1 - Introduction

(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.

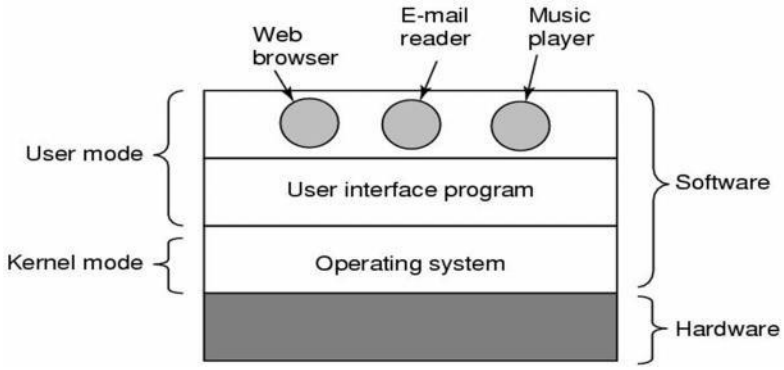


Figure 1-1. A computer system consists of hardware, system programs, and application programs.

- 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.

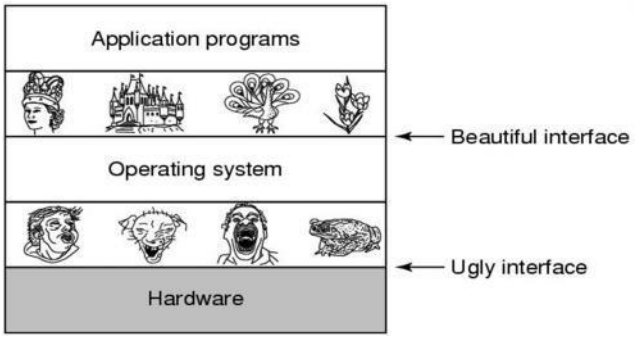


Figure 1-2. Operating Systems turn ugly hardware into beautiful abstractions.

- 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
    - Many of these are found in industrial process control, avionics, and military and similar application areas.
    - These systems must provide absolute guarantees that a certain action will occur be a certain time.
  - ✓ Soft real time system
    - Missing an occasional deadline, while not desirable is acceptable and does not cause any permanent damage.
    - 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

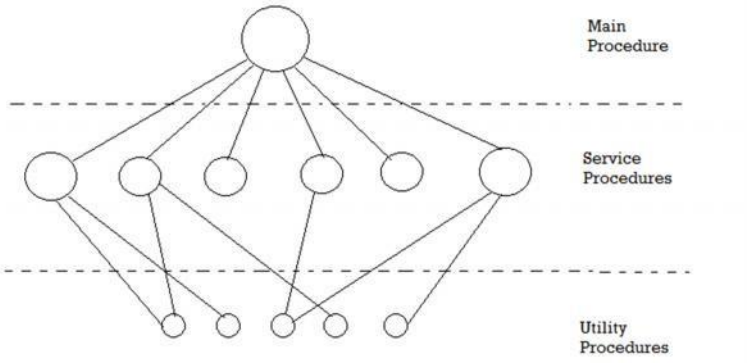


Figure 1-3. A simple structuring model for a monolithic system.

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.

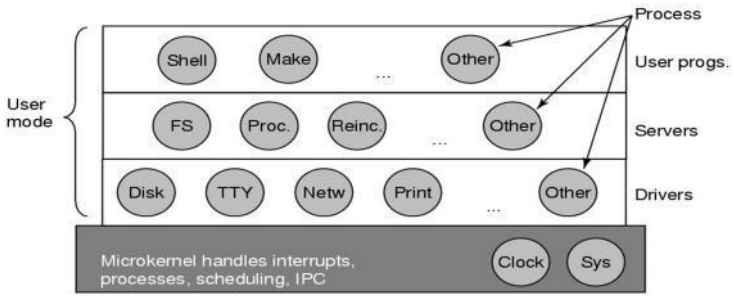


Figure 1-5. Structure of MINIX 3 system.

- 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.

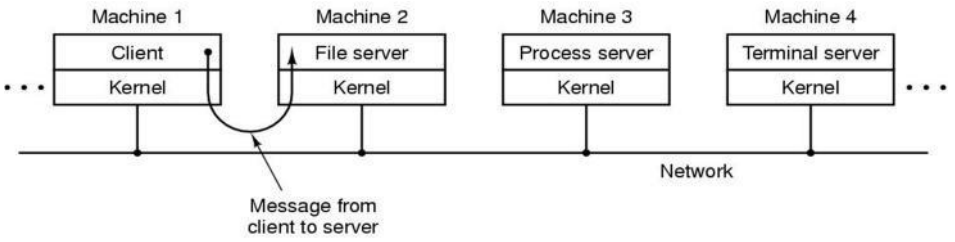


Figure 1-6. The client server model over a network.

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.

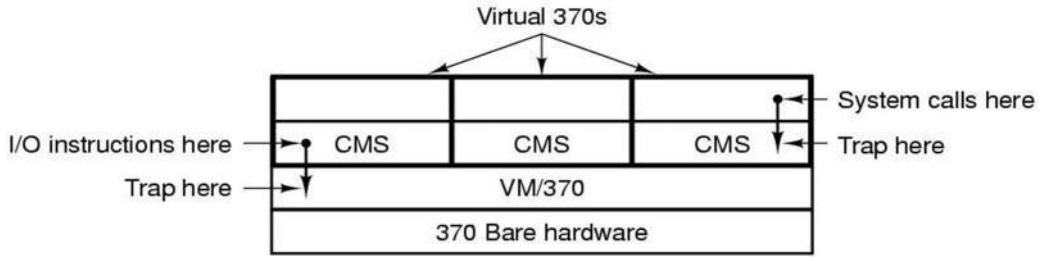


Figure 1-7. The structure of VM/370 with CMS.

- 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.**

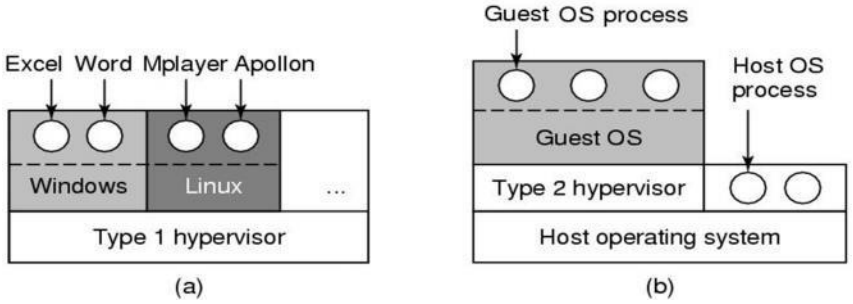


Figure 1-8. Type 1 & 2 Hypervisor.

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).

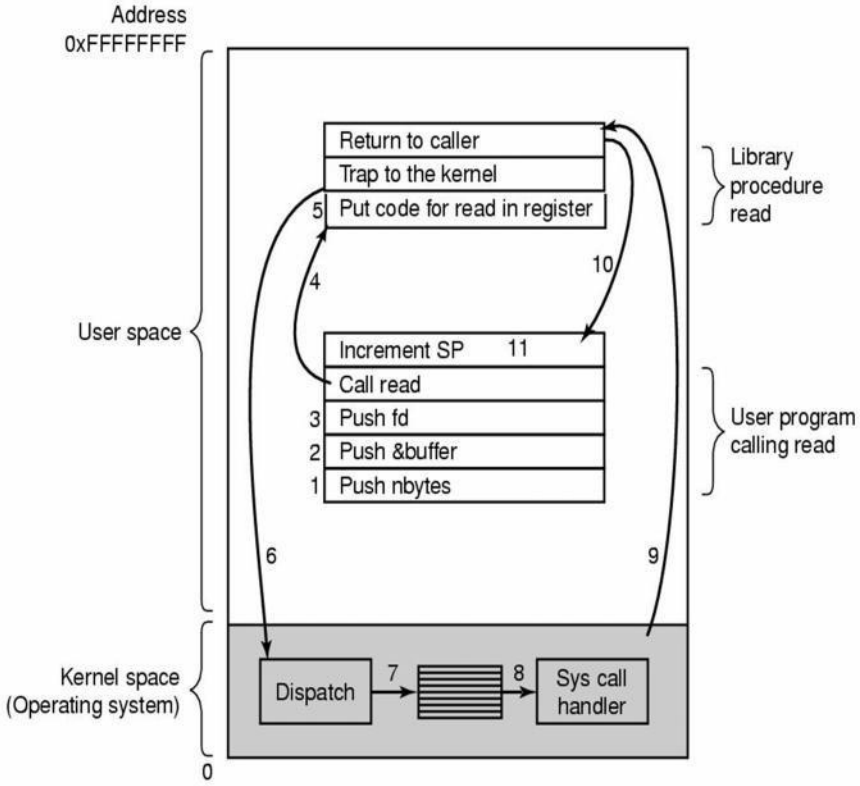


Figure 1-9. The 11 steps in making the system call read(fd, buffer, nbytes).

- 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).

2 – Processes & Threads

(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.

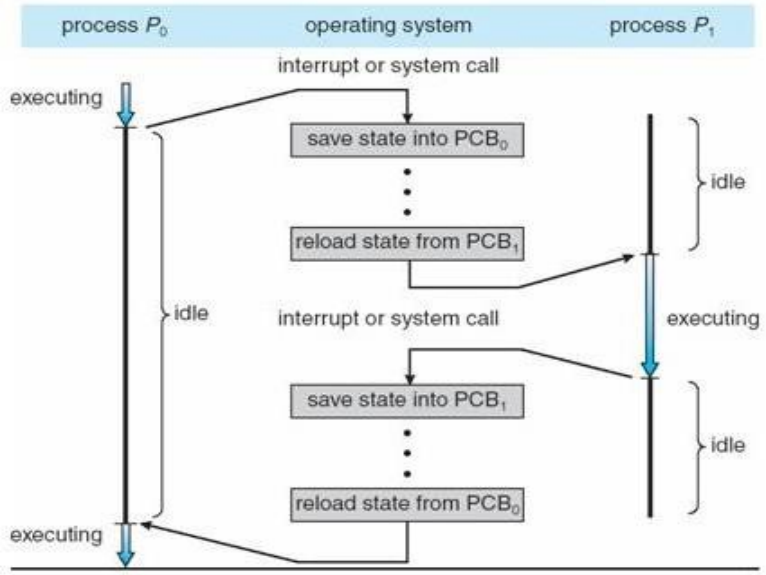


Figure 2-1. Context Switching

(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.

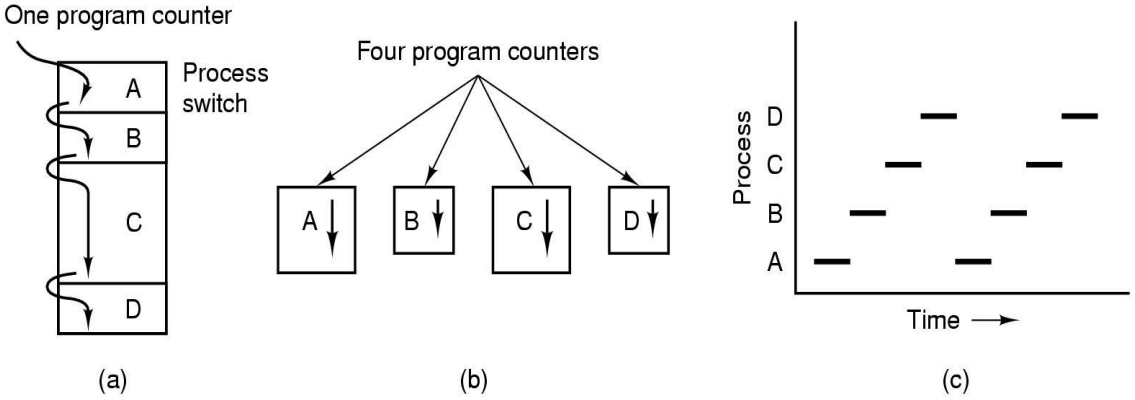


Figure 2-2. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

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).

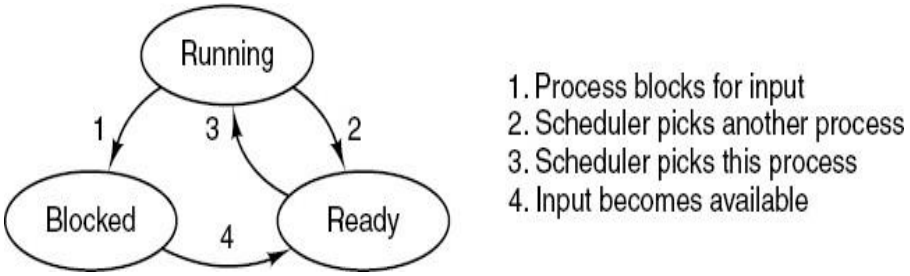


Figure 2-3. Process state transition diagram

- 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.

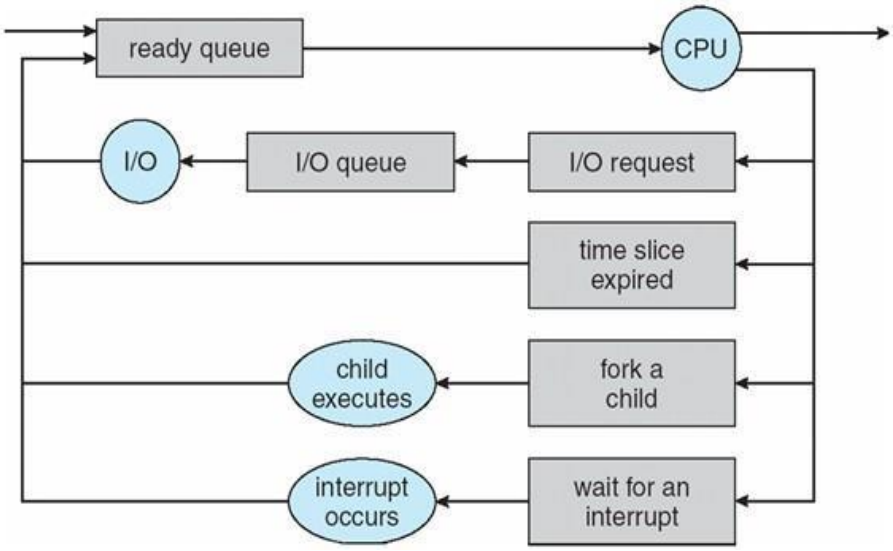


Figure 2-4. Queuing diagram representation of Process Scheduling

- 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.

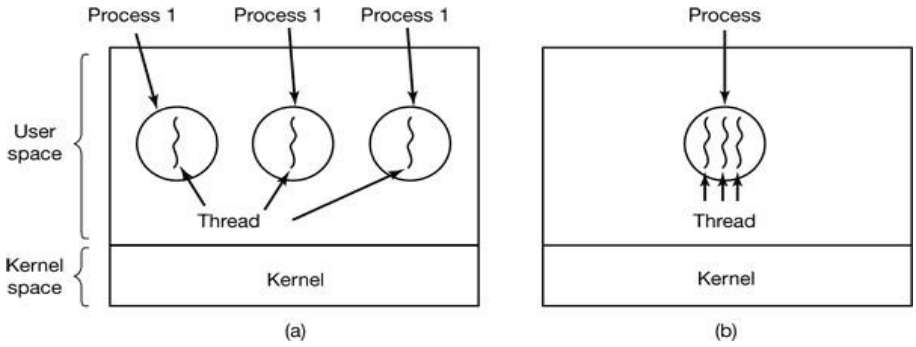


Figure 2-7. (a) Three processes each with one thread. (b) One process with three threads.

- 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.

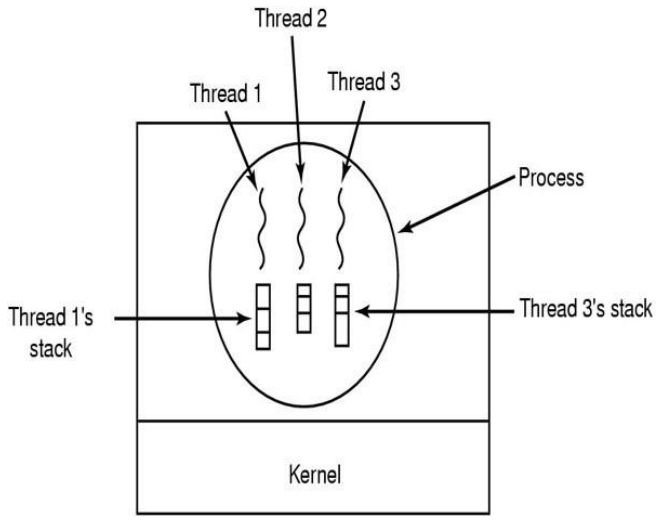


Figure 2-8. Each thread has its own stack.

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.

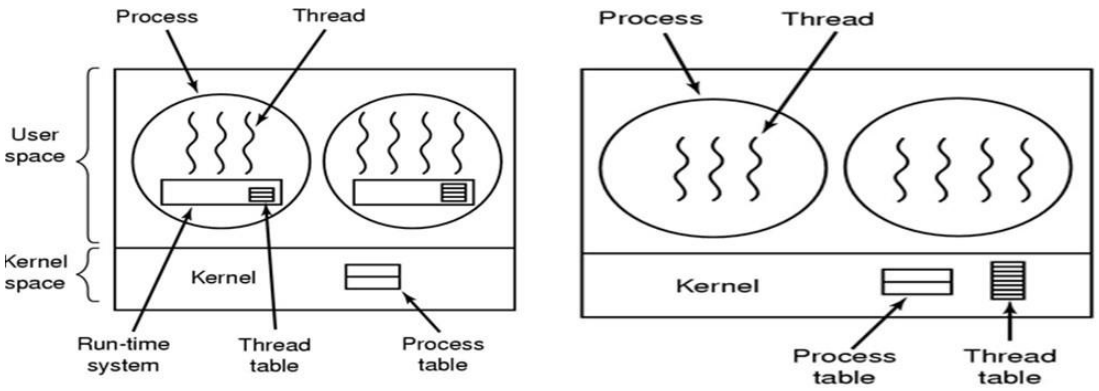


Figure 2-9. (a) A user-level threads package. (b) A threads package managed by the kernel.

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.

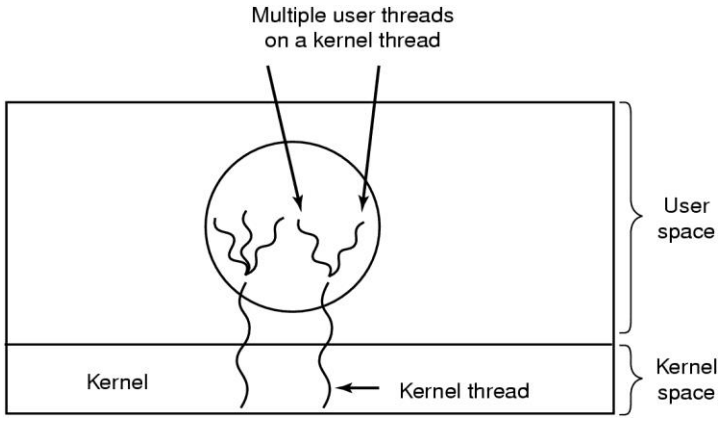


Figure 2-10. Multiplexing user-level threads onto kernel-level threads.

(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

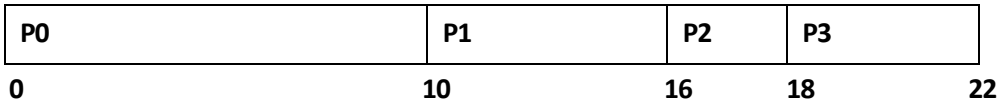
- Unlike processes threads are not independent of one another, threads within the same process share an address space.
- Unlike processes all threads can access every address in the task.
- Unlike processes threads are design to assist one other. Note that processes might or might not assist one another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

2 – Process Scheduling

(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 :**



• 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 **<u>Shortest Job First (SJF):</u>**

- **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 :**

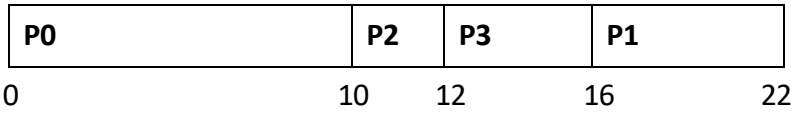
This strategy can be implemented by using sorted FIFO queue. All processes in a queue are sorted in ascending order based on their required CPU bursts. When CPU becomes free, a process from the first position in a queue is selected to run.

- **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.

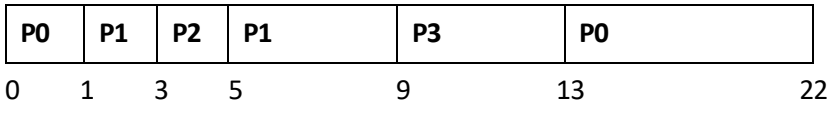
III **<u>Shortest Remaining Time Next (SRTN):</u>**

- **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 **<u>Non Preemptive Priority Scheduling:</u>**

- **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.

2 – Process Scheduling

- **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.

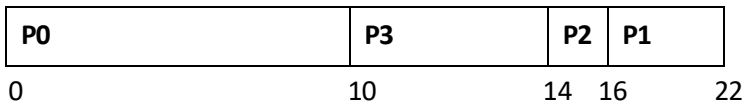
- **Example :**

  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:

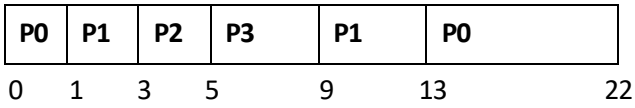
- **Selection criteria :**
  The process, that has highest priority, is served first.
- **Decision Mode:**
  Preemptive: When a new process arrives, its priority is compared with current process priority. If the new job has higher priority than the current, the current process is suspended and new job 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

2 – Process Scheduling

- **Gantt chart:**



- 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.

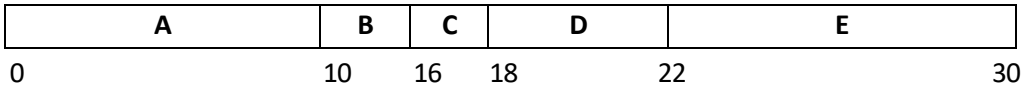
2 – Process Scheduling

(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:

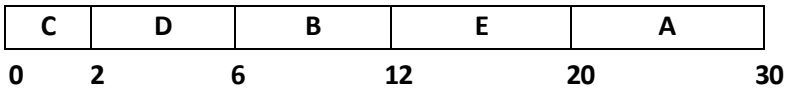


Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (TAT-ΔT)
A	0	10	10	10	0
B	0	6	16	16	10
C	0	2	18	18	16
D	0	4	22	22	18
E	0	8	30	30	22

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

2 – Process Scheduling

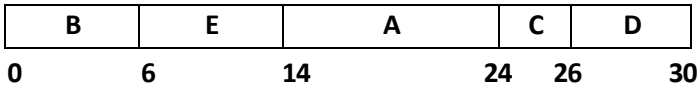
Shortest Job First:



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:



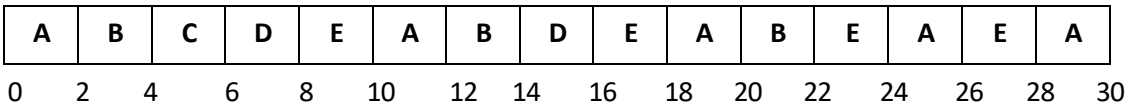
Process	Arrival time (T0)	Completion Time (ΔT)	Finish Time (T1)	Turnaround Time (TAT=T1-T0)	Waiting Time (TAT-ΔT)
A	0	10	24	24	14
B	0	6	6	6	0
C	0	2	26	26	24
D	0	4	30	30	26
E	0	8	14	14	6

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.



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?

2 – Process Scheduling

(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.

3 – IPC

(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

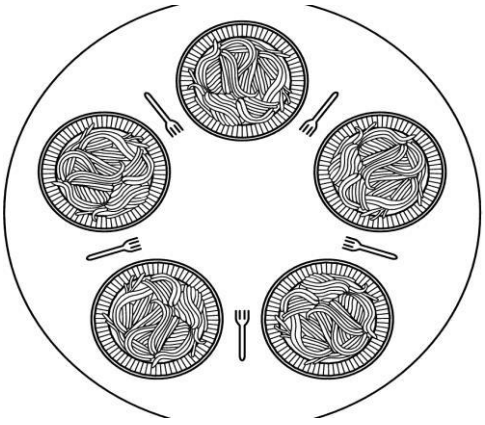
Solution of Producer Consumer Problem with Mutex

- 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.

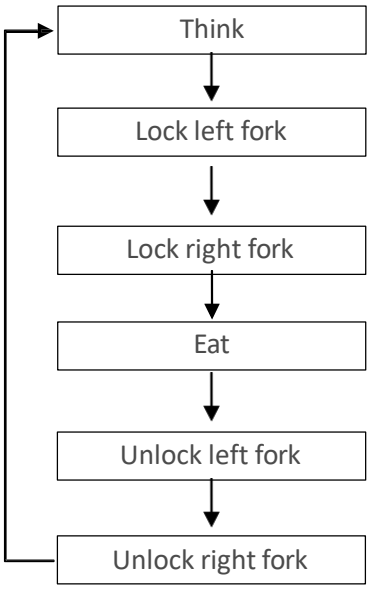
- Dining philosopher's problem is a classic multiprocess synchronization problem.
- Problem is summarized as 5 philosophers sitting at a round table doing 2 things, eating and thinking.
- While eating they are not thinking and while thinking they are not eating.
- Each philosopher has plates that are 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 */
      }
 }
```

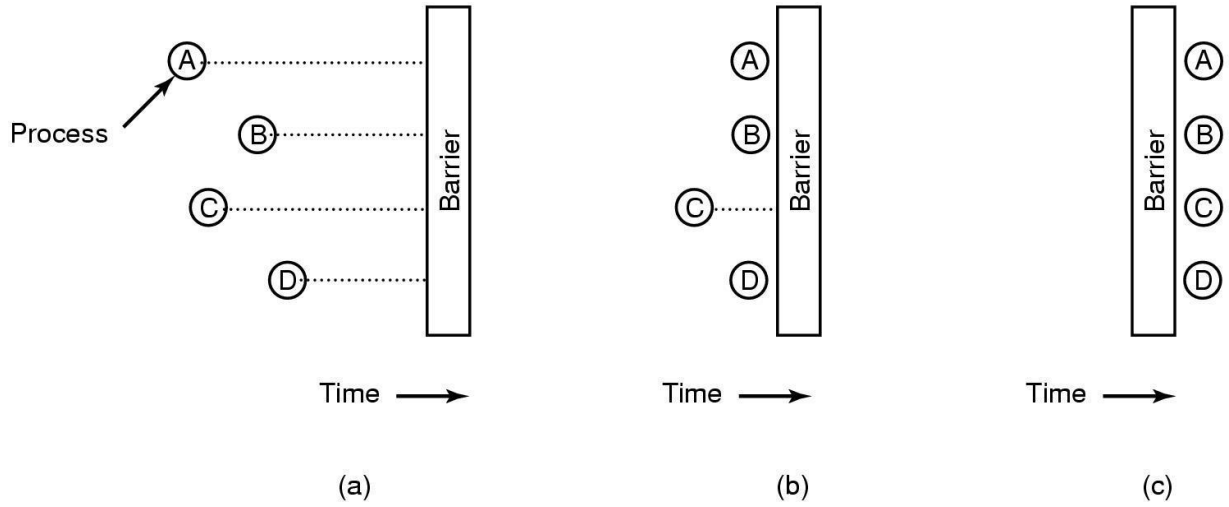
Figure 3-7. The producer-consumer problem with *N* messages.

- 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.

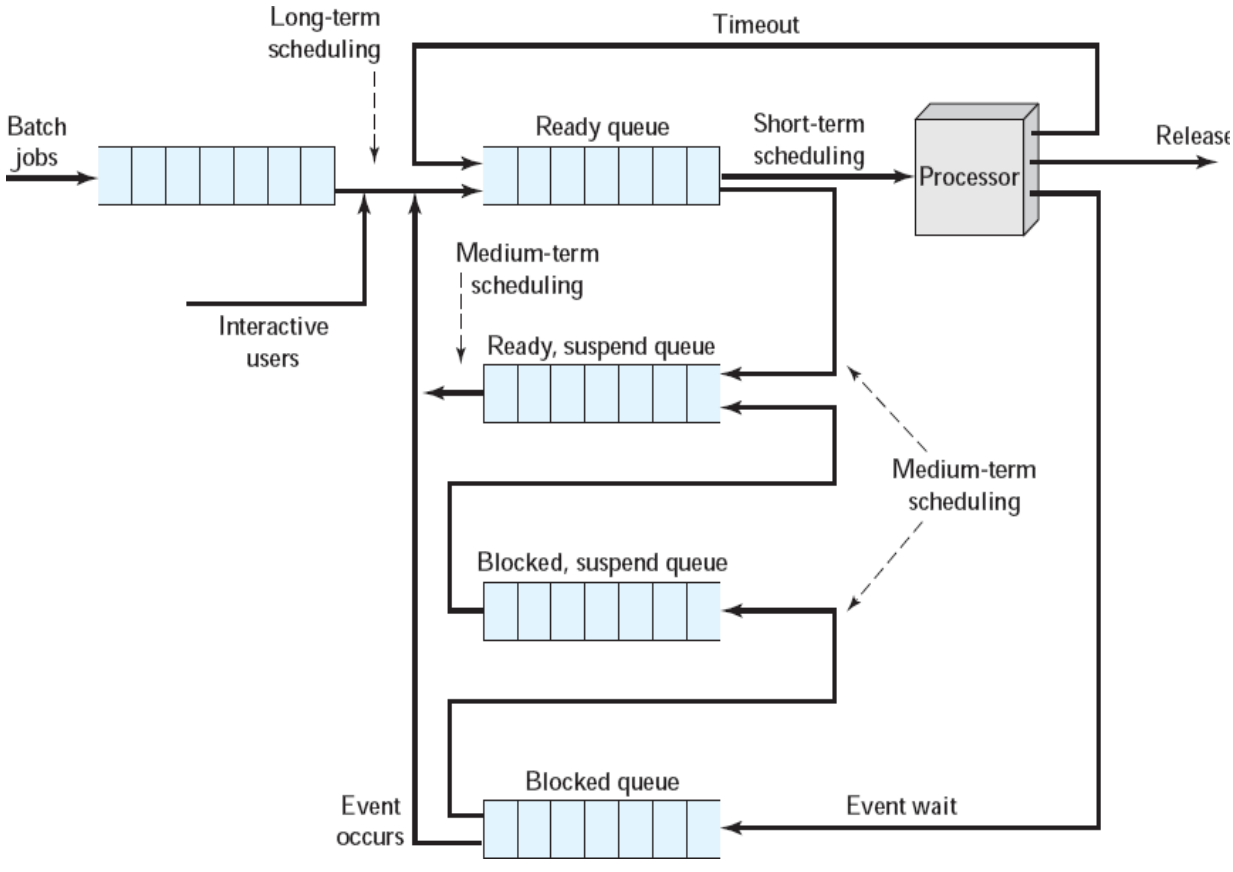


Figure 3-6 A Queuing diagram representation of process scheduler:

- 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.