Q1 (a) List the major activities of an Operating System with respect to process management and memory management?

Answer: Major activities of an operating system in respect to Process management are as follows:

- The creation and deletion of both user and system processes.
- The suspension and resumption of processes.
- The provision of mechanisms for process synchronization.
- The provision of mechanisms for process communication.
- The provision of mechanisms for deadlock handling.

(b) Discuss the differences between user level threads and kernel level threads.

Answer: Difference between User Level threads and Kernel Level threads are as follows:

- User threads are supported above the kernel and are implemented by a thread library at the user level. Whereas, kernel threads are supported directly by the operating system.
- For user threads, the thread library provides support for thread creation, scheduling and management in user space with no support from the kernel, as the kernel is unaware of user-level threads. In case of kernel threads, the kernel performs thread creation, scheduling and management in kernel space.
- As there is no need of kernel intervention, user-level threads are generally fast to create and manage. As thread management is done by the operating system, kernel threads are generally slower to create and manage that is user threads.
- If the kernel is single-threaded, then any user-level thread performing blocking system call, will cause the entire process to block, even if other threads are available to run within the application. However, since the kernel is managing the kernel threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution.
- User-thread libraries include POSIX P threads, Mach C-threads and Solaris 2 UI-threads. Some of the cotemporary operating systems that support kernel threads are Windows NT, Windows 2000, Solaris 2, BeOS and Tru64 UNIX (formerly Digital UNIX).

(c) How process synchronization is achieved in Pthreads?

Answer:
The Pthreads API provides mutex locks, condition variables, and read-write locks for thread synchronization. This API is available for programmers and is not part of any particular kernel. Mutex locks represent the fundamental synchronization technique used with Pthreads. A mutex lock is used to protect critical sections of code—that is, a thread acquires the lock before entering
a critical section and releases it upon exiting the critical section. Many systems that implement Pthreads also provide semaphores, although they are not part of the Pthreads standard and instead belong to the POSIX SEM extension. Other extensions to the

Pthreads API include spinlocks, although not all extensions are considered portable from one implementation to another.

(d) When can the OS bind instructions and data to memory?

Answer:
Classically, the binding of instructions and data to memory addresses can be done at any step along the way:
• **Compile time.** If you know at compile time where the process will reside in memory, then **absolute code** can be generated. For example, if you know that a user process will reside starting at location *R*, then the generated compiler code will start at that location and extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code. The MS-DOS .COM-format programs are bound at compile time.
• Load **time.** If it is not known at compile time where the process will reside in memory, then the compiler must generate **relocatable code. In** this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.
• **Execution time.** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

(e) What are the different security measures to be taken to protect a computer system?

Answer:
To protect a system, we must take security measures at four levels:
1. Physical. The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders. Both the machine rooms and the terminals or workstations that have access to the machines must be secured.
2. Human. Authorizing users must be done carefully to assure that only appropriate users have access to the system. Even authorized users, however, may be "encouraged" to let others use their access (in exchange for a bribe, for example). They may also be tricked into allowing access via **social engineering.** One type of social-engineering attack is **phishing.** Here, a legitimate-looking e-mail or web page misleads a user into entering confidential information. Another technique is **dumpster diving,** a general term for attempting to gather information in order to gain unauthorized access to the computer (by looking through trash, finding phone books, or finding notes containing passwords, for example). These security problems are management and personnel issues, not problems pertaining to operating systems.
3. **Operating system.** The system must protect itself from accidental or purposeful security breaches. A runaway process could constitute an accidental denial-of-service attack. A query to a service could reveal passwords. A stack overflow could allow the launching of an unauthorized process. The list of possible breaches is almost endless.

4. Network. Much computer data in modern systems travels over private leased lines, shared lines like the Internet, wireless connections, or dial-up lines. Intercepting these data could be just as harmful as breaking into a computer; and interruption of communications could constitute a remote denial-of-service attack, diminishing users' use of and trust in the system.

(f) What are the reasons for using Process Migration in Distributed Operating Systems?

Answer:
A logical extension of computation migration is **process migration.** When a process is submitted for execution, it is not always executed at the site at which it is initiated. The entire process, or parts of it, may be executed at different sites. This scheme may be used for several reasons:
• **Load balancing.** The processes (or subprocesses) may be distributed across the network to even the workload.
• **Computation speedup.** If a single process can be divided into a number of subprocesses that can run concurrently on different sites, then the total process turnaround time can be reduced.
• **Hardware preference.** The process may have characteristics that make it more suitable for execution on some specialized processor (such as matrix inversion on an array processor, rather than on a microprocessor).
• **Software preference.** The process may require software that is available at only a particular site, and either the software cannot be moved, or it is less expensive to move the process.
• **Data access.** Just as in computation migration, if the data being used in the computation are numerous, it may be more efficient to have a process run remotely than to transfer all the data.

(g) Write the advantages and disadvantages of Single-Coordinator approach for Locking Protocol?

Answer:
The Single-Coordinator approach for locking protocol has the following advantages:
• **Simple implementation.** This scheme requires two messages for handling lock requests and one message for handling unlock requests.
• **Simple deadlock handling.** Since all lock and unlock requests are made at one site, the deadlock-handling algorithms can be applied directly to this environment.

The disadvantages of the Single-Coordinator approach include the following:
• **Bottleneck**. The site S, becomes a bottleneck, since all requests must be processed there.
• **Vulnerability.** If the site S, fails, the concurrency controller is lost. Either processing must stop or a recovery scheme must be used.

Q2 (a) What is a Process? Briefly discuss different process states?

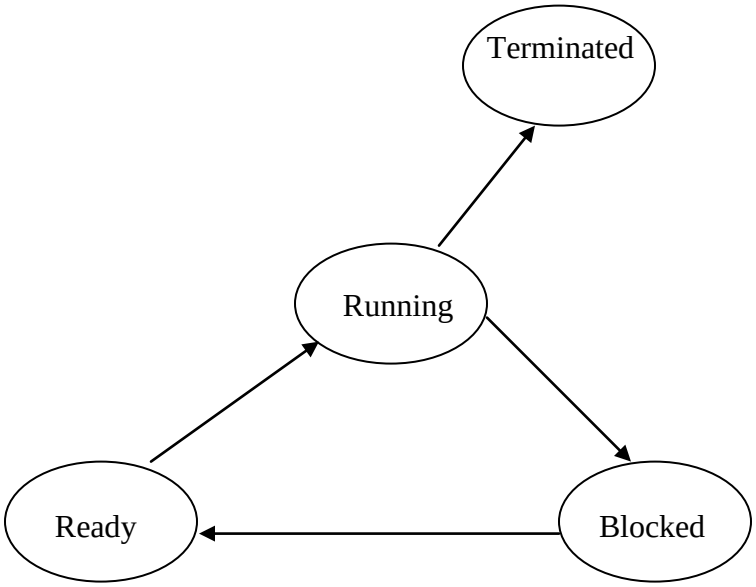
Answer:

A process or task is a portion of a program in some stage of execution. A program can consist of several processes, each working on their own or as a unit, perhaps communicating with each other. Each process that runs in an operating system is assigned a process control block that holds information about the process, such as a unique process ID (a number used to identify the process), the saved state of the process, the process priority and where it is located in memory.

A process in a computer system may be in one of the possible state as follows:

- Running: A CPU is currently allocated to the process and the process is in execution.
- Blocked: The process is waiting for a request to be satisfied, or an event to occur. Such a process cannot execute even if a CPU is available.
- Ready: The process is not running, however it can execute if a CPU is allocated to it, means the process is not blocked.
- Terminated: The process has finished its execution.

A state transition is caused by the occurrence of some event in the system. When a process in the running state makes an I/O request, it has to enter blocked state awaiting completion of the I/O. When the I/O completes, the process state changes from blocked to ready. Similar state changes occur when a process makes some request, which cannot be satisfied by OS straightway. The process state changes to blocked until the request is satisfied, when its state changes to ready once again. A ready process becomes running when the CPU allocated to it. The fundamental state transition is shown in figure below.



(b) What are System programs? How are they divided into different categories?

Answer:
System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex.
They can be divided into these categories:

- **File management.** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
- **Status information.** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.
- **File modification.** Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
- **Programming-language support.** Compilers, assemblers, debuggers and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system.
- **Program loading and execution.** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
- **Communications.** These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to log in remotely, or to transfer files from one machine to another.

(c) Define Deadlock? Discuss the four necessary conditions of Deadlock prevention?

Answer:
Deadlock is a situation, in which processes never finish executing and system resources are tied up, preventing other jobs form starting. A process requests resources; if the resources are not available at that time, the process enters a wait state. Waiting processes may never again change state, because other waiting processes, thereby causing deadlock, hold the resources they have requested.

The four necessary conditions for deadlocks to occur are mutual exclusion, hold and wait, no preemption and circular wait. If any one of the above four conditions does not hold, then deadlocks will not occur. Thus **prevention of deadlock** is possible by ensuring that at least one of the four conditions cannot hold.

- **Mutual exclusion**: Resources that can be shared are never involved in a deadlock because such resources can always be granted simultaneously access by processes. Hence processes requesting for such a sharable resource will never have to wait. Examples of such resources include read-only files. Mutual exclusion must therefore hold for non-sharable resources. But it is not always possible to prevent deadlocks by denying mutual exclusion condition because some resources are by nature non-sharable, for example printers.
- **Hold and wait**: To avoid hold and wait, the system must ensure that a process that requests for a resource does not hold on to another. There can be two approaches to this scheme:
    - a process requests for and gets allocated all the resources it uses before execution begins.
    - a process can request for a resource only when it does not hold on to any other.

  Algorithms based on these approaches have poor resource utilization. This is because resources get locked with processes much earlier than they are actually used and hence not available for others to use as in the first approach. The second approach seems to applicable only when there is assurance about reusability of data and code on the released resources. The algorithms also suffer from starvation since popular resources may never be freely available.
- **No preemption**: This condition states that resources allocated to processes cannot be preempted. To ensure that this condition does not hold, resources could be preempted. When a process requests for a resource, it is allocated the resource if it is available. If it is not, than a check is made to see if the process holding the wanted resource is also waiting for additional resources. If so the wanted resource is preempted from the waiting process and allotted to the requesting process. If both the above is not true that is the resource is neither available nor held by a waiting process, then the requesting process waits. During its waiting period, some of its resources could also be preempted in which case the process will be restarted only when all the new and the preempted resources are allocated to it.

  Another alternative approach could be as follows: If a process requests for a resource which is not available immediately, then all other resources it currently holds are preempted. The process restarts only when the new and the preempted resources are allocated to it as in the previous case.

  Resources can be preempted only if their current status can be saved so that processes could be restarted later by restoring the previous states. Example, CPU memory and main memory. But resources such as printers cannot be preempted, as their states cannot be saved for restoration later.
- **Circular wait**: Resource types need to be ordered and processes requesting for resources will do so in increasing order of enumeration. Each resource type is mapped to a unique integer that allows resources to be compared and to find out the precedence order for the resources. Thus F: R → N is a 1:1 function that maps resources to numbers. For example:

  F (tape drive) = 1, F (disk drive) = 5, F (printer) = 10.

  To ensure that deadlocks do not occur, each process can request for resources only in increasing order of these numbers. A process to start with in the very first

instance can request for any resource say R_i. There after it can request for a resource R_j if and only if $F(R_j)$ is greater than $F(R_i)$. Alternately, if $F(R_j)$ is less than $F(R_i)$, then R_j can be allocated to the process if and only if the process releases R_i.

The mapping function F should be so defined that resources get numbers in the usual order of usage.

Q3 (a) Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5 all at time 0.

(i) Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a nonpreemptive priority (a smaller number implies a higher priority), and RR (quantum = 1) scheduling.

(ii) What is the turnaround time and waiting time of each process for each of the scheduling algorithms in part (i)? Which of the schedules results in the minimal average waiting time (over all processes)?

Answer:

(i) The four Gantt charts are

1	2	3	4	5	FCFS

1	2	3	4	5	1	3	5	1	5	1	5	1	5	1	RR

2	4	3	5	1	SJF

2	5	1	3	4	Priority

(ii) Turnaround time

Process	FCFS	RR	SJF	Priority
P_1	10	19	19	16
P_2	11	2	1	1
P_3	13	7	4	18
P_4	14	4	2	19
P_5	19	14	9	6

(iii)Waiting time (turnaround time minus burst time)

Process	FCFS	RR	SJF	Priority
P_1	0	9	9	6
P_2	10	1	0	0
P_3	11	5	2	16
P_4	13	3	1	18
P_5	14	9	4	1

(iv) Shortest Job First

(b) What is Readers-Writers problem? Give a solution for Readers-Writers problem using conditional critical regions

Answer:

Readers-writers problem: Let a data object (such as a file or record) is to be shared among several concurrent processes. Readers are the processes that are interested in only reading the content of shared data object. Writers are the processes that may want to update (that is, to read and write) the shared data object. If two readers access the shared data object simultaneously, no adverse effects will result. However if a writer and some other process (either a reader or writer) access the shared object simultaneously, anomaly may arise. To ensure that these difficulties do not arise, writers are required to have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem.

Solution for readers-writers problem using conditional critical regions.

Conditional critical region is a high-level synchronization construct. We assume that a process consists of some local data, and a sequential program that can operate on the data. The local data can be accessed by only the sequential program that is encapsulated within same process. One process cannot directly access the local data of another process. Processes can, however, share global data.

Conditional critical region synchronization construct requires that a variable v of type T, which is to be shared among many processes, be declared as

v: shared T;

The variable v can be accessed only inside a region statement of the following form:

region v when B do S;

This construct means that, while statement S is being executed, no other process can access the variable v. When a process tries to enter the critical-section region, the Boolean expression B is evaluated. If the expression is true, statement S is executed. If it is false, the process releases the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v.

Now, let A is the shared data object. Let readcount is the variable that keeps track of how many processes are currently reading the object A. Let writecount is the variable that

keeps track of how many processes are currently writing the object A. Only one writer can update object A, at a given time.

Variables readcount and writecount are initialized to 0. A writer can update the shared object A when no reader is reading the object A.

```
region A when( readcount = = 0 AND writecount = = 0){
        ……
    writing is performed
        ……  }
```

A reader can read the shared object A unless a writer has obtained permission to update the object A.

```
region A when(readcount >=0 AND writecount = = 0){
        ……
    reading is performed ……  }
```

Q4 (a) With the help of example, discuss overlay.

Answer:

To enable a process to be larger than the amount of memory allocated to it, we can use overlays. The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed.

As an example, consider a two-pass assembler. During pass1, it constructs a symbol table; then, during pass2, it generates machine-language code. We may be able to partition such an assembler into pass1 code, pass2 code, the symbol table, and common support routines used by both pass1 and pass2. Assume that the sizes of these components are as follows:

Pass1	:	70 KB
Pass2	:	80 KB
Symbol table	:	20 KB
Common routines	:	30 KB

To load everything at once, we would require 200 KB of memory. If only 150 KB is available, we cannot run our process. However, notice that pass1 and pass2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass1, and overlay B is the symbol table, common routines, and pass2.

We add an overlay driver (10 KB) and start with overlay ***A*** in memory. When we finish pass1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass2. Overlay A needs only 120 KB, whereas overlay B needs 130 KB. We can now run our assembler in the 150 KB of memory. It will load somewhat faster because fewer data need to be transferred before execution

starts. However, it will run somewhat slower, due to the extra I/O to read the code for overlay B over the code for overlay A.
The code for overlay A and the code for overlay B are kept on disk as absolute memory images, and are read by the overlay driver as needed. Special relocation and linking algorithms are needed to construct the overlays. As in dynamic loading, overlays do not require any special support from the operating system. They can be implemented completely by the user with simple file structures, reading from the files into memory and then jumping to that memory and executing the newly read instructions. The operating system notices only that there is more I/O than usual.

(b) Consider a paging system with the page table stored in memory.

(i) If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?

If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there.)

Answer:

(i) 400 nanoseconds; 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.

(ii) Effective access time = 0.75 * (200 nanoseconds) + 0.25 * (400 nanoseconds) = 250 nanoseconds.

(c) What is the cause of thrashing? How does the system detect thrashing?

Answer:
Thrashing is caused by under allocation of the minimum number of pages required by a process, forcing it to continuously page fault. The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming. It can be eliminated by reducing the level of multiprogramming.

Q5 (a) Discuss the various attributes of a file? What are the methods of accessing the information stored in a file? Discuss them.

Answer:
A file has certain attributes, which may vary from one operating system to another, but typically consist of these:

- **Name** The symbolic file name is the only information kept in human readable form.
- **Type** This information is needed for those systems that support different types.
- **Location** This information is a pointer to a device and to the location of the file on that device.
- **Size** The current size of the file (in bytes, words or blocks), and possibly the maximum allowed size are included in this attribute.

- **Protection** Access-control information controls that can do reading, writing, executing, and so on.
- **Time, date, and user identification** This information may be kept for creation, last modification and last use. These data can be useful for protection, security, and usage monitoring.

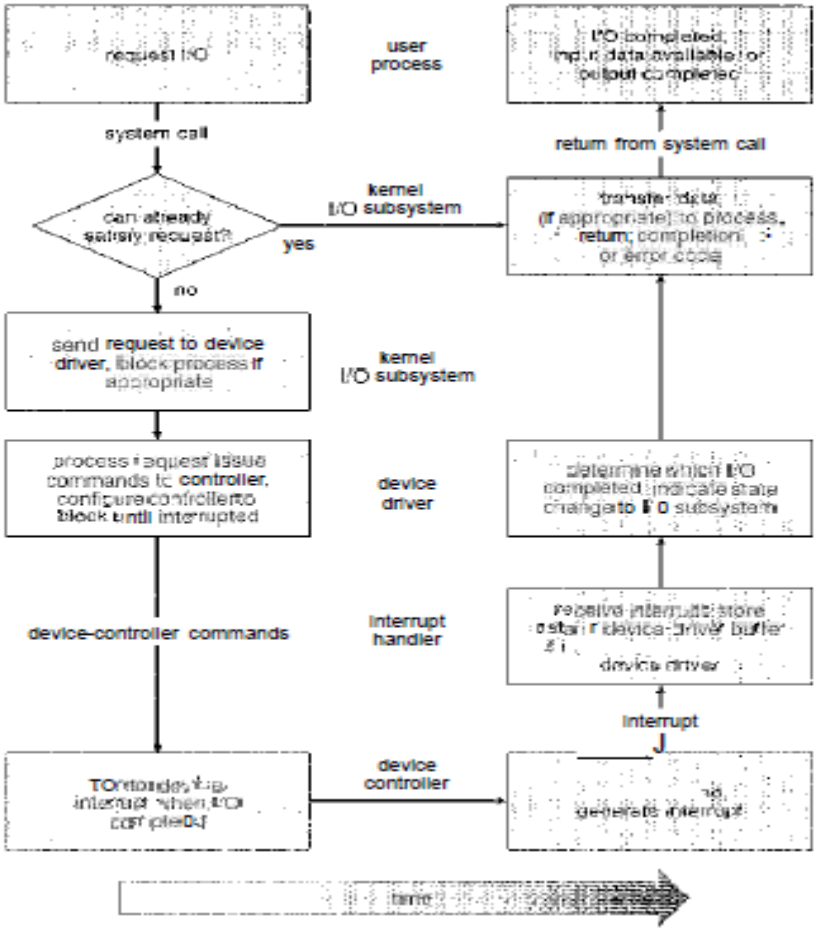
Different accessing methods are:

- **Sequential Access:** The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the most common, for example, editor and compilers usually access files in sequential methods.
- **Direct Access:** A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order. The direct access method is based on a disk model of a file, since disks allow random access to any file block. For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. There are no restrictions on the order of reading or writing for a direct-access file. Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. When a query concerning a particular subject arrives, the block containing the answer is computed, and then that block is accessed directly to provide the desired information.

(b) With the help of a figure describe the life cycle of a blocking read request?

Answer:

The typical life cycle of a blocking read request, as depicted in following figure suggests that an I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.



The Life Cycle of an I/O request

1. A process issues a blocking read () system call to a file descriptor of a file that has been opened previously.
2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.
3. Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.
4. The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.

5. The device controller operates the device hardware to perform the data transfer.
6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.
7. The correct interrupt handler receives the interrupt via the interrupt vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
8. The device driver receives the signal, determines which I/O request has completed, determines the request's status, and signals the kernel I/O subsystem that the request has been completed.
9. The kernel transfers data or returns codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

Q6 (a) Suppose that a disk drive has 5000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is

86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests, for each of the following disk scheduling algorithms?

(i) FCFS (ii) SSTF
(iii) SCAN (iv) LOOK
(v) C-SCAN

Answer:

(i) The FCFS schedule is 143, 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130. The total seek distance is 7081.
(ii) The SSTF schedule is 143, 130, 86, 913, 948, 1022, 1470, 1509, 1750, 1774. The total seek distance is 1745.
(iii) The SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 130, 86. The total seek distance is 9769.
(iv) The LOOK schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 130, 86. The total seek distance is 3319.
(v) The C-SCAN schedule is 143, 913, 948, 1022, 1470, 1509, 1750, 1774, 4999, 86, 130. The total seek distance is 9813.

(b) Describe the different methods of implementing the access matrix. Compare these methods.

Answer:
The different methods of implementing the access matrix are as follows:

- **Global Table:** The simplest implementation of the access matrix is a global table consisting of a set of ordered triples <*domain, object, rights-set*>. Whenever an

operation *M* is executed on an object O_j, within domain D_i, the global table is searched for a triple $<D_i, O_j, R_k>$, with $M \in R_k$. If this triple is found, the operation is allowed to continue; otherwise, an exception (or error) condition is raised.

This implementation suffers from several drawbacks. The table is usually large and thus cannot be kept in main memory, so additional I/O is needed. Virtual memory techniques are often used for managing this table. In addition, it is difficult to take advantage of special groupings of objects or domains. For example, if everyone can read a particular object, it must have a separate entry in every domain.

- **Access Lists for Objects:** Each column in the access matrix can be implemented as an access list for one object. Obviously, the empty entries can be discarded. The resulting list for each object consists of ordered pairs *<domain, rights-set>*, which define all domains with a nonempty set of access rights for that object.

  This approach can be extended easily to define a list plus a *default* set of access rights. When an operation *M* on an object O_j is attempted in domain *Dj,* we search the access list for object O_j, looking for an entry $<D_i, R_k>$ with $M \in R_k$. If the entry is found, we allow the operation; if it is not, we check the default set. If M is in the default set, we allow the access. Otherwise, access is denied, and an exception condition occurs. For efficiency, we may check the default set first and then search the access list.
- **Capability Lists for Domains:** Rather than associating the columns of the access matrix with the objects as access lists, we can associate each row with its domain. A capability list for a domain is a list of objects together with the operations allowed on those objects. An object is often represented by its physical, name or address, called a capability. To execute operation M on object *0,,* the process executes the operation *M,* specifying the capability (or pointer) for object O/ as a parameter. Simple possession of the capability means that access is allowed.

  The capability list is associated with a domain, but it is never directly accessible to a process executing in that domain. Rather, the capability list is itself a protected object, maintained by the operating system and accessed by the user only indirectly. Capability-based protection relies on the fact that the capabilities are never allowed to migrate into any address space directly accessible by a user process (where they could be modified). If all capabilities are secure, the object they protect is also secure against unauthorized access.

  Capabilities were originally proposed as a kind of secure pointer, to meet the need for resource protection that was foreseen as multiprogrammed computer systems came of age. The idea of an inherently protected pointer provides a foundation for protection that canbe extended up to the applications level. To provide inherent protection, we must distinguish capabilities from other kinds of objects and they must be interpreted by an abstract machine on which higher-level programs run.

  Capabilities are usually distinguished from other data in one of two ways:

  - Each object has a tag to denote its type either as a capability or as accessible data. The tags themselves must not be directly accessible by an

application program. Hardware or firmware support may be used to enforce this restriction. Although only 1 bit is necessary to distinguish between capabilities and other objects, more bits are often used. This extension allows all objects to be tagged with their types by the hardware. Thus, the hardware can distinguish integers, floating-point numbers, pointers, Booleans, characters, instructions, capabilities, and uninitialized values by their tags.

- Alternatively, the address space associated with a program can be split into two parts. One part is accessible to the program and contains the program's normal data and instructions. The other part, containing the capability list, is accessible only by the operating system. A segmented memory space is useful to support this approach.

- **A Lock-Key Mechanism:** The lock-key scheme is a compromise between access lists and capability lists. Each object has a list of unique bit patterns, called locks. Similarly, each domain has a list of unique bit patterns, called keys. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

Comparison between different methods

We now compare the various techniques for implementing an access matrix. Using a global table is simple; however, the table can be quite large and often cannot take advantage of special groupings of objects or domains. Access lists correspond directly to the needs of users. When a user creates an object, he can specify which domains can access the object, as well as the operations allowed. However, because access-rights information for a particular domain is not localized, determining the set of access rights for each domain is difficult. In addition, every access to the object must be checked, requiring a search of the access list. In a large system with long access lists, this search can be time consuming.

Q7 (a) Write shorts notes on fully distributed Deadlock-detection algorithm.

Answer:

In the **fully distributed deadlock-detection algorithm**, all controllers share equally the responsibility for detecting deadlock. Every site constructs a wait for graph that represents a part of the total graph, depending on the dynamic behavior of the system. The idea is that, if a deadlock exists, a cycle will appear in at least one of the partial graphs. We present one such algorithm, which involves construction of partial graphs in every site.

Each site maintains its own local wait-for graph. A local wait-for graph in this scheme we add one additional node P_{ex} to the graph. An arc $P_i \rightarrow P_{ex}$ exists in the graph if P_i is waiting for a data item in another site being held by *any* process. Similarly, an arc $P_{ex} \rightarrow P_j$ exists in the graph if a process at another site is waiting to acquire a resource currently being held by P_j in this local site.

To illustrate this situation, we consider the two local wait-for graphs as in figure (a) below. The addition of the node P_{ex} in both graphs results in the local wait-for graphs shown in figure (b).

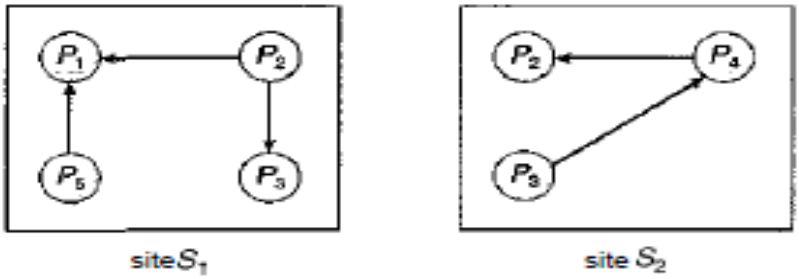


Figure (a): Two local wait-for graphs

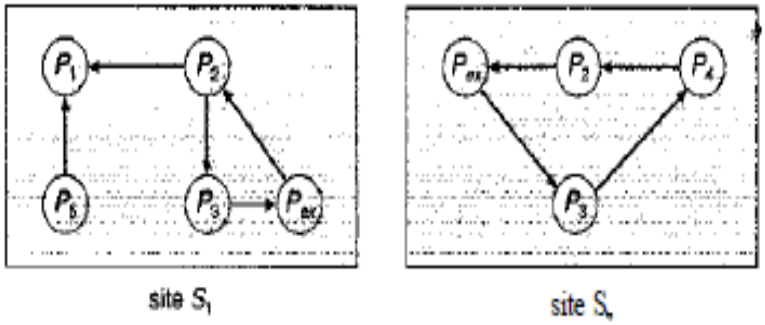


Figure (b): Augmented local wait-for graphs of figure (a)

If a local wait-for graph contains a cycle that does not involve node P_{ex}, then the system is in a deadlocked state. If, however, a local graph contains a cycle involving P_{ex}, then this implies the *possibility* of a deadlock.

To ascertain whether a deadlock does exist, we must invoke a distributed deadlock-detection algorithm.

Suppose that, at site S_i, the local wait-for graph contains a cycle involving node P. This cycle must be of the form

$$P_{ex} \rightarrow P_{k1} \rightarrow P_{k2} \rightarrow \ldots\ldots \rightarrow P_{kn} \rightarrow P_{ex}$$

which indicates that process P_{kn} in site S_i is waiting to acquire a data item located in some other site—say, S_j. On discovering this cycle, site S_i sends to site S_j a deadlock-detection message containing information about that cycle.

When site S_j receives this deadlock-detection message, it updates its local wait-for graph with the new information. Then it searches the newly constructed wait-for graph for a cycle not involving P_{ex}. If one exists, a deadlock is found, and an appropriate recovery scheme is invoked. If a cycle involving P_{ex} is discovered, then S_j transmits a deadlock-detection message to the appropriate site—say, S_k. Site S_k, in return, repeats the procedure. Thus, after a finite number of rounds, either a deadlock is discovered or the deadlock-detection computation halts.

To illustrate this procedure, we consider the local wait-for graphs of figure (b). Suppose that site S_1 discovers the cycle

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}$$

Since P_3 is waiting to acquire a data item in site S_2, a deadlock-detection message describing that cycle is transmitted from site S_1 to site S_2. When site S_2 receives this

message, it updates its local wait-for graph, obtaining the wait-for graph of figure (c). This graph contains the cycle

$P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2$

which does not include node P_{ex}. Therefore, the system is in a deadlocked state, and an appropriate recovery scheme must be invoked.

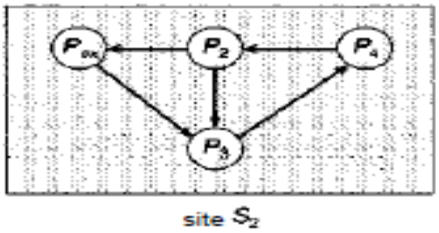


Figure (c): Augmented local wait-for graph in site S_2 of figure (b)

(b) What are the various parameters that define the Quality of Service (QoS) for multimedia applications? Discuss each of them.

Answer:

The numbers of parameters defining QoS for multimedia applications are as follows:

- **Throughput:** Throughput is the total amount of work done during a certain interval. For multimedia applications, throughput is the required data rate.
- **Delay:** Delay refers to the elapsed time from when a request is first submitted to when the desired result is produced. For example, the time from when a client requests a media stream to when the stream is delivered is the delay.
- **Jitter:** Jitter is related to delay; but whereas delay refers to the time a client must wait to receive a stream, jitter refers to delays that occur during playback of the stream. Certain multimedia applications, such as on-demand real-time streaming, can tolerate this sort of delay. Jitter is generally considered unacceptable for continuous-media applications, however, because it may mean long pauses—or lost frames—during playback. Clients can often compensate for jitter by buffering a certain amount of data—say, 5 seconds worth—before beginning playback.
- **Reliability:** Reliability refers to how errors are handled during transmission and processing of continuous media. Errors may occur due to lost packets in the network or processing delays by the CPU. In these—and other—scenarios, errors cannot be corrected, since packets typically arrive too late to be useful.

(c) Differentiate between location independence and static location transparency in a distributed file system?

Answer:

A few aspects that can differentiate location independence and static location transparency are as follows:

- Divorce of data from location, as exhibited by location independence, provides a better abstraction for files. A file name should denote the file's most significant attributes, which are its contents rather than its location. Location-independent files can be viewed as logical data containers that are not attached to a specific

storage location. If only static location transparency is supported, the file name still denotes a specific, although hidden, set of physical disk blocks.

- Static location transparency provides users with a convenient way to share data. Users can share remote files by simply naming the files in a location transparent manner, as though the files were local. Nevertheless, sharing the storage space is cumbersome, because logical names are still statically attached to physical storage devices. Location independence promotes sharing the storage space itself, as well as the data objects. When files can be mobilized, the overall, system-wide storage space looks like a single virtual resource. A possible benefit of such a view is the ability to balance the utilization of disks across the system.
- Location independence separates the naming hierarchy from the storage devices hierarchy and from the intercomputer structure. By contrast, if static location transparency is used (although names are transparent), we can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This configuration may restrict the architecture of the system unnecessarily and conflict with other considerations. A server in charge of a root directory is an example of a structure that is dictated by the naming hierarchy and contradicts decentralization guidelines.

Text Books

1. **Abraham Silberschatz, Peter Baer Galvin, Greg Gagne (2010), "Operating System Principle" John wiley & Sons (Asia) Pvt. Ltd.**

2. **Andrew S Tanenbaum "Modern Operating Systems" (2009) Pearson Education.**