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

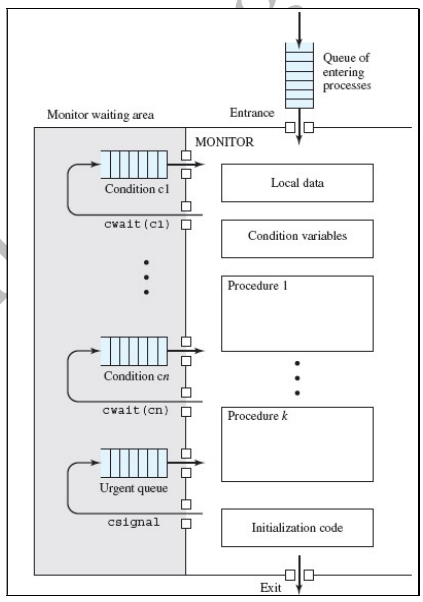


Figure: structure of monitor

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

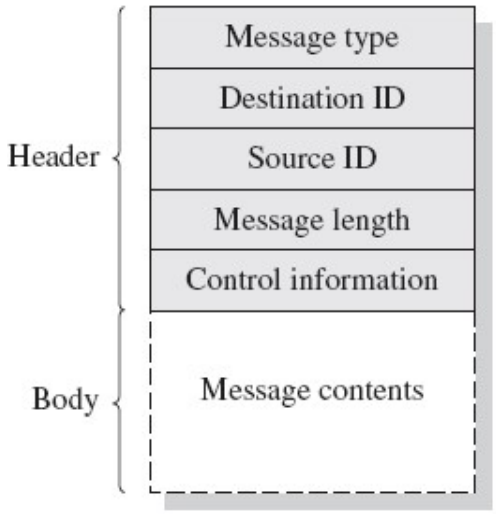


Figure: Message Format in IPC using message passing

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.

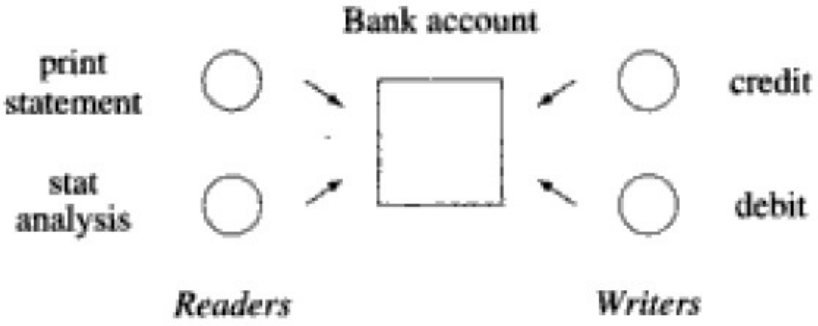
Q. What is Barbershop problem?
Ans. The Barbershop problem is defined as: A barbershop has three chairs, three barbers and a waiting area that can accommodate four customers on a sofa and a standing room for additional customers. Fire codes limit the total number

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