Process Synchronization

- As we understand in a multiprogramming environment a good number of processes compete for limited number of resources.
- Concurrent access to shared data at same time may result in data inconsistency for e.g.

```
P ()
{
        read ( i );
        i = i + 1;
        write( i );
}
```

- **Race Condition** - if we change the order of execution of different process with respect to other process the output may change.
- That is why we need some kind of synchronization to eliminate the possibility of data inconsistency.

Q The following two functions P_1 and P_2 that share a variable B with an initial value of 2 execute concurrently.

$P_1()$	$P_2()$
{	{
C = B – 1;	D = 2 * B;
B = 2 * C;	B = D - 1;
}	}

The number of distinct values that B can possibly take after the execution is **(GATE-2015) (1 Mark)**

Ans. 3

Q Consider three concurrent processes P1, P2 and P3 as shown below, which access a shared variable D that has been initialized to 100. **(GATE-2019) (2 Marks)**

P_1	P_2	P_3
.	.	.
.	.	.
D = D + 20	**D = D - 50**	**D = D + 10**
.	.	.
.	.	.

The process are executed on a uniprocessor system running a time-shared operating system. If the minimum and maximum possible values of D after the three processes have completed execution are X and Y respectively, then the value of Y–X is ___________.
Ans: 80

Q When the result of a computation depends on the speed of the processes involved, there is said to be **(GATE-1998) (1 Marks)**
a) cycle stealing **b)** race condition **c)** a time lock **d)** a deadlock
Ans: b

Critical Section Problem

General Structure of a process

- **Initial Section:** Where process is accessing private resources.
- **Entry Section:** Entry Section is that part of code where, each process request for permission to enter its critical section.
- **Critical Section:** Where process is access shared resources.
- **Exit Section:** It is the section where a process will exit from its critical section.
- **Remainder Section:** Remaining Code.

```
Pi()
{
    While(T)
    {
        Initial Section
        Entry Section
        Critical Section
        Exit Section
        Remainder Section
    }
}
```

Criterion to Solve Critical Section Problem

- **Mutual Exclusion:** No two processes should be present inside the critical section at the same time, i.e. only one process is allowed in the critical section at an instant of time.

- **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next(means other process will participate which actually wish to enter), and there should be no deadlock.

- **Bounded Waiting:** There exists a bound or a limit on the number of times a process is allowed to enter its critical section and no process should wait indefinitely to enter the CS.

Some Points to Remember:

- Mutual Exclusion and Progress are mandatory requirements that needs to be followed in order to write a valid solution for critical section problem.
- Bounded waiting is optional criteria, if not satisfied then it may lead to starvation.

Q Suppose P, Q and R are co-operating processes satisfying Mutual Exclusion condition. Then, if the process Q is executing in its critical section then **(NET-DEC-2018)**
a) Both 'P' and 'R' execute in critical section
b) Neither 'P' nor 'R' executes in their critical section
c) 'P' executes in critical section
d) 'R' executes in critical section
Ans: b

Q Which of the following conditions does not hold good for a solution to a critical section problem? **(NET-DEC-2014)**
(A) No assumptions may be made about speeds or the number of CPUs.
(B) No two processes may be simultaneously inside their critical sections.
(C) Processes running outside its critical section may block other processes.
(D) Processes do not wait forever to enter its critical section if bounded wait is satisfied.
Answer: C

Q Part of a program where the shared memory is accessed and which should be executed indivisibly, is called: **(NET-JUNE-2007)**
a) Semaphores **b)** Directory **c)** Critical Section d) Mutual exclusion
Answer: C

Solutions to Critical Section Problem:

We generally have the following solutions to a Critical Section Problems:

1. Two Process Solution
   (a) Using Boolean variable turn
   (b) Using Boolean array flag
   (c) Peterson's Solution

2. Operating System Solution
   (a) Counting Semaphore
   (b) Binary Semaphore

3. Hardware Solution
   (a) Test and Set Lock

4. Computer and Programming Support Type
   (a) Monitors

Two Process Solution

- In general, it will be difficult to write a valid solution in the first go to solve critical section problem among multiple processes, so it will be better to first attempt two process solution and then generalize it to N-Process solution.
- There are 3 Different idea to achieve valid solution, in which some are invalid while some are valid.

1- Using Boolean variable turn

2- Using Boolean array flag

3- Peterson's Solution

- Here we will use a Boolean variable turn, which is initialize randomly (0/1)

P_0	P_1

while (1)
{
    while (turn! = 0);
    Critical Section
    turn = 1;
    Remainder section
}
	while (1)
	{
	    while (turn! = 1);
	    Critical Section
	    turn = 0;
	    Remainder Section
	}

- The solution follows **Mutual Exclusion** as the two processes cannot enter the CS at the same time.
- The solution does not follow the **Progress,** as it is suffering from the strict alternation. We never asked the process whether it wants to enter the CS or not?

- Here we will use a Boolean array flag with two cells, where each cell is initialized to 0.

P_0	P_1

while (1)
{
    flag [0] = T;
    while (flag [1]);
    Critical Section
    flag [0] = F;
    Remainder section
}
	while (1)
	{
	    flag [1] = T;
	    while (flag [0]);
	    Critical Section
	    flag [1] = F;
	    Remainder Section
	}

- This solution follows the **Mutual Exclusion** Criteria.
- But in order to achieve the progress the system ended up being in a **deadlock state**.

Peterson’s Solution

- It is a classic software-based solution to the critical-section problem.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P_0 and P_1.

We will be using two data items: **int turn and Boolean flag**.

P_0	P_1
while (1)	while (1)
{	{
flag [0] = T;	flag [1] = T;
turn = 1;	turn = = 0;
while (turn = = 1 && flag [1] = = T);	while (turn = = 0 && flag [0] = =T);
Critical Section	Critical Section
flag [0] = F;	flag [1] = F;
Remainder section	Remainder Section
}	}

- This solution ensures **Mutual Exclusion, Progress and Bounded Wait.**

Q Consider the methods used by processes P_1 and P_2 for accessing their critical sections whenever needed, as given below. The initial values of shared Boolean variables S_1 and S_2 are randomly assigned. **(GATE-2010) (1 Marks)** **(NET-JUNE-2012)**

$P_1()$	$P_2()$
While (S_1==S_2);	While (S_1!= S_2);
Critical section	Critical section
S_1= S_2;	SS_2=not (S_1);

Which one of the following statements describes the properties achieved?

(A) Mutual exclusion but not progress
(B) Progress but not mutual exclusion
(C) Neither mutual exclusion nor progress
(D) Both mutual exclusion and progress

Answer: (A)

Q Consider the following two-process synchronization solution. **(GATE-2016) (2 Marks)**

Process 0	Process 1
Entry: loop while (turn == 1);	Entry: loop while (turn == 0);
(critical section)	(critical section)
Exit: turn = 1;	Exit: turn = 0;

The shared variable turn is initialized to zero. Which one of the following is TRUE?

(a) This is a correct two-process synchronization solution.

(b) This solution violates mutual exclusion requirement

(c) This solution violates progress requirement.

(d) This solution violates bounded wait requirement

Answer: (C)

Q Two processes X and Y need to access a critical section. Consider the following synchronization construct used by both the processes.

Here, varP and varQ are shared variables and both are initialized to false. Which one of the following statements is true? **(GATE-2015) (1 Mark)**

Process X	Process Y
While(t)	While(t)
{	{
varP = T;	varQ = T;
While (varQ == T)	While (varP == T)
{	{
Critical section	Critical section
varP = F;	varQ = F;
}	}
}	}

(A) The proposed solution prevents deadlock but fails to guarantee mutual exclusion

(B) The proposed solution guarantees mutual exclusion but fails to prevent deadlock

(C) The proposed solution guarantees mutual exclusion and prevents deadlock

(D) The proposed solution fails to prevent deadlock and fails to guarantee mutual exclusion

Answer: (A)

Q Two processes, P_1 and P_2, need to access a critical section of code. Consider the following synchronization construct used by the processes: Here, $wants_1$ and $wants_2$ are shared variables, which are initialized to false. Which one of the following statements is TRUE about the above construct? **(GATE-2007) (2 Mark)**

P_1()	P_2()
While(t)	While(t)
{	{
$wants_1$ = T	$wants_1$ = T
While ($wants_1$== T);	While ($wants_1$== T);
Critical section	Critical section
$wants_1$ = F	$wants_1$ = F
Remainder section	Remainder section
}	}

(A) It does not ensure mutual exclusion.
(B) It does not ensure bounded waiting.
(C) It requires that processes enter the critical section in strict alternation.
(D) It does not prevent deadlocks, but ensures mutual exclusion.
Answer: (D)

Q Consider Peterson's algorithm for mutual exclusion between two concurrent processes i and j. The program executed by process is shown below. **(GATE-2001) (2 Mark)**

Repeat
flag[i] = T;
turn = j;
while(P) do no-op;
Enter critical section, perform actions, then critical section
flag[i] = f;
Perform other non-critical section actions
Until false;

For the program to guarantee mutual exclusion, the predicate P in the while loop should be.
(A) flag[j] = true and turn = I
(B) flag[j] = true and turn = j
(C) flag[i] = true and turn = j
(D) flag[i] = true and turn = i
Answer: (B)

Operating System Solution

- Semaphores are synchronization tools using which we will attempt n-process solution.
- A semaphore S is a simple integer variable that, but apart from initialization it can be accessed only through two standard atomic operations: wait(S) and signal(S).
- The wait(S) operation was originally termed as P(S) and signal(S) was originally called V(S).

Wait(S)
{
while(s<=0);
s--;
}

Signal(S)
{
s++;
}

- Peterson's Solution was confined to just two processes, and since in a general system can have n processes, Semaphores provides n-processes solution.
- While solving Critical Section Problem only we initialize semaphore S = 1.

$P_i()$
{
While(T)
{
Initial Section
wait(s)
Critical Section
signal(s)
Remainder Section
}
}

- Semaphores are going to ensure Mutual Exclusion and Progress but does not ensures bounded waiting.

- It is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the wait() and signal() operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.
- In a multiprocessor environment, interrupts must be disabled on every processor. Otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques —such as compare and swap() or spinlocks —to ensure that wait() and signal() are performed atomically.

Q Two atomic operations permissible on Semaphores are ___________ and ___________. (NET-NOV-2017)
a) wait, stop **b)** wait, hold **c)** hold, signal **d)** wait, signal
Answer: D

Q A relationship between processes such that each has some part (critical section) which must not be executed while the critical section of another is being executed, is known as **(NET-JUNE-2011)**
a) Semaphore **b)** Mutual exclusion
b) Multiprogramming **d)** Message passing
Answer: B

Q In order to allow only one process to enter its critical section, binary semaphore are initialized to **(NET-JUNE-2010)**
a) 0 **b)** 1 **c)** 2 **d)** 3
Answer: B

Q.................. synchronize critical resources to prevent deadlock. **(NET-JUNE-2006)**
a) P-operator **b)** V-operator **c)** Semaphores **d)** Hard disk
Answer: C

Q Semaphores are used to: **(NET-DEC-2004)**
a) Synchronise critical resources to prevent deadlock
b) Synchronise critical resources to prevent contention
c) Do I/o
d) Facilitate memory management
Answer: A

Q A critical section is a program segment? **(GATE-1996) (2 Mark)**
(a) which should run in a certain specified amount of time
(b) which avoids deadlocks
(c) where shared resources are accessed
(d) which must be enclosed by a pair of semaphore operations, P and V
Answer: (A)

Q A critical region is **(GATE-1987) (1 Marks)**
a) One which is enclosed by a pair of P and V operations on semaphores.
b) A program segment that has not been proved bug-free.
c) A program segment that often causes unexpected system crashes.
d) A program segment where shared resources are accessed.
Answer: (D)

Q A certain computation generates two arrays a and b such that a[i]=f(i) for 0 ≤ i < n and b[i]=g(a[i]) for 0 ≤ i < n. Suppose this computation is decomposed into two concurrent processes X and Y such that X computes the array a and Y computes the array b. The processes employ two binary semaphores R and S, both initialized to zero. The array a is shared by the two processes. The structures of the processes are shown below. **(GATE-2013) (2 Marks)**

Ans: c

Process X:	Process X:
private i;	private i;
for (i=0; i < n; i++) {	for (i=0; i < n; i++) {
a[i] = f(i);	EntryY(R, S);
ExitX(R, S);	b[i]=g(a[i]);
}	}

ExitX(R, S) {
P(R);
V(S);
}
EntryY(R, S) {
P(S);
V(R);
}

ExitX(R, S) {
V(R);
V(S);
}
EntryY(R, S) {
P(R);
P(S);
}

ExitX(R, S) {
P(S);
V(R);
}
EntryY(R, S) {
V(S);
P(R);
}

ExitX(R, S) {
V(R);
P(S);
}
EntryY(R, S) {
V(S);
P(R);
}

Q Two concurrent processes P_1 and P_2 use four shared resources R_1, R_2, R_3 and R_4, as shown below.

P_1	P_2
Compute:	Compute;
Use R_1;	Use R_1;
Use R_2;	Use R_2;
Use R_3;	Use R_3;.
Use R_4;	Use R_4;

Both processes are started at the same time, and each resource can be accessed by only one process at a time The following scheduling constraints exist between the access of resources by the processes:

- P2 must complete use of R_1 before P_1 gets access to R_1
- P1 must complete use of R_2 before P_2 gets access to R_2.
- P2 must complete use of R_3 before P_1 gets access to R_3.
- P1 must complete use of R_4 before P_2 gets access to R_4.

There are no other scheduling constraints between the processes. If only binary semaphores are used to enforce the above scheduling constraints, what is the minimum number of binary semaphores needed? **(GATE-2005) (2 Marks)**

(A) 1 **(B)** 2 **(C)** 3 **(D)** 4

Answer: (B)

Q Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S and T. The code for the processes P and Q is shown below.

Process P	Process Q
While(t)	While(t)
{	{
W:	Y:
print '0';	print '0';
print '0';	print '0';
X:	Z:
}	}

Synchronization statements can be inserted only at points W, X, Y and Z.

***Which of the following will always lead to an output staring with '001100110011' ?* (GATE-2004) (2 Marks)**

(A) P(S) at W, V(S) at X, P(T) at Y, V(T) at Z, S and T initially 1

(B) P(S) at W, V(T) at X, P(T) at Y, V(S) at Z, S initially 1, and T initially 0

(C) P(S) at W, V(T) at X, P(T) at Y, V(S) at Z, S and T initially 1

(D) P(S) at W, V(S) at X, P(T) at Y, V(T) at Z, S initially 1, and T initially 0

Answer: (B)

Q Which of the following will ensure that the output string never contains a substring of the form 01^n0 or 10^n1 where n is odd? **(GATE-2004) (2 Marks)**

(A) P(S) at W, V(S) at X, P(T) at Y, V(T) at Z, S and T initially 1

(B) P(S) at W, V(T) at X, P(T) at Y, V(S) at Z, S and T initially 1

(C) P(S) at W, V(S) at X, P(S) at Y, V(S) at Z, S initially 1

(D) V(S) at W, V(T) at X, P(S) at Y, P(T) at Z, S and T initially 1

Answer: (C)

Deadlock

Q Suppose S and Q are two semaphores initialized to 1. P1 and P2 are two processes which are sharing resources. **(NET-JUNE-2013)**

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
Critical section 1;	critical-section 2;
signal(S);	signal(Q);
signal(Q);	signal(S);

Their execution may sometimes lead to an undesirable situation called
(A) Starvation **(B)** Race condition
(C) Multithreading **(D)** Deadlock
Ans: d

Q Three concurrent processes X, Y, and Z execute three different code segments that access and update certain shared variables. Process X executes the P operation (i.e., wait) on semaphores a, b and c; process Y executes the P operation on semaphores b, c and d; process Z executes the P operation on semaphores c, d, and a before entering the respective code segments. After completing the execution of its code segment, each process invokes the V operation (i.e., signal) on its three semaphores. All semaphores are binary semaphores initialized to one. Which one of the following represents a deadlock-free order of invoking the P operations by the processes? **(GATE-2013) (1 Marks)**
(A) X: P(a)P(b)P(c) Y: P(b)P(c)P(d) Z: P(c)P(d)P(a)
(B) X: P(b)P(a)P(c) Y: P(b)P(c)P(d) Z: P(a)P(c)P(d)
(C) X: P(b)P(a)P(c) Y: P(c)P(b)P(d) Z: P(a)P(c)P(d)
(D) X: P(a)P(b)P(c) Y: P(c)P(b)P(d) Z: P(c)P(d)P(a)
Answer: (B)

Q Consider two processes P1 and P2 accessing the shared variables X and Y protected by two binary semaphores SX and SY respectively, both initialized to 1. P and V denote the usual semaphone operators, where P decrements the semaphore value, and V increments the semaphore value. The pseudo-code of P1 and P2 is as follows: **(GATE-2004) (2 Marks)**

P_0	P_1
While true do {	While true do {
L1 : ………..	L3 : ………..
L2 : ………..	L4 : ………..
X = X + 1;	Y = Y + 1;
Y = Y – 1;	X = Y – 1;
V(SX);	V(SX);
V(SY);	V(SY);
}	}

In order to avoid deadlock, the correct operators at L1, L2, L3 and L4 are respectively

(A) P(SY), P(SX); P(SX), P(SY)

(B) P(SX), P(SY); P(SY), P(SX)

(C) P(SX), P(SX); P(SY), P(SY)

(D) P(SX), P(SY); P(SX), P(SY)

Answer: (D)

Q A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution? **(GATE-2013) (2 Marks)**

(A) -2 **(B)** -1 **(C)** 1 **(D)** 2

Answer: (D)

Q Barrier is a synchronization construct where a set of processes synchronizes globally i.e. each process in the set arrives at the barrier and waits for all others to arrive and then all processes leave the barrier. Let the number of processes in the set be three and S be a binary semaphore with the usual P and V functions. Consider the following C implementation of a barrier with line numbers shown on left.

```
void barrier (void) {
1:  P(S);
2:  process_arrived++;
3.  V(S);
4:  while (process_arrived !=3);
5:  P(S);
6:  process_left++;
7:  if (process_left==3) {
8:     process_arrived = 0;
9:     process_left = 0;
10: }
11: V(S);
}
```

The variables process_arrived and process_left are shared among all processes and are initialized to zero. In a concurrent program all the three processes call the barrier function when they need to synchronize globally.

Q The above implementation of barrier is incorrect. Which one of the following is true? **(GATE-2006) (2 Marks)**
a) The barrier implementation is wrong due to the use of binary semaphore S.
b) The barrier implementation may lead to a deadlock if two barrier in invocations are used in immediate succession.
c) Lines 6 to 10 need not be inside a critical section.

d) The barrier implementation is correct if there are only two processes instead of three.
Answer: (B)

Q Given below is a program which when executed spawns two concurrent processes:
semaphore X : = 0 ;
/* Process now forks into concurrent processes P1 & P2 */

P_1	P_2
repeat forever	repeat forever
V (X) ;	P(X) ;
Compute ;	Compute ;
P(X) ;	V(X) ;

Consider the following statements about processes P1 and P2:

1. It is possible for process P1 to starve.
2. It is possible for process P2 to starve.

Which of the following holds? **(GATE-2005) (2 Marks)**
(A) Both I and II are true
(B) I is true but II is false
(C) II is true but I is false
(D) Both I and II are false
Answer: (A)

Classical Problems on Synchronization

- There are number of actual industrial problem we try to solve in order to improve our understand of Semaphores and their power of solving problems.
- Here in this section we will discuss a number of problems like
    - Producer consumer problem/ Bounder Buffer Problem
    - Reader-Writer problem
    - Dining Philosopher problem

Producer-Consumer Problem

- Problem Definition – There are two process Producer and Consumers, producer produces information and put it into a buffer which have n cell, that is consumed by a consumer.
- Both Producer and Consumer can produce and consume only one article at a time.

Producer()	Consumer()
{	{
while(T)	while(T)
{	{
Produce()	wait(F)//UnderFlow
wait(E)//OverFlow	wait(S)
wait(S)	pick()
append()	signal(S)
signal(S)	wait(E)
wait(F)	consume()
}	}
}	}

- Producer-Consumer Problem needs to sort out three major issues:
- A producer needs to check whether the buffer is overflowed or not after producing an item before accessing the buffer.
- Similarly, a consumer needs to check for an underflow before accessing the buffer and then consume an item.
- Also, *the producer and consumer must be synchronized, so that once a producer and consumer it accessing the buffer the other must wait.*

Solution Using Semaphores

Now to solve the problem we will be using three semaphores:

Semaphore S = 1

Semaphore E = n

Semaphore F = 0

Total three resources are used

- semaphore E take count of empty cells and over flow
- semaphore F take count of filled cells and under flow
- Semaphore S take care of buffer

Q Consider the following solution to the producer-consumer synchronization problem. The shared buffer size is N. Three semaphores *empty, full* and *mutex* are defined with respective initial values of 0, N and 1. Semaphore *empty* denotes the number of available slots in the buffer, for the consumer to read from. Semaphore *full* denotes the number of available slots in the buffer, for the producer to write to. The placeholder variables, denoted by P, Q, R and S, in the code below can be assigned either *empty* or *full*. The valid semaphore operations are: *wait()* and *signal()*. **(GATE-2018) (2 Marks)**

Producer:	**Consumer:**
Do{	Do{
Wait(P);	Wait(R);
Wait(mutex);	Wait(mutex);
//Add item to buffer	//Consume item from buffer
Signal(mutex);	Signal(mutex);
Signal(Q);	Signal(S);
} while(1);	} while(1);

Which one of the following assignments to P, Q, R and S will yield the correct solution?

(A) P: *full*, Q: *full*, R: *empty*, S: *empty*
(B) P: *empty*, Q: *empty*, R: *full*, S: *full*
(C) P: *full*, Q: *empty*, R: *empty*, S: *full*
(D) P: *empty*, Q: *full*, R: *full*, S: *empty*

Answer: (C)

Q The Bounded buffer problem is also known as ___________. **(NET-NOV-2017)**

a) Producer - consumer problem
b) Reader - writer problem
c) Dining Philosophers problem
d) Both (2) and (3)

Answer: A

Q Consider the procedure below for the Producer-Consumer problem which uses semaphores:
Semaphore n = 0;
Semaphore s = 1;

Void Producer ()	**Void Consumer ()**
{	{
While(true)	While(true)
{	{
Produce ();	semWait(s);
SemWait(s);	semWait(n);
addToBuffer();	RemovefromBuffer();
semSignal(s);	semSignal(s);
SemSignal(n);	consume();
}	}
}	}

Which one of the following is TRUE? **(GATE-2014) (2 Marks)**
(A) The producer will be able to add an item to the buffer, but the consumer can never consume it.
(B) The consumer will remove no more than one item from the buffer.
(C) Deadlock occurs if the consumer succeeds in acquiring semaphore s when the buffer is empty.
(D) The starting value for the semaphore n must be 1 and not 0 for deadlock-free operation.
Answer: (C)

Q Producer consumer problem can be solved using: **(NET-DEC-2005)**
a) semaphores **b)** event counters **c**) monitors d) all the above
Answer: D

Q The semaphore variables full, empty and mutex are initialized to 0, n and 1, respectively. Process P_1repeatedly adds one item at a time to a buffer of size n, and process P_2 repeatedly removes one item at a time from the same buffer using the programs given below. In the programs, K, L, M and N are unspecified statements. **(GATE-2004) (2 Marks)**

```
P1
while (1) {
   K;
   P(mutex);
   Add an item to the buffer;
   V(mutex);
    L;
 }

P2
while (1) {
    M;
    P(mutex);
    Remove an item from the buffer;
    V(mutex);
    N;
}
```

The statements K, L, M and N are respectively

(A) P(full), V(empty), P(full), V(empty)
(B) P(full), V(empty), P(empty), V(full)
(C) P(empty), V(full), P(empty), V(full)
(D) P(empty), V(full), P(full), V(empty)

Answer: (D)

Reader-Writers Problem

- Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database (readers), whereas others may want to update (that is, to read and write) the database(writers). The former are referred to as readers and to the latter as writers.
- If two readers access the shared data simultaneously, no adverse effects will result. But, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.
- <u>Points that needs to be taken care for generating a Solutions:</u>
- The solution may allow more than one reader at a time, but should not allow any writer.
- The solution should strictly not allow any reader or writer, while a writer is performing a write operation.
- <u>Solution using Semaphores</u>
- The reader processes share the following data structures:
- semaphore mutex = 1, wrt =1; // Two semaphores
- int readcount = ; // Variable
- Three resources are used
- Semaphore Wrt is used for synchronization between WW, WR, RW
- Semaphore reader is used to synchronize between RR
- Readcount is simple int variable which keep counts of number of readers

Writer()	Reader()
Wait(wrt)	Wait(mutex)
CS //Write	Readcount++
Signal(wrt)	If(readcount ==1)
	wait(wrt)
	signal(mutex)
	CS //Read
	Wait(mutex)
	Readcount--
	If(readcount ==0)
	signal(wrt)
	signal(mutex)

Q To overcome difficulties in Readers-Writers problem, which of the following statement/s is/are true? **(NET-DEC-2018)**
1) Writers are given exclusive access to shared objects
2) Readers are given exclusive access to shared objects
3) Both readers and writers are given exclusive access to shared objects.
Choose the correct answer from the code given below:
a) 1 **b)** 2 **c)** 3 **d)** 2 and 3 only
Ans: a

Q Let P_i and P_j two processes, R be the set of variables read from memory, and W be the set of variables written to memory. For the concurrent execution of two processes pi and Pj which of the conditions are not true? **(NET-JUNE-2015)**
a) $R(Pi)\cap W(Pj)=\Phi$ **b)** $W(Pi)\cap R(Pj)=\Phi$
c) $R(Pi)\cap R(Pj)=\Phi$ **d)** $W(Pi)\cap W(Pj)=\Phi$
Ans: c

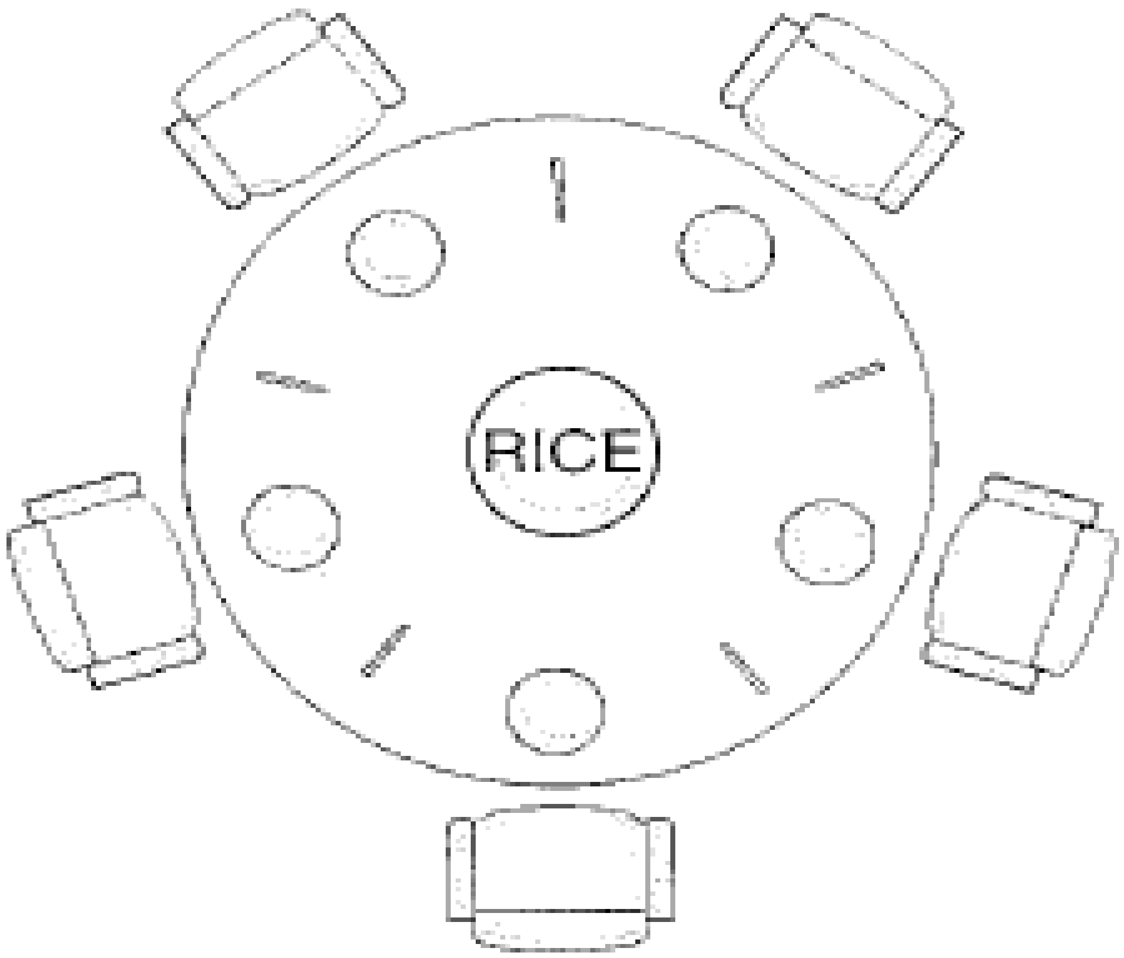
Q Synchronization in the classical readers and writers problem can be achieved through use of semaphores. In the following incomplete code for readers-writers problem, two binary semaphores mutex and wrt are used to obtain synchronization **(GATE-2007) (2 Marks)**

wait (wrt)
writing is performed
signal (wrt)
wait (mutex)
readcount = readcount + 1
if readcount = 1 then S1
S2
reading is performed
S3
readcount = readcount – 1
if readcount = 0 then S4
signal (mutex)
The values of S1, S2, S3, S4, (in that order) are
(A) signal (mutex), wait (wrt), signal (wrt), wait (mutex)
(B) signal (wrt), signal (mutex), wait (mutex), wait (wrt)
(C) wait (wrt), signal (mutex), wait (mutex), signal (wrt)
(D) signal (mutex), wait (mutex), signal (mutex), wait (mutex)
Answer: (C)

Dining-Philosopher Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she can’t pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.



Code that needs to be followed by philosophers is:

```
Void Philosopher (void)

{

        while ( T )

        {

                Thinking ( ) ;

                wait(chopstick [i]);

                wait(chopstick([(i+1)%5]);

                Eat( );

                signal(chopstick [i]);

                signal(chopstick([(i+1)%5]);

        }

}
```

- Here we have used an array of semaphores called chopstick[]
- Solution is not valid because there is a possibility of deadlock.
- Here we have used an array of semaphores called chopstick[]
- Solution is not valid because there is a possibility of deadlock.
- The proposed solution for deadlock problem is
  - Allow at most four philosophers to be sitting simultaneously at the table.
  - Allow six chopstick to be used simultaneously at the table.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
  - One philosopher picks up his right chopstick first and then left chop stick, i.e. reverse the sequence of any philosopher.

- Odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Q Dining Philosopher's problem is a: **(NET-JUNE-2015)**
a) Producer - consumer problem
b) Classical IPC problem
c) Starvation problem
d) Synchronization primitive
Ans: b

Q Let m[0]…m[4] be mutexes (binary semaphores) and P[0] …. P[4] be processes.
Suppose each process P[i] executes the following:

wait (m[i]); wait(m[(i+1) mode 4]);
------
release (m[i]); release (m[(i+1)mod 4]);
This could cause: **(GATE-2000) (1 Marks)**
(A) Thrashing
(B) Deadlock
(C) Starvation, but not deadlock
(D) None of the above
Answer: (B)

Q A solution to the Dining Philosophers Problem which avoids deadlock is: **(GATE-1996) (2 Marks)**
(A) ensure that all philosophers pick up the left fork before the right fork
(B) ensure that all philosophers pick up the right fork before the left fork
(C) ensure that one particular philosopher picks up the left fork before the right fork, and that all other philosophers pick up the right fork before the left fork
(D) None of the above
Answer: (C)

Types of Semaphore

There are of two types of semaphores:

1. Binary Semaphores: The value of a binary semaphore can range only between 0 and 1.

2. Counting Semaphores: can range over an unrestricted domain Counting semaphore can range over an unrestricted domain. i.e. -∞ to +∞. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Implementation of semaphore

- This simple implementation of semaphore with this wait(S) and signal(S) function suffer from busy waiting.

Wait(S)
{
while(s<=0);
s--;
}

Signal(S)
{
s++;
}

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct{
    int value;
    struct  process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list.

Now, the wait() semaphore operation can be defined as

```
wait(semaphore *S){
    S->value--;
    if  (S->value  < 0){
        add this process to S->list;
        block();
    }
}
```

Signal() semaphore operation can be defined as

```
signal(semaphore  *S){
    S->value++;
    if  (S->value  <= 0){
        remove a process P from S->list;
        wakeup(P);
    }
}
```

- The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.
- Note that in this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.
- The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue.

Q At a particular time of computation, the value of a counting semaphore is 10. Then 12 P operations and "x" V operations were performed on this semaphore. If the final value of semaphore is 7, x will be: **(NET-JULY-2018)**

a) 8 **b)** 9 **c)** 10 **d)** 11

Ans: b

Q There are three processes P1, P2 and P3 sharing a semaphore for synchronizing a variable. Initial value of semaphore is one. Assume that negative value of semaphore tells us how many processes are waiting in queue. Processes access the semaphore in following order : **(NET-JAN-2017)**

(a) P2 needs to access
(b) P1 needs to access
(c) P3 needs to access
(d) P2 exits critical section
(e) P1 exits critical section

The final value of semaphore will be:

a) 0 **b)** 1 **c)** -1 **d)** -2

Answer: A

Q Consider a non-negative counting semaphore S. The operation P(S) decrements S, and V(S) increments S. During an execution, 20 P(S) operations and 12 V(S) operations are issued in some order. The largest initial value of S for which at least one P(S) operation will remain blocked is _________. **(GATE-2016) (2 Marks)**

(A) 7 **(B)** 8 **(C)** 9 **(D)** 10

Answer: (A)

Q A semaphore count of negative n means (s=–n) that the queue contains ………….. waiting processes. **(NET-DEC-2009)**

a) n + 1 **b)** n **c)** n – 1 **d)** 0

Answer: B

Q The P and V operations on counting semaphores, where s is a counting semaphore, are defined as follows:

```
P(s) : s =  s - 1;
  if (s  < 0) then wait;

V(s) : s = s + 1;
  if (s <= 0) then wakeup a process waiting on s;
```

Assume that P_b and V_b the wait and signal operations on binary semaphores are provided. Two binary semaphores X_b and Y_b are used to implement the semaphore operations P(s) and V(s) as follows:

```
P(s) : Pb(Xb);
  s = s - 1;
  if (s < 0) {
   Vb(Xb) ;
   Pb(Yb) ;
  }
  else Vb(Xb);

V(s) : Pb(Xb) ;
  s = s + 1;
  if (s <= 0) Vb(Yb) ;
  Vb(Xb) ;
```

The initial values of X_b and Y_b are respectively **(GATE-2008) (2 Marks)**

(A) 0 and 0 **(B)** 0 and 1 **(C)** 1 and 0 **(D)** 1 and 1

Answer: (C)

Q A counting semaphore was initialized to 10. Then 6P (wait) operations and 4V (signal) operations were completed on this semaphore. The resulting value of the semaphore is **(GATE-1998) (1 Marks)**

a) 0 **b)** 8 **c)** 10 **d)** 12

Ans: b

Q At a particular time of computation the value of a counting semaphore is 7. Then 20 P operations and 15V operations were completed on this semaphore. If the new value of semaphore is will be **(GATE-1992) (1 Marks)**

(A) 42 **(B)** 2 **(C)** 7 **(D)** 12

Answer: (B)

Hardware Type Solution Test and Set Lock (TSL)

software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. In the following discussions, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software, all these solutions are based on the premise of locking —that is, protecting critical regions through the use of locks.

The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.

```
Boolean test and set(Boolean *target){
    Boolean rv = *target;
    *target  = true;
    return rv;
}



While(1){
    while  (test and set(&lock));
    /* critical section */
   lock = false;
   /* remainder section */
}
```

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word atomically —that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner.

The important characteristic of this instruction is that it is executed atomically. Thus, if two test and set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

Q Fetch_And_Add(X,i) is an atomic Read-Modify-Write instruction that reads the value of memory location X, increments it by the value i, and returns the old value of X. It is used in the pseudocode shown below to implement a busy-wait lock. L is an unsigned integer shared variable initialized to 0. The value of 0 corresponds to lock being available, while any non-zero value corresponds to the lock being not available.

```
AcquireLock(L){
      while (Fetch_And_Add(L,1))
          L = 1;
  }

 ReleaseLock(L){
      L = 0;
  }
```

This implementation **(GATE-2012) (2 Marks)**

(A) fails as L can overflow

(B) fails as L can take on a non-zero value when the lock is actually available

(C) works correctly but may starve some processes

(D) works correctly without starvation

Answer: (B)

Q The enter_CS() and leave_CS() functions to implement critical section of a process are realized using test-and-set instruction as follows:

```
void enter_CS(X)
{
   while test-and-set(X) ;
}
```

```
void leave_CS(X)
{
  X = 0;
}
```

In the above solution, X is a memory location associated with the CS and is initialized to 0. Now consider the following statements: **(GATE-2009) (2 Marks)**

I. The above solution to CS problem is deadlock-free

II. The solution is starvation free.

III. The processes enter CS in FIFO order.

IV More than one process can enter CS at the same time.

Which of the above statements is TRUE?

(A) I only **(B)** I and II **(C)** II and III **(D)** IV only

Answer: (A)

Q The atomic fetch-and-set x, y instruction unconditionally sets the memory location x to 1 and fetches the old value of x In y without allowing any intervening access to the memory location x. consider the following implementation of P and V functions on a binary semaphore S.

```
void P (binary_semaphore *s)
{
   unsigned y;
   unsigned *x = &(s->value);
   do
   {
      fetch-and-set x, y;
   }
   while (y);
}
void V (binary_semaphore *s)
{
   S->value = 0;
}
```

Which one of the following is true? **(GATE-2006) (2 Marks)**

(A) The implementation may not work if context switching is disabled in P
(B) Instead of using fetch-and –set, a pair of normal load/store can be used
(C) The implementation of V is wrong
(D) The code does not implement a binary semaphore
Answer: (A)

Disable interrupt

- This could be a hardware solution where process have a privilege instruction, i.e. before entering into critical section, process will disable all the interrupts and at the time of exit, it again enables interrupts.
- This solution is only used by OS, as if some user process enter into critical section, then can block the entire system.

```
$P_i$()
{
    While(T)
    {
        Initial Section
        Entry Section//Disable interrupt
        Critical Section
        Exit Section//Enable interrupt
        Remainder Section
    }
}
```

Q In an operating system, indivisibility of operation means: **(NET-DEC-2015)**

(A) Operation is interruptible
(B) Race – condition may occur
(C) Processor cannot be pre-empted
(D) All of the above

Answer: (C)