existing file such an entry is called a link.

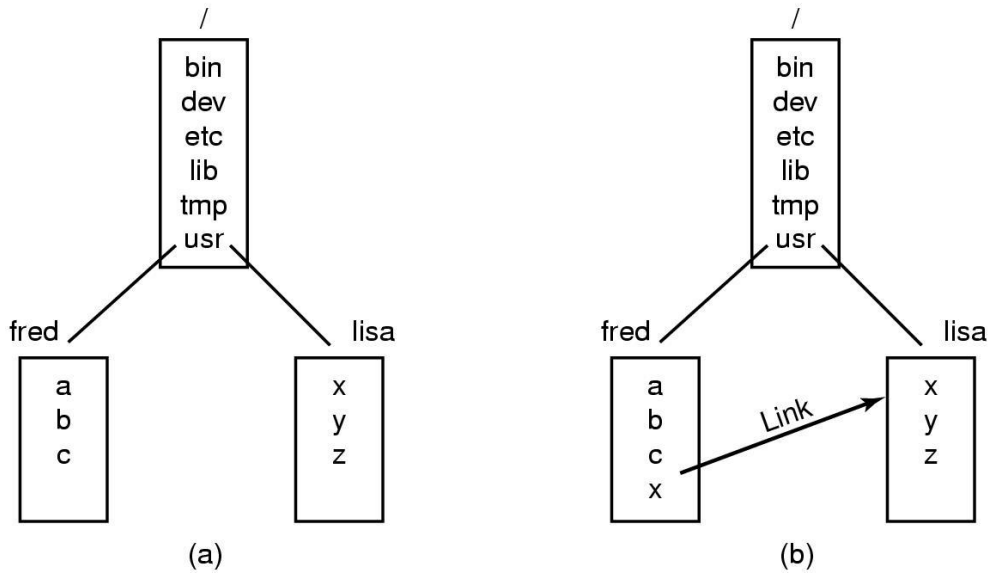


Figure (a) Before linking. (b) After linking.

- For example consider a situation as shown in the figure. Fred and Liza are working together on project and each of them needs access to the other files. If fred has /usr/fred as is working directory then he can refer to the file x in Liza's directory as /usr/liza/x. alternatively Fred can create a new directory in his directory as in the figure.

(6) Explain the role or function of the kernel.

- Each process asks for the system resources like computing power, memory network connectivity etc. The kernel is the bulk of the executable code in charge of handling such request.
- The kernel is the main component of most computer operating systems. It is a bridge between applications and the actual data processing done at the hardware level.
  The following are the major role of the kernel :
- **Resource allocation :-**
  - The kernel's primary function is to manage the computer's resources and allow

other programs to run and use these resources. These resources are- CPU, Memory and I/O devices.

- **Process Management :-**
  - The kernel is in charge of creating, destroying and handling the input output of the process.
  - Communications amongst the different processes is the responsibility of the kernel.
- **Memory Management :-**
  - The memory is the major resource of the computer and the policy used to deal with it is a critical.
  - The kernel builds a virtual address space for all the process on the top of the resource. The different parts of the kernel interact with the memory management subsystem by using the function calls.
- **File System :-**
  - The kernel builds the structured file system on the top of the unstructured hardware.
  - Kernel also supports the multiple file system types that is different way of managing the data.
- **Device Control :-**
  - Every system maps into the physical device.
  - All the device control operations are performed by the code that is specific to the device being addressed. This code is called device driver.
- **Networking :-**
  - It will be managed by the operating system and also control the program execution according to the network activity.
  - Routing and address resolution issues are also handled by the kernel.
- **Inter - Process Communication**
  - Kernel provides methods for Synchronization and Communication between processes called Inter- Process Communication (IPC).
  - There are various approaches of IPC say, semaphore, shared memory, message queue, pipe (or named fifo), etc.
- **Security or Protection Management**
  - Kernel also provides protection from faults (error control) and from malicious behaviors (Security).
  - One approach toward this can be Language based protection system, in which the kernel will only allow code to execute which has been produced by a trusted

language compiler.

(7) Explain in brief roles/ functions of shell.

- UNIX shell act as a command interpreter.
- It gathers input from user and executes programs based on that input, when a program finishes executing; it displays that program's output.
- It is primary interface between a user sitting at his terminal and operating system, unless the user is not using a graphical interface.
- A shell is an environment in which user can run our commands, programs, and shell scripts.
- There can various kinds of shells such as sh (bourne shell), csh (C shell), ksh (korn shell) and bash.
- When any user logs in, a shell is started.
- The shell has the terminal as standard input and standard output.
- It starts out by typing the prompt, a character such as $, which tells user that shell is waiting to accept command.
- For example if user types date command,
  $ date
  sat Mar 12 08:30:19 IST 2016
- The shell creates a child process and runs date program as child.
- While the child process is running, the shell waits for it to terminate.
- When child finishes, the shell types the prompt again and tries to read the next input line.
- Shell is work as interface, command interpreter and programming language.

- **Shell - As interface**
  - Shell is interface between user and computer.
  - User can directly interact with shell.
  - Shell provide command prompt to user to execute commands.
- **Shell - As command interpreter**
  - It read command enter by user on prompt.
  - It Interpret the command, so kernel can understand it easily.
- **Shell - As programming language**
  - Shell is also work as programming language.
  - It provides all features of programming language like variables, control structures and loop structures.

(8) Give comparison of UNIX and Windows Operating System.

- UNIX is a open source operating system (we can modify the code), easily available free of cost. Windows requires licensing and is a paid operating system.
- UNIX is a command based operating system while windows is menu based.
- In UNIX I/O components are stored as file while windows does not support this functionality.
- Windows must boot from a primary partition. UNIX can boot from either a primary partition or a logical partition inside an extended partition.
- Windows allows programs to store user information (files and settings) anywhere. This makes it impossibly hard to backup user data files and settings and to switch to a new computer.
- UNIX is CLUI (Command Line User Interface) operating system while a window is GUI (Graphical User Interface) Operating System.
- Windows program are event driven i.e. main program waits for some events to happen and calls a procedure to handle it. In contrast unix program consist of code that does something or makes system calls to get some services done.
- In UNIX there is one to one relation between system calls and library procedures, while in windows library calls and system calls are decoupled. Win 32 API(Application Program Interface) are used to get operating system services.
- UNIX supports various functionality as linking files, mounting and unmounting of files, file permissions using chmod command, which are not present in windows.
- UNIX is less affected by viruses and malwares and hence more secure compared to windows.
- Examples of UNIX operating system are Ubuntu, Fedora, Red Hat, Debian, Archlinux, Androidetc while examples of windows operating system are Windows 8, 8.1, 7, Vista, XP.

1) **cal**:- Displays a calendar

Syntax:- cal [options] [month] [year]

Description :-

- cal displays a simple calendar. If arguments are not specified, the current month is displayed. The switching options are as follows:

-1	Display single (current) month output. (This is the default.)
-3	Display prev/current/next month output
-s	Display Sunday as the first day of the week (This is the default.)
-m	Display Monday as the first day of the week
-j	Display Julian dates (days one-based, numbered from January 1)
-y	Display a calendar for the current year

Example:-

$cal

or

$cal 02 2016

Feb 2016

Su	**Mo**	**Tu**	**We**	**Th**	**Fr**	**Sa**
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					

2) **clear** :- It clears the terminal screen.

Syntax :- clear

Description :-

- Clear clears your screen if this is possible, including its scroll back buffer.
- Clear ignores any command-line parameters that may be present.

3) **pwd :-** Displays path from root to current directory

Syntax :- pwd [options]

Example:

$ pwd

/home/kumar/progs

4) **cd**:- It is used to change the directory.

Syntax :- cd [directory]

Description:-

- Used to go back one directory on the majority of all UNIX shells. It is important that the space be between the cd and directory name or ..

Example:-

$ pwd
/home/kumar
$ cd progs
$ pwd
/home/kumar/progs
$ cd ..
/home/kumar

5) **ls**:- Lists the contents of a directory

Syntax :- ls [options]

Description :-

-a	Shows you all files, even files that are hidden (these files begin with a dot.)
-A	List all files including the hidden files. However, does not display the working directory (.) or the parent directory (..).
-d	If an argument is a directory it only lists its name not its contents
-l	Shows you huge amounts of information (permissions, owners, size, and when last modified.)
-p	Displays a slash (/) in front of all directories
-r	Reverses the order of how the files are displayed
-R	Includes the contents of subdirectories

Example:-

$ ls – l
-rw-r----- 1 student student 23 Jan 16 15:27 file1.txt

Field Explanation:

- If first character is – then it is normal file
- If it is d then it is directory
- **Field 1 – File Permissions**: Next 9 character specifies the files permission. Each 3 characters refers to the read, write, execute permissions for user, group and world In this example, -rw-r—– indicates read-write permission for user, read permission for group, and no permission for others.
- **Field 2 – Number of links**: Second field specifies the number of links for that file. In this example, 1 indicates only one link to this file.
- **Field 3 – Owner**: Third field specifies owner of the file. In this example, this file is owned by username 'student'.
- **Field 4 – Group**: Fourth field specifies the group of the file. In this example, this file belongs to "student' group.
- **Field 5 – Size**: Fifth field specifies the size of file. In this example, '13' indicates the

file size.

- **Field 6 – Last modified date & time**: Sixth field specifies the date and time of the last modification of the file. In this example, 'Jan 16 15:27' specifies the last modification time of the file.
- **Field 7 – File or directory name**: The last field is the name of the file or directory. In this example, the file name is file1.txt

6) **exit** :- It is used to terminate a program, shell or log you out of a network normally.
Syntax :- exit

7) **echo** :- It prints the given input string to standard output.
Syntax :- echo string
Example:-

$ echo "hi.. hello unix"
hi.. hello unix

8) **who** :- who command can list the names of users currently logged in, their terminal, the time they have been logged in, and the name of the host from which they have logged in.
Syntax :- who [options] [file]
Description:-

am i	Print the username of the invoking user, The 'am' and 'i' must be space separated.
-b	Prints time of last system boot.
-d	print dead processes.
-H	Print column headings above the output.
-l	Include idle time as HOURS:MINUTES. An idle time of . indicates activity within the last minute.
-m	Same as who am i.
-q	Prints only the usernames and the user count/total no of users logged in.

Example :-

$ who
dietstaffpts/1 2016-02-20 22:42 (:0.0)
dietstaffpts/2 2016-02-20 09:30 (:0.0)

- Here first column shows user name, second shows name of the terminal the user is working on. Third& fourth column shows date and time of logging, last column shows machine name.

9) **who am i**:- Print effective userid
Syntax :- who am i

Description: - Print the user name associated with the current effective user id.
Example :-

$ who am i
dietstaffpts/3 2016-02-10 08:52 (:0.0)

- Here first column shows user name, second shows name of the terminal the user is working on. Third & fourth column shows date and time of logging, last column shows machine name.

10) **mkdir**:- This command is used to create a new directory
Syntax :- mkdir [options] directory
Description :-

-m	Set permission mode (as in chmod)
-p	No error if existing, make parent directories as needed.
-v	Print a message for each created directory
directory	The name of the directory that you wish to create

Example:-

$ mkdir aaa

- The above command will create directory named aaa under the current directory. We can also create number of subdirectories with one mkdir command.

11) **rmdir**:- It is used to delete/remove a directory and its subdirectories.
Syntax :- rmdir [options..] Directory
Description :-

- It removes only empty directory.

-p	Allow users to remove the directory and its parent directories which become empty.

12) **bc**:- bc command is used for command line calculator. It is similar to basic calculator. By using which we can do basic mathematical calculations.
Syntax :- bc [options]
Description :-

- bc is a language that supports arbitrary precision numbers with interactive execution of statements.
- bc starts by processing code from all the files listed on the command line in the order listed. After all files have been processed, bc reads from the standard input. All code is executed as it is read.

-q	To avoid bc welcome message
-l	To include math library functionalities

Example:-

$ bc

bc 1.06 Copyright 1991-1994,1997,1998,2000 Free Software Foundation,Inc. This is free software with ABSOLUTELY NO WARRANTY. For details type `warranty'. 2*3
6

- The above command used is for mathematical calculations.

$ bc -l

bc 1.06 Copyright 1991-1994,1997,1998,2000 Free Software Foundation,Inc. This is free software with ABSOLUTELY NO WARRANTY.For details type `warranty'.11+2
13

- The above command displays the sum of '11+2'.

$ bc calc.txt

bc 1.06 Copyright 1991-1994,1997,1998,2000 Free Software Foundation,Inc. This is free software with ABSOLUTELY NO WARRANTY. For details type `warranty'. 13

- 'calc.txt' file have the following code:11+2. Get the input from file and displays the output.

13) **uname**:- It is used to print system information.

Syntax :- uname [options]

Description :-

- Print certain system information.

-s	print the kernel name
-n	print the network node hostname
-r	print the kernel release
-v	print the kernel version
-m	print the machine hardware name
-o	print the operating system

Example:-

$ uname

Linux

14) **tty**:- Print the file name of the terminal connected to standard input.

Syntax :- tty

Description :-

- tty writes the name of the terminal that is connected to standard input onto

standard output.

- Command is very simple and needs no arguments.

Example :-

$tty

/student/tty10

15) stty:- Change and print terminal line settings.

Syntax :- sty

Description :-

- stty sets certain terminal I/O modes for the device that is the current standard input.
- Without arguments, it writes the settings of certain modes to standard output.

Example :-

$ sty

speed 19200 baud, 25 rows, 79 columns

kill = ^U

16) cat:- It is used to create, display and concatenate file contents.

Syntax : - cat [options] [FILE]...

Description :-

-A	Show all.
-b	Omits line numbers for blank space in the output.
-e	A $ character will be printed at the end of each line prior to a new line.
-E	Displays a $ (dollar sign) at the end of each line.
-n	Line numbers for all the output lines.
-s	If the output has multiple empty lines it replaces it with one empty line.
-T	Displays the tab characters in the output.
-v	Non-printing characters (with the exception of tabs, new-lines and form-feeds) are printed visibly.

- Two basically three uses of the cat command.

1) Create new files.
2) Display the contents of an existing file.
3) Concatenate the content of multiple files and display.

Example :-

$ cat file1.c

- Above syntax will display the content of file1.c

$ cat > file1.c

- Above syntax creates file1.c and allow us to insert content for this file.

- After inserting content you can use ctrl+d to exit the file.
- If file with same name exist then it will overwrite that file.

$ cat file1.c
process management
memory management
file mgmt
$ cat file1.c >> file2.c

- It can concatenate the contents of two files. For this you have to use append output redirection operator.
- The contents of file2.cwill be appended to file1.c.

17) **cp**:- cp command copy files from one location to another. If the destination is an existing file, then the file is overwritten; if the destination is an existing directory, the file is copied into the directory (the directory is not overwritten).

Syntax :- cp [options]... source destination

Description:-

- Here, after cp command contents of both source file and destination file files are the same.
- It will copy the content of source file to destination file.
- If the destination file doesn't exist, it will be created.
- If it exists then it will be overwritten without any warning.
- If there is only one file to be copied then destination can be the ordinary file or the directory file.

-a	archive files
-f	force copy by removing the destination file if needed
-i	interactive - ask before overwrite
-l	link files instead of copy
-L	follow symbolic links
-n	no file overwrite
-u	update - copy when source is newer than dest

Example:-

$ cp file1 file2

- The above cp command copies the content of file1.php to file2.php.

Copy folder and subfolders:

$ cp-R scripts scripts1

- The above cp command copy the folder and subfolders from scripts to scripts1

18) **rm**:- It is used to remove/delete the file from the directory.

Syntax :- rm [options..] [file|directory]

Description :-

- Files can be deleted with rm. It can delete more than one file with a single invocation. For deleting a single file we have to use rm command with filename to be deleted.
- Deleted file can't be recovered. rm can't delete the directories. If we want to remove all the files from the particular directory we can use the * symbol.

-f	Ignore nonexistent files, and never prompt before removing.
-i	Prompt before every removal.

Examlple :-

$ rm myfile.txt

- Remove the file myfile.txt. If the file is write-protected, you will be prompted to confirm that you really want to delete it.

  $ rm *
- Remove all files in the working directory. If it is write-protected, you will be prompted before rm removes it.

  $ rm -f myfile.txt
- Remove the file myfile.txt. You will not be prompted, even if the file is write-protected; if rm can delete the file, it will.

  $ rm -f *
- Remove all files in the working directory. rm will not prompt you for any reason before deleting them.

  $ rm -i *
- Attempt to remove every file in the working directory, but prompt before each file to confirm.

19) mv:- It is used to move/rename file from one directory to another.

Syntax :- mv [options] oldname newname

Description :-

- mv command which is short for move.
- mv command is different from cp command as it completely removes the file from the source and moves to the directory specified, where cp command just copies the content from one file to another.
- mv has two functions: it renames a file and it moves a group of files to a different directory. Mv doesn't create a copy of the file , it merely renames it. No additional space is consumed on disk during renaming. For example if we rename a file os to os1 and then if we try to read file os we will get error message as it is renamed to os1 there is no existence of file named os.

-f	mv will move the file(s) without prompting even if it is writing over an existing target. Note that this is the default if the standard input is not a terminal
-i	Prompts before overwriting another file

Example:-

$ cat file1

Memory
Process
Files

$ mv file1 file2

- rename file1 to file2
- If the destination file doesn't exist it will be created. mv can also be used to rename a directory. A group of files can also be moved to a directory. mv doesn't prompt for overwriting destination file if it exists.

20) nl :-

nl numbers the lines in a file.

Syntax: - nl [OPTION] [FILE]

Example :-

$cat list.txt

apples
oranges
potatoes
lemons
garlic

$nl list.txt

1 apples
2 oranges
3 potatoes
4 lemons
5 garlic

- In the above example, we use the cat command to display the contents of list.txt. Then we use nl to number each line and display the result to standard output.

$nl list.txt > nlist.txt

$cat nlist.txt

1 apples
2 oranges
3 potatoes
4 lemons

5 garlic

- In the above example, we run the same nl command, but redirect the output to a new file, nlist.txt. Then we use cat to display the results.

21) **cut** :- cut command is used to cut out selected fields of each line of a file. The cut command uses delimiters to determine where to split fields.

Syntax :- cut [options] filename

Description :-

file	A path name of an input file. If no file operands are specified, or if a file operand is -, the standard input will be used
-c	The list following -c specifies character positions
-d	The character following -d is the field delimiter
-f	select only these fields on each line
-b	Select only the bytes from each line as specified in LIST

Example :-

- For example, let's say you have a file named data.txt which contains the following text:
  one two three four five
  alpha beta gamma delta epsilon
- In this example, each of these words is separated by a tab character, not spaces. The tab character is the default delimiter of cut, so it will by default consider a field to be anything delimited by a tab.
- To "cut" only the third field of each line, use the command:
  $ cut -d' ' -f 3 data.txt
  three
  gamma
- let's say you want the third field and every field after it, omitting the first two fields. In this case, you could use the command:
  $ cut -d' ' -f 3- data.txt
  three four five
  gamma delta epsilon
  $ cut -c 3 file.txt
  r
  m
- For example, to output only the third-through-twelfth character of every line of data.txt, use the command:
  $ cut -c 3-12 data.txt
  e two thre
  pha beta g

22) **paste**:- paste command is used to paste the content from one file to another file. It is also used to set column format for each line.

Syntax :- paste [option] file

Description :-

- Paste prints lines consisting of sequentially corresponding lines of each specified file. In the output the original lines are separated by TABs. The output line is terminated with a newline.

-d	Specify of a list of delimiters.
-s	Paste one file at a time instead of in parallel.
File	A path name of an input file. If - is specified for one or more of the file s, the standard input will be used.

Example:-

$ cat > a
Unix
Linux
Windows
$ cat > b
Dedicated server
Virtual server
$ paste a b
Unix Dedicated server
Linux Virtual server
Windows
$ paste -d"|" a b
Unix|Dedicated server
Linux|Virtual server
Windows|
$ paste –s a b
Unix Linux Windows
Dedicated server Virtual server
$ paste –s –d”,” a b
Unix,Linux,Windows
Dedicated server,Virtual server

23) **more**:- Displays text one screen at a time.

Syntax :- more [options] filename

Description :-

- More command displays its output a page at a time. For example we are having a

big file with thousands of records and we want to read that file then we should use more command. Consider a file named employees then we will use following command to read its contents a page at a time.

-c	Clear screen before displaying.
-e	Exit immediately after writing the last line of the last file in the argument list.
-n	Specify how many lines are printed in the screen for a given file.
+n	Starts up the file from the given number.

Example:-

$ more -c myfile.txt

- Clears the screen before printing the file .

$ more -3 myfile.txt

- Prints first three lines of the given file. Press Enter to display the file line by line.

$ more +/"hope" myfile.txt

- Display the contents of file myfile.txt, beginning at the first line containing the string "hope".

24) cmp:- It compares two files and tells you which line numbers are different.

Syntax : - cmp [options..] file1 file2

Description :-

- Let's create a file named os2. And use cmp command to compare os and os1files.

- c	Output differing bytes as characters.
- l	Print the byte number (decimal) and the differing byte values (octal) for each difference.
- s	Prints nothing for differing files, return exit status only.
- c	Output differing bytes as characters.

Example:-

$ cat file1
memory
process
files
$ cat > file2
memory
process
files mgmt
$ cmp file1 file2
File1 file2 differ: char 21, line 3

- The two files are compared byte by byte and the location of the first mismatch is echoed to the screen. cmp doesn't bother about possible subsequent mismatches.

25) comm:- compare two sorted files line by line

Syntax:- comm [option]... FILE1 FILE2

Description :-

- Compare sorted files FILE1 and FILE2 line by line.
- Requires two sorted files and lists differing entries in different columns. produces three text columns as output:
  1 Lines only in file1.
  2 Lines only in file2.
  3 Lines in both files.

-1	suppress lines unique to FILE1
-2	suppress lines unique to FILE2
-3	suppress lines that appear in both files

- For example let's create two files named file1 and file2 with following data.
  $ cat > file1
  c.k.shukla
  chanchalsanghvi
  s.n.dasgupta
  sumit [1] + Stopped

  $ cat > file2
  anilaggarwal
  barunsengupta
  c.k.shukla
  lalit
- Now let's use comm. Command with these two files.
  $ comm file1 file2
  anilaggarwal
  barunsengupta
  c.k.shukla
  chanchalsanghvi
  lalit
  s.n.dasgupta
  sumit
- In the above output we can see that first column contains two lines unique to the first file and second column contains three lines unique to the second file and the third column contains two lines common to both the files. Comm. Can produce the

single column output using 3 options -1,-2 or -3. To drop a particular column, simply use its column number as a prefix.

Example :-

$ cat file1

f1.c
f2.c
f3.c
f4.c
f5.c

$ cat file2

f1.c
f3.c
f4.c
f6.c
f7.c

- Now, When you run the comm command on these files, this is what you get:

$ comm file1 file2

```
                f1.c
f2.c
                f3.c
                f4.c
f5.c
        f6.c
        f7.c
```

- The output is split in 3 columns. Column1 indicates files which are unique in file1, column 2 indicates files unique to file2. Column 3 indicates files common between them. comm command provides some real good options with which you can filter the output better.
- Now, say you want to find out only the list of files which were there in the older version but not in the newer version:

$ comm -23 file1 file2

f2.c
f5.c

- The option -23 indicates to remove the second and third columns from the comm command output, and hence we are left with only the first column which is the files unique in file1.
- Similarly, to find out the list of files which were not there in the old version, but has been added in the new version:

$ comm -13 file1 file2
f6.c
f7.c

- As explained above, -13 option tells to remove the first and third columns from the comm output.
- Finally, to know the list of files which have been retained, or common in both the versions:
  $ comm -12 file1 file2
  f1.c
  f3.c
  f4.c
- When you apply comm command on files, the files should be sorted. This command works only on sorted files.

26) diff:- It is used to find differences between two files.

Syntax :- diff [options..] fileone filetwo

Description :-

- Diff is the third command that can be used to display file differences. Unlike its fellow members ,cmp and comm. , it also tells us which lines in one file have to be changed to make the two files identical.

-b	Ignore any changes which only change the amount of whitespace (such as spaces or tabs).
-w	Ignore whitespace entirely.
-B	Ignore blank lines when calculating differences.
-y	Display output in two columns.
-i	Ignore changes in case.consider upper- and lower-case letters equivalent.

Example:-

$ cat 1.txt
aaa
bbb
ccc
ddd
eee
fff
ggg
$ cat 2.txt
bbb
c c
ddd
eee
fff

ggg
hhh
$ diff 1.txt 2.txt
1d0
< aaa
3c2
< ccc

> c c
7a7
> hhh

- Lines like "1d0" and "3c2" are the coordinates and types of the differences between the two compared files, while lines like "< aaa" and "> hhh" are the differences themselves.
- diff change notation includes 2 numbers and a character between them. Characters tell you what kind of change was discovered:
  - **d – a line was deleted**
  - **c – a line was changed**
  - **a – a line was added**
- Number to the left of the character gives you the line number in the original (first) file, and the number to the right of the character tells you the line number in the second file used in comparison.
- So, looking at the two text files and the diff output above, you can see what happened:
- **1d0**
  < aaa
- This means that 1 line was deleted. < aaa suggests that the aaa line is present only in the original file.
  3c2
  < ccc

  > c c
- And this means that the line number 3 has changed. You can see how this confirms that in the first file the line was "ccc", and in the second it now is "c c".
  7a7
  > hhh
- Finally, this confirms that one new line appeared in the second file, it's "hhh" in the line number 7.

27) chmod:- chmod command allows you to alter / Change access rights to files and directories.

Syntax:- chmod [options] [MODE] FileName

Description :-

- chmod command is used to set the permissions of one or more files for all three categories of users (user,group and others). It can be run only by the user and super user. Command can be used in two ways. Let's first take a look at the abbreviations used by chmod command.

Category		Operation		Permission	
u	User	+	Assigns permission	r	Read permission
g	Group	-	Removes permission	w	Write permission
o	Others	=	Assigns absolute	x	Execute permission

File Permission :

#	File Permission
0	none
1	execute only
2	write only
3	write and execute
4	read only
5	read and execute
6	read and write
7	set all permissions

Relative permissions

- When changing permissions in a relative manner, chmod only changes the permissions specified in the command line and leaves the other permissions unchanged. In this mode it uses the following syntax:
- let's first check the permission of file shortlist. It shows read/write permission. Let's assign execute permission to this file for owner.

  $ ls –l

  -rw-r--r—1 shortlist None 241 Feb 21 04:02

  $ chmod u+x shortlist

  $ ls -l

  -rwxr--r-- 1 shortlist None 241 Feb 21 04:02
- Here chmod uses expression u+x means – u shows user category that is user, + shows we need to assign the permission and x shows execute permission. Sameway we can assign other permissions.

Absolute permissions

- Sometimes you don't need to know what a file's current permissions are, but want to set all nine permission bits explicitly. The expression used by chmod here is a string of three octal numbers. Each type of permission is assigned a number as shown:
  Read permission-4
  Write permission-2
  Execute permission -1
- For each category we add numbers that represent the assigned permission. For instance , 6 represents read and write permission , and 7 represents all permissions.

Example :

$ chmod 666 myfile or(chmod u=rw,g=rw,o=rwmyfile) shortlist

- Shows read and wrie (4 +2) permissions for all three types of users.

28) **chown**:- Command for system V that changes the owner of a file.
Syntax :- chown [options] newowner filename/directoryname
Description:-

-R	Change the permission on files that are in the subdirectories of the directory that you are currently in.
-c	Change the permission for each file.
-f	Prevents chown from displaying error messages when it is unable to change the ownership of a file.

Example :-

$ ls -l demo.txt
-rw-r--r-- 1 root root 0 Jan 31 05:48 demo.txt
$ chownvivek demo.txt
-rw-r--r-- 1 vivek root 0 Jan 31 05:48 demo.txt

29) **chgrp**:- chgrp command is used to change the group of the file or directory. This is an admin command. Root user only can change the group of the file or directory.
Syntax:- chgrp [options] newgroup filename/directoryname
Description:-

-R	Change the permission on files that are in the subdirectories of the directory that you are currently in.
-c	Change the permission for each file.
-f	Force. Do not report errors.

Example :-

$ ls -l demo.txt
-rw-r--r-- 1 root root 0 Jan 31 05:48 demo.txt

$ chgrp vivek demo.txt
-rw-r--r-- 1 root vivek 0 Jan 31 05:48 demo.txt

30) file:- file command tells you if the object you are looking at is a file or a directory.
Syntax:- file [options] directoryname/filename
Description:-

- File command is used to determine the type of file, especially of an ordinary file. We can use it with one or more filenames as arguments. For example we can use file command to check the type of the os1 file that we have created.
  $ file os1
  os1: short english text

31) finger:- finger command displays the user's login name, real name, terminal name and write status (as a ''*'' after the terminal name if write permission is denied), idle time, login time, office location and office phone number.
Syntax:- finger [username]
Description :-

-l	Force long output format
-s	Force short output format

Example :-

$ finger rahul
Login: abc Name: (null)
Directory: /home/abc Shell: /bin/bash
On since Mon Nov 1 18:45 (IST) on :0 (messages off)
On since Mon Nov 1 18:46 (IST) on pts/0 from :0.0
New mail received Thu Feb 7 10:33 2015 (IST)
Unread since Mon Feb 8 12:59 2016 (IST)
No Plan.

32) sleep:- Delay for a specified amount of time
Syntax :- sleep NUMBER[SUFFIX]
Description:-

- The sleep command pauses for an amount of time defined by NUMBER.
- SUFFIX may be "s" for seconds (the default), "m" for minutes, "h" for hours, or "d" for days.

Example :-

- To sleep for 5 seconds, use:
  $ sleep 5
- To sleep for 2 minutes, use:

$ sleep 2m

- To sleep for 3 hours, use:
  $ sleep 3h
- To sleep for 5 days, use:
  $ sleep 5d

33) kill:- kill command is used to kill the background process.
Syntax:- kill [options] pid
Description :-

- The command kill sends the specified signal to the specified process or process group.
- If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal.
- For other processes, it may be necessary to use the KILL (9) signal, since this signal cannot be caught.

pid.	Specify the list of processes that kill should signal
-s	send the specified signal to the process
-l	list all the available signals.
-9	Force to kill a process.

34) users :- users command displays currently logged in users.
Syntax:- users
Output:-

$ users
dietstaff dietstaff

35) ps:- It is used to report the process status. ps is the short name for Process Status.
Syntax:- ps [options]

Description :-

-a	List information about all processes most frequently requested: all those except process group leaders and processes not associated with a terminal
-A	List information for all processes. Identical to -e, below
-f	Generate a full listing
-j	Print session ID and process group ID
-l	Generate a long listing

Example :-

$ps

```
PID  TTY      TIME  CMD
291 console 0:00   bash
```

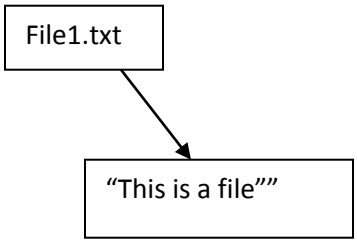
36) ln :- ln command is used to create link to a file (or) directory. It helps to provide soft link for desired files.

Syntax:- ln [options] existingfile(or directory)name newfile(or directory)name

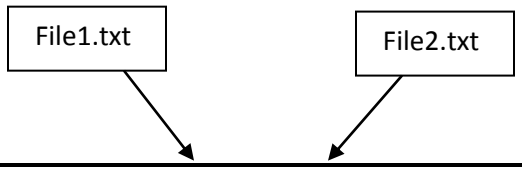
Description:-

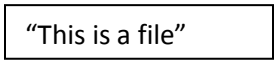
What Is A Link?

- A link is an entry in your file system which connects a filename to the actual bytes of data on the disk. More than one filename can "link" to the same data. Here's an example. Let's create a file named file1.txt:
  $ echo "This is a file." > file1.txt
- This command echoes the string "This is a file". Normally this would simply echo to our terminal, but the > operator redirects the string's text to a file, in this case file1.txt
- When this file was created, the operating system wrote the bytes to a location on the disk and also linked that data to a filename, file1.txt so that we can refer to the file in commands and arguments.
- If you rename the file, the contents of the file are not altered; only the information that points to it.
- The filename and the file's data are two separate entities.

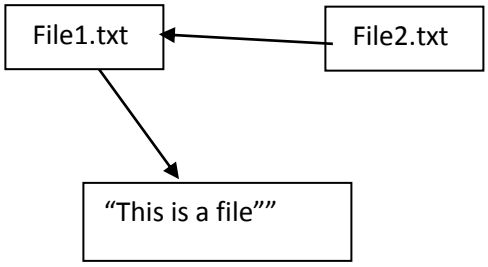


- What the link command does is allow us to manually create a link to file data that already exists.
- So, let's use link to create our own link to the file data we just created. In essence, we'll create another file name for the data that already exists.
  $ link file1.txt file2.txt
- The important thing to realize is that we did not make a copy of this data. Both filenames point to the same bytes of data on the disk. Here's an illustration to help you visualize it:





- If we change the contents of the data pointed to by either one of these files, the other file's contents are changed as well. Let's append a line to one of them using the >>operator:
  $ echo "It points to data on the disk." >> file1.txt
- Now let's look at the contents of file1.txt:
  $ cat file1.txt
  This is a file.
- now let's look at the second file, the one we created with the link command.
  $ cat file2.txt
  This is a file.
- ln, by default, creates a hard link just like link does. So this ln command:
  $ ln file1.txt file2.txt
- It is the same as the following link command. Because, both commands create a hard link named file2.txt which links to the data offile1.txt.
  $ link file1.txt file2.txt
- However, we can also use ln to create symbolic links with the -s option. So the command:
  $ ln -s file1.txt file2.txt
- It will create a symbolic link to file1.txt named file2.txt. In contrast to our hard link example, here's an illustration to help you visualize our symbolic link:



- You should also be aware that, unlike hard links, removing the file (or directory) that a symlink(symbolic linkl) points to will break the link. So if we create file1.txt:
  $ echo "This is a file." > file1.txt
- Now, create a symbolic link to it:
  $ ln -s file1.txt file2.txt
- we can cat either one of these to see the contents:

$ cat file1.txt
This is a file.
$ cat file2.txt
This is a file.

- But, if we remove file1.txt:
  $ rm file1.txt
- we can no longer access the data it contained with our symlink:
  $ cat file2.txt
  cat: file2.txt: No such file or directory

-s	Makes it so that it creates a symbolic link
-f	If the destination file or files already exist, overwrite them.

37) **head**:- head command is used to display the first ten lines of a file, and also specifies how many lines to display.
Syntax:- head [options] filename
Description:-

- Head command displays the top of the file. When used without an option , it displays the first ten lines of the file.

-n	To specify how many lines you want to display.
-n number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in lines.
-c number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes.

Example :-

$ head myfile.txt

- Display the first ten lines of myfile.txt.
  $ head -15 myfile.txt
- Display the first fifteen lines of myfile.txt.
  $ head -c 20 myfile.txt
- Will output only the first twenty bytes (characters) of myfile.txt. Newlines count as a single character, so if head prints out a newline, it will count it as a byte.
  $ head -n 5K myfile.txt
- Displays the first 5,000 lines of myfile.txt.

38) **tail**:- tail command is used to display the last or bottom part of the file. By default it displays last 10 lines of a file.
Syntax :- tail [options] filename
Description:-

-l	To specify the units of lines.
-b	To specify the units of blocks.
-n	To specify how many lines you want to display.
-c number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes.
-n number	The number option-argument must be a decimal integer whose sign affects the location in the file, measured in lines.

Example : -

$ tail index.txt

- It displays the last 10 lines of 'index.txt'.

$ tail -2 index.txt

- It displays the last 2 lines of 'index.txt'.

$ tail -n 5 index.txt

- It displays the last 5 lines of 'index.txt'.

$ tail -c 5 index.txt

- It displays the last 5 characters of 'index.txt'.

39) **sort**:- It is used to sort the lines in a text file.

Syntax:- sort [options] filename

-b	Ignores spaces at beginning of the line
-c	Check whether input is sorted; do not sort
-r	Sorts in reverse order
-u	If line is duplicated only display once
-r	Reverse the result of comparisons.

Example:-

$ cat > data.txt
apples
oranges
pears
kiwis
bananas
$ sort data.txt
apples
bananas
kiwis
oranges
pears

- Note that this command does not actually change the input file, data.txt. If you want to write the output to a new file, output.txt, redirect the output like this:

$ sort data.txt > output.txt

- it will not display any output, but will create the file output.txt with the same sorted data from the previous command. To check the output, use the cat command:

  $ cat output.txt

  apples
  bananas
  kiwis
  oranges
  pears
- You can also use the built-in sort option -o, which allows you to specify an output file:

  $ sort -o output.txt data.txt
- Using the -o option is functionally the same as redirecting the output to a file; neither one has an advantage over the other.

Sorting In Reverse Order :

- You can perform a reverse-order sort using the -r flag. For example, the following command:

  $ sort -r data.txt

  pears
  oranges
  kiwis
  bananas
  apples

Checking For Sorted Order

- If you just want to check to see if your input file is already sorted, use the -c option:

  $ sort -c data.txt
- If your data is unsorted, you will receive an informational message reporting the line number of the first unsorted data, and what the unsorted data is.

40) find:- Finds one or more files assuming that you know their approximate path.

Syntax :- find [options] path

Description :-

- Find is one of the powerful utility of Unix (or Linux) used for searching the files in a directory hierarchy

path	A path name of a starting point in the directory hierarchy
-maxdepth	Descend at most levels (a non-negative integer) levels of directories below the command line arguments.
-i	ignore the case in the current directory and sub-directories.

-size	Find file based on size

Example:-

$ find -name "sum.java"
./bkp/sum.java
./sum.java
$ find -iname "sum.java"
./SUM.java
./bkp/sum.java
./sum.java
$ find -maxdepth 1 -name "sum.java"
./sum.java

- Find Files Under Home Directory
- Find all the files under /home directory with name myfile.txt.
  $ find /home -name myfile.txt
  /home/myfile.txt
  $ find . -size 10M
- Display files whose size is exactly 10M
  $ find . -size +10M
- Display files larger than 10M size
  $ find -newer "sum.java"
- Display the files which are modified after the modification of a give file.
  $ find . -perm 777
- display the files based on the file permissions.
- This will display the files which have read, write, and execute permissions. To know the permissions of files and directories use the command **"ls -l".**
  $ find -name '*' -size +1000k
- The system would search for any file in the list that is larger than 1000k.

41) uniq:- Report or filter out repeated lines in a file.
Syntax:- uniq [option] filename
Description : -

-c	Precede each output line with a count of the number of times the line occurred in the input
-d	Suppress the writing of lines that are not repeated in the input
-D	Print all duplicate lines
-f	Avoid comparing first N fields
-i	Ignore case when comparing
-s	Avoid comparing first N characters.
-u	Prints only unique lines

-w	Compare no more than N characters in lines

Example:-

$ cat example.txt
Unix operating system
unix operating system
unix dedicated server
linux dedicated server

- The default behaviour of the uniq command is to suppress the duplicate line. Note that, you have to pass sorted input to the uniq, as it compares only successive lines.

  $ uniq example.txt
  unix operating system
  unix dedicated server
  linux dedicated server
- The -c option is used to find how many times each line occurs in the file. It prefixes each line with the count.

  $ uniq -c example.txt
  2 unix operating system
  1 unix dedicated server
  1 linux dedicated server
- You can print only the lines that occur more than once in a file using the -d option.

  $ uniq -d example.txt
  unix operating system
- The -D option prints all the duplicate lines.

  $ uniq -D example.txt
  unix operating system
  unix operating system

42) **tr**:- Translate characters.

Syntax:- tr [options] set1 [set2]

Description:-

-c	Complement the set of characters specified by string1
-d	Delete all occurrences of input characters that are specified by string1
-s	replaces repeated characters listed in the string1 with single occurrence
Set1	First string or character to be changed
set2	Second string or character to change the string1

Example:-

- Convert lower case letters to upper case
- The following tr command translates the lower case letters to capital letters in the

give string:
$ echo "linux dedicated server" | tr "[:lower:]" "[:upper:]"
LINUX DEDICATED SERVER
$ echo "linux dedicated server" | tr "[a-z]" "[A-Z]"
LINUX DEDICATED SERVER

- The -c option is used to replace the non-matching characters with another set of characters.
  $ echo "unix" | tr -c "u" "a"
  uaaa
- You can squeeze more than one occurrence of continuous characters with single occurrence. The following example squeezes two or more successive blank spaces into a single space.
  $ echo "linux server" | tr -s " "
  linux server
- The following example removes the word linux from the string.
  $ echo "linuxserver" | tr -d "linux"
  Server

43) history:- history command is used to list out the recently executed commands in the number line order.
Syntax:- history [options]
Description:-

- The history command performs one of several operations related to recently-executed commands recorded in a history list. Each of these recorded commands is referred to as an ``event''.

-c	clear the history list by deleting all of the entries.

Example :-

- To list the recently executed commands:
  $ history
- This command is used to list the history.
  To find specific command in history list:
  $ history | grep cd
  33 cd Pictures/
  37 cd ..
  39 cd Desktop/
  61 cd /usr/bin/
  68 cd
- This command is used to list only cd commands from history list.
  To copy history list to file

$ history -a history.txt

- This command is used to copy history list to history.txt file .

To Clear all history:

$ history -c

- This command can be used to clear all history from history list.

44) write:- Send a message to another user.

Syntax:- write person [ttyname]

Description:-

- The write utility allows you to communicate with other users, by copying lines from your terminal to theirs.
- When you run the write command, the user you are writing to gets a message of the format:
  Message from yourname@yourhost on yourtty at hh:mm ...
- Any further lines you enter will be copied to the specified user's terminal. If the other user wants to reply, they must run write as well.
- When you are done, type an end-of-file or interrupt character. The other user will see the message 'EOF' indicating that the conversation is over.

person	If you wish to talk to someone on your own machine, then person is just the person's login name. If you wish to talk to a user on another host, then person is of the form 'user@host'.
ttyname	If you wish to talk to a user who is logged in more than once, the ttyname argument may be used to indicate the appropriate terminal name, where ttyname is of the form 'ttyXX' or 'pts/X'.

45) grep:- It selects and prints the lines from a file which matches a given string or pattern.

Syntax:- grep [options] pattern [file]

Description:-

- This command searches the specified input fully for a match with the supplied pattern and displays it.
- While forming the patterns to be searched we can use shell match characters, or regular expressions.
- Let us begin with the simplest example of usage of grep.

-i	Ignore case distinctions
-v	Invert the sense of matching, to select non-matching lines.
-w	Select only those lines containing matches that form whole words.
-x	Select only matches that exactly match the whole line.
-c	print a count of matching lines for each input file.

Example :-

$ grep "Error" logfile.txt

- This searches for the string "Error" in the log file and prints all the lines that has the word "Error".

$ grep "string" file1 file2

- Searching for a string in multiple files.

$ grep -i "UNix" file.txt

- The -i option enables to search for a string case insensitively in the give file. It matches the words like "UNIX", "Unix", "unix".

$ grep -w "world" file.txt

- By default, grep matches the given string/pattern even if it found as a substring in a file. The -w option to grep makes it match only the whole words.

$ grep -c "sting" file.txt

- We can find the number of lines that matches the given string/pattern

$ grep -l "string" *

- We can just display the files that contain the given string/pattern.

$ grep -n "string" file.txt

- We can make the grep command to display the position of the line which contains the matched string in a file using the -n option

46) pwd:-Displaying your current directory name (Print working directory).

Syntax:-pwd [options]

Description:-

- At the time of logging in user is placed in the specific directory of the file system. You can move around from one directory to another, but any point of time, you are located in only one directory. This directory is known as your current directory. pwd command tells your current directory.

Example:-

$ pwd

/home/abc

47) wc:- Word Count (wc) command counts and displays the number of lines, words, character and number of bytes enclosed in a file.

Syntax: - wc [*options*] [filename]

Description:-

- This command counts lines, words and characters depending on the options used. It takes one or more filenames as its arguments and displays four-columnar output. For example let's read our os1 file. And we use wc command with that filename.

-l	print the newline counts.
-w	print the word counts.

-c	print the byte counts.
-m	print the character counts.
-L	print the length of the longest line.

Example :-

$cat myfile.txt
Red Hat
CentOS
Fedora
Debian
Scientific Linux
OpenSuse
Ubuntu
Xubuntu
Linux Mint
Pearl Linux
Slackware
Mandriva

- The ‘wc‘ command without passing any parameter will display a basic result of ”myfile.txt‘ file. The three numbers shown below are 12 (number of lines), 16 (number of words) and 112 (number of bytes) of the file.
  $wc myfile.txt
  12 16 112 myfile.txt
- To count number of newlines in a file use the option ‘-l‘, which prints the number of lines from a given file. Say, the following command will display the count of newlines in a file. In the output the first filed assigned as count and second field is the name of file.
  $wc -l myfile.txt
  12 myfile.txt
- Using ‘-w‘ argument with ‘wc‘ command prints the number of words in a file. Type the following command to count the words in a file.
  $wc -w myfile.txt
  16 myfile.txt
- When using options ‘-c‘ and ‘-m‘ with ‘wc‘ command will print the total number of bytes and characters respectively in a file.
  $wc -c myfile.txt
  112 myfile.txt
- The ‘wc‘command allow an argument ‘-L‘, it can be used to print out the length of

longest (number of characters) line in a file. So, we have the longest character line ('Scientific Linux') in a file.

$wc -L myfile.txt

16 myfile.txt

48) **man**:- man command which is short for manual, provides in depth information about the requested command (or) allows users to search for commands related to a particular keyword.

Syntax:- man commandname [options]

Description :-

-a	Print a one-line help message and exit.
-k	Searches for keywords in all of the manuals available.

Example:-

$ man mkdir

- Display the information about mkdir command

49) **| (Pipeline command)**:- Used to combine more than one command. Takes output of 1st command as input of 2nd command.

Syntax:- commmand1| command2|……

Example:-

$ls -l | grep "Feb"

```
-rw-r--r-- 1   dietstaffdietstaff    336   Feb 19 14:41 calc1.sh
-rw-r--r—1   dietstaffdietstaff    410    Feb 19 14:28 calc.sh
drwxr-xr-x 7 dietstaffdietstaff    4096  Feb 24 09:04 Desktop
drwxr-xr-x 7 dietstaffdietstaff    12288Feb 26 07:44 Downloads
drwxr-xr-x 3 dietstaffdietstaff    4096  Feb 26 08:55 lab
drwxr-xr-x 2 dietstaffdietstaff    4096  Feb 24 08:00 shellscript
-rw-r--r—1   dietstaffdietstaff    81      Feb 24 12:34 temp3.sh
```

Explain the basic system administration commands of unix

1) **date** : - Prints or sets the date and time.

Syntax :- date[options] [+format] [date]

Description :-

- Display the current date with current time, time zone.
- The command can also be used with suitable format specifies as arguments. Each format is preceded by a + symbol, followed by the % operator, and a single character describing the format.

Format

%a	Abbreviated weekday(Tue).

%A	Full weekday(Tuesday).
%b	Abbreviated month name(Jan).
%B	Full month name(January).
%c	Country-specific date and time format..
%D	Date in the format %m/%d/%y.
%j	Julian day of year (001-366).
%p	String to indicate a.m. or p.m.
%T	Time in the format %H:%M:%S.
%t	Tab space.
%V	Week number in year (01-52); start week on Monday.

Example:-

$ date
Fri Feb 19 09:55:15 IST 2016
$ date +%m
02
$ date +%h
Feb
$ date +%y
16
$ date +"Date is %D %t Time is %T"
date is 19/02/16 Time is 10:52:34

- To know the week number of the year,
  $ date -V
  11
- To set the date,
  $ date -s "10/08/2016 11:37:23"
- The above command will print Wed Oct 08 11:37:23 IST 2016

2) wall :- send a message to everybody's terminal.

Syntax :- wall [message]

- Wall sends a message to everybody logged in with their mesg(1) permission set to yes. The message can be given as an argument to *wall*, or it can be sent to *wall*'s standard input. When using the standard input from a terminal, the message should be terminated with the EOF key (usually Control-D).
- The length of the message is limited to 20 lines.

Example : -

$sudo wall message.txt

- Using the sudo command to run wall as the superuser, sends the contents ofmessage.txt to all users.

ulimit : Setting limits on file size

Syntax: - ulimit [limit]

Description :-

- The ulimit command imposes a restriction on the maximum size of the file that a user is permitted to create , so that there is less wastage of disk space.

3) passwd:- It is used to change your password.

Syntax:- passwd

Description :-

- Passwd changes the password or shell associated with the user given by name or the current user if name is omitted.
- First user has to insert current password. Then new password will be asked followed by confirm new password field.
- passwd command can also be used to change the home directory where the path stands for the home directory.

4) suid : - set user id.

Description :-

- suid (Set owner User ID up on execution) is a special type of file permissions given to a file.
- Normally in Linux/Unix when a program runs, it inherits access permissions from the logged in user.
- suid is defined as giving temporary permissions to a user to run a program/file with the permissions of the file owner rather that the user who runs it.
- In simple words users will get file owner's permissions as well as owner UID and GID when executing a file/program/command.

Example:-

$ chmod u+s a.out

$ ls –l a.out

-rw**s**r-xr-rw 1 dietstaff dietstaff 121 Feb 21 11:18 a.out

1) Write a shell script to scans the name of the command and executes it.

Program :-

```
echo "enter command name"
read cmd
$cmd
```

Output :-

```
enter command name
cal
February 2016
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29
```

2) Write a shell script Which works like calculator and performs below operations Addition , Subtract ,Division ,Multiplication

Program :-

i) using if..elif statement

```
echo " Enter one no."
read n1
echo "Enter second no."
read n2

echo "1.Addition"
echo "2.Subtraction"
echo "3.Multiplication"
echo "4.Division"
echo "Enter your choice"
read ch
if [ $ch = "1" ]
then
   sum=`expr $n1 + $n2`
    echo "Sum ="$sum
elif [ $ch = "2" ]
then
      sum=`expr $n1 - $n2`
    echo "Sub = "$sum
elif [ $ch = "3" ]
then
```

```
   sum=`expr $n1 \* $n2`
    echo "Mul = "$sum
elif [ $ch = "4" ]
then
   sum=`expr $n1 / $n2`
    echo "Div = "$sum
fi
```

Output :-

```
Enter one no.
32
Enter second no.
12
1.Addition
2.Subtraction
3.Multiplication
4.Division
Enter your choice
2
Sub = 20
```

ii) using while loop and switch statement

```
i="y"
while [ $i = "y" ]
do
echo " Enter one no."
read n1
echo "Enter second no."
read n2
echo "1.Addition"
echo "2.Subtraction"
echo "3.Multiplication"
echo "4.Division"
echo "Enter your choice"
read ch
case $ch in
   1)sum=`expr $n1 + $n2`
    echo "Sum ="$sum;;
    2)sum=`expr $n1 - $n2`
    echo "Sub = "$sum;;
   3)sum=`expr $n1 \* $n2`
```

```
    echo "Mul = "$sum;;
   4)sum=`expr $n1 / $n2`
    echo "Div = "$sum;;
   *)echo "Invalid choice";;
esac
echo "Do u want to continue ? y/n"
read i
if [ $i != "y" ]
then
   exit
fi
done
```

Output :-

```
Enter one no.
32
Enter second no.
22
1.Addition
2.Subtraction
3.Multiplication
4.Division
Enter your choice
2
Sub = 10
Do u want to continue ? y/n
N
```

3) Write a shell script to print the pyramid structure for the given number.

Program :-

```
echo "enter the number"
read n
printf "\n"
for((i=1;i<=$n;i++))
do
   for((j=1;j<=$i;j++))
do
   printf "$j"
done
   printf "\n"
done
```

Output :-

```
enter the number
4
1
12
123
1234
```

4) Write a shell script to find the largest among the 3 given numbers.

Program :-

```
clear
echo "Enter first number: "
read a
echo "Enter second number: "
read b
echo "Enter third number: "
read c

if [ $a -ge $b -a $a -ge $c ]
then
        echo "$a is largest integer"
elif [ $b -ge $a -a $b -ge $c ]
then
        echo "$b is largest integer"
elif [ $c -ge $a -a $c -ge $b ]
then
        echo "$c is largest integer"
fi
```

Output :-

```
Enter first number:
22
Enter second number:
33
Enter third number:
42
44 is largest integer
```

5) Write a shell script to find factorial of given number n.

Program :-

```
clear
```

```
fact=1
echo "Enter number to find factorial : "
read n
a=$n
#if enter value less than 0
if [ $n -le 0 ]
then
echo "invalid number"
exit
fi
#factorial logic
while [ $n -ge  1 ]
do
 fact=`expr $fact \* $n`
 n=`expr $n - 1`
done
echo "Factorial for $a is $fact"
```

Output :-

```
Enter number to find factorial :
5
Factorial for 5 is 120
```

6) Write a shell script to print all prime numbers from 1 to n.

Program :-

```
clear
echo "enter the range"
read n
echo "the prime no are:"
m=2
while [ $m -le $n ]
do
i=2
flag=0
while [ $i -le `expr $m / 2` ]
do
if [ `expr $m % $i` -eq 0 ]
then
flag=1
break
fi
```

```
i=`expr $i + 1`
done
if [ $flag -eq 0 ]
then
echo $m
fi
m=`expr $m + 1`
done
```

Output :-

```
enter the range
10
the prime no are
2
3
5
7
```

7) Write a shell script to reverse a number supplied by a user.

Program :-

```
if [ $# -eq 1 ]
      then
          if [ $1 -gt 0 ]
          then
             num=$1
             sumi=0
             while [ $num -ne 0 ]
             do
             lnum=`expr $num % 10`
             sumi=`expr $sumi \* 10 + $lnum`
             num=`expr $num / 10`
             done
             echo "Reverse of digits is  $sumi of $1"
              else
             echo " Number is less than 0"
           fi
  else
        echo "Insert only one parameter "
  fi
```

Output :-

```
bash pr81.sh 123
Reverse of digits is  321 of 123
```

8) Write a shell script to find first n Fibonacci numbers like: 0 1, 1, 2, 3, 5, 13,…

Program :-

```
clear
 echo "How many number of terms to be generated ?"
 read n
 x=0
 y=1
 i=2
 echo "Fibonacci Series up to $n terms :"
 echo "$x"
 echo "$y"
 while [ $i -lt $n ]
 do
    i=`expr $i + 1 `
    z=`expr $x + $y `
    echo "$z"
    x=$y
    y=$z
 done
```

Output :-

```
How many numbers of terms to be generated?
5
Fibonacci Series up to 5 terms :
0
1
1
2
3
```

9) Write a shell script to check whether the given number is Perfect or not.

Program :-

```
echo Enter a number
read no
i=1
ans=0
while [ $i -le `expr $no / 2` ]
```

```
  do
    if [ `expr $no % $i` -eq 0 ]
    then
      ans=`expr $ans + $i`
    fi
    i=`expr $i + 1`
done
if [ $no -eq $ans ]
then
echo $no is perfect
else
echo $no is NOT perfect
fi
```

Output :-

```
Enter a number
6
6 is perfect

Enter a number
10
10 is NOT perfect
```

10) Write a shell script which displays a list of all files in the current directory to which you have read, write and execute permissions

Program :-

```
for File in *
do
if [ -r $File -a -w $File -a -x $File ]
then
echo $File
fi
done
```

Output :-

```
Desktop
Documents
Downloads
lab
Music
```

Pictures
Public
shellscript
Templates
Videos

11) Write a shell script that deletes all the files in the current directory which are 0 bytes in length.

Program :-

```
clear
find . -name "*" -size -1k –delete
echo “files deleted”
```

Output :-

files deleted

12) Write a shell script to check whether the given string is Palindrome or not.

Program :-

```
clear
echo "Enter the string:"
read str
echo
len=`echo $str | wc -c`
len=`expr $len - 1`
i=1
j=`expr $len / 2`
while test $i -le $j
do
k=`echo $str | cut -c $i`
l=`echo $str | cut -c $len`
if test $k != $l
then
echo "String is not palindrome"
exit
fi
i=`expr $i + 1`
len=`expr $len - 1`
done
echo "String is palindrome"
```

Output :-

```
Enter the string:
abba
String is palindrome

Enter the string:
abc
String is not palindrome
```

13) Write a shell script to display the digits which are in odd position in a given 5 digit number

Program :-

```
echo "Enter a 5 digit number"
read num
n=1
while [ $n -le 5 ]
do
a=`echo $num | cut -c $n`
echo $a
n=`expr $n + 2`
done
```

Output :-

```
Enter a 5 digit number
12345
1
3
5
```

14) Write a shell script to check given year is leap year or not.

Program :-

```
clear
echo "enter  any year"
read num
if [ `expr $num % 4` -eq 0 ]
then
if [ `expr $num % 100` -eq 0 -a `expr $num % 400` -ne 0 ]
then
echo "Not a leap year"
else
echo "Leap year "
```

```
fi
else
echo "Not a leap year"
fi
```

Output :-

```
enter  any year
2016
Leap year

enter  any year
2100
Not a leap year
```

15) Write a shell script to find the value of one number raised to the power of another.

Program :-

```
echo "Input number"
read no
echo "Input power"
read power
counter=0
ans=1
while [ $power -ne $counter ]
do
      ans=`echo $ans \* $no | bc`
      counter=`echo $counter + 1 | bc`
done
echo "$no power of $power is $ans"
```

Output :-

```
Input number
5
Input power
3
5 power of 3 is 125
```

16) Write a shell script to display the following details in a pay list Pay slip details, House rent allowance, Dearness allowance, Provident fund. HRA is to be calculated at the rate of 20% of basic, DA at the rate of 40% of basic and PF at the rate of 10% of basic.

Program :-

```
i="y"
while [ $i = "y" ]
do
echo "Please enter your Basic:"
read basic
echo "PAY SLIP DETAILS"
echo "1. HOUSE RENT ALLOWANCE"
echo "2. DEARNESS ALLOWANCE"
echo "3. PROVIDENT FUND"
echo "your choice:"
read ch
case $ch in
1) hra=`expr $basic \* 20 / 100`
echo Your HOUSE RENT ALLOWANCE is Rs. $hra;;
2) da=`expr $basic \* 40 / 100`
echo Your DEARNESS ALLOWANCE is Rs. $da;;
3) pf=`expr $basic \* 10 / 100`
echo Your PPOVIDENT FUND is Rs. $pf;;
*) echo "Not a valid choice";;
esac
echo "Do u want to continue ?"
read i
if [ $i != "y" ]
then
   exit
fi
done
```

Output :-

```
Please enter your Basic:
1000
PAY SLIP DETAILS
1. HOUSE RENT ALLOWANCE
2. DEARNESS ALLOWANCE
3. PROVIDENT FUND
your choice:
1
Your HOUSE RENT ALLOWANCE is Rs. 200
Do u want to continue ?
```

17) Write a shell script to find sum of digits of a number.

Program :-

```
clear
echo "enter the number"
read n
sum=0
a=$n
while(($n >0))
do
x=`expr $n % 10`
sum=`expr $sum + $x`
n=`expr $n / 10`
done
echo "the sum of $a is $sum"
```

Output :-

```
enter the number
3355
the sum of 3355 is 16
```

18) Write a shell script that greets the user by saying Good Morning, Good Afternoon, and Good Evening according to the system time.

Program :-

```
clear
  #hours=`date|cut -c 12-13`
  hours=`date +%H`
  if [ $hours -le 12 ]
  then
      echo "Good Morning"
  elif [ $hours -le 16 ]
  then
      echo "Good Afternoon"
  elif [ $hours -le 20 ]
  then
  echo "Good Evening"
  else
  echo "Good Night"
   fi
```

Output :-

```
Good Afternoon
```

19) Write a shell script to generate mark sheet of a student. Take 3 subjects, calculate and display total marks, percentage and Class obtained by the student.

Program :-

```
Clear
echo "Enter the five subject marks for the student"
read s1 s2 s3
sum1=`expr $s1 + $s2 + $s3`
echo "Sum of 5 subjects are: " $sum1
per=`expr $sum1 / 3`
echo " Percentage: " $per
if [ $per -ge 60 ]
then
echo "You get Distinction"
elif [ $per -ge 50 ]
then
echo â€œYou get First classâ€
elif [ $per -ge  40 ]
then
echo "You get Second class"
else
    echo "You get Fail"
fi
```

Output :-

```
Enter the five subject marks for the student
78 88 92
Sum of 5 subjects are:  258
 Percentage:  86
You get Distinction
```

20) Write a shell script that finds total no. of users and finds out how many of them are currently logged in.

Program :-

```
cat /etc/passwd>user.txt
set `wc -l user.txt`
log=`who|wc -l`
echo "There are $1 users in network "
echo "There are $log user loged in right now"
```

Output :-

```
There are 49 users in network
There are 2  user loged in right now
```