Introduction to Operating Systems

1. **Basic Terminology & Definitions**
   System, kernel, shell, program, hardware, software, system software, user applications, elements of computing system, operating systems, main memory, secondary memory, system call, multiprogramming, multiprocessing, timesharing, uniprocessor system, multiprocessor systems, file, process, interrupt, batch processing systems

2. **Operating System Objectives and functions**
3. **Operating system structure**
   Monolithic and layered structure, microkernel architecture, UNIX system architecture, block diagram of system kernel, OS design hierarchy, General structure of OS control tables, characteristics of modern OS, SMP

4. **Summary of System calls**
   System calls for process management, device management, file management, time management, signaling, protection

1. *Basic terminology and definitions*

Q. What is System, kernel, shell with reference to Operating System

Ans:
System: A system is a collection of various components interacting with each other to achieve a specific goal. In a Computer System, Hardware and Software work together for solving the task assigned by user.

Kernel :- Kernel is a fundamental part of operating system that is loaded into primary memory on starting up the computer. This core part of Operating System has control over operations of the computer. This manages the functioning of input & output requests from application software.

Shell: Shell command interpreter of the Operating system (OS). It interacts with OS by invoking System Calls.

Q. What is the role of hardware & software in a computing system?

Ans: Hardware includes the CPU, Memory and other communication Devices while the software includes the programs for using these devices.

Q. Differentiate between system software & application software

Ans:
System Software: System software can be defined as a program that runs computer hardware and software.

- Examples: Operating System, BIOS, firmware
- Special Types of System Software: Translators, Compilers, DBMS Programs, Other Diagnostic tools
- Operates at the lowest computer level
- Manages & Operates Hardware
- Interface between Application Software and Hardware
- Synchronizes & controls data flow between memory, secondary storage device, Display, printers etc.

Application software: Application softwares are defined as the softwares those run on system software to serve end users.

- Examples: Animation, Graphics, Microsoft Office, etc.
- Developed to perform single or multiple tasks

Q. What are the elements of computing system?

Ans:- A Computing System Consists of -

Processor: For execution of instructions.
Memory Devices: For staring data and instructions (programs)
Input /Output /Communication Devices: Used for communicating with the real world. Receiving the date from the user, presenting data to the user or communicating with other devices.

Q. What is an operating systems

Ans: Operating System is collection of software that controls hardware for scheduling the execution of other programs, mange storage; input, output & Communication Devices and the CPU.

Q. **Compare main memory and secondary memory.**

Ans: Main Memory is the CMOS/ RAM in the Computer System. The program must be loaded in main memory to get executed along with it's data. Typical size of primary memory these days is 4 GB.

Secondary memory is typically hard disks in the computer. The OS partially resides in secondary memory too. Typical sizes of contemporary hard disk is 250GB

Q. What is system call?

Ans: Means of invoking Operating System functions is called System Calls. They are used for Process Management, File Management, Time Management, Memory and I/O Management.

Q. Differentiate between multiprogramming Timesharing , multiprocessing
Ans:
Multiprogramming & Timesharing:
The ability of OS to run multiple programs at a time is called multiprogramming. The time-sharing is the concept that helps the OS to achieve multiprogramming.

When CPU time is shared by multiple processes on the same processor it is called time sharing. With the help of timesharing, multiple processes are executed on same processor giving pseudo impression to the user of executing many processes simultaneously.

Multiprocessing :
The ability of OS to run multiple programs on multiple processors concurrently is called multiprocessing.

Multiprogramming can be done on one or more processors but multiprocessing can be done only on multiple processors.

Q. Define Uniprocessor system and multiprocessor systems giving example.
Ans:

Uniprocessor systems:- When only one processor exists for execution of processes, the computer system is termed as Uniprocessor System. The speed of operation is slower as compared to multiprocessing as each processes gets a time slice and time is also spent in context switching.
Multiprocessor systems:- In Multiprocessor environment many processors executes the processes in synchronization with each other, thus reducing the time for execution.

Q. Define interrupt.

Ans: Interrupt is a request by any device for service. The device needing resources send signal to the processor which intern directs operating system to perform the requested task.
The interrupts could be hardware or software interrupts.

Q. What are batch processing systems?

Ans: The batch operating system maintains files that contains commands to be executed in a sequence. Typically these commands do not need human intervention to complete their execution.
The batch processing is classical OS characteristic, but in modern days too, the periodic executions are scheduled as batch commands.

Q. what are the operating system objectives?

Ans: Operating systems has got three main objectives:-

1. Convenience:- The OS must make the computer convenient to use.
2. Efficiency:- The OS must make efficient use of computing resources.
3. Ability to evolve:- The OS must be designed and implemented in such a way as to provide some means to introduce new functionalities without interfering with OS services.

Q. What are the functions carried out by an Operating Systems?

Ans: the operating systems provides following main functionalities-

1. The OS functions as User/Computer interface
2. The OS functions as resource manager

Q. What are the issues handled by OS as resource manager

Ans: The operating system is responsible for managing all the computing resources . These resources combinely store, process and move the data.

1. Managing Processor:- The OS functions are executed by processor and also the user programs. To manage the balance, OS gives control to processor and then depends on OS to execute instructions which give control back to OS. This way, processor keeps on executing interleaved sequence of user programs and OS programs. The OS decides how much CPU time should be awarded to execution of user program. If the system has more than one processor, then the OS takes decision for all processors. There are more than one policies those decide which processor should run OS in case of multiprocessor system.

2. Managing Memory: The main part of OS that resides in main memory is kernel; rest OS resides in secondary memory. The remaining part of Main memory contains user data, user programs and the OS programs that are invoked by kernel to process user requests. The allocation of main memory to various user and system programs is managed by operating system functions for memory management and memory management hardware in the processor.

3. Managing I/O:- The OS decides which program to execute next, and if the program needs to use I/O then its OS that decides when the program gets that access. Also, if required, the OS can take the I/O from the program before it completes execution.

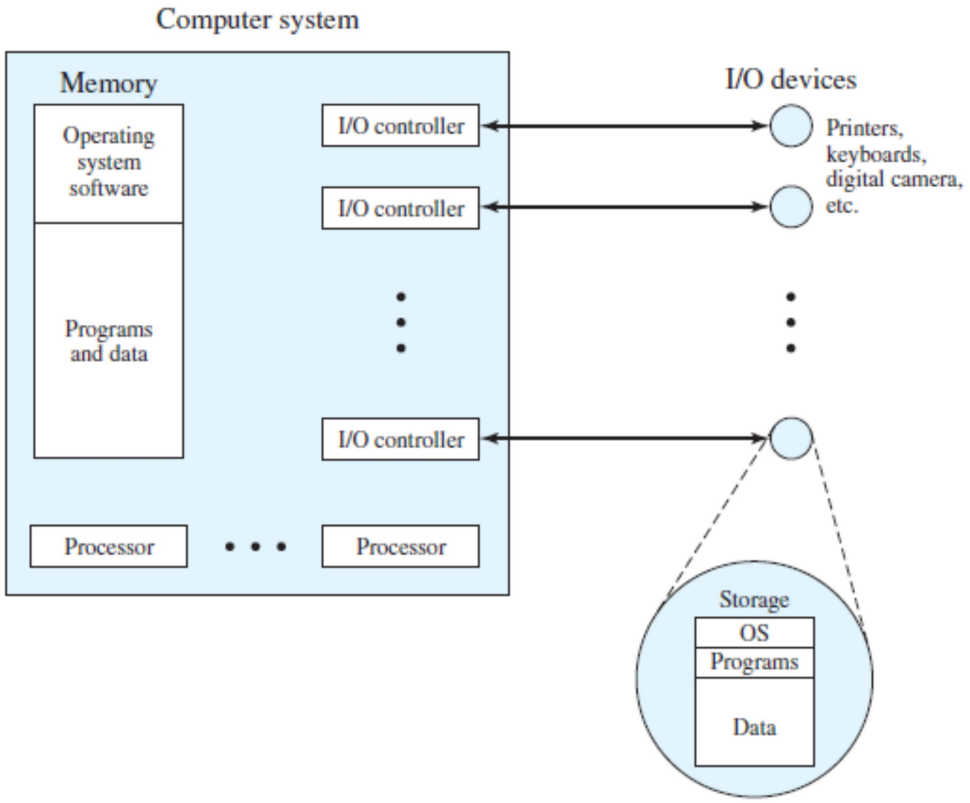


Figure 2.2 The Operating System as Resource Manager

Image courtesy: "Operating systems", William Stallings, Pearson education, 6th edition.

The operating system manages all these resources by using time and space multiplexing concepts.

3. *Operating system structure*

Q. What are different OS structures?
Ans: The OS has had following structure since inception it's concept

1. Monolithic: The OS was developed as a big program which contained functions written for OS functionalities. All the functions could call each other and modification in any module would require recompiling of whole OS and reinstallation of the same. There was no concept of data hiding , encapsulation or protecting data from any function. The modification in one module had chances of introducing errors in other modules.

   Drawbacks:

   1. The entire OS used to reside in main memory and thereby it left very less space for user programs.
   2. In this structure all the functions are executed in kernel mode.

2. Layered: The operating system functions were divided into modules which were arranged in form of layers. In this structure, only the required layer was modified and other were not affected with the same. The interfaces between any two layers were clearly defined and minimal information could flow from one module to another. Modification in one layer required compilation of the only that module.

   Drawbacks:

   1. The information had to pass through all the layers sequentially even if all the layers wouldn't need access to the same.
   2. Also, The entire OS used to reside in main memory and thereby it left very less space for user programs.
   3. In this structure most the functions are executed in kernel mode.

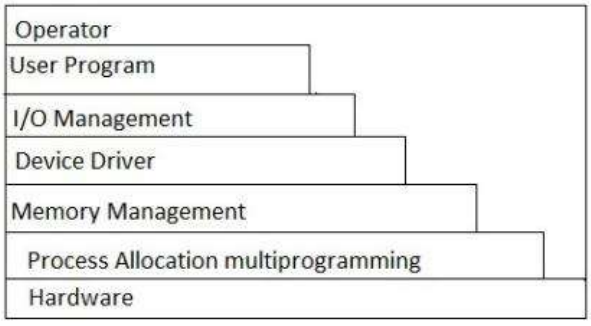


fig:- layered Architecture

Image courtesy: http://www.sciencehq.com/wp-content/uploads/layered-architecture.jpg

3. Microkernel architecture: This is modern operating system architectures.

   Here the entire OS need not reside in main memory and thereby it leaves more space for user programs. The OS partially resides in main memory and secondary memory. The part of OS that always resides in main memory is called kernel and rest of the part residing in secondary memory is invoked by special managers in kernel as and when needed.

   In this structure, Most of the functions run in user mode and only high privileged operations are executed in kernel mode.

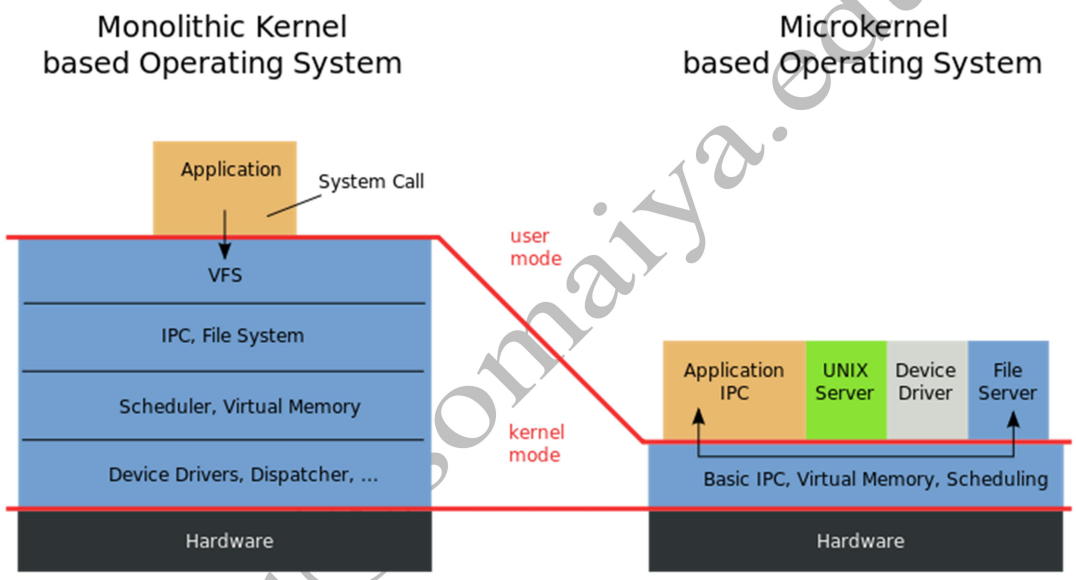


Image courtesy: https://en.wikipedia.org/wiki/Microkernel

Q. Explain the Unix system architecture

Ans: The Operating system works as an interface between hardware and user by providing common services to programs and hiding complexities from the user. The OS provides services as kernel and the utilities and user programs have clear separation from the OS. The programs like shell and various editors interact with OS by invoking different system calls.

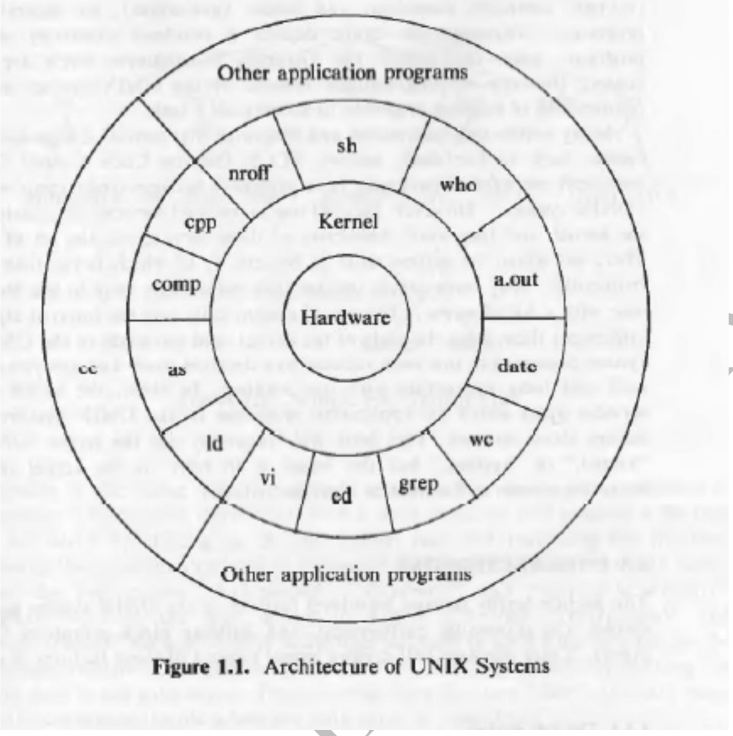


Figure 1.1. Architecture of UNIX Systems

Image courtesy: "The design of Unix operating system" Maurice j Bach , PHI

Q. Draw and explain Block diagram of OS system Kernel

Ans: The operating system is typically divided into three main layers: hardware level, kernel level and user level.

- Hardware level: The hardware level handles interrupts and low level communication for the machine. The interrupts are processed by special kernel functions which are invoked in the context of the process currently being executed by the processor.

- Kernel level: the kernel level contains the hardware control device drivers those manage the different hardware devices. The devices also include I/O devices connected to system. Generally these devices read and write data into systems which are saved in terms of files. The device drivers could be serial or block drivers. The block drivers have access to buffer cache module that caches the frequently/recently used information.

The processes those access memory/files are managed by process control subsystem. This subsystem manage communication amongst processes, scheduling them on the processors and manage memory allocated to the processes.

All the high privileged processes are executed in kernel mode.

- User level: User can invoke the system functions through system calls. The system has predefined system calls for file, time, device, memory, communication, security, i.e. all kinds of managements. System calls access file system and process control subsystem on user's behalf.

  The users can also develop programs using system defined libraries which in turn can invoke system calls to execute what user program intends to. In short, one has to execute system calls directly or indirectly to use the computing machine .

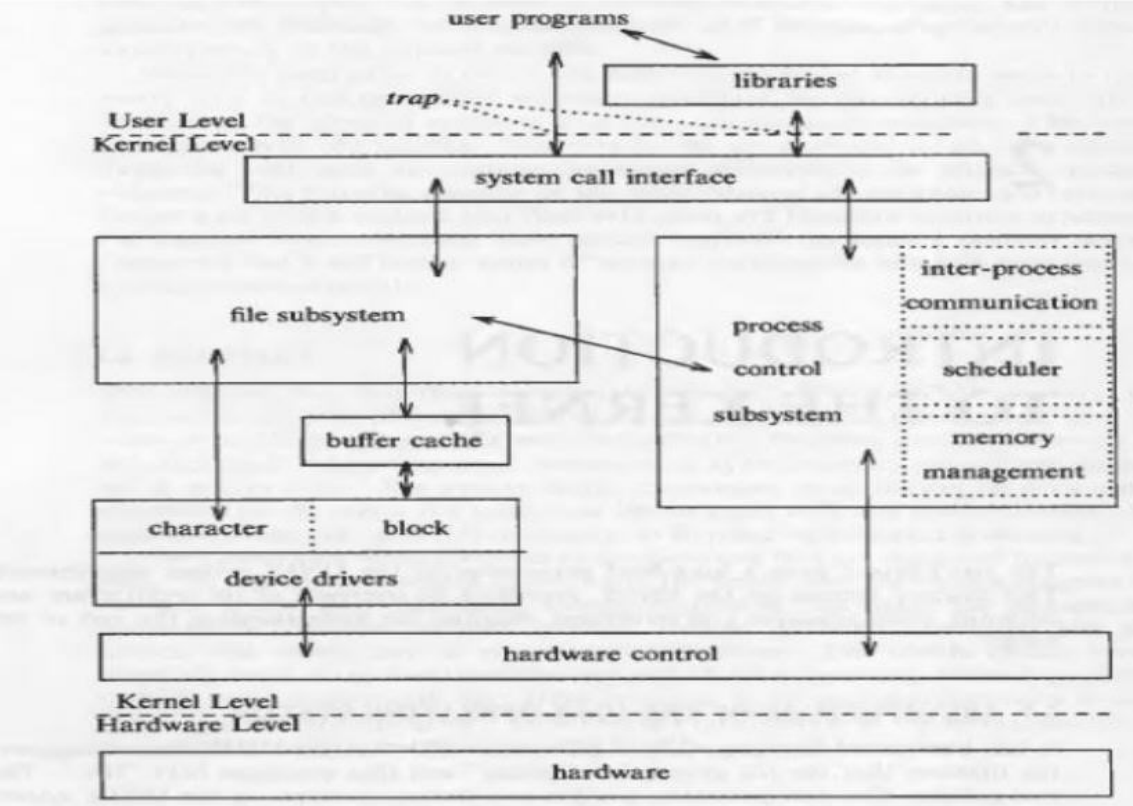


Figure 2.1. Block Diagram of the System Kernel

Image courtesy: "The design of Modern Operating System", Maurice Bach, PHI, 2nd edition.

Q. Explain the OS design hierarchy

Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printers, displays, and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write
7	Virtual memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive processes, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalar and array data	Load, store, add, subtract, branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

Gray shaded area represents hardware.

Image courtesy: “Operating systems”, William Stallings, Pearson education, 6th edition

The operating system is interface between hardware and user. To do so, it has been divided into modules those run OS resource management programs and also serve the user requests. The above figure explains the OS’s division into different layers.

Q. What are the different control tables that OS maintains?

Ans: The Operating systems maintains four main tables such as,

- Memory table
- File table
- I/O table
- Process table

Q. General structure of OS control tables

Ans:

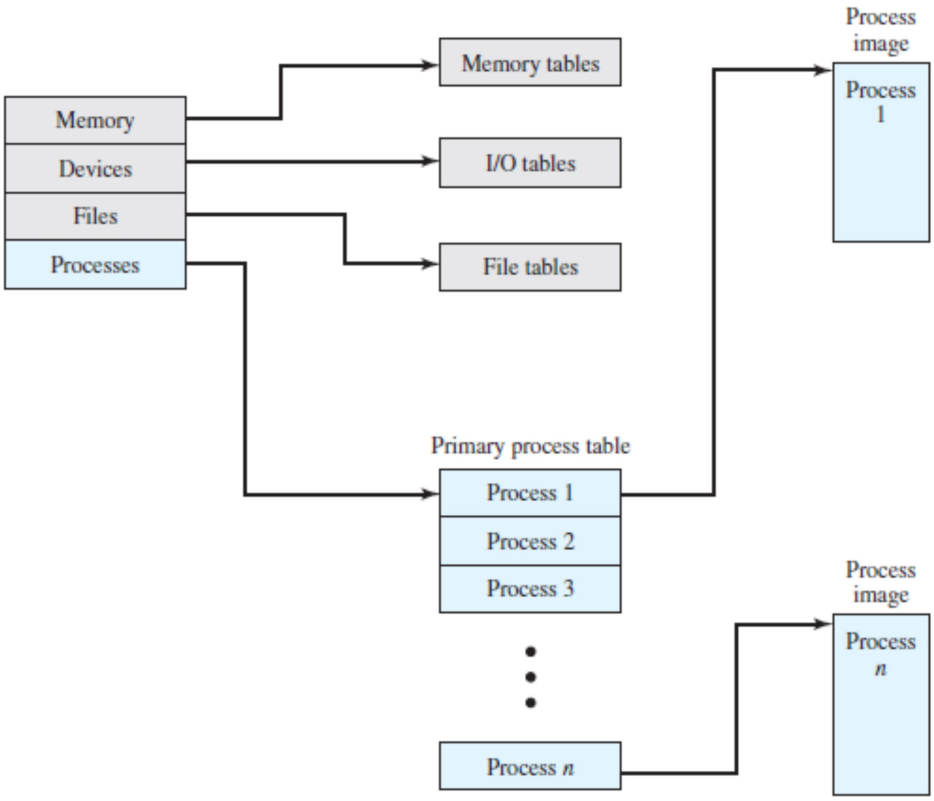


Figure 3.11 General Structure of Operating System Control Tables

Image courtesy: "Operating systems", William Stallings, Pearson education, 6th edition

Q. What are the characteristics of modern OS?

Ans:-

The modern operating system is identified with following four characteristics.

1. Microkernel architecture: Microkernel architecture has come up with minimal amount of modules that constitute microkernel, thereby has left more space for the user programs. The rest of the modules are loaded into the main memory as and when needed.
2. Symmetric multiprocessing: The system contains multiple processors. These processors in system can execute any process, there no processor exclusively dedicated to OS. This way, the available process load can be distributed well amongst all the processors thereby achieving efficiency, availability and provision for incremental growth. All processors share the same main memory and I/O subsystems which are connected very well with each other.
3. Distributed OS: The distrubted OS gives the illusion of single main memory, secondary memory space and file system, too. .
4. Multithreading:- The modern OS supports multithreading thereby striking the perfect balance between processes and threads. The multiple

processes can run multiple threads from multiple processes in parallel. Threads of one process too can run concurrently.

5. Object oriented design : Object oriented design helps in building the modular extensions to micro kernel. The object oriented feature helps in system integrity and modularity .

4. System calls

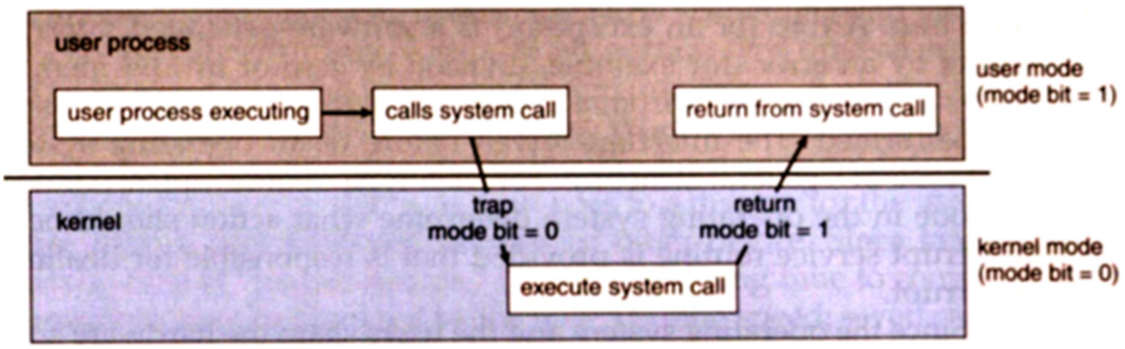


Image courtesy:http://faculty.salina.k-state.edu/tim/ossg/Introduction/sys_calls.html

Q. List System calls for process management

Ans:- Following are the various system calls those are used for process management.

- fork ()- to create a new process
- exec() – to execute a process
- wait () - to make the process to wait
- exit () - to terminate the process
- getpid()-To find the unique process id
- getppid () - To find the parent process id
- nice () - To change currently running process's priority

Q. List system calls for file management

- Create()- to create a new file
- Open()- to open an existing file
- Read()- to read file contents
- Write()- to write into a file
- Close()- to close an open file
- Seek()- to change the file offset position while reading or writing
- *Get()* and *set() – retrieve and set* file attributes.

Q. List system calls for device management,

- Open()- To open an I/O connection.
- Close()- To close the connection with associated device.
- Read() – to read data from device associated with opened device id.
- Write()- to write into the device associated with opened device id.

Process

A. Basic Terminology & Definitions
Task, program, process, thread, job, process state, process state transitions, process scheduling, process context, process mode, event, ready queue, blocked queue, suspended queue, resource, degree of multiprogramming, process priority, I/O scheduling, preemption, preemptive, nonpreemptive, multithreading

B. State Transition Models And Process Life Cycles
User mode, kernel mode, Two state model, five state model, seven state model, eleven state model, Phases in and reasons of Process creation, suspension, termination

C. Process Management
PCB, Process image, mode switch Vs context switch

D. Threads
Thread management, ULT, KLT, process switch Vs thread switch,

E. System Calls For Process Management
Fork, kill, signal, exec, wait, exit, getpid, getppid, nice, brk

F. Process scheduling
Types of schedulers: Mid-term-short term-long term schedulers, response time, waiting time, turnaround time, scheduling policies, uniprocessor scheduling, multiprocessor scheduling, real time scheduling, UNIX scheduling, LINUX scheduling, Comparative assessment of scheduling

G. Processes in Unix, Solaris and windows 2000 thread and SMP Management.

H. Multiple Choice Questions

Basic Terminology & Definitions

Q. 1 Define Task, program, process, thread and job with an example of each.
Ans:

- **Task:** Task is a unit of assigned work. It can also be defined as the unit of programming controlled by OS. Depending on the OS design the task may involve one or more processes.
  Example: Bake a cake

- **Program**: A program is defined as sequence of instruction written to accomplish a task. A program may comprise of one or more processes depending on the statement being executed. Generally referred as a passive entity that does not perform any action.
  Example: A particular recipe given in book.

- **Process**: This is an instance of program in execution. The static statements in the program when executed, take the process form. In contrast to the program, a process is an active entity which needs a set of resources to perform its function. The Linux kernel internally represents processes as tasks. A process elements are: a program, data and process state.
  Example: actually following the steps given the book.

- **Thread**: A thread is a lightweight process. It is also defined as the smallest processing unit that is scheduled by an operating system.
  A thread must be part of a process as it shares the process environment viz, code, data and resources with other threads. Threading concept has increased the computing efficiency to significant extent.

- **Job**: A job is unit of work submitted by user to the system. A job may be interactive or a batch job which may in turn consist of one or more processes.

Q.2 Define process states, process context, process mode, event, degree of multiprogramming, process priority, preemption, preemptive, nonpreemptive, Multithreading.

Ans:

- **Process State:** A process state is a process' internal data maintained by OS for the purpose of supervision and control of the process. It is also called as executional context of the process.

- **Process Context:** whenever a running process is taken away from processor, some of the process state's information needs to be retained. This information called as process context helps the process to resume from the point where it was stopped last time.

The context of a process includes its address space, stack space, virtual address space, register set image i.e. Program Counter, Stack Pointer, Program Status Word, Instruction Register and other general processor registers, accounting information, associated kernel data structures and current state of the process (waiting, ready, etc).

- **Process Mode:** the operational or the privilege mode of process execution is called process mode. A process executes in user mode or a kernel mode. Processor switches the process in between these modes depending on the code the process is running.

- **Event:** An activity that is happens or is expected to happen. Generally this a software message exchanged when the activity occurs. In operating systems, the events may cause the processes to change their state.
  e.g. mouse click, file lock reset, etc.

- **Degree Of Multiprogramming**: with multiprogramming, the CPU can run multiple programs simultaneously or concurrently. The degree of multiprogramming is the maximum number of allowed processes at a time in a system that does not let the CPU performance degrade than a certain threshold. The long term scheduler can limit this number so that all the admitted processes get fair CPU share and they all execute well.

- **Process priority:** In a multiprogramming system, the processes are assigned numerical privileged values indicating their relative importance and/or urgency and/or value.

- **Preemption:** Preemption is the ability of the system to take over a currently executing process by another one (possibly with high privileged one) with an intention to resume the preempted process later on.

- **Preemptive:** Preemptive entities are the one those can be taken over by another similar type of entities.

- **Non-preemptive:** Non-preemptive entities are the one those cannot be taken over by other entities.

- **Multithreading:** Multithreading is defined as the ability of an OS to support multiple, concurrent paths of execution within a single process.
  The thread process relationship is shown in following diagram

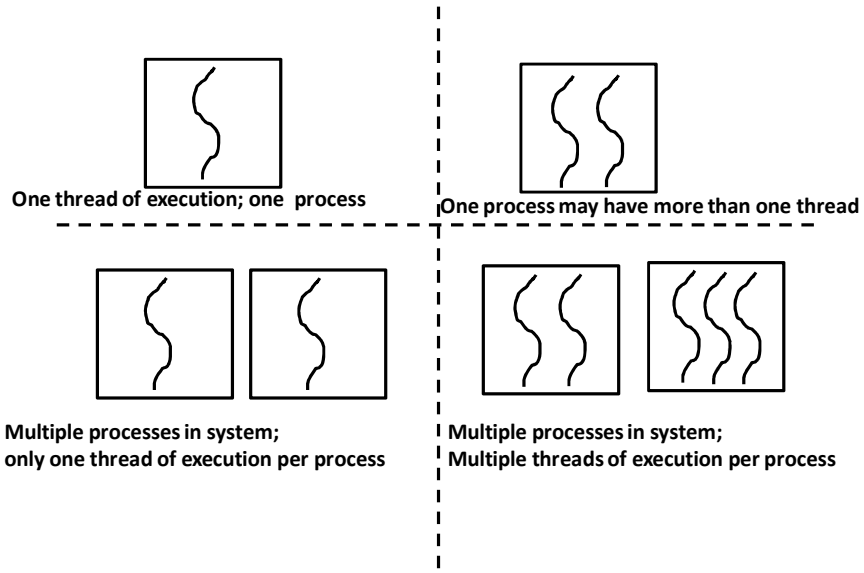


Figure: Threads and process association.

Q.3 Define Process State Transitions, Process Scheduler, I/O Scheduling, Processor Scheduling, Ready Queue, Blocked Queue, Suspended Queue.

- **Process State Transition Diagram:** A process transits among various states from its initiation to termination. The pictorial representation of this changes in states is called as process state transition diagram.

- **Process Scheduler:** Process scheduler or the Dispatcher is a software module that works for CPU(processor) scheduling and chooses the next process to run. Its main activities involve switching the process context, switching the execution mode to user. The dispatcher is required to work very fast to improve the efficiency of execution.

- **I/O Scheduling:** I/O scheduling is a process that is involved in making the decision as to which process's pending I/O request should be handled by an available I/O device.

- **Processor Scheduling:** Processor scheduling is a process that makes a decision of which process should get hold of processor next. This process needs the 'dispatcher' – a software module of short term scheduler to make this decision.

- **Ready Queue:** Ready Queue is a data structure that holds the processes which have acquired all required resources for the execution except CPU and are ready to run.

Any process submitted for execution undergoes many states during its lifetime. In certain states, only one process can have that state at any time of instance, while in some states multiple processes are allowed at a time. Such processes are stored in data structure 'Queue'.

- **Blocked Queue:** Blocked queue is a data structure that holds the processes which were executing and now waiting for some event to occur. Unless specified by priority, these processes are removed from this queue on FIFO basis as and when the corresponding events occur.

- **Suspended Queue:** Suspended queue is a data structure that holds the processes which were blocked and are swapped out to secondary memory to make room for processes which are getting blocked newly.

  Unless specified by priority, these processes are removed from this queue on FIFO basis as and when the corresponding events occur. After removal, processes may go to Blocked queue if memory is available but event is still awaited else, they may be shifted to Ready-Suspended state which indicates that event has occurred but memory isn't available.

Q.4 what are the I/O bound and CPU bound processes?
Ans.

I/O bound process: I/O bound processes are the ones those spend more time doing I/O operations than computation, though it may have many short CPU bursts.

CPU bound processes: CPU bound processes spend more time in computations and so they have long CPU bursts. Although they may also involve in very short durations of I/O operations.

Q. 5 Differentiate between parallelism and concurrency.
Ans.

Parallelism: Parallelism is the quality of the processes to execute at the same time. Two processes or events are said to be parallel if they occur at the same time. In parallelism, multiple processes can be active at a time.

Concurrency: Concurrency does not mean parallel. With concurrency, multiple processes are executed one after another by interleaving their execution in such a way that it creates an illusion of parallelism. In concurrency only one process can be active at a time.

Generally the concurrency is achieved by interleaving process execution on (may be single) CPU which creates an illusion that processes run in parallel, while parallelism is obtained by multiple processors operating in parallel at a time. Both the techniques achieve computation speedup, though the inherent mechanism used by both concepts is different.

Q. 6. State advantages of process concurrency.
Ans.

The processes in a multiprogramming environment are a blend of I/O bound and CPU bound processes. If these processes are executed sequentially, they underutilize the CPU and I/O devices both. If executed in interleaved fashion, the system gives better efficiency, relatively lower response, turn around and waiting times.

Example: consider three processes A,B and C with different requirements of resources and execution times. The following diagram shows the effect when they are executed in interleaved and non-interleaved mode.

Figure: Non-interleaved execution of processes

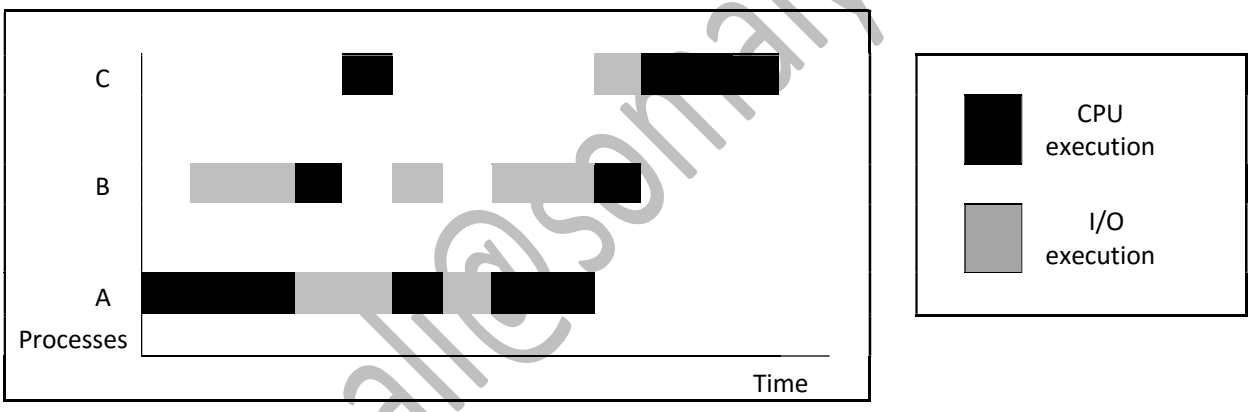


Figure: Interleaved process execution

Q. 7 What are the basic elements of a process?

Ans:

A process is comprised of six components as,

1. Process ID: This is a unique identifier assigned to the process
2. Code: This is program code.
3. Data : These are the data and files required for execution
4. Resources : These are different types of resources allocated by OS
5. Stack: This contains parameters for the functions/procedures called and their return addresses.
6. Process state: This is one of the eleven states process is in.

Q. 8. What are the elements of process environment?

Ans.

The components of process environment are program code and data, memory allocation information, status of the file processing activities, information about the process interaction, information about the resources required for process execution and some miscellaneous information needed by the OS. These elements can be detailed as,

- **Program code and data:** This includes the program code including all the functions and procedures, and the program data, including the stack information.
- **Memory allocation information:** This section has information of the memory areas allocated to process. This proves to be the vital information as it is required to manage the memory accesses by concurrent processes.
- **Status of file processing activities:** This holds the pointers to files opened by process and the current positions in the respective files.
- **Process interaction information:** This holds information necessary to manage interprocess communication through signals and messages, and the IDs of child-parent processes.
- **Resource information:** The resource information of the resources acquired by process is stored.
- **Miscellaneous information :** miscellaneous information needed for a process to communicate with OS is maintained as part of process environment.

Q. 8 What are the fundamental kernel functions of process control?

Ans

The kernel does three main functions to control the processes in hand as given below,

1. Scheduling: Choose the process as per the scheduling policy to be executed next on the CPU.
2. Dispatching: Set up execution of the chosen process on the CPU.
3. Context save: Save information concerning an executing process when its execution gets suspended.

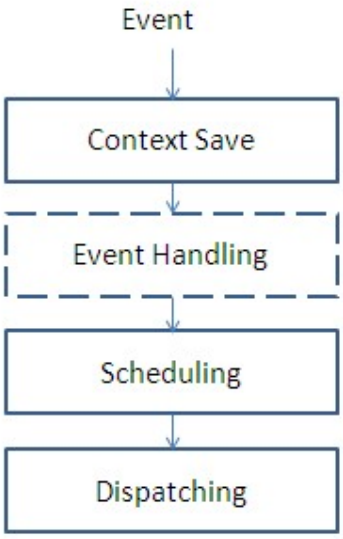


Figure: Fundamental functions to control processes

Here in the diagram, occurrence of the event calls the context save functionality and an appropriate event handling procedure. This event handling may initiate some processes, hence the scheduling function gets invoked to choose the process and in turn, the dispatching function transfers control to the new process.

Q. 9. Define and explain the concept: process state transition.

Ans:

Definition: A state transition for process is defined as a change in its state.

The state transition occurs in response to some event in the computing system. E.g. transition from ready state to running when dispatched by scheduler.

State Transition Models and Process Life Cycles

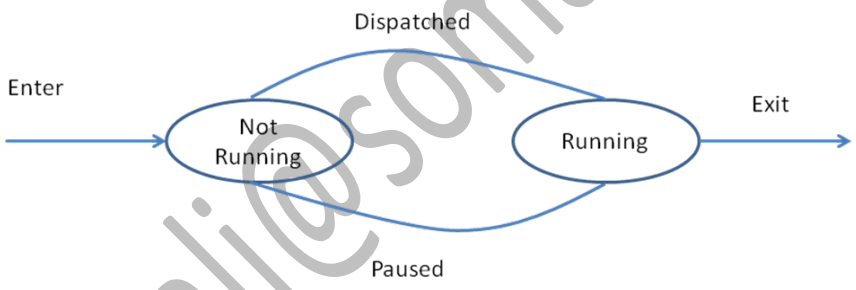
Q. 1. List and explain different states in process state-transition diagram

Ans. The different possible states in a complete state transition diagram are: new, ready, running, suspended, blocked, terminated, etc.

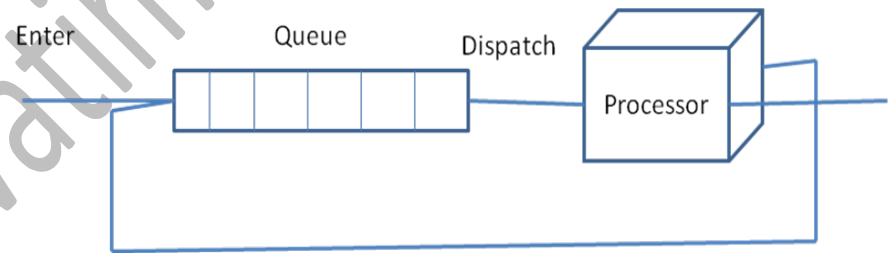
- New: The process is being created
- Ready: the process is ready with all required resources and waiting to be assigned to a processor
- Waiting/Blocked: the process is waiting for some event to occur
- Suspended: the process was waiting for some event to occur, but then is swapped to secondary memory to make room for new processes getting 'blocked'.
- Terminated: the process has finished its execution.

Q. 2. Explain the two-state process model.

Ans:



A



B.

Figure A and B: Two State Model

In two process model, the process is in either of the states, Running or Not Running. A newly created process gets its PCB and enter the system in the Not Running state. It acquires all resources required for its execution and waits for its turn to execute. The dispatcher keeps on interrupting the Running process time to time and selects the other deserving processes to run. Then the Running process is moves to Not Running state and the other selected process transits from Not Running to Running state. The Not Running

processes can be more than one in number and they must be processed in some order. Hence, they are stored in an ordered queue.

Q. 3. Explain the five state process model with the transitions
Ans.

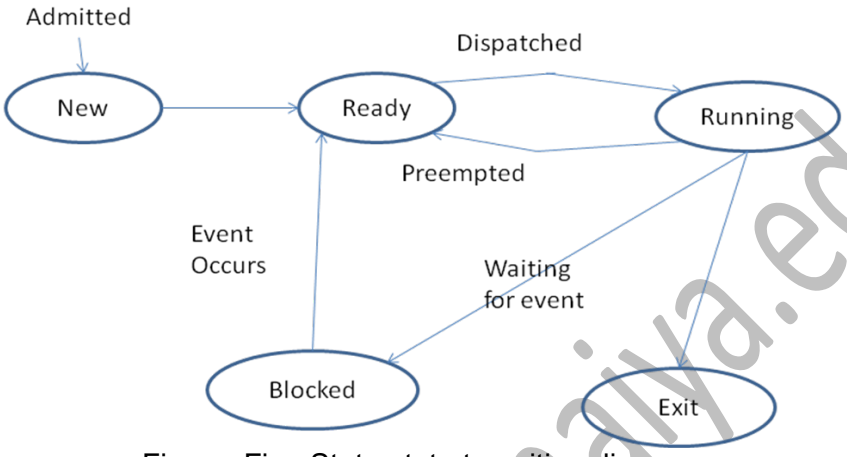


Figure: Five-State state transition diagram

The diagram above depicts the simple life cycle of the process. The state transitions can further be explained as,

State transition	Description
Null→ New	A new process is created to execute a program.
New→ ready	The OS may move a process from New to Ready state depending on the predefined maximum number of processes allowed (Degree of multiprogramming).
Ready→ Running	The process is scheduled by dispatcher. The CPU starts or resumes execution of the instruction codes
Blocked → Ready	The request initiated by process is satisfied or the event on which it is waiting occurs.
Running → Ready	The process is preempted by the OS decides to execute some other process. This transition takes place may be because of expiration of time quantum or arrival of high priority process.
Running → Blocked	The running process makes request for resource(s) or needs some event to occur to proceed further. The process then calls for a system call to indicate its wish to wait till the resource or the event becomes available.

Running → Termination	The program execution is completed or terminated.

Q. 4. What are the causes of process initiation?

Ans:

The main reasons for process creation are: New batch job initiation, Interactive logon, Initiation by OS to provide some service or creation by some other process.

New batch job: while processing the new batch of jobs submitted, the OS creates a process to execute the same.

Interactive logon: when a user logs into the system, a new process is created.

Created by OS to provide some service: The OS initiates a process to perform the service requested by user directly or indirectly, without making the user to wait.

Spawned by an existing process: To support Modularity and/or parallelism, a user program can create some number of new processes.

Q. 5. What are the causes of process blocking?

Ans: The five major reasons of process blocking are:

- Process requests an I/O operation
- Process requests memory or some other resource
- Process wishes to wait for a specific interval of time
- Process waits for message from some other process
- Process wishes to wait for some action to be performed by another process.

Q. 6. What are the causes of process termination?

Ans:

The main reasons of process termination can be listed as,

- **Normal Completion:** The process executes an OS system call to intimate that it has completed its execution.
- **Self termination** (e.g. incorrect file access privileges, inconsistent data)
- **Termination by the parent process:** a parent process calls a system call to kill/terminate its child process when the execution of child process is no longer necessary.
- **Exceeding resource utilization:** An OS may terminate a process if it is holding resources more than it is allowed to. This step can also be taken as part of deadlock recovery procedure.
- **Abnormal conditions during execution:** the OS may terminate a process if an abnormal condition occurs during the program execution. (e.g. memory protection violation, arithmetic overflow etc)
- **Deadlock detection and recovery.**

Q. 7. Draw and explain nine state state-transition diagram.

Ans.

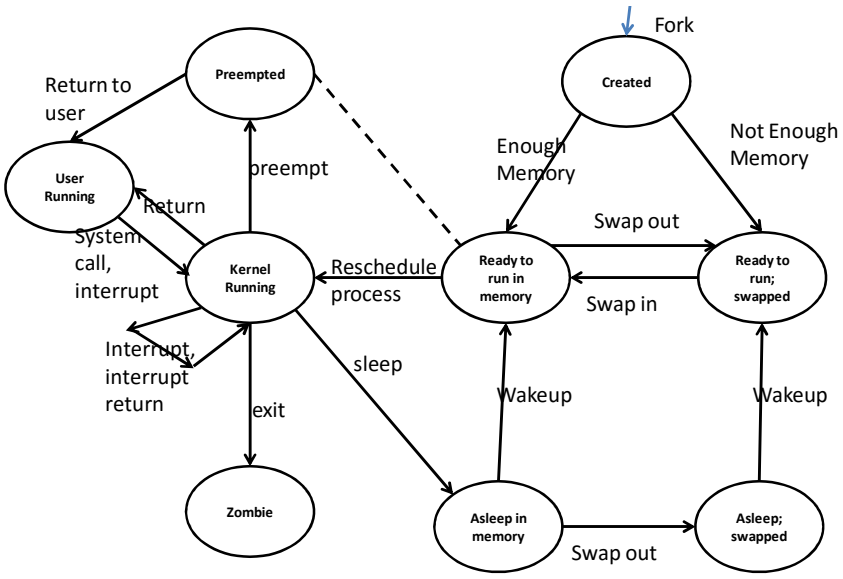


Figure: Nine-state/ UNIX process state transition diagram

UNIX operating system clearly specifies the two modes of execution: user mode, kernel mode. The process gets created on execution for the system call fork(). All the states shown above can be detailed as given in the table below.

Process State	Description
User Running	The process is running in user mode
Kernel Running	The process is running in Kernel mode
Ready to Run, in Memory	The process is ready to run as soon as the kernel dispatches it.
Asleep in Memory	The process is blocked on some event, process is in main memory.
Ready to Run, Swapped	The process is ready to run, but the swapping module must swap it in the main memory, so that kernel can schedule it.
Sleeping, swapped	The process is blocked on some event and it is in secondary memory.
Preempted	The process was returning from kernel mode to user mode, but the kernel preempts it and performs a process switch to dispatch another process.
Created	The process is just created and is not yet ready to run. (may not have all the resources, including memory, which are required to run)
Zombie	The process no longer exists, but it leaves some information (probably accounting information) to its parent process to collect.

Q. 8. What are the events pertaining to a Process?

Ans: The processes wait on some events to execute to complete their execution. Though not complete, some of the events can be listed as follows,

1. Process creation event: a new process gets created
2. Process termination event: a process completes its execution.
3. Timer event: occurrence of timer interrupt
4. Resource request event: a resource request is made by a process
5. Resource release event: process releases a resource and notifies.
6. I/O initiation request event: a process wishes to intiate I/O operation
7. I/O completion event: a process finishes I/O operation
8. Message send event: A message is sent by one process to an another one.
9. Message receive event: a process receives a massage by another one.
10. Signal send event: a signal is sent by a process to an another one
11. Signal receive event: a process receives a signal
12. A program interrupt: An instruction in currently executing process executes some illegal operation and malfunctions.

Q. 9. What are the reasons of process suspension?

Ans:

The basic reasons of process suspension are: swapping, interactive user request, timing, parent process request or some OS reason.

Swapping: The main memory may not be enough to accommodate some ready process. So a currently not running process is shifted from main memory to secondary memory.

Interactive User request: a user may wish to suspend a currently running process for debugging or to manage the use of resources

Timing: A process may be executed on periodic basis and may be suspended while waiting for its next turn of execution.

Parent process request: A parent process may wish to suspend its child process to examine or modify the suspended process or to coordinate the child processes.

Other OS reasons of suspension: The OS may suspend a background or utility process or a process that is causing some problem in normal activities.

Process Management

Q. 1. How Operating System executes a program?

Ans. Every program that is loaded for execution contains declarations of variables and functions if any, statements which may or may not contain data explicitly. During program execution, the instructions use values stored in data area and the stack to perform the intended computation. Any process' address space is formed by its instructions, data and program stack. The OS first needs to allocate some memory to accommodate this address space to realize process execution.

In simple programs the control flows between the main program and the functions according to program logic. The OS treats it as a single program as it is not aware of existence of functions. In such cases, the program execution constitutes a single process (one-to-one relationship).

If the program is coded in a language that contains special features for concurrent programming (e.g. JAVA), then during execution, the OS is informed to execute some portions of the program concurrently. In such a case, one program may have multiple processes (one-to-many relationship) or threads running on its behalf.

Q. 2. What are the child processes? State the advantages of having parent-child process relationships.

Ans. The Operating System creates a process when a program is submitted to it for execution. Depending on the coding style used for writing the program this process may create other processes while in execution. Such processes are called child processes and the processes generating them are called the parent ones. These child processes can also become parents and give birth to new level of child processes.

Advantages of parent-child process relationship:

Sr No.	**Advantage**	**Description**

Q. 3. What are the different mechanism processes use to interact with each other?

Ans. In a typical multiprogramming environment, the processes are required to interact with each other. The basic mechanism can be listed as follows,

Sr No	Interaction Mechanism	Description
1	Data Sharing	The processes interact with each other by altering data values. If more than one processes update the data the same time, they may leave

		the shared in inconsistent state. So, shared data items are protected against simultaneous access to avoid such situation.
2	Message Passing	In this mechanism, the processes exchange information by sending messages to each other.
3	Synchronization	In certain computing environments, the processes are required to execute their actions in some particular order. To help this happen, the processes synchronize with each other to maintain their relative timings and execute in the desired sequence.
4	Signals	The processes may wait for events to occur. It can be intimated to processes through the signaling mechanism.

Q. 4. What are the components associated with process switching?

Ans: Broadly, process switching has two components pertaining to overhead: execution related overhead, resource use related overhead.

Execution related overhead: During process switching the context of running process is saved and to that of the new process needs to be loaded. This overhead is unavoidable in process switching.

Resource use related overhead: A process holds many resources required for execution during its runtime. This resource information leads to large size of process state information, which adds to the process switching overhead.

Q. 5. What are the different control structures maintained by OS to manage processes?

Ans:

The operating system needs the universal information about the current status of each process and resources in system. To have this information handy, OS maintains tables of information about it each entity that system is managing.

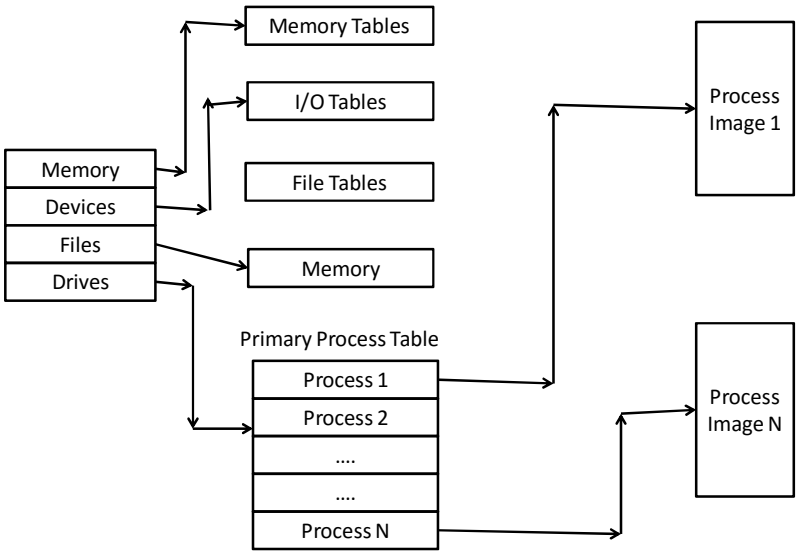


Figure: Operating system Control Structures

Memory Tables: Memory tables keep track of both main and secondary memory. Active processes are stored in main memory and when required, they are moved to secondary memory through the mechanism called 'swapping'. The memory tables maintain the following information:

- The main memory allocation to all processes in system
- The secondary memory allocation to all processes in system
- Shared memory regions in main and virtual memory and their attributes
- Miscellaneous information required to manage virtual memory.

I/O Tables: I/O tables keep track of I/O devices and channels in the computing system. The I/O devices are also resources required by processes. So at any given instance, I/O devices may be available or allocated to a particular process.

File Tables: File tables keeps track of all files, their locations on secondary memory, their current statuses and other attributes such security, sharing, etc. Most of the operating systems, this information is maintained by a module called File Management System.

Process table: Process tables manage processes. They maintain information of processes, their child process references, statuses, allocated resources, process contexts, information required for process synchronization and so on. These pieces of information are stored in process images.

Q. 1. What are the typical fields of a PCB?

Ans:

The Process Control Block contains all information pertaining to a process that is used in controlling the process operation, resource information and information needed for inter-process communication.

Sr No	PCB Field	Contents
	Identifiers	
1	Process ID	This is unique process ID assigned to it at the time of creation
2	Child and Parent IDs	The child and parent process IDs are required for process synchronization
3	User Identifier	The unique identifier associated with the user in multiuser systems.
	Processor State Information	
3	User Accessible Registers	Every processor offers some general purpose registers those are accessible to user. The number of available registers differs with every processor type.
4	Control and Status registers	This field is also called PSW(Program Status Word) which typically contains **Program counter** (Contains the address of the next instruction to be fetched), **Condition codes** (Result of the most recent arithmetic or logical operation), **Status information**(Includes interrupt enabled/disabled flags, execution mode)
5	Stack Pointers	Each process needs one or more LIFO stacks to store parameters and return addresses in case of procedure or system calls. The stack pointer points to stack top.
	Process Control Information	
6	Scheduling and state information	This is information about process state, priority, scheduling related information and event information in case the concerned process is waiting on that!
7	Data Structuring	A process may linked to other processes in queue, ring or some other structure. This information is stored in data structuring field.
8	Interprocess communication	The processes need various flags, messages and signals be exchanged to interact with each other. Some or all this information in stored in PCB.
9	Process Priority	The priority is a numeric value, which may be assigned to a process at the time of its creation. Some priorities can be altered by user and some change with process age, too.
10	Memory Management	This field holds the pointer to segments or to the page tables which describe portions of memory assigned to the process.
11	Resource ownership and Utilization	All the resources or the number of instances of resources assigned to process are described in this field.

Figure: Fields of Process Control Block

Q. 6. What the typical contents of a process image?

Ans:

The collection of program, stack, data and process attributes are called process image. The process image is generally maintained as a continuous or contiguous block of memory which resides in secondary memory. To execute a process, its process image is loaded in main memory or virtual memory.

Typical elements of process image

Sr No	Process image element	Description
1	User Stack	This is part of user space that contains program data, a user stack area, and modifiable/editable programs.
2	User Program	The program under execution.
3	Stack	Each process needs one or more LIFO stacks to store parameters and return addresses in case of procedure or system calls.
4	Process Control Block	Process control block(PCB) contains the data which is used by OS to control and manage the process.

Figure: Typical Elements of Process Image

Q. 7. Write the step by step sequence of actions taken up by OS when a process is created?

Ans:

1. Create a process
2. Assign a unique process ID to newly created process
3. Allocate the memory for the process and create its process image
4. Initialize process control block
5. Set the appropriate linkages to the different data structures such as ready queue etc.
6. Create or expand the other data structures if required.

Q. 8. Differentiate between process context switch and mode switch.

Ans:

Context Switch: Execution of a process may be stopped to respond an interrupt. This exercise needs to save Process Image be saved of one process and load process image of the new process loaded.

Here the processes are switched and processes keep on changing their status as Running and Not running.

Mode switch: every process may switch in between a low privileged user mode and high privileged kernel mode in its lifetime.

Here, the process remains the same. Its status always remains as 'Running' and only the mode of execution keeps on changing. Process switch requires more effort than a mode switch.

Q. 9. Detail the process switching operation.

Ans:

If the process switch changes its state, process switch or context switch occurs. The steps involved in this exercise are as follows:

1. Process context, including program counter and other registers is saved.
2. Update the PCB of the process that was currently running state. The new state assigned may be one of the other states (Ready, Blocked, Ready/Suspend or Exit).
3. Move this updated PCB to appropriate queue (Ready; Blocked on Event i; Ready/Suspend).
4. Select another deserving process for execution.
5. Update the PCB of chosen process and change the process state as Running.
6. Update the memory management data structures.
7. Restore the context of the chosen process so that it can resume from the point if it was interrupted last time, or can start its execution if it was loaded for the first time.

Threads

Q. 1. What are the advantages of threads?

Ans:

The primary advantages of threads include low computation overhead, speed-up and efficient communication.

Low switching overhead: A process thread state does not contain resource allocation state and communication state, instead it contains only computation state. This makes thread switch operation far lighter than process switch.

Speed-up: A process can be broken into multiple threads, which can run concurrently. This mechanism can speed-up the execution speed of an application on uniprocessors and multiprocessors, both.

Efficient Communication: Threads of a process can communicate with each other using shared data space, thus this avoids the necessity of communication system calls.

Q. 2. What are different types of thread?

Ans.

Threads are implemented in different ways. The main difference is on the basis of kernel's knowledge of thread existence. The methods of thread implementation are **Kernel Level Threads(KLTs), User Level Threads(ULTs), hybrid threads.**

Out of these three types, the kernel level threads are most expensive to manage and the user level threads are the cheapest ones. The third type tries to strike the balance between the rest two types. Experiments have proved that switching between kernel level threads is 10 times faster than process switching, and switching user level threads is over 100 times faster than process switching. Though so fast, ULTs cannot support pure multiprocessing and though slow, KLTs provide better parallelism and speed-up in multiprocessor systems.

Q. 3. Explain the working of Kernel Level Threads.

Ans:

The threads require some mechanism to create them, manage them in their lifetime and then destroy them. In Kernel Level Threads (KLTs), kernel provides library to create, schedule, manage and destroy the threads. as these library functions are executed by kernel, it always invokes system calls for every activity involving threads.

When a process calls upon create_thread system call, the kernel assigns a thread ID to it and allocates a thread control block(TCB). This TCB has a pointer to the PCB of its parent process. In case of thread switch, the system saves context of the interrupted thread in its TCB. Then the dispatcher considers all the TCBs in hand, and selects a ready thread. If this new thread belongs to the same process whose thread was earlier interrupted, new process context is not required to be loaded. Otherwise, process switch occurs along with thread switch and the execution resumes. Basically, if both

the threads belong to the same process, then actions to save and load process context are unnecessary. This concept reduces the switching overhead to great context. Moreover, in case of multiprocessor systems, the kernel can schedule multiple ready threads belonging to same process on multiple processors, thus can exploit parallelism to the fullest!

Q. 4. State advantages and disadvantages of Kernel Level Threads.

Ans:

Advantages:

1. KLT offers all advantages that a process has, and still it proves better than a process being 'lightweight'.
2. KLTs belonging to same process can be scheduled in parallel on multiple processes, thus exploiting the parallelism in true sense

Disadvantages:

1. KLTs are solely handled by Kernel and so thread switching is also carried out by kernel in case of event handling. This proves to be very expensive operation even if the interrupted and next scheduled threads both belong to the same process.

Q. 5. What are User Level Threads?

Ans:

The User Level Threads(ULTs) are created and implemented by thread library that is provided by the programming language in which the threads are coded. The thread code is linked with the process. This management does not involve kernel, i.e. in other words, the kernel has no knowledge of ULTs existence. The system schedules the process, and the process threads are identified by the thread library as ready, and internal scheduling is again done by the thread library.

ULT creation and operation:

1. A process calls upon a library function *create_thread* for creation of the new thread.
2. The library creates a TCB(Thread Control Block) for the newly created thread and starts considering if this thread is worth 'scheduling'.
3. When the thread in *running* state invokes the library function to synchronize with other thread, library function checks the new *runnable* thread and switches to that thread of process. This all happens without any intervention of kernel.
4. If the thread library cannot find any runnable thread, the process invokes a '*block me*' system call. This process gets unblocked when some event activates on its threads.
5. The thread library code is part of each process. It performs 'scheduling' to select a thread and organizes its execution. The information stored in TCB is used by thread library while selecting the next *runnable* thread.
6. When a particular thread is scheduled, its CPU state becomes the process' CPU state, and the process stack pointer points the thread stack.

Q. 6. What are the advantages and disadvantages of User level Threads?

Ans:

Advantages:

1. As the thread synchronization and management is handled by thread library, this avoids the overhead of invoking the system calls.
2. The thread switching is simpler and cheaper than in KLTs.
3. This kernel independence enables each process to have its own scheduling policies that suits its own nature.

Disadvantages:

1. The kernel isn't aware of ULT existence; hence, it cannot distinguish between a thread and a process.
2. Blocking of a thread blocks the entire parent process.
3. Since the kernel schedules a process and then thread library schedules a process within a process, at most only one thread can be in operation at any time.
4. ULTs does not support pure multiprocessing as only one thread in one process is active at a time.

Solutions:

1. The program can be written to create multiple processes than multiple threads. this overcomes all the disadvantages given above. But this solution cannot have advantages of thread switch and has to incur overhead of process switch.
2. Blocking of threads mentioned in disadvantage number 2 cannot be avoided with the technique of jacketing. In this, instead of executing a blocking call to system calls, the thread invokes an application level I/O jacket routine which in turn does the thread's calling job.

Q. 7. What is a hybrid thread model?

Ans:

A hybrid thread model has both user-level and kernel-level threads and a method of associating ULTs with KLTs. These different methods of association provide various combinations of the low switching overhead of ULTs and the high concurrency and parallelism of KLTs.

These association methods can be summarized as: **Many-TO-One, One-to-One and Many-to-Many.**

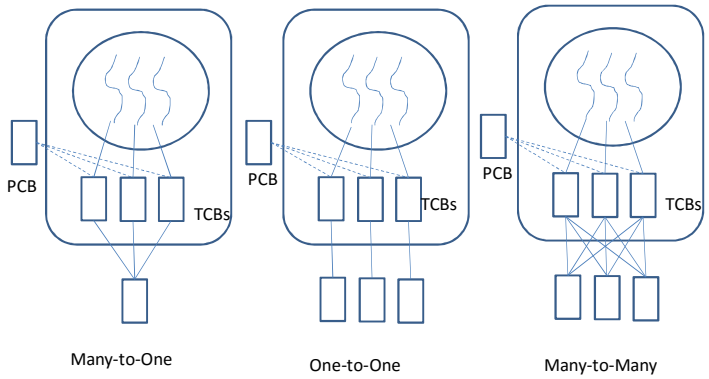


Figure: Association of KLTs with ULTs

The kernel creates KLTs in a process and associates a Kernel Thread Control Block with each corresponding KLT.

In **Many-to-One** association method, a single KLT is created in each process by the kernel and all ULTs created in a process by thread library associated with this KLT. This approach gives an effect similar to more ULTs.

Advantages:

1. ULTs can be concurrent without being parallel.
2. Thread switching is cheap

Disadvantage: Blocking of an ULT in a process blocks all threads in the process.

In the **One-to-One** association, each ULT is mapped permanently into a KLT. This provides an effect very similar to KLTs.

Advantages:

1. This association gives pure multiprocessing on multiprocessor systems
2. Blocking of one thread in a process does not block rest of the threads in the same process.

Disadvantage: switching between threads is done at kernel level and thus incurs high overhead.

The **Many-to-Many** association produces an effect which is very similar to one ULT being mapped into any KLT. This achieves parallelism between ULTs by mapping them into different KLTs.

Advantages:

1. Achieves high parallelism
2. Low thread switching overhead
3. Blocking of one thread does not block rest of the threads in the process.

Disadvantages: system is quite complex to implement.

Q. 8. What is a multithreaded process model?

Ans:

In a multithreaded environment, a thread is called lightweight process while, a process is simply a unit of resource allocation and unit of protection.

The process maintains only:

- A virtual address space to store the process image
- Protected access to processors and with rest of the processes for interprocess communication, with files and with I/O devices.

The thread contains:

- A thread execution state
- A thread context, generally saved when process isn't in *Running* state.
- A thread execution stack
- Thread storage area to store local and temporary data
- A shared access to resources of its process.

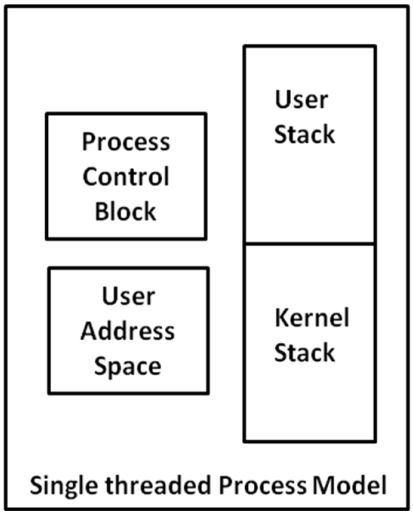


Figure: Single threaded Process Model

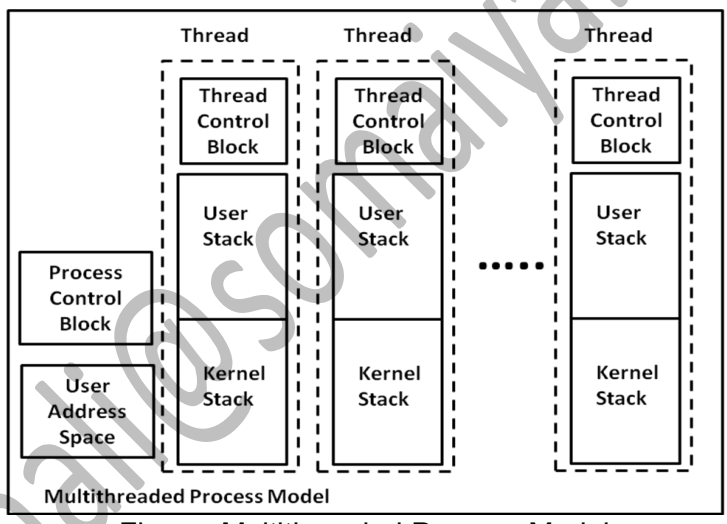


Figure: Multithreaded Process Model

As shown in above figure, all the threads of the same process share process control block and user address space, i.e. they share the execution state and the resources assigned to the same process.

Q. 9. List the operations associated with process threads?

Ans:

There are four basic thread operations associated with a change in thread state viz: spawn, block, unblock, finish.

- **Spawn:** Typically, when a new process is spawned, a thread for that process is also spawned. Also, a thread can also spawn another threads in the same process. The new thread also gets its own registers, context and the stack space.
- Block: when a thread waits on some event, it blocks itself and the thread context is saved.
- **Unblock:** when the event on which the process is waiting occurs, the thread gets shifted to ready queue.

- **Finish:** when a thread finishes its execution, its register and stack contexts are deallocated.

Q. 10. Explain the different process-thread relationships.

Ans.

The process and its thread share either of four relationships amongst them as: One-to-One, Many-to-Open

Q. 11. List some typical uses of threads in single user environment.

Ans:

Though not a complete list, some applications of threads in single user environment are:

- **Foreground and background work:** multiple threads of a single program can execute some processes in background and foreground independent of each other. E.g. the HTML pages when load, different processes load text, images, videos and rest of the graphics independent of each other.
- **Asynchronous processing:** asynchronous elements of program are implemented as threads.
- **Execution speed:** as threads can execute independent of each other, one batch of jobs can be processed by some threads while the other thread may read next batches of job.
- **Modular program development :** projects or programs those accept inputs from various sources and pass on information to various modules of processing, are better be implemented using threads.

System Calls for Process Management

Q. 1. List the different system calls used for process management

Ans:

The process management related system calls are listed as follows,

Sr No	System Call	Description
1	fork()	This system calls creates a new process.
2	exec()	This call is used to execute a new program on a process.
3	wait()	This call makes a process wait until some event occurs.
4	exit()	This call makes a process to terminate
5	getpid()	This system call helps to get the identifier associated with the process.
6	getppid()	This system call helps to get the identifier associated with the parent process.
7	nice()	The current process priority can be changed with execution of this system call.
8	brk()	This call helps to increase or decrease the data segment size of the process.
9	Kill()	The forced termination of any process can be executed with this system call.
10	Signal()	This system call is invoked for sending and receiving software interrupts

Q. 2. Explain working of fork() system call

Ans:

The **fork()** system call is used to create a new process. When the system executes this system call in response to process creation request, following steps are carried out.

1. The operating system allocates a slot in the process table for the newly created process.

2. This new process, i.e. child process is then assigned a unique ID.
3. System then creates a logical copy of the parent process context.
4. The file and inode table counters associated with parent process are incremented as this area may be shared between parent process and the child process.
5. The child process ID is returned to the process and '0' value is assigned to the child process.
6. All the above steps are carried out in kernel of the parent process.
7. The control of execution may remain with the parent process, may be transferred to child process or handed over to a third process leaving the parent and child processes in 'Ready-to-Run' state.

Q. 3. What happens when the system terminates a process with exit() system call?

Ans: The exit() system call is executed to terminate a process. This call is generally executed with some status code which is passed as an argument to the call. This status_code indicates the termination status of the corresponding process. The sequence of actions taken by OS in response to exit() is as follows,

1. Kernel sends a close call to all open files of the process.
2. Kernel also releases the memory assigned to the pr0ocess for execution.
3. The User area of the process is destroyed by the system.
4. The process structure is not destroyed though, and is retained till the parent process destroys the same.
5. The process though terminated, still exists as a dead process and is generally called a Zombie process.
6. The exit call also sends signal to the parent process indicating the termination of its child process.

Processes in Unix, Solaris and windows 2000 thread and SMP Management.

Q. 1 what are the Windows process attributes?

Ans:

Windows supports concurrent processing and multithreading. Windows process object attributes can be listed as,

Attribute	Description
Process ID	Unique process identifier
Security Descriptor	Describes process owner, process access rights for users, process access permissions for the users of the system.
Base priority	A baseline execution priority for the process and its threads
Default processor affinity	In a multiprocessor environment, the set of processors on which processes or threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all the threads of the process have spent with processor.
I/O counters	The variables those keep record of the number and types of I/O operations that the process's threads have performed.
VM operation counters	The variables those keep record of the number and types of virtual memory operations that the process's threads have performed.
Exception/debugging ports	These are interprocess communication channels to which the process manager can communicate in case the process causes any kind of exception.
Exit status	The reason of process termination.

Q. 2 what are the windows thread attributes?

Ans:

The windows thread attributes can be listed as given in the following table:

Attributes	Description
Thread ID	A unique ID assigned to thread
Thread context	This consists of register values and other data that defines the executions state of the thread
Dynamic priority	The execution priority of the thread at any given moment
Base priority	The basic priority assigned to the thread. This is also the lowest priority level the thread can have.
Thread processor affinity	In a multiprocessor environment, the set of processors on which processes or threads can run.
Thread execution time	The total time a thread time has spent with processor; which includes user mode and kernel mode processing, both.
Alert status	This is a flag that denotes in case a waiting thread can execute an asynchronous procedure call.
Suspension count	This is count of the number of times the thread's execution gets suspended without being resumed..
Impersonation token	This is a temporary access token that permits a thread to execute operations on behalf of another process.
Termination port	These are interprocess communication channels to which the process manager can communicate when the thread terminates.
Thread exit status	This field describes the reason for the thread's termination.

Q. 3 what are the different types of processes in UNIX?

Ans:

The Unix OS has three types of processes as: **user process, kernel process and daemon process.**

User process: The low privileged processes created on behalf of user are called user processes.

Kernel process: The high privileged processes initiated by operating system in response to calls made by user processes are called kernel processes. The system calls are the kernel processes.

Daemon process: these processes perform the functions in a system wide basis. The functions are generally of auxiliary kind, but they are very important from the point of controlling of the computing environment of the system. These processes once created, exist throughout the life time of the process. E.g. print spooling, network management.

Concurrency

1. Basic Terminology & Definitions

   Concurrency, Process synchronization, Mutual Exclusion, Race Condition, Critical Section, Data Coherence, Deadlock, Livelock, starvation, atomic operation, busy waiting, Semaphore, Monitor, message passing, mutex,

2. Principles, Approaches and Requirements

   Process computation synchronizations, properties of Critical Section implementation, solution mechanisms to achieve concurrency, requirements for mutual exclusion, design issues for concurrency.

3. . Software approaches

   Dekker's solution, Peterson's solution, Comparison of both solutions

4. Hardware approaches

   Interrupt disabling, special instructions

5. Support from Operating systems and Programming Languages

Semaphores, Monitors, Message passing

6. Concurrency Mechanisms in different OS

   Concurrency in Unix, Solaris and windows

7. Solutions to classical problems

   Solutions to producer-consumer problem, readers-writers problem, sleeping barbershop problem using semaphore, monitors and message passing

8. Multiple Choice Questions

1. Basic Terminology & Definitions

Q. Define Concurrency, Process synchronization, Mutual Exclusion, Race Condition, Critical Section, Data Coherence, Deadlock, Livelock, starvation

Ans.
Concurrency : Concurrency is defined as a property of systems in which several processes are executing at the same time.

Process synchronization: This refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of actions.

Mutual Exclusion: Mutual Exclusion is the ability to exclude all other processes from a course of actions while one process is granted that ability

Race Condition: Race condition is defined as a flaw in process whereby the output or result of the process is unexpectedly and critically dependent on the sequence or timing of other events. The term originates with the idea of two processes racing each other to influence the output first.

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

Example : consider two processes, P1 and P2, share the global variable a. if while executing P1 updates a to the value 1, and some time later P2 updates a to the value 2.Thus, the two tasks are in a race to write variable a. In this example the "loser" of the race (the process that updates last) determines the final value of a.

Critical Section: Critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

Data Coherence: Data coherence refers to the consistency of data or variables stored in shared memory.

Example: Consider two data items a and b that are required to be maintained in a relationship a=b in a certain application.

Now consider two processes:

```
P1:     a=a+1;
        b=b+1;

P2:     a=2*a;
        b=2*b;
```

if these processes P1 and P2 execute concurrently respecting mutual exclusion on a and b in the sequence as:

```
a=a+1;
b=2*b;
b=b+1;
a=2*a;
```

Then this execution does not maintain the a=b relationship. i.e. execution of P1 and P2 has violated data coherence.

Deadlock: Deadlock is defined as a situation when multiple processes are waiting for the availability of a resource that will not become available as it is held by another process which is in similar wait state.

Livelock: A condition in which one or more processes continuously change their state in response to changes in the other process(es) without doing any useful work. This is similar to deadlock in that no progress is made but differs in that neither process is blocked nor waiting for anything.

Starvation: A condition in which a process is indefinitely delayed because other processes are always given the preference.

Q. Define atomic operation, busy waiting, Semaphore, Monitor, message passing, mutex.

Ans:

Atomic Operation: A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.

Busy Waiting: it is repeated execution of some code while it is waiting for some event to occur.

This should generally be avoided as it simply wastes the CPU cycles and does not give any fruitful output.

Semaphore: Semaphore is an integer variable that can be initialized to some value.

The semaphore values are used for signaling the processes for communication. The semaphores can be operated upon by three operations viz, initialize, increment and decrement. All these operations are atomic in nature. Depending on exact definition of semaphore, the decrement (wait) operation may block a process and the increment (signal) operation may unblock a process.

Semaphores are supported by operating system to achieve concurrency and synchronization.

Monitor: Monitor is a programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type.

The processes have only one entry and exit points to the monitor. Processes can access the same by invoking one of the procedures contained in monitor, which in turn can access the data and variables local to the monitor. To achieve mutual exclusion, only one process can be active inside a monitor. Generally, the access

procedures are *critical sections*. A monitor may have a queue of processes that are waiting to access it.

A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions:

- cwait(c): Suspend execution of the calling process on condition *c*. The monitor is now available for use by another process.
- csignal(c): Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

Message Passing: message passing is a mechanism by which processes interact with each other to achieve synchronization and communication.

Message passing is implemented with two primitives as *send* and *receive*. It can be used in uniprocessor, shared-memory multiprocessors as well as in distributed systems.

Mutex: Short for **mu***tual* **ex***clusion object*. A mutex is a program object that allows multiple program threads to share the same resource, such as file access, but not simultaneously. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

When a program is started, a mutex is created with a unique name. After this stage, any thread that needs the resource must lock the mutex from other threads while it is using the resource. The mutex is set to unlock when the data is no longer needed or the routine is finished.

Principles, Approaches and Requirements

Q. what are the different types of process computation synchronizations?
Ans: The processes of computation use two kinds of synchronizations: **Control Synchronization** and **Data Synchronization**
Control Synchronization: This kind of synchronization is needed if a process wishes to perform some action a_i only after some other process have executed a set of actions $\{a_j\}$ or only when a set of conditions $\{c_k\}$ hold.

A simple example is a process which waits for its child process to complete before terminating itself.
Data Access Synchronization: Race conditions may arise if processes access shared data in an uncoordinated manner.

Data access synchronization is used to access shared data in a mutually exclusive manner. It avoids race conditions and safeguards consistency of shared data.

Q. What is the basic principle of synchronization?
Ans: The basic principle used to implement control or data synchronization is to block a process until an appropriate condition is fulfilled.

Thus to implement synchronization a process p_i can be blocked till some process p_k reaches a specific point in its execution. Mutual exclusion over shared data is implemented by blocking a process till another process finishes accessing the shared data.

Q. What are the properties of Critical Section implementation?
Ans: A CS implementation for any data item d_s should work as a scheduler for a resource. It must keep track of all processes those wish to enter a CS for d_s and select a process for entry in a CS in accordance with the notions of mutual exclusion, efficiency and fairness.
The essential properties of any CS implementation can be summarized as follows:

1. Correctness: at any moment, at most one process may execute a CS for a data item d_s.
2. Progress: When a CS is not in use, one of the processes wishing to enter it will be granted entry to the CS.
3. Bounded wait: After a process P_i has indicated its desire to enter a CS for d_s, the number of times other processes can gain entry to a Cs for d_s ahead of p_i is bounded by a finite integer.
4. Deadlock freedom: The implantation should be free of deadlock.

Q. What are the solution mechanisms to achieve concurrency?
Ans. The solution mechanisms to achieve concurrency are:

- **Software approach:** Here no hardware, OS, or programming language level supported is assumed. This mechanism burdens programmers to write the programs (which when executed becomes processes) to achieve concurrency themselves. Dekker's or Peterson's algorithms are known for the software approach solutions to mutual exclusion.
- **Hardware approach** : This approach makes use of special hardware instructions and control over the hardware such as disabling the interrupts.

- **Support from operating systems and programming languages**: Here, the operating system offers support to some signals those may help in bringing mutual exclusion or the programmers can be provided with language constructs or library with which they can easily write the code for the process synchronize with each other.

Q. what are the requirements for mutual exclusion?

Ans: Any mechanism that strives to achieve mutual exclusion must respect following set of requirements.

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section
2. A process that halts in its noncritical section must do so without interfering with other processes.
3. The solution must ensure no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

Q. What are the design issues where concurrency is relevant?

Ans. The design issues where concurrency is relevant:

- Communication among processes,
- sharing of and competing for resources,
- synchronization of the activities of multiple processes, and
- allocation of processor time to processes.

/* PROCESS 1 */	/* PROCESS 2 */	...	/* PROCESS n */

void P1
{
while (true) {
/* preceding code /;
EnterCriticalSection (Ra);
/* Execute Critical Section */;
ExitCriticalSection (Ra);
/* remaining code */;
}
}
	void P2
	{
	while (true) {
	/* preceding code */;
	EnterCriticalSection (Ra);
	/* Execute Critical Section */;
	ExitCriticalSection (Ra);
	/* remaining code */;
	}
	}
		...	void Pn
			{
			while (true) {
			/* preceding code */;
			EnterCriticalSection (Ra);
			/* Execute Critical Section */;
			ExitCriticalSection (Ra);
			/* remaining code */;
			}
			}

Figure: Ilustration of Mutual Exclusion

Q. What are advantages of Mutual Exclusion?

Mutual exclusion techniques can be used to resolve conflicts, such as competition for resources, and to synchronize processes so that they can cooperate.

3. Software approaches

Q. Explain the Dekker's four attempts with mutual exclusion.
Ans. Dekker suggested total five solutions to mutual exclusion one by one to achieve mutual exclusion. The first four solutions were not complete and all of them suffered with busy waiting.

If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is. If one process is already in the critical section, the other process will busy wait for the first process to exit. This is done by the use of two flags, flag[0] and flag[1], which indicate an intention to enter the critical section and a turn variable which indicates who has priority between the two processes.

<table>
<tr><td colspan="4"></td></tr>
<tr><td colspan="2">Attempt One</td><td colspan="2">Attempt Two</td></tr>
<tr><td>/* Process 0*/
.
.
.
while(turn!=0)
/* do nothing */;
/*critical section*/;
turn =1;
.
.</td><td>/* Process 1*/
.
.
.
while(turn!=1)
/* do nothing */;
/*critical section*/;
turn =0;
.
.</td><td>/* Process 0*/
.
.
.
while(flag[1])
/* do nothing */;
flag[0]=true;
/*critical section*/;
flag[0]=false;
.
.
.</td><td>/* Process 1*/
.
.
.
while(flag[0])
/* do nothing */;
flag[1]=true;
/*critical section*/;
flag[1]=false;
.
.
.</td></tr>
<tr><td colspan="2">First Attempt:
Succeeds in enforcing mutual exclusion
Uses variable to control which thread can execute
Constantly tests whether critical section is available
Busy waiting
Wastes significant processor time
Problem known as strict alternation synchronization
Each thread can execute only in strict alternation</td><td colspan="2">Second Attempt:
Removes strict alternation synchronization
Violates mutual exclusion
Thread could be preempted while updating flag variable
Not an appropriate solution</td></tr>
</table>

Attempt Three		**Attempt Four**	

/* Process 0*/
.
.
.
flag[0]=true;
while(flag[1])
/* do nothing */;
/*critical section*/;
flag[0]=false;
;
.
.
	/* Process 1*/
	.
	.
	.
	flag[1]=true;
	while(flag[0])
	/* do nothing */;
	/*critical section*/;
	flag[1]=false;
	;
	.
	.
		/* Process 0*/
		.
		.
		.
		flag[0]=true;
		while(flag[1])
		{ flag[0]=false;
		/*delay*/
		flag[0]=true;
		}
		/*critical section*/;
		flag[0]=false;
		.
		.
			/* Process 1*/
			.
			.
			.
			flag[1]=true;
			while(flag[0])
			{ flag[1]=false;
			/*delay*/
			flag[1]=true;
			}
			/*critical section*/;
			flag[1]=false;
			.
			.

Third Attempt: • Set critical section flag before entering critical section test ○ Once again guarantees mutual exclusion • Introduces possibility of deadlock ○ Both threads could set flag simultaneously ○ Neither would ever be able to break out of loop • Not a solution to the mutual exclusion		**Fourth Attempt:** • Sets flag to false for small periods of time to yield control • Solves previous problems, introduces indefinite postponement ○ Both threads could set flags to same values at same time ○ Would require both threads to execute in tandem (unlikely but possible) • Unacceptable in mission- or business-critical systems	

Unfortunately all of the above solutions did not work well but finally he had a perfect solution wherein all the processes run in synchronization.

Dekker's correct solution:

boolean flag [2];
int turn;
	void main()
	{ flag[0]=flag[1]=false;
	turn=1;
	parbegin(P0,P1);
	}

```
void P0()
{
  while (true)
   {
     flag[0]=true;
      while (flag[1])
              if (turn==1)
                {
                   flag[0]=false;
                          while (turn==1)
                             /* Do Nothing*/
                    flag[0]=true;
                   }
        /*Critical Section */
        turn=1;
         flag[0]=false;
          /*Remainder Code*/
   }
}
```

```
void P1()
{
  while (true)
   {
     flag[1]=true;
      while (flag[0])
              if (turn==0)
                {
                   flag[1]=false;
                          while (turn==0)
                              /* Do Nothing*/
                    flag[1]=true;
                   }
        /*Critical Section */
        turn=0;
         flag[1]=false;
          /*Remainder Code*/
   }
}
```

The algorithm uses two variables, *flag* and *turn*. A *flag[n]* value of *true* indicates that the process wants to enter the critical section. The variable *turn* holds the ID of the process whose turn it is. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting *turn* to 0.The code above gives the solution by using notion of favored threads to determine entry into critical sections. It resolves conflict over which thread should execute first. The central concept here is, 'each thread temporarily unsets critical section request flag and the favored status alternates between threads'. Thus this algorithm guarantees mutual exclusion and avoids previous problems of deadlock, indefinite postponement.

The characteristics of Dekker's solution,

- Proper solution
- Guarantees mutual exclusion
- Avoids previous problems of deadlock, indefinite postponement
- Uses notion of favored threads to determine entry into critical sections
  - Resolves conflict over which thread should execute first
  - Each thread temporarily unsets critical section request flag
  - Favored status alternates between threads

Dekker's algorithm wasn't accepted as a solution to mutual exclusion as it could support only two processes and in the real scenario system has many processes to handle.

Q. Explain the Peterson's solution to Mutual Exclusion.

Ans. Peterson's solution for the mutual exclusion is far simpler than that of the Dekker's solution and as it could also support n number of processes, it is considered as the very first complete solution for the concurrency control in that particular category.

```
void Pi()
{
    While (true)
    {
        flag[i] := true;
        turn := j;
            while (flag[ j] and turn = j )
                /*Do Nothing*/ ;
       /*Critical Section */
       flag[i] := false;
       /*Remainder Section*/
   }
}
```

```
boolean flag[n];
Int turn;

void main()
{
     flag[i] := false;
     flag[ j] := false;

parbegin(Pi,Pj,….Pn);
}
```

Figure: Peterson's Solution

Q. Compare Dekker's and Peterson's algorithms

The comparison between two algorithms can be enumerated as

- The Peterson's solution is simpler than Dekker's algorithm
- If process 1 has set *p1wantstoenter* to true, process 2 cannot enter its critical section.
- Mutual blocking is prevented -- if process 1 is blocked in its while loop, then *p2 wantstoenter* is true and *favoredprocess* = second.
- Process 2 cannot monopolize access to the critical section because it has to set *favoredprocess* to first before each attempt to enter its critical section.
- This algorithm can be generalized to the case of n processes.

Dekker's algorithm is the historically first software solution to mutual exclusion problem for 2-process case. The first software solution for *n*-process case was subsequently proposed by Dijkstra. These two algorithms have become de facto examples of mutual exclusion algorithms, for their historical importance. Since the publication of Dijkstra's algorithm, there have been many solutions proposed in the literature. In that, Peterson's algorithm is one among the very popular algorithms. Peterson's algorithm has been extensively analyzed for its elegance and compactness. .

Q. Write the failure causes of Dekker's four attempts with software solution to mutual exclusion.

Ans. The reasons why the first four attempts to enforce mutual exclusion can be summarized as given in the following table.

Attempt	Failure causes
Attempt 1	• Busy waiting

	• Process one always enters first. • Strict alteration • Execution speed is dictated by slower process in set • Failure of one process permanently blocks another one
Attempt 2	• Busy waiting • Failure of one process may permanently block another one • Solution is not independent of relative process execution speeds • Does not guarantee mutual exclusion
Attempt 3	• Busy waiting • Causes deadlock
Attempt 4	• Causes livelock • indefinite postponement

Q. What is the problem of busy wait?
Ans. A process may have a CS implemented with a simple code as,

```
While(some other process is in CS )
        {do nothing}
        Critical
        Section
```

In the above while loop, the process checks for the given condition and if the condition is true, it keeps looping until the other process exits its CS. Such a situation in which a process repeatedly keeps on checking if a condition that would enable it to get past a situation point is satisfied is called *Busy Waiting*. Thus the busy wait simply keeps CPU busy in executing a process even when the process does nothing. If more processes with lower priority are waiting, they are denied the CPU, thus their response times and turnaround times suffer. All these things make the overall system performance suffer. It can also cause **deadlock** and make the processes **starve** for CPU.

To avoid busy waits, a process waiting for an entry to a CS should be put into a *blocked* state. This state should be changed to *ready* only when it can be allowed to enter its CS.

4. Hardware approaches

Q. what are the hardware approaches to enforce mutual exclusion?
Ans. The hardware approach suggests two solutions to enforce mutual exclusion as: **disabling interrupt** and **use of special instructions**.

Disabling interrupt: In a uniprocessor system, concurrent processes are not overlapped but they can only be interleaved. Furthermore, a process executes until it calls an OS service or until it is interrupted. Thus, it is sufficient to prevent a process from being interrupted to guarantee mutual exclusion, This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts.

The pseudo code:

```
while (true) {
/* disable interrupts */;
/* critical section */;
/* enable interrupts */;
/* remainder */;
}
```

As the processes are not interrupted, the mutual exclusion is guaranteed. Though seems simple, the price paid for the same is very high. As the processor is not allowed to interleave the processes, the efficiency and overall system performance degrades significantly.

Also, this particular solution will not work with multiprocessor architecture. When the computer includes more than one processor, it is possible (and typical) for more than one process to be executing at a time. In this case, disabled interrupts do not guarantee mutual exclusion.

Special Machine Instructions:
In a multiprocessor systems, several processors share main memory. Moreover, the interrupt disabling cannot help the mutual exclusion. So to give mutually exclusive access to any memory location the processor designers have proposed several machine instructions that carry out some actions atomically, with one instruction fetch cycle.

Examples of such some atomic instructions are given here.
Compare & Swap Instruction

```
int compare_and_swap (int *word, int testvalue, int newvalue)
{
int oldvalue;
oldvalue = *word;
if (oldvalue == testvalue) *word = newvalue;
return oldvalue;
}
```

Exchange Instruction

```
void exchange (int register, int memory)
{
int temp;
temp = memory;
memory = register;
register = temp;
}
```

Q. Discusses advantages and disadvantages of hardware approach to enforce mutual exclusion.

Ans.

The use of a special machine instruction to enforce mutual exclusion has a number of **advantages**:

- The solution works with any number of processes on uniprocessor as well as multiprocessors architectures.
- It is very simple and therefore easy to verify.
- It can also support multiple critical sections; and each critical section can be defined by its own variable.

Disadvantages:

- **Busy waiting is employed.** While the other process is in critical section, the said process has to wait for the same. This continues to consume processor time giving poor system performance
- **Starvation is possible.** When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access.
- **Deadlock is possible.** Consider the following scenario on a single-processor system. Process P1 executes the special instruction (e.g., compare&swap, exchange) and enters its critical section. P1 is then interrupted to give the processor to P2, which has higher priority. If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanism. Thus it will go into a busy waiting loop. However, P1 will never be dispatched because it is of lower priority than another ready process, P2. Because of the drawbacks of both the software and hardware solutions just outlined, we need to look for other mechanisms.

Q. What is indivisible or atomic operation?

Ans. An atomic operation on a asset of data items { d_s } is an operation that can not be executed concurrently with any other operation involving a data item included in { d_s }

Example. Special operation : Exchange (XCHG)

```
void exchange (int register, int memory)
{
int temp;
```

```
temp = memory;
memory = register;
register = temp;
}
```

The atomic operations can prove solution to *race conditions* and *data coherence* as they do not allow preemption of process while in atomic mode.

5. Support from Operating systems and Programming Languages

Q. what is the significance of signals in process synchronizations?
Ans. The processes use control synchronization to coordinate their activities with respect to one another. A frequent requirement in process synchronization is that a process pi should perform an action a_i only after process p_j has performed some action a_j. This synchronization requirement is met using the technique of signaling.

The signaling can be achieved with the use of Boolean variables or flags which will indicate whether a process P_i has performed action a_i for a process P_j to perform action $_{aj}$. All the processes those should precede P_i can check the contents of flag variables before performing the coordination-requiring actions and accordingly can proceed further or block themselves.

Signaling when implemented well with a blend of atomic operations can help a lot and can avoid the race conditions by guarantying the proper coordinated synchronization among the processes.

Example: when finished writing, the writer in readers-writers problem should activate one of the waiting writer or activate all the waiting readers by sending them respective signals.
Semaphores and monitors make use of signals for synchronizations.

A. Semaphores

Q. What are different types of semaphore?
Ans: The different types of semaphore are: General or counting Semaphore, Binary Semaphore, Strong Semaphore, and Weak Semaphore.

Binary Semaphore: Binary Semaphore is a semaphore that takes on only the values 0 and 1.

General or counting Semaphore: Any Semaphore that is not restricted to have value only 0 and 1 and can have values >=2 are called general or counting semaphores.
In other words, the nonbinary semaphore is often referred to as either a counting semaphore or a general semaphore.

Strong Semaphore: A semaphore whose definition includes FCFS policy when it is unblocked from the blocked queue, is called a Strong Semaphore.

Weak Semaphore: A semaphore whose definition does not include any policy when it is unblocked from the blocked queue, is called a Weak Semaphore.

Q. Differentiate between mutex and semaphore.
Mutex: Mutexes are typically used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only

allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section

Semaphore: A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).
Example: Mutex is a key to a toilet. One person can have the key - occupy the toilet - at the time. When finished, the person gives (frees) the key to the next person in the queue. While semaphore is the number of free identical toilet keys. If one has four toilets with identical locks and keys then the semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.

Q. List and explain semaphore primitives.
Ans. Semaphore is an integer variable that can be initialized to some value.

The semaphore values are used for signaling the processes for communication. The semaphores can be operated upon by three operations viz, initialize, increment and decrement. All these operations are atomic in nature. Depending on exact definition of semaphore, the decrement (wait) operation may block a process and the increment (signal) operation may unblock a process.

```
struct semaphore {
int count;
queueType queue;
};

void semWait(semaphore s)
{
s.count--;
if (s.count < 0) {
/* place this process in s.queue */;
/* block this process */;
}
}

void semSignal(semaphore s)
{
s.count++;
if (s.count <= 0) {
/* remove a process P from s.queue */;
/* place process P on ready list */;
}
}
```

Figure: Definition of Semaphore Primitives

The above primitives can be used to bring synchronization among communicating processes as,

```
/* program mutualexclusion */

const int n = /* number of processes */;

semaphore s = 1;
void P(int i)
{
while (true) {
semWait(s);
/* critical section */;
semSignal(s);
/* remainder */;
}
}

void main()
{
parbegin (P(1), P(2), . . ., P(n));
}
```

Q. What are advantages and disadvantages of semaphore implementation?
Ans.

The **advantages** of semaphore implementation are:

1. The first and foremost advantage is simplicity.
2. In semaphores there is no spinning, hence no waste of resources due to no busy waiting. i.e. unnecessary CPU time is not spent on checking if a condition is satisfied to allow the thread to access the critical section.
3. Semaphores permit more than one thread to access the critical section, in contrast to alternative solution of synchronization like monitors, which follow the mutual exclusion principle strictly. Hence, semaphores allow flexible resource management.
4. Finally, semaphores are machine independent, as they are implemented in the machine independent code of the microkernel services.

The **disadvantages** of semaphore:

1. The technique of semaphore is more prone to programmer error. The programmer may loose track of *signal* and *wait* signals or those operations can be misplaced. The implementation can also lead to deadlock or violation of mutual exclusion due to programmer error.
2. The semaphore implementations are difficult to debug and fix.
3. Semaphores does not support modularity.
4. Semaphores are quite impractical when it comes to large scale use.

Monitors

Q. Explain the structure of monitor.

A monitor is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

In some applications threads attempting an operation may need to wait until some condition P holds true. The solution is **condition variables**. Conceptually a condition variable is a queue of threads, associated with a monitor, on which a thread may wait for some condition to become true. Thus each condition variable c is associated with an assertion P_c. While a thread is waiting on a condition variable, that thread is not considered to occupy the monitor, and so other threads may enter the monitor to change the monitor's state. In most types of monitors, these other threads may signal the condition variable c to indicate that assertion P is true in the current state.

Thus there are two main operations on condition variables:

- **Wait(**c) is called by a thread that needs to wait until the assertion P_c is true before proceeding. While the thread is waiting, it does not occupy the monitor.
- **Signal**(c) (sometimes written as **notify** c) is called by a thread to indicate that the assertion P is true.

When a signal happens on a condition variable that at least one other thread is waiting on, there are at least two threads that could then occupy the monitor: the thread that signals and any one of the threads that is waiting. In order to at most one thread occupies the monitor at each time, a choice must be made. Two schools of thought exist on how best to resolve this choice. This leads to two kinds of condition variables which will be examined next:

- *Blocking condition variables* or *Signal and Wait* give priority to a signaled thread.
- *Nonblocking condition variables* or *Signal and Continue* give priority to the signaling thread.

```
enter the monitor:
    enter the method
    if the monitor is locked
        add this thread to e
        block this thread
    else
        lock the monitor

leave the monitor:
    schedule
    return from the method
```

```
wait c :
    add this thread to c.q
    schedule
    block this thread

signal c :
    if there is a thread waiting on c.q
        select and remove one such thread t from c.q
        (t is called "the signaled thread")
        add this thread to s
        restart t
        (so t will occupy the monitor next)
        block this thread

schedule :
    if there is a thread on s
      select and remove one thread from s and restart it
      (this thread will occupy the monitor next)
    else if there is a thread on e
      select and remove one thread from e and restart it
      (this thread will occupy the monitor next)
    else
      unlock the monitor
      (the monitor will become unoccupied)
```

Figure: pseudopodia for monitor operations

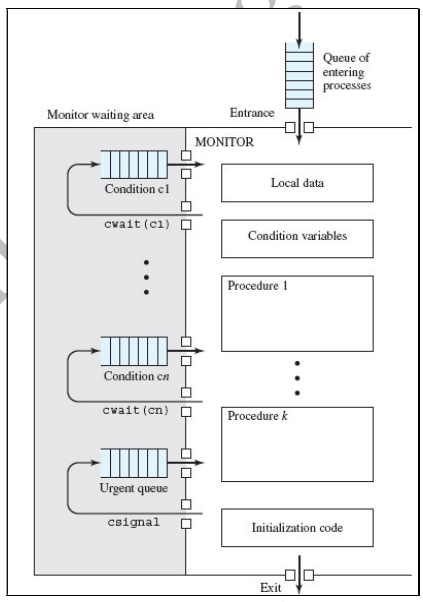


Figure: structure of monitor

Q. what are the aspects of monitor type?

Ans. The four aspects of monitor type are: **Data declaration, Data initialization, Operations on shared data and the synchronization operations**. They can be detailed in brief as follows.

a. Data declaration: shared data and condition variables are declared in this section. Copies of this data exist in every object of a monitor type.
b. Data initialization: data are initialized when a monitor i.e. an object type is created.
c. Operations on shared data: operations on shared data are coded as procedures of the monitor type. The monitor insures that these operations are executed in a mutually exclusive manner.
d. Synchronization operation: procedures of the monitor type use the synchronization operations **wait** and **signal** over condition variables to synchronize execution of processes.

C. Message passing

Q. How does the message passing works?
The message passing is normally provided in the form of a pair of primitives:
send (destination, message)
receive (source, message)
This is the minimum set of operations needed for processes to engage in message passing. A process sends information in the form of a message to another process designated by a destination. A process receives information by executing the receive primitive, indicating the source and the message.

Q. What are the design characteristics of message passing systems for Interprocess Communication?
The main issues in designing IPC using message passing are:

1. Synchronization on sending and receiving side
2. Sender and receiver addressing
3. The actual message format
4. The queuing discipline for handling the waiting queues

Q. Write a program for mutual exclusion using messages.

```
/* program MutualExclusion */
const int n = /* number of process */
void P(int i)
{
     message msg;
     while (true) {
                receive (box, msg);
                     /* critical section */;
                     send (box, msg);
                /* remainder */;
          }
}

void main()
{
          create mailbox (box);
          send (box, null);
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure: Definition primitive for mutual exclusion using message passing

Q. what is general message format in IPC using message passing?
The message format actually depends on the messaging facility and if the facility is running on standalone computer or on a distributed system. But in broader sense, the message format can be depicted as

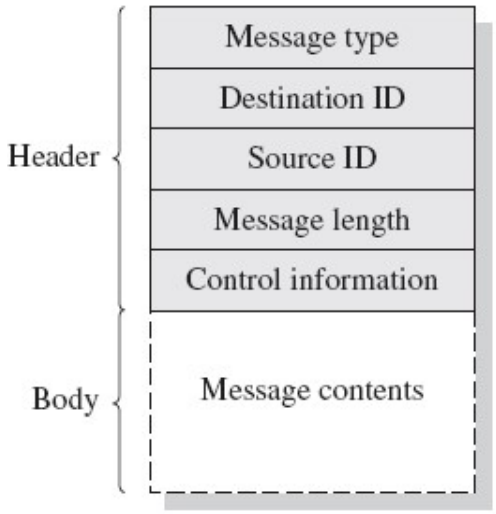


Figure: Message Format in IPC using message passing

The message is typically divided into two parts viz a header and a body. The header contains message information and the body contains the actual message.

Q. What are the advantages of having message passing as the solution to mutual exclusion?

Message passing achieves synchronization and communication to enforce mutual exclusion. Also, it can extend itself to implementation in distributed systems as well as in shared-memory multiprocessor and uniprocessor systems.

Q. What are the advantages of using mailboxes in message passing systems? Give example scenarios those may use mailboxes for IPC.

The mailboxes are used in indirect addressing mode of message passing.
The advantages of this scheme are:

1. The decoupling sender and receiver bring in greater flexibility in use of messages.
2. The sender-receiver relationship can be extended to one-to-many, many-to-one and many-to-many from the simple one-to-one interaction.
3. The sender and receiver processes can be geographically wide apart and they need not be running at all instances of times.

6. Concurrency Mechanisms in different OS

Q. Unix concurrency mechanisms.

Ans

Unix uses different concurrency mechanisms as: Pipes, Messages, shared memory, semaphores and signals.

Pipes: Pipes are the circular buffers those allow two processes to communicate. Pipes are generally written by one process and read by another, thus they act as shared buffer for reader and writer processes. Operating system enforced mutual exclusion on pipes for reading and writing processes. As the shared buffer is not infinite, the write requests are executed iff there is any room for more data to accommodate, otherwise the process gets blocked. Similarly read request is blocked in case it wants to read more bytes than available in the pipe.

For this, unix uses two types of pipes called as **Named pipes** and **unnamed pipes**.

Messages: Messages are text blocks with accompanying type which are exchanged between processes who wish to communicate with each other. The receiver process can either retrieve messages in FIFO order or by message type. A process may get suspended if it tries to send a message to a full message queue. Also the process reading an empty queue gets suspended. On the contrary if a process tries to read message of a certain type which is not currently available in message queue does not get suspended or blocked but fails.

UNIX provides **msgsnd()** and **msgrcv()** system calls for processes to engage in message passing. Every process has an associated message queue, which functions like a mailbox.

Shared memory: Apart from pipes and messages, shared virtual memory blocks can be used by multiple processes for the communication. Amongst all, this is the fastest form of interprocess communication. Here, the mutual exclusion is required to be provided by the processes themselves and not by the OS.

Semaphores: semaphores are the special variables those can be initialized and operated upon by the operations wait() and signal(). Operating systems handle the given semaphore operations.

Unix SVR4 uses a generalization of the semWait() and semSignal() primitives. The semaphores have associated queues of processes blocked on that semaphore. A semaphore used here consists of the following elements:

- Current value of the semaphore
- Process ID of the last process to operate on the semaphore
- Number of processes waiting for the semaphore value to be greater than its current value
- Number of processes waiting for the semaphore value to be zero

Associated with the semaphore are queues of processes suspended on that semaphore.

Signals: signals are software mechanism that inform a process of the occurrence of asynchronous events.

Here a signal is delivered by updating a field in the process table for the process to which the signal is being sent.

Q. comment on synchronization in Solaris.

Ans. To achieve synchronization, Solaris uses adaptive mutexes, condition variables, semaphores, reader-writer locks and turnstiles.

The adaptive mutex controls access to every critical resource in mutually exclusive manner. On multiprocessor systems, It works as a standard semaphore that runs as a spinlock, i.e. if the data are locked and thus inaccessible, the adaptive mutex follows one of the two solutions as,

1. If a lock is held by some process that is running on another processor, the thread spins because the lock is likely to get released soon.
2. If the thread is not running, the thread blocks and sleeps to get awakened by the lock release event. i.e. it will not spin while waiting as the lock may not get freed immediately.

The adaptive mutexes will be held if a lock will be held for less than a few hundred instructions. For the longer code segments condition variables are used. Readers-Writers lock are used if the data are accessed very frequently but they are used in read-only fashion. As they can allow multiple data reading threads concurrently, this readers-writers locks prove better than semaphores.

Solaris uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or a reader-writer lock. Turnstile is actually a queue structure containing threads blocked on a lock.

The locking technique used by kernel is implemented for user level threads, too. Thus provides same types of locks both inside and outside kernel.

Q. what are Windows NT concurrency mechanisms?

Windows NT uses synchronization objects as

1. Process
2. Thread
3. File
4. Console input
5. file change notification
6. mutex
7. semaphore
8. event
9. waitable timer

Solutions to classical problems

Q. What is Producer-consumer problem? What are the properties of a good solution to producer-consumer problem?
Ans. The Producer-consumer problem can be stated as: "There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer."

This problem has multiple solutions and various problem relaxations as well. For example, this problem can be viewed as with infinite buffer in the simple case and later on it can also have the solution with bounded buffers, too.
A solution to the Producer-consumer problem must satisfy following conditions:

1. A producer must not overwrite a full buffer.
2. A consumer must not consume an empty buffer.
3. Producers and consumers must access buffer in mutually exclusive manner.
4. Information must be consumed in the same sequence in which it is put into the buffer, i.e. FIFO order.

Q. What is readers-writers problem? What are the correctness conditions of readers-writers problem solution?
Ans. The readers/writers problem is defined as follows: There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

The correctness conditions those can be imposed on a readers-writers problem solution are:

1. Many readers can read the data concurrently.
2. No reader-writer combination is allowed at a time.
3. Only one writer can write at a time.
4. A reader has a non-preemptive priority over writers. i.e it gets access to shared data ahead of a waiting writer, though it does not preempt an active writer.

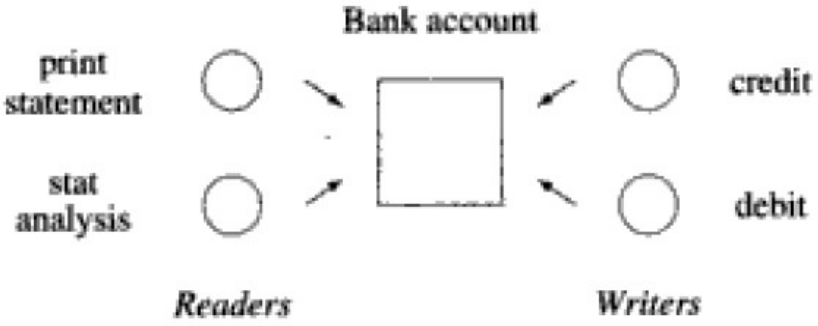
Q. What is Barbershop problem?
Ans. The Barbershop problem is defined as: A barbershop has three chairs, three barbers and a waiting area that can accommodate four customers on a sofa and a standing room for additional customers. Fire codes limit the total number

of customers in the shop to be 20. Any new customer entering shop will check the max number of customers allowed in shop. Then he checks first sofa and then barber chair if free. The barber goes to sleep if there is no customer in the chair and the customer leaves the shop when barber notifies him that the work is done. The problem is to design a solution in which all the customers coordinate with each other irrespective of where they are-in barber chair, sofa, standing area, entering or exiting the shop, the barber should not cut in air when no customer in chair, and the customer need not wait for longer time when he's already in chair and maximum of the users should be serviced and the solution should also respect mutual exclusion and should avoid deadlock, livelock and starvation.

Q. Give examples of Readers-Writers and Producer-Consumer problem.

1. **Ans.** A print service is a good example of producer-consumer concept in OS domain. A print daemon is a consumer process. A fixed sized queue of print requests is the bounded buffer. A process that adds a print request to the queue is a producer process.

2. **Ans.** For example, the readers and writers may share a bank account. The reader processes print statement and stat analysis also reads data from the same bank accounts so they can run concurrently. While the debit and credit functions modify the balance in account so obviously they can be termed as writers and only one of them should be active at any instance of time when they want to modify the data.



Q. Give the solution for Readers-Writers problem using semaphores. (readers have priority)

```
/* program Readers-Writers */
int ReadCount;
semaphore x = 1,wsem = 1;
void ProcessReader()
{
        while (true){
                wait (x);
                readcount++;

                if(readcount == 1)
                        wait (wsem);
                signal (x);
                READUNIT();
                wait (x);
                readcount;

                if(readcount == 0)
                        signal (wsem);
                signal (x);
        }
}

void ProcessWriter()
{
        while (true){
                 wait (wsem);
                WRITEUNIT();
                signal (wsem);
        }
}

void main()
{
        readcount = 0;
        parbegin (reader,writer);
}
```

This solution works for one or more readers and writers. The semaphore *wsem* enforces mutual exclusion. Thus when one writer is writing, no other writer or any reader is allowed to access the data area. The code given above obeys all the correctness characteristics discussed above. The global variable *ReadCount* is used to keep track of the number of readers, and the semaphore x is used to assure that *ReadCount* is updated properly.

Q. Write a solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores.

Ans.

```
/* program BoundedBuffer */

const int BufferSize = /* buffer size */;
semaphore s = 1, n = 0, e = BufferSize;

void producer()
{
        while (true) {
                produce();
                wait(e);
                wait(s);
                append();
                signal(s);
                signal(n);
        }
}

void consumer()
{
        while (true) {
                wait(n);
                wait(s);
                take();
                signal(s);
                signal(e);
                consume();
        }
}

void main()
{
        parbegin (producer, consumer);
}
```


The above program gives the perfect solution to the producer-consumer problem with bounded buffer using semaphores. As with the definition, wait() and signal() functions protect the CS by decrementing and incrementing the semaphore values. The semaphore 's' is a binary semaphore that takes care of either produce or a consumer being active at a time in the buffer while 'e' and 'SizeBuffer' are the counting semaphores which see to that neither producer nor consumer gets involved in busy waiting.

Q. Give Solution to sleeping Barber's problem using semaphores.

In computer science, the **sleeping barber problem** is a classic inter-process communication and synchronization problem between multiple operating system processes. The problem is analogous to that of keeping a barber working when there are customers, resting when there are none and doing so in an orderly manner.

A *multiple sleeping barbers problem* has the additional complexity of coordinating several barbers among the waiting customers.

```
# These three are mutexes (only 0 or 1 possible)
Semaphore custReady = 0          # if 1, at least one customer
is ready
Semaphore barberReady = 0
Semaphore accessWRSeats = 1
int numberOfFreeWRSeats = N

def Barber():
  while true:                    # Run in an infinite loop.
{
   # Try to acquire a customer - if none is available, go to
sleep.
   wait(custReady);
# Awake - try to get access to modify # of available seats,
otherwise sleep.
   wait(accessWRSeats);
    numberOfFreeWRSeats++ ;      # One waiting room chair
becomes free.
    signal(barberReady);         # I am ready to cut.
    signal(accessWRSeats);    # Don't need  lock on the chairs
anymore
    # (Cut hair here.)
}

def Customer():

{
while true:                  # Run in an infinite loop.
# Try to get access to the waiting room chairs.
    wait(accessWRSeats);
   if numberOfFreeWRSeats > 0: # If there are any free seats:
      numberOfFreeWRSeats--      #   sit down in a chair
      signal(custReady)          #notify the barber waiting for
customer
      signal(accessWRSeats)      # don't need to lock the
chairs anymore
      wait(barberReady)          #  wait until the barber is
ready
      # (Have hair cut here.)
    else:                        # otherwise, there are no free
seats;
    tough luck --
      signal(accessWRSeats)      # don't forget to release the
seat lock
    # (Leave without a haircut.)

}
```

Q. Give a solution to Reader-Writers problem using Monitors.

Monitors can be used to restrict access to the database. In this example, the read and write functions used by processes which access the database are in a monitor called *ReadersWriters*. If a process wants to write to the database, it must call the *writeDatabase* function. If a process wants to read from the database, it must call the *readDatabase* function.

Here the monitor is using the primitives **Wait** and **Signal** to put processes to sleep and to wake them up again. In *writeDatabase*, the calling process will be put to sleep if the number of reading processes, stored in the variable *count*, is not zero. Upon exiting the *readDatabase* function, reading processes check to see if they should wake up a sleeping writing process.

```
monitor ReadersWriters
  condition OKtoWrite, OKtoRead;
  int ReaderCount = 0;
  Boolean busy = false;


procedure StartRead()
{
  if (busy)             // if database is not free, block
     OKtoRead.wait;
  ReaderCount++;       // increment reader  ReaderCount
  OKtoRead.signal();

}

procedure EndRead()
{
   ReaderCount-- ;      // decrement reader ReaderCount
   if ( ReaderCount == 0 )
      OKtoWrite.signal();
}


procedure StartWrite()
{
  if ( busy || ReaderCount != 0 )
     OKtoWrite.wait();
  busy = true;
}
procedure EndWrite()
{
  busy = false;
  If (OKtoRead.Queue)
    OKtoRead.signal();
  else
    OKtoWrite.signal();
 }

Reader()
{
```

```
     while (TRUE)         // loop forever
     {
        ReadersWriters.StartRead();
       readDatabase();    // call readDatabase function in
monitor
        ReadersWriters.EndRead();
     }
  }

  Writer()
  {
     while (TRUE)         // loop forever
     {
       make_data(&info);          // create data to write
        ReaderWriters.StartWrite();
       writeDatabase();   //call writeDatabase monitor function
        ReadersWriters.EndWrite();
     }
  }
```

Q. Give solution to Producer-Consumer Problem using monitors

Monitors make solving the producer-consumer a little easier. Mutual exclusion is achieved by placing the critical section of a program inside a monitor. In the code below, the critical sections of the producer and consumer are inside the monitor *ProducerConsumer*. Once inside the monitor, a process is blocked by the **Wait** and **Signal** primitives if it cannot continue.

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

  procedure enter();
  {
    if (count == N) wait(full);         //block  if buffer
full,
    put_item(widget);                    // put item in buffer
    count = count + 1;              // increment count of full
slots
    if (count == 1)
      signal(empty);                  //awake consumer
  }

  procedure remove();
  {
    if (count == 0)  wait(empty);       // block if buffer
empty
    remove_item(widget);                // remove item from
buffer
    count = count - 1;                 // decrement count of full
slots

    if (count == N-1)
    signal(full);   // wakeup producer

  }
```

```
  count = 0;
  end monitor;

  Producer();
  {
    while (TRUE)
   {
      make_item(widget);            // make a new item
      ProducerConsumer.enter;       // call monitor function
enter
   }
  }

  Consumer();
  {
    while (TRUE)
    {
      ProducerConsumer.remove;    //call monitor function
remove
      consume_item;               // consume an item
   }
  }
```

Q. Give solution to the Bounded-Buffer Producer/Consumer Problem Using Message Passing.

Ans:

```
const int capacity = /* buffering capacity */ ;
null = /* empty message */ ;
int i;
void producer()
{ message pmsg;
while (true) {
receive (mayproduce,pmsg);
pmsg = produce();
send (mayconsume,pmsg);
}
}
void consumer()
{ message cmsg;
while (true) {
receive (mayconsume,cmsg);
consume (cmsg);
send (mayproduce,null);
}
}


void main()
{
create_mailbox (mayproduce);
create_mailbox (mayconsume);
for (int i = 1;i <= capacity;i++) send (mayproduce,null);
parbegin (producer,consumer);
```

```
}
```

8. Multiple Choice Questions

Q. what does IPC achieves……
1. Coordination between computations spread over several processes.
2. Mutual exclusion for the processes competing for resources
3. Command interpretation for the user
4. Deadlock avoidance
Ans. 1

Q. A process is said to be starved
1 if it is permanently waiting for a resource
2 if semaphores are not used
3 if a queue is not used for scheduling
4 if demand paging is not properly implemented
Ans) 1

Q. Situations where two or more processes are reading or writing some shared data and the final results depend on the order of usage of the shared data, are called _____.
1 Race conditions
2 Critical section
3 Mutual exclusion
4 Dead locks
Ans) 1

Q. The solution to Critical Section Problem is : Mutual Exclusion, Progress and Bounded Waiting.
1 The statement is false
2 The statement is true.
3 The statement is contradictory.
4 None of the above
Ans) 2

Q. A critical region is defined as
1 is a piece of code which only one process executes at a time
2 is a region prone to deadlock
3 is a piece of code which only a finite number of processes execute
4 is found only in Windows NT operation system
Ans) 1

Q. _____ is a high level abstraction over Semaphore.
1 Shared memory
2 Message passing
3 Monitor
4 Mutual exclusion
Ans) 3

Q. A binary semaphore
1 has the values one or zero

2 is essential to binary computers
3 is used only for synchronization
4 is used only for mutual exclusion
Ans) 1

Q. The Purpose of Co-operating Process is _____.
1 Information Sharing
2 Convenience
3 Computation Speed-Up
4 All of the above
Ans) 4

Q. Mutual exclusion
1 if one process is in a critical region others are excluded
2 prevents deadlock
3 requires semaphores to implement
4 is found only in the Windows NT operating system
Ans) 1

Q. The section of code which accesses shared variables is called as _____.
1 Critical section
2 Block
3 Procedure
4 Semaphore
Ans) 1

Q. Semaphore can be used for solving _____.
1 Wait & signal
2 Deadlock
3 Synchronization
4 Priority
Ans) 3

Q. A binary semaphore
1 has the values one or zero
2 is essential to binary computers
3 is used only for synchronisation
4 is used only for mutual exclusion
Ans) 1

Q. Inter process communication can be done through _____.
1 Mails
2 Messages
3 System calls
4 Traps
Ans) 2

Q. Mutual exclusion
1 if one process is in a critical region others are excluded

2 prevents deadlock
3 requires semaphores to implement
4 is found only in the Windows NT operating system
Ans) 1

Q. The section of code which accesses shared variables is called as _____.
1 Critical section
2 Block
3 Procedure
4 Semaphore
Ans) 1

Q. a strong semaphore is
1 one that follows FIFO policy for unblocking the processes waiting for this semaphore
2 one that cannot ever be preempted
3 one that is initialized to some higher value
4 one with the highest priority
Ans) 1

Q. a weak semaphore
1 one that follows no policy for unblocking the processes waiting for this semaphore
2 one that always gets preempted
3 one that is initialized to either 0 or 1
4 one with the lowest priority
Ans) 1

Deadlock

- **Basic Terminology & Definitions**
  Deadlock, livelock, starvation, resource allocation graph
- **Conditions To Deadlock, Approaches To Deadlock**
  Mutual exclusion, hold and wait, no preemption, circular wait, deadlock prevention, avoidance, detection and recovery, Ostrich Algorithm
- **Deadlock Prevention**
  Solutions and demerits of the solutions for the conditions Mutual exclusion, hold and wait, no preemption, circular wait
- **Deadlock Avoidance**
  Process initiation denial, resource allocation denial, banker's algorithm, safe-unsafe state
- **Deadlock Detection And Recovery**
  Detection algorithm, resource category, solutions to every category, integrated deadlock strategy
- **Dining philosophers problem**
- **Solved problems**
  Problems on process initiation denial, resource allocation denial, detection algorithm, determine if the states are safe or unsafe.
- **Multiple Choice Questions**

Basic concepts and Terminologies

Q. Define Deadlock, resource, system state, safe state, unsafe state, consumable nonconsumable resources, livelock,
Ans.
Deadlock: Deadlock is defined as a situation wherein a set of two or more processes are waiting for the resources which are currently held by other members, which in turn are waiting for the resources in possession of the some other processes in the same set, thus none of them can proceed further.

In other words, the deadlock occurs if two processes need the same two resources to continue and each has ownership of one. Unless some action is taken, each process will wait indefinitely for the missing resource. The deadlock leads to permanent blockage of all the deadlocked processes.

Ans. Resource: A **resource**, or **system resource**, is any physical or virtual component of limited availability within a computer system. Every device connected to a computer system is a resource.

The basic resources of computing machine are: CPU, memory, files and I/O devices. Every internal system component is a resource. Virtual system resources include files, network connections and memory areas.

Ans. System State: The state of the system at any point of time is defined as the allocation of resources to processes at that particular instance.

Ans. Safe State: the system state in that does not lead to deadlock is termed as safe state/

Ans. Unsafe state: The system state that leads to deadlock is called unsafe state.

Ans. Reusable and consumable resources:

non-consumable resources: resources that are made available again after use.

Examples of reusable resources are:

- processors,
- I/O channels,
- Main and secondary memory, disk space
- Devices, and
- Data structures such as files, databases, and semaphores.
- Glasses, space in street!

Consumable resources: one task creates the resource, another can use it once only.

Examples of consumable resources are:

- Interrupts,
- Signals,
- Messages, and
- Information in I/O buffers.
- Spoken words

The resources in any computing system are broadly classified into two categories as, reusable and consumable resources. The consumable resources are created and destroyed. Also there is no limit on consumable resources of a particular type.

Ans. live lock : Live lock can be defined as a condition in which one or more processes continuously change their state in response to changes in the other process(es) without doing any useful work. This is similar to deadlock in that no progress is made but differs in that neither process is blocked nor waiting for anything.

The system can come out of livelock if their relative pace of execution is changed while in execution.

Q. List three examples of deadlocks that are not related to a computer system environment.

Ans. Deadlock is not only associated with computer systems, but they are possible with general life scenarios, too. Some such examples are listed below.

- A scenario when two cars are crossing a single-lane bridge from opposite directions.
- A scenario where a person is going down a ladder while another person is climbing up the ladder at the same time.
- A scenario when two trains are traveling toward each other on the same track.

2. Process resource representation methodologies

Q. Comment on the resource state modeling methods.

Ans. Operating systems needs to analyze the process-resource states to determine if it is in deadlocked state or not. Two kinds of models are used to represent the resource allocation state of system as : **Graph model** and **Matrix model**.

Graph model:

- This model can depict the allocation state of a restricted class of system in which any process can request and use exactly one resource unit of each resource class.
- Uses a simple graph algorithm to determine if the circular wait condition is attained by the processes.

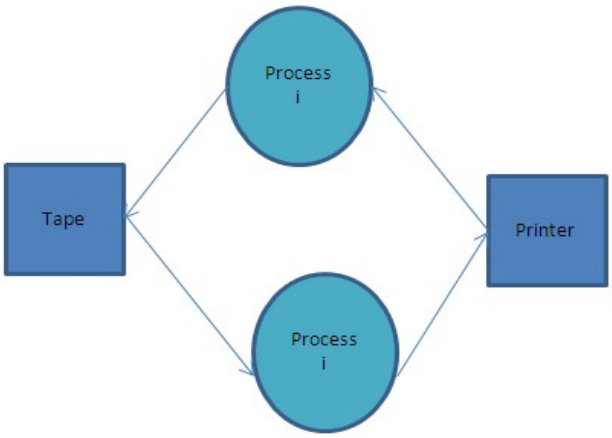


Figure: Resource allocation graph for two process and two resources.

Matrix model:

- This model has the advantage of generality.
- It can model allocation state in a system that permits a process to request and acquire any number of units of resource class and instances.

	R1	R2	…	…	Rm
P1	A_{11}	A_{12}	…	…	A_{1m}
P2	A_{21}	A_{22}	…	…	A_{2m}
…	…	…	…	…	…
…	…	…	…	…	…
Pn	A_{n1}	A_{n2}	…	…	A_{nm}

Allocation Matrix of n resources for m types of resource classes

Q. Explain the concept of resource allocation graphs in detail.
Ans. Resource allocation graphs
The resource allocation to processes is the root cause for the deadlocks to occur. So, the deadlock management requires some tools to depict the resource allocation scenario at any instance. Resource allocation graphs characterize the exactly the same thing.

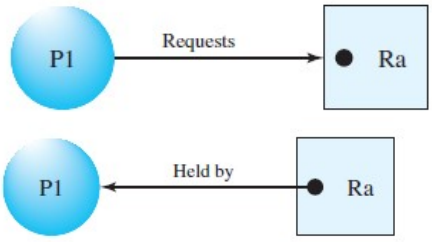
The resource allocation graph is basically a digraph in which each process and resource is represented by a node. Within a resource node, a dot is shown for each instance of that resource. All the directed edges from process to resource indicate the resources that have been requested but not yet granted. A graph edge directed from a reusable resource node dot to a process indicates a request that has been granted; that is, the process has been assigned one unit of that resource. A graph edge directed from a consumable resource node dot to a process indicates that the process is the producer of that resource.

Resource Graph Model:
The resource allocation graphs(RAG) contain two kinds of nodes: circles and squares. Circles represent processes and squares are used to model resources.

Process

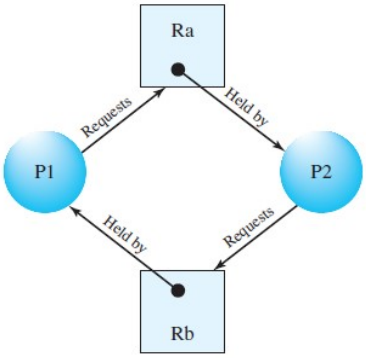
Resources

The RAGs represent two kinds of actions:
a. Process needs a resource
b. Resource is allocated to a process

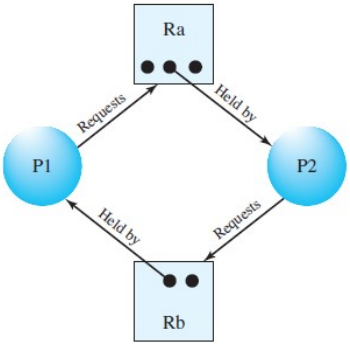


a. A requested resource

b. An allocated resource



c. Circular wait



d. No deadlock

Why the system resources and their allotment to processes are mostly represented with matrix rather than with resource allocation graphs?
Ans. The system has multiple resources. Also there is no control over number of processes those can exist in system. These processes request multiple copies of each resource which becomes difficult to model with resource allocation graph. So, the matrix representation wins over the same.

	R1	R2	…	…	Rm
P1	C_{11}	C_{12}	…	…	C_{1m}
P2	C_{21}	C_{22}	…	…	C_{2m}
…	…	…	…	…	…
…	…	…	…	…	…
Pn	C_{n1}	C_{n2}	…	…	C_{nm}

Figure: Resource Claimed Matrix for n Processes and m Types Of Resources

	R1	R2	…	…	Rm
P1	A_{11}	A_{12}	…	…	A_{1m}
P2	A_{21}	A_{22}	…	…	A_{2m}
…	…	…	…	…	…
…	…	…	…	…	…
Pn	A_{n1}	A_{n2}	…	…	A_{nm}

Figure: Resource Allocation Matrix for n Processes and m Types Of Resources

3. Conditions to Deadlock, Approaches to Deadlock

Q. State the conditions for deadlock.

A deadlock has four potential reasons to occur. Out of them, the first three do not guarantee a deadlock, but when any one of them is combined with the fourth one, the deadlock occurs for sure. These four conditions can be listed as: Mutual Exclusion, Hold and wait, No preemption and Circular wait.

- Mutual exclusion:
- Hold and wait
- No preemption
- Circular wait

Q. What are the strategies to handle deadlock?
Ans. The different strategies to handle deadlock are: deadlock prevention, deadlock avoidance and deadlock detection and recovery.
Deadlock prevention: This is name of the design policy of operating system to get rid of deadlocks. It suggests how resource requests are made and how they are serviced.

Deadlock prevention is constraing how resource request can be submitted and how they can be handled by system. The foremost goal is to ensure that deadlock occurring conditions do not hold.
Deadlock Avoidance: This concept allows checks the possibility of deadlock on dynamic basis and then decides if resource granting will be safe or not.

The deadlock avoidance considers every resource request in run time and decides if it should grant the resources. This concept also requires all the processes to submit their total resource requirements in advance. Deadlock avoidance method allows more concurrency.
Deadlock detection and recovery: The detection technique checks if the system is in deadlock. If the answer is a “yes”, then gives solution to recover from the same. This concept has nothing to do with resource granting. The recovery method depends on process and resource characteristics those have contributed to deadlock.

4. Deadlock prevention policy

Q. What is deadlock prevention policy of OS?
Ans. The deadlock prevention policy is to design system in such a way that excludes all the possibilities of deadlock occurrences. These prevention methods can be categorized into two categories: indirect method and direct method.
The indirect method tries to prevent the occurrence of one of the conditions i.e. mutual exclusion, hold and wait and no preemption. The direct method directly works on the circular wait to get prevented.

Mutual Exclusion prevention policy: Actually, this particular condition cannot be disallowed. If any resource requires mutual access then mutual exclusion must be supported by OS.

Hold and Wait prevention policy: The hold and wait condition can be prevented by requiring that a process should request all its required resources at once and blocking the process until all resources can be granted simultaneously. This approach takes much time for a process to get started and thus results in more response time, waiting time and obviously turnaround time. Also, this approach keeps most of the resources with the process for its lifetime and they may get underutilized.

No preemption prevention policy: This situation can be handled in many ways. One approach is, when a process holding certain resources and is denied further ones, then the process must release its original resources. This process when scheduled next time, can request for these resources afresh. Another approach is, when a process holding certain resources and requests for some more ones which are in possession with other process, the OS must preempt second process and make it to release all its possessions.

This method is useful only when it is applied to resources whose state can be easily saved and restored later on.

Circular wait prevention policy: In this case, all the resources in system are listed and are given some liner order numbers. If a process is allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.
i.e. Then resource Ri precedes Rj in the ordering if i < j. Now suppose that two processes, A and B, are deadlocked because A has acquired Ri and requested Rj, and B has acquired Rj and requested Ri. This condition would be impossible here because it implies i < j and j < i.

This approach to deal with circular-wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.

These prevention policies leads to inefficient use of resources and overall inefficient execution of processes.

5. Deadlock avoidance policy

Q. What is deadlock avoidance policy?
Ans: **Deadlock** avoidance allows the first three conditions but ensures that deadlock point is never checked. With avoidance, a decision is dynamically made whether the next resource allocation, if granted, can lead to deadlock. This decision is based on two policies as, **Process Initiation Denial** and **Resource Allocation Denial.**

A. **Process Initiation Denial: Do not start a process if its demands might lead to deadlock.**

- This approach enlists all system resources in resource vector R.
- The processes involved state all their resource requirements beforehand.
- With every resource allocation resource available vector A is computed, which initially equals the resource vector R.
- The process requirements if are less than or equal to as mentioned in available vector A, then the request is granted and process is initiated.
- If the process requirements are greater than those mentioned in available vector A, then the process is denied its initiation.

This strategy is hardly optimal, because it assumes the worst: all processes will make their maximum claims together.

B. **Resource allocation denial: Do not grant an incremental resource request to a process if it might lead to deadlock.**

This approach, also called as Banker's algorithm, always try to keep the system is safe state. The state concept is defined with reference to a system of fixed number of resources and a fixed number of processes as follows,
State: the State of the system depicts the current allocation of resources to the processes.
Safe state: Safe state is characterized as there exist at least one sequence of resource allocation that does not result in deadlock.
Unsafe state: the state which is not safe, i.e. which does not assure even one sequence that doesn't end up in deadlock.

Here, there is no restriction on process initiation. Instead, processes are allowed to get created. All the processes give their overall resource requirements and their requirements in initial state. At every resource allocation request, the OS evaluates a hypothetical situation 'what if the required resources are granted, the process in question completes and returns back the possessions, will there be at least one sequence in which all the processes in set can get executed?" i.e. if the resource granting will result in a safe state? If yes, the allocation is completed otherwise it gets denied.

Q. Explain the process denial policy of deadlock avoidance in detail.

Process Initiation Denial is one the deadlock prevention policies. It is based on the assumption that any process requesting its initiation should claim all of its resources at the same time.

To implement this policy, the system maintains two vectors and two matrices as,

Vector R = all resources indicating various resource types and their instances.

$R = (R_1, R_2, \ldots, R_m)$

Vector V = the types and instances of available resources at any instance of time.

$V = (V_1, V_2, \ldots, V_m)$

Matrix C = resources claimed by processes.(C_{ij} = requirement of process i for resource j, with one row dedicated to each process)

	R1	R2	…	…	Rm
P1	C_{11}	C_{12}	…	…	C_{1m}
P2	C_{21}	C_{22}	…	…	C_{2m}
…	…	…	…	…	…
…	…	…	…	…	…
Pn	C_{n1}	C_{n2}	…	…	C_{nm}

Claimed Matrix C

Matrix A= a matrix of resources allocated to processes. (A_{ij} = current allocation to process i of resource j)

	R1	R2	…	…	Rm
P1	A_{11}	A_{12}	…	…	A_{1m}
P2	A_{21}	A_{22}	…	…	A_{2m}
…	…	…	…	…	…
…	…	…	…	…	…
Pn	A_{n1}	A_{n2}	…	…	A_{nm}

Allocation Matrix A

All these above data structures are bound in following relationships,

1. All resources are either allocated or available.

$$R_j = V_j + \sum_{i=1 \text{ to } n} A_{ij} \text{ for all } j$$

2. No process can claim more than the total amount of resources in the system.

$$C_{ij} <= R_j \text{ for all } i, j$$

3. No process is allocated more resources of any type than the process originally claimed to need.

$$A_{ij} <= C_{ij} \text{ for all I,j}$$

When allowing any (n+1)th process to initiate while already n (n>=0) processes exist in system , the following condition is checked.

$$R_{ij} >= C_{(n+1)j} + \sum_{i=1 \text{ to } n} C_{ij} \text{ for all } j$$

That means, a process is initiated only if the maximum claims so far plus the new request can be met. This approach is hardly optimal as its based on the worst possible assumption: all processes will make their maximum claims together.

Q. Give an example of process initiation denial approach of deadlock management.

Ans.

Consider a system with given resource vector R=

6	5	9

Assume process P1,P2,P3,P4 join the system with some claims for resources in the same order as they are written.

C=

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

Initially the available vector is same as the resource vector.

So V=

7	5	9

Every process received is checked against the available vector one by one in their order of arrival.

.

Process	Request	Available resources Vk=Vk-Cik	Is Request < Available?	Process initiated?
P1	{4,3,3}	{7,5,9}	Yes	Yes
P2	{7,2,4}	{3,2,6}	No	No
P3	{2,2,5}	{3,2,6}	Yes	Yes
P4	{0,0,1}	{1,0,1}	Yes	Yes

Thus in the above example only P1,P3, and P4 are allowed to initiate while P2 is denied its initiation.

Q. Explain in detail the resource allocation denial policy of deadlock avoidance.

The deadlock possibility can be avoided by not awarding the resources whenever they may have potential of leading to deadlock. This strategy is most popularly known as **Banker's algorithm**. For a system of fixed number of resources and a fixed number of processes, the OS defines state as the current resource allocation to processes. This way, a safe state is considered the one in which there exists at least one sequence of process execution. Thus, any allocation leading to unsafe state is avoided.

Data structures used in computation:

Vector R = all resources indicating various resource types and their instances.

$R = (R_1, R_2, \ldots, R_m)$

Vector V = the types and instances of available resources at any instance of time.

$V = (V_1, V_2, \ldots, V_m)$

Matrix C = resources claimed by processes.(C_{ij} = requirement of process i for resource j, with one row dedicated to each process)

C=

C_{11}	C_{12}	…	…	C_{1m}
C_{21}	C_{22}	…	…	C_{2m}
…	…	…	…	…
…	…	…	…	…
C_{n1}	C_{n2}	…	…	C_{nm}

Matrix A= a matrix of resources allocated to processes. (A_{ij}= current allocation to process i of resource j)

A=

A_{11}	A_{12}	…	…	A_{1m}
A_{21}	A_{22}	…	…	A_{2m}
…	…	…	…	…
…	…	…	…	…
A_{n1}	A_{n2}	…	…	A_{nm}

Matrix M = a matrix of more resources required which is computed as,

M = C-A

Given all the above values the stafe status can be calculated as,

1. For any process i=1 to n, compare its associated row of matrix M with vector V.
2. If that Mij is less than or equal to Aj for all j, this process i can be assumed to be completed and thereby its resources can be given back to system.
   Thus, after hypothetical successful completion of process i the available vector A is updated as,
   V= V+ Aij where Aij indicates the resources j allocated earlier to process i.
   Also update Aij=0 for all js as the process holds no more resources.
3. Repeat step 1 and 2 until all processes are considered.

If all the processes in the set have the allocation matrix entries as 0s, the system is in safe sate, and sequence in which the process were selected can be termed as one of the possible execution sequence without any deadlock.

This approach seems simple and good, but it has certain preconditions X as,

1. All the resource requirements must be stated in advance.
2. The order in which processes get executed should not be constrained by synchronization or any that kind of mechanism.
3. The system must have fixed number of resources to allocate
4. The processes participating here are not allowed to exit holding the resources

Banker's Algorithm makes sure that only processes that will run to completion are scheduled to run. However, if there are deadlocked processes, the will remain deadlocked. **Banker's Algorithm does not eliminate a deadlock.**

Q. Explain the resource allocation denial concept with an example.
Ans. Resource allocation denial or Banker's algorithm
Example: consider a set of four processes p1,p2,p3,p4 and three resource types as R1,R2 and R3 with their values as,

Resource vector R(R1,R2,R3)=

13	5	9

Note: In the following matrices rows indicate process and columns represent the resources.

Claimed matrix C =

4	3	3
7	2	4
4	2	5
5	3	3

Allocation Matrix A =

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

More requirement M =

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

Available vector V (R1,R2,R3)=

0	0	1

Solution: From the given processes P1,P2,P3 and p4, only p2 is the process whose corresponding row in matrix M satisfies the criteria of being less than values in vector V.
So, **first P2 may run to completion.** Then the above tables will have values as,

1. After P2 runs to completion

4	3	3
0	0	0
4	2	5
5	3	3

Claimed Matrix C

2	0	0
0	0	0
3	2	2
1	1	3

Allocation Matrix A

2	2	2

0	0	0
1	0	3
4	2	0

More Requirements M

7	2	4

Available vector V

Now, seeing the entries in M and available vector V, **any process** can be awarded with the resources and they can run to completion.

1. After P1 runs to completion

0	0	0
0	0	0
4	2	5
5	3	3

Claimed Matrix C

0	0	0
0	0	0
3	2	2
1	1	3

Allocation Matrix

A

Available vector V=

9	2	4

More Requirements M=

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

2. After P3 runs to completion

0	0	0
0	0	0
0	0	0
5	3	3

Claimed Matrix C

0	0	0
0	0	0
0	0	0
1	1	3

Allocation Matrix

A

0	0	0
0	0	0
0	0	0
4	2	0

More Requirements M

Available vector V

12	4	6

3. After P4 runs to completion

0	0	0
0	0	0
0	0	0
0	0	0

0	0	0
0	0	0
0	0	0
0	0	0

Claimed Matrix C Allocation Matrix A

0	0	0
0	0	0
0	0	0
0	0	0

More Requirements M

Available vector V

13	5	9

As the resultant computation is giving the process execution sequence as **P2-P1-P3-P4**, the system is in **Safe state**.

Q. Give one example of potential unsafe state that can be avoided with deadlock avoidance (Banker's algorithm).
Ans.

consider a set of four processes p1,p2,p3,p4 and three resource types as R1,R2 and R3 with their values as,

Resource vector R(R1,R2,R3)=

13	5	9

Note: In the following matrices rows indicate process and columns represent the resources.

If process P2 from earlier example requests one more resource of type R3, and if it gets granted by system, then it enters in unsafe state.

Claimed matrix C =

4	3	3
7	2	**5**
4	2	5
5	3	3

Allocation Matrix A =

2	0	0
7	2	**3**
3	2	2
1	1	3

More requirement M =

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

Available vector V (R1,R2, R3)=

0	0	1

In the above scenario, none of the row the More Requirement Matrix M has requirement that can be satisfied by available vector V. so, **this is an unsafe state.** To overcome this, the requests given in claimed matrix should be served in such a way that there exists one possible sequence of execution.

6. Deadlock Detection And Recovery

Q. How does deadlock detection and recovery work?

This is the most liberal policy amongst all. This approach does not limit resource access or restricts process actions. Instead, the requested resources are granted whenever possible and OS performs periodic or aperiodic checks to detect if deadlock has occurred and it is recovered accordingly.

The frequency of deadlock detection is one of the mentioned below:

- On some regular interval
- On each resource request

The first policy reduces OS overhead but may lead the system into a serious deadlock state while the later one results in early detection.

One the deadlock is detected, every kind of resource that has caused the deadlock, has a different deadlock recovery strategy.

Q. Explain in detail the deadlock detection algorithm.

Ans.

This particular approach has the least restrictions on the processes and resources. It lets the processes claim and get resources, all resources need not be claimed in beginning and system runs checks if system is in deadlock state and recovers from the same accordingly.

The detection and recovery algorithm:

Inputs:

1. Resource Vector R
2. Available vector V (optional)
3. Claimed matrix C
4. Allocation matrix A

Steps:

1. Compute the available vector V if not given.
2. Compute the more requirement matrix M= C-A
3. Mark each process that has a row in the Allocation matrix of all zeros.
4. Find an index I such that process i is not marked and its corresponding ith row of M is less than or equal to Available vector V.i.e., Mik <= Vk for all 1<=k<=m where m is total number of resources. If no such row is found, terminate the algorithm.
5. If such a row is found, mark process I and add the corresponding row of the allocation matrix to available vector V. i.e. Vk=Vk+Aik for all 1<=k<=m. return to step 4.

A deadlock is detected if there are any unmarked processes at the end of the algorithm. It says that every unmarked process is deadlocked. this algorithm does not guarantee to prevent deadlock; that will depend on the order in which future requests are granted. All that it does is it determines if deadlock currently exists accordingly calls the recovery procedure.

Q. What are the general recovery policies of system?
Ans: one of the following options can be taken whenever system detects that it is in a deadlock state.

1. Abort all deadlocked processes (one of the most common solution adopted in OS!!)
2. Rollback each deadlocked process to some previously defined checkpoint and restart them (original deadlock may reoccur)
3. Successively abort deadlock processes until deadlock no longer exists (each time we need to invoke the deadlock detection algorithm)\
4. Successively preempt some resources from processes and give them to other processes until deadlock no longer exists (a process that has a resource preempted must be rolled back prior to its acquisition)

For the options offered in 3 &4, the criteria to choose a process is one or more of the following ones.

- least amount of CPU time consumed so far
- least total resources allocated so far
- least amount of "work" produced so far...

The following table gives example of three types of the resources device, file and memory being requested, acquired by a process and released by the same.

Request a resource	**Acquire/use a resource**	**Release a resource**
Request a device	Read from/write to a device	Release a device
Open a file	Read/write a file	Close a file
allocate memory	Use the memory	Free memory

Q. what is integrated deadlock strategy?

There are several ways to address the problem of deadlock in an operating system.

- Just ignore it and hope it doesn't happen
- Detection and recovery - if it happens, take action
- Dynamic avoidance by careful resource allocation. Check to see if a resource can be granted, and if granting it will cause deadlock, don't grant it.
- Prevention - change the rules

Actually, every deadlock handling strategy has some merits and demerits associated with them. So instead of employing any one the strategies, it's

better to follow different strategy in different situations and with different kinds of resources.

The general integrated deadlock strategy is:

1. Group the resources into number of different resource categories.
2. Use the linear ordering strategy defined for deadlock prevention of circular wait to prevent deadlocks between resource classes.
3. Within a resource class, use the algorithm that is most appropriate for that class.

Some example strategies within a resource class are as listed below,

1. Resource type: **Swappable space**
   Recovery strategy: **Deadlock prevention** as the one used with hold-and-wait prevention solution and **deadlock avoidance**
2. Resource type: **Process resources**
   Recovery strategy**: deadlock avoidance** with resource allocation denial solution type and **Deadlock prevention** by means of resource ordering within the resource class.
3. Resource type: **Main memory**
   Recovery strategy: **Deadlock prevention** solution used for no-preemption situation.
4. Resource type: **Internal resources**
   Recovery strategy: **Deadlock prevention** by means of resource ordering can be used.

7. Deadlocks in UNIX

Q. Comment on deadlocks in Unix.
Like any other OS, Unix also employs multiple policies to handle the deadlock issue. They can be enumerated as,

- Ostrich approach-It simply ignores the possibility of deadlocks involving user processes.
- Deadlock prevention through resource ordering strategy is used for the processes that are executing the kernel code as a result of interrupt or system calls. Data structures in kernel are locked and released in a standard order. However, not all kernel functionalities can lock the data structures so deadlocks are possible.
- One approach to deal with issue above is, the process issuing request avoids getting blocked on its lock. Instead it tries to get the next resource of the same type if possible. This strategy avoids deadlocks by avoiding circular waits, especially with buffer memory allocations.
- For the processes working with file systems, in case they need access to two different directories, both directory locks are not set at the same time. It first locks the first directory, updates it in desired manner and releases the lock and then accesses another directory and updates it as well. Thus it gets only one lock at a time thus avoiding hold and wait condition.

Dining philosophers problem

Q. Dining philosopher's problem
Ans. The dining philosophers problem is a ``classical'' synchronization problem. Taken at face value, it is a pretty meaningless problem, but it is typical of many synchronization problems that one can see when allocating resources in operating systems.
Problem definition: There are 5 philosophers sitting at a round table. Between each adjacent pair of philosophers is a fork. So they have only five fork. Each philosopher does two things: think and eat. The philosopher thinks for a while, and then stops thinking and becomes hungry. When the philosopher becomes hungry, she cannot eat until she owns the both the forks on left and right. When the philosopher is done eating she puts down the forks and begins thinking again.

The challenge in the dining philosophers problem is to design a protocol so that the philosophers do not deadlock (i.e. every philosopher has a fork), and so that no philosopher starves (i.e. when a philosopher is hungry, she eventually gets the chopsticks). Additionally, our protocol should try to be as efficient as possible -- in other words, one should try to minimize the time that philosophers spent waiting to eat.

- Solution 1: Resource ordering

This approach establishes the convention that all resources will be requested in order, and released in reverse order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources are the forks. They are numbered one through five. All the philosopher's have to pick up the lower numbered fork first and then the higher numbered one. Then, she will always put down the higher numbered fork first, followed by the lower numbered fork. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so he will be able to eat using two forks. When he finishes using the forks, he will put down the highest-numbered fork first, followed by the lower-numbered fork, freeing another philosopher to grab the latter and begin eating.

This solution to the problem is the one originally proposed by Dijkstra. And it is not always practical, especially when the list of required resources is not completely known in advance.

- Solution 2: monitor

  Philosophers can eat if neither of their neighbors is eating. This is comparable to a system where philosophers that cannot get the second fork must put down the first fork before they try again. (no hold and wait condition)

  This solution uses a single mutual exclusion lock. This lock is not associated with the forks but with the decision procedures that can change the states of the philosophers. This is ensured by the monitor. The procedures **test**, **pickup** and **putdown** are local to the monitor and share a mutual exclusion lock. The philosophers wanting to eat do not hold a fork. When the monitor allows a philosopher who wants to eat to continue, the philosopher will reacquire the first fork before picking up the now available second fork. When done eating, the philosopher will signal to the monitor that both forks are now available.

  To also guarantee that no philosopher starves, one could keep track of the number of times a hungry philosopher cannot eat when his neighbors put down their forks. If this number exceeds some limit, the state of the philosopher could change to Starving, and the decision procedure to pick up forks could be augmented to require that none of the neighbors are starving.

  A philosopher, who cannot pick up forks because a neighbor is starving, is effectively waiting for the neighbor's neighbor to finish eating. This additional dependency reduces concurrency. Raising the threshold for transition to the Starving state reduces this effect.

Multiple choice questions.

Q. Let S and Q be two semaphores initialized to 1, where P0 and P1 processes the following statements wait(S);wait(Q); ---; signal(S);signal(Q) and wait(Q); wait(S);---;signal(Q);signal(S); respectively. The above situation depicts a _____ .
1 Semaphore
2 Deadlock
3 Signal
4 Interrupt
Right Ans) 2

Q. _____ is the situation in which a process is waiting on another process, which is also waiting on another process , which is waiting on the first process. None of the processes involved in this circular wait are making progress.
1 Deadlock
2 Starvation
3 Dormant
4 None of the above
Right Ans) 1

Q. The Banker's algorithm is used
1 to prevent deadlock in operating systems
2 to detect deadlock in operating systems
3 to rectify a deadlocked state
4 none of the above
Right Ans) 1

Q. In one of the deadlock prevention methods, impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. This overcomes the _____ condition of deadlock
1 Mutual exclusion
2 Hold and Wait
3 Circular Wait
4 No Preemption
Right Ans) 3

Q. Resource locking _____.
1 Allows multiple tasks to simultaneously use resource
2 Forces only one task to use any resource at any time
3 Can easily cause a dead lock condition
4 Is not used for disk drives
Right Ans) 2

Q. A set of resources' allocations such that the system can allocate resources to each process in some order, and still avoid a deadlock is called _____.
1 Unsafe state

2 Safe state
3 Starvation
4 Greeedy allocation
Right Ans) 2

Q. A process said to be in _____ state if it was waiting for an event that will never occur.
1 Safe
2 Unsafe
3 Starvation
4 Dead lock
Ans) 4

Q. Consider a system with m resources of same type being shared by n processes. Resources can be requested and released by processes only on at a
time. The system is deadlock free if and only if
1. The sum of all max needs is < m+n
2. The sum of all max needs is > m+n
3. Both of above
4. None
Ans: 1

Q. Consider a system consisting of 4 resources of same type that are share by 3 processes each of which needs at most two resources. Show that the system is deadlock free
Ans:
If the system is deadlocked, it implies that each process is holding one resource and is waiting for one more. Since there are 3 processes and 4 resources, one process must be able to obtain two resources. This process requires no more resources and therefore it will return its resources when done.

Q. A system has four processes P1 through P4 and two resource types R1 and R2. It has 2 units of R1 and 3 units of R2.
Given that: P1 requests 2 units of R2 and 1 unit of R1, P2 holds 2 units of R1 and 1 unit of R2, P3 holds 1 unit of R2, P4 requests 1 unit of R1
Show the resource graph for this state of the system. Is the system in deadlock, and if so, which processes are involved?

	R1	R2
P1	1	2
P2	0	0
P3	0	0

Ans:

	R1	R2
P4	1	0

Given:

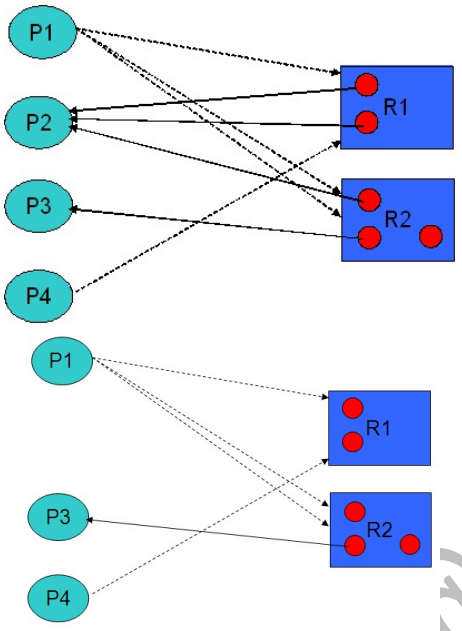
R1	R2
2	3

R

	R1	R2
P1	0	0
P2	1	2
P3	0	1
P4	0	0

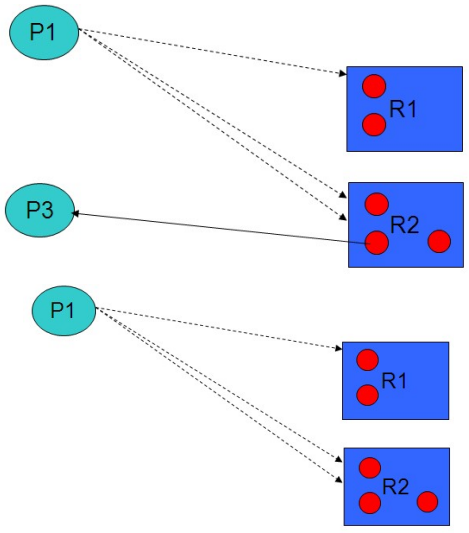
C

A



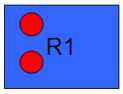
Step 1: Initial stage
(dashed lines indicate requests)

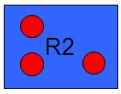
Step 2: Process P2 completes



Step 3: Process P4 completes

Step 4: Process P3 completes





Step 5: Process P1 completes

There are many sequences in which the processes could be realize, but process P2 must complete before process P1 as it hold the resources requested by former.

Q. Given 5 total units of the resource of the same type, tell whether the following system is in a safe or unsafe state.

R=5,

C=

P1	P2	P3	P4
2	3	4	5

A=

P1	P2	P3	P4
1	1	2	0

Ans. safe state

Q. Given a total of 5 units of resource type 1 and 4 units of resource type 2, tell whether the following system is in a safe or unsafe state. Show your work.

	R1	R2
P1	2	3
P2	3	2
P3	4	4

Claimed Matrix C

	R1	R2
P1	1	1
P2	1	1
P3	2	1

Allocation Matrix A

Ans: unsafe

Process	Allocation	Claimed	Available

Q. Given a total of 10 units of a resource type, and given the safe state shown below, should process 2 be granted a request of 2 additional resources? Show your work.

C=

P1	P2	P3	P4	P5
5	6	6	2	4

A=

P1	P2	P3	P4	P5
2	1	2	1	1

Ans. Unsafe state

Q.
Consider the snapshot of a system.

P0	{0,1,0}	{7,5,3}	{3,3,2}
P1	{2,0,0}	{3,2,2}	
P2	{3,0,2}	{9,0,2}	
P3	{2,1,1}	{2,2,2}	
P4	{0,0,2}	{4,3,3}	

Answer the following question using the banker's Algorithm
a. What is the content of matrix need?
b. Is the system in safe state?
c. If a request from process P1 arrives for (1,0,2) can the request be granted immediately?

Q. Consider a system with three processes and three types of resources with the following data.

	A	B	C
R=	3	2	1

Resource vector

	A	B	C
P1	2	1	1
P2	2	1	1
P3	1	2	0

Matrix

	A	B	C
P1	1	1	0
P2	2	0	1
P3	0	1	0

Allocation Matrix Claim

Run the deadlock detection algorithm on the above example and check is the system in a deadlock? If yes, then what are the process(es) that need to be pre-empted and in what order, to ensure that deadlock is overcome?