UNIT-V
Synchronization

Objectives:

- Students will be able to introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- Students will be able to present both software and hardware solutions of the critical-section problem.
- To discuss various inter process communication and synchronization problems.

Syllabus:

The critical section problem, Peterson's solution, synchronization hardware, semaphores, classic problems of synchronization (Bounded-Buffer problem, Readers-Writers problem, Dining-philosophers problem), monitors.

Outcomes:

Students will be able to

- Understand the concepts of critical section problems and its solutions.
- Outline the solutions of critical section problems.
- Develop algorithms for various Inter Process Communication and Synchronization problems

Learning Material

INTRODUCTION:

- Race condition: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be synchronized**.**

5.1. Critical Section Problem:

Definition: Consider a system consisting of *n* processes {P_0, P_1, …, P_{n-1}}. Each process has a segment of code, called critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.

- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section** followed by **exit section;** the remaining code is the **remainder section.**

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (TRUE);
```

Figure: General structure of a typical process p_i.

➢ A solution to the critical-section problem must satisfy the following three requirements:

- **Mutual exclusion:** If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.
- **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, and this selection cannot be postponed indefinitely.
- **Bounded waiting.** There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

5.2. Peterson's solution:

- A classic software-based solution to the critical-section problem known as Peterson's solution**.**
- 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.
- For convenience, when presenting P_i, we use P_j to denote the other process; that is, j equals 1 - i.
- Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

- The variable turn indicates whose turn it is to enter its critical section.
  - If turn ==i, then process *Pi* is allowed to execute in its critical section.
- The flag array is used to indicate if a process *is ready* to enter its critical section.
  - if flag [i] is true, this value indicates that *Pi* is ready to enter its critical section.
- To enter the critical section, process *Pi* first sets flag [i] to be true and then sets turn to the value j.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure: The structure of process A in Peterson's solution

- The eventual value of turn determines which of the two processes is allowed to enter its critical section first.
- We now prove that this solution is correct. We need to show that:

  1. Mutual exclusion is preserved.

  2. The progress requirement is satisfied.

  3. The bounded-waiting requirement is met.

  - To prove property 1, we note that each P_i enters its critical section only if either flag [j] == false or turn == i. If both processes can be executing in their critical sections at the same time, then flag [0] == flag [1] ==true.
  - These two observations imply that *Po* and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.
  - One of the processes say, P_i -must have successfully executed the while statement, whereas P_i had *to* execute at least one additional statement ("turn== j").
  - To prove properties 2 and 3, we note that a process *P;* can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] ==true and turn=== j; this loop is the only one possible.

- If P_i is not ready to enter the critical section, then flag [j] ==false, and P_i can enter its critical section. If P_j has set flag [j] to true and is also executing in its while statement, then either turn === i or turn === j. If turn == i, then P_i will enter the critical section.
- If turn== j, then *Pi* will enter the critical section. However, once *Pi* exits its critical section, it will reset flag [j] to false, allowing P_i to enter its critical section.
- If P_i resets flag [j] to true, it must also set turn to i. Thus, since P_i does not change the value of the variable turn while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

5.3. Synchronization hardware

- By using locks critical section problem is solved.
- Race conditions are prevented by requiring that critical regions be protected by locks.
- That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE);
```

Figure: Solution to the critical-section problem using locks.

- If lock= false, then no process is executing in critical section.

5.3.1. Test And Set() :

- Whenever a process is ready to enter in the critical section and it calls Test And Set() which sets lock=true.

- Here (critical section process is executing) at this condition if any process want to enter into critical section it should wait until the process executes in critical section.

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure: The definition of the TestAndSet () instruction.

- If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.
- The structure of process P_i is shown below

```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Figure: Mutual-exclusion implementation with TestAndSet ().

5.3.2. Swap () instruction:

- This instruction, in contrast to the TestAndSet () instruction, operates on the contents of two words; the common data structures are
- A global Boolean variable lock is declared and is initialized to false.
- In addition, each process has a local Boolean variable key.

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

Figure: The definition of the Swap () instruction

- The structure of process P_i *for* is shown below:

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock, &key);

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Figure: Mutual-exclusion implementation with the Swap() instruction.

- These algorithms satisfy the mutual-exclusion requirement; they do not satisfy the bounded-waiting requirement.

5.3.3. Modified Test and Set(): proving bounded waiting requirement

```
boolean waiting[n];
boolean lock;
```

- These data structures are initialized to false.
- To prove that the mutual exclusion requirement is met, we note that process *P;* can enter its critical section only if either waiting [i] == false or key == false.
- The value of key can become false only if the TestAndSet () is executed.
- The first process to execute the TestAndSet () will find key== false; all others must wait.
- The variable waiting [i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual-exclusion requirement.

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
} while (TRUE);
```

Figure: Bounded-waiting mutual exclusion with TestAndSet ().

5.4. Semaphores:

- Semaphore is nothing but a synchronization tool.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations:
  - wait () and signal ().
  - The wait () operation was originally termed P (from the Dutch *proberen,* "to test");
  - signal () was originally called V (from *verhogen,* "to increment").
- The definition of wait () is as follows:

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

- The definition of signal () is as follows:

```
signal(S) {
    S++;
}
```

- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

5.4.1. Usage:

Semaphore is of two types:

1. The value of a **counting semaphore** can range over an unrestricted domain
2. The value of a **binary semaphore** can range only between 0 and 1.
    a. In some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

- Semaphores are used to solve various synchronization problems.
- For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2.
- Suppose we require that S_2 be executed only after S_1 has completed.
- We can implement this scheme readily by letting P_1 and P_2 share a common semaphore synch, initialized to 0, and by inserting the statements in process P_1 and the statements in process P_2.

```
S1;
signal(synch);

wait(synch);
S2;
```

5.4.2. Implementation:

- By using semaphores we have one disadvantage is busy waiting.
- If one process is executing in critical section the other processes waiting outside is known as **busy waiting.**
- Spin **Lock:** wastage of CPU cycles is known as Spin lock
- Solution to busy waiting:
    - Define a semaphore as a record

```
typedef struct
{
        int value;
        struct process *L;
```

```
} semaphore;
```

- Assume two simple operations:
  - ✦ block suspends the process that invokes it.
  - ✦ wakeup(P) resumes the execution of a blocked process P.

➢ Semaphore operations now defined as

- wait(S)

```
{
        S.value--;
        if (S.value < 0)
        {
                add this process to S.L;
                block();
        }
}
```

- signal(S)

```
{
        S.value++;
        if (S.value <= 0)
        {
                remove a process P from S.L;
                wakeup(P);
        }
}
```

5.4.3. Deadlocks and Starvation:

➢ The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.

➢ The event in question is the execution of a signal () When such a state is reached, these processes are said to be deadlocked.

➢ We consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

- Suppose that P_0 executes wait (S) and then P_1, executes wait (Q).
- When P_0 executes wait (Q), it must wait until P_1, executes signal (Q).
- Similarly, when P_1, executes wait (S), it must wait until P_0executes signal(S).
- Since these signal () operations cam1ot be executed, P_0 and P_1, are deadlocked.
- We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.
- Problem related to deadlocks is **indefinite blocking** or **starvation** a situation in which processes wait indefinitely within the semaphore.

5.5. Classic problems of synchronization

5.5.1. The Bounded-Buffer Problem:

- We assume that the pool consists of *n* buffers, each capable of holding one item.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
- The empty and full semaphores count the number of empty and full buffers.
- The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
- The code for the producer process is shown below

```
do {
     . . .
   // produce an item in nextp
     . . .
   wait(empty);
   wait(mutex);
     . . .
   // add nextp to buffer
     . . .
   signal(mutex);
   signal(full);
} while (TRUE);
```

Figure: The structure of the producer process.

- The code for the consumer process is shown below:

```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
} while (TRUE);
```

Figure: The structure of the consumer process

- The producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

5.5.2. Readers-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, whereas others may want to update the database.
- These two types of processes are distinguished as readers and writers
- If two readers access the shared data simultaneously, no adverse effects will result.
- If a writer and some other process (either a reader or a writer) access the database then problem arises.
- The information in the shared data is read by a process that processor is known as reader process. This performs in shared lock.
- The writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the *readers-writers problem.*
- In the solution to the first readers-writers problem, the reader processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

- The semaphores mutex and wrt are initialized to 1;
- readcount is initialized to 0.
- The semaphore wrt is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
- The readcount variable keeps track of how many processes are currently reading the object.
- The semaphore wrt functions as a mutual-exclusion semaphore for the writers.

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

Figure: The structure of a reader process.

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Figure: The structure of a writer process

Reader-writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader writer lock.

5.5.3. Dining-philosophers 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

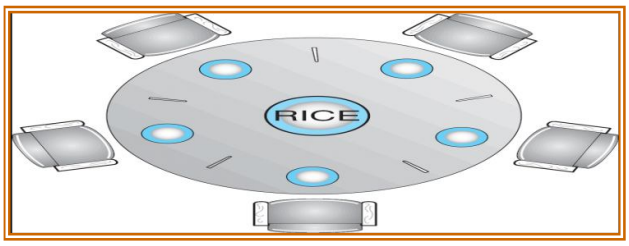

Figure: The situation of the dining philosophers

- 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.
- *A* philosopher may pick up only one chopstick at a time.
- Obviously, she cannot 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.

- One simple solution is to represent each chopstick with a semaphore.
- *A* philosopher tries to grab a chopstick by executing a wait () operation on that semaphore;
- she releases her chopsticks by executing the signal () operation on the appropriate semaphores.

semaphore chopstick[5];

- The structure of philosopher i is shown below

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
} while (TRUE);
```

Figure: The structure of philosopher *i.*

- Several possible remedies to the deadlock problem are listed next.
    - Allow at most four philosophers to be sitting 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).
    - Use an asymmetric solution; that is, an 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.

5.6. Monitors:

- Semaphores provide a convenient and effective mechanism for process synchronization.
- By using them incorrectly can result in timing errors that are difficult to detect.
- These errors happen only if some particular execution sequences take place and these sequences do not always occur.
  - Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
   ...
  critical section
   ...
wait(mutex);
```

  - This sequence violating the mutual-exclusion requirement.
  - Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes

```
wait(mutex);
   ...
  critical section
   ...
wait(mutex);
```

  - In this case, a deadlock will occur.
  - Suppose that a process omits the wait (mutex), or the signal (mutex), or both.
  - In this case, either mutual exclusion is violated or a deadlock will occur.

5.6.1. Usage:

- A *monitor type* is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.

- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.

```
monitor monitor name
{
    // shared variable declarations
    procedure P1 ( . . . ) {
        . . .
    }
    procedure P2 ( . . . ) {
        . . .
    }
            .
            .
            .
    procedure Pn ( . . . ) {
        . . .
    }
    initialization code ( . . . ) {
        . . .
    }
}
```

Figure: Syntax of a monitor.

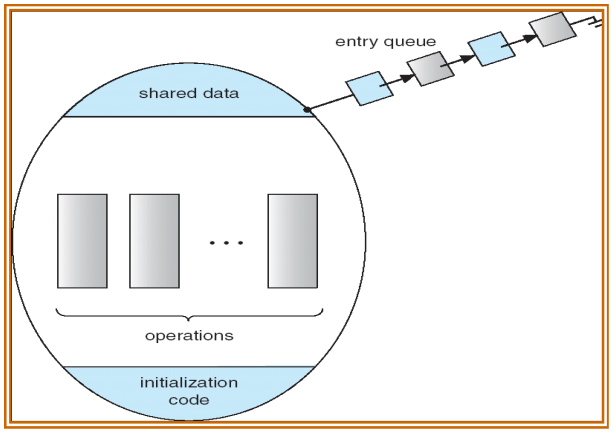

Figure: Schematic view of a monitor.

- A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition:*

condition x, y;

- The only operations that can be invoked on a condition variable are wait () and signal ().

x. wait();

- The operation means that the process invoking this operation is suspended until another process invokes

x. signal();

- The x. signal () operation resumes exactly one suspended process.

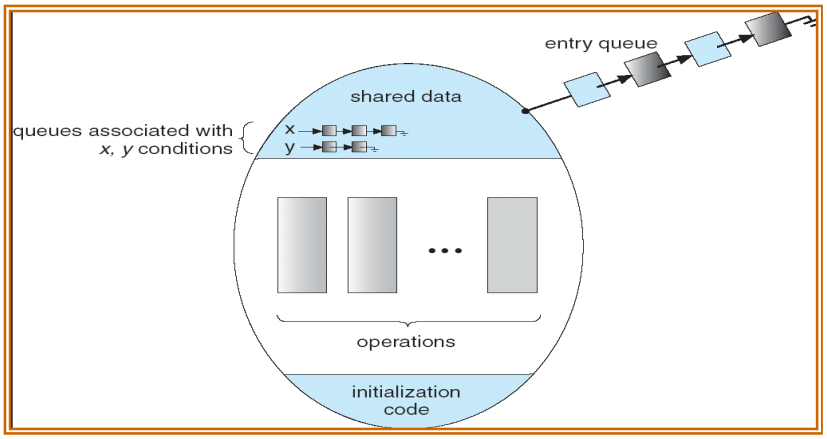

Figure Monitor with condition variables

- Suppose that, when the x. signal () operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- If the suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor.
- Both processes can conceptually continue with their execution. Two possibilities exist:

1. **Signal and wait.** *P* either waits until *Q* leaves the monitor or waits for another condition.
2. **Signal and continue.** *Q* either waits until *P* leaves the monitor or waits for another condition.

5.6.2. Dining-Philosophers Solution Using Monitors:

- Presenting a deadlock-free solution to the dining-philosophers problem.
- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- Data structure to distinguish among three states in which we may find a philosopher

```
enum {THINKING, HUNGRY, EATING} state[5];
```

Philosopher *i* can set the variable `state[i] = EATING` only if her two neighbors are not eating: `(state[(i+4) % 5] != EATING)` and `(state[(i+1) % 5] != EATING)`.

We also need to declare

```
condition self[5];
```

- Each philosopher, before starting to eat, must invoke the operation pickup().
- After the successful completion of the operation, the philosopher may eat.
- The philosopher invokes the put down() operation.
- Thus, philosopher *i* must invoke the operations pickup() and put down() in the following sequence:

```
DiningPhilosophers.pickup(i);
                ...
                eat
                ...
DiningPhilosophers.putdown(i);
```

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
          state[i] = EATING;
          self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Figure: A monitor solution to the dining-philosopher problem

UNIT-V
Assignment-Cum-Tutorial Questions
SECTION A

Objective Questions

1. When several processes access the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called [ ]
   a) dynamic condition. b) race condition
   c) essential condition d) critical condition
2. If a process is executing in its critical section, then no other processes can be executing in their critical section. This condition is called [ ]
   a) mutual exclusion b) critical exclusion
   c) synchronous exclusion d) asynchronous exclusion
3. Which one of the following is a synchronization tool? [ ]
   a) thread b) pipe
   c) semaphore d) socket
4. Mutual exclusion can be provided by the [ ]
   a) mutex locks
   b) binary semaphores
   c) both mutex locks and binary semaphores
   d) none of the mentioned
5. To enable a process to wait within the monitor, [ ]
   a) a condition variable must be declared as condition
   b) condition variables must be used as boolean objects
   c) semaphore must be used
   d) all of the mentioned
6. The segment of code in which the process may change common variables, update tables, write into files is known as : [ ]
   a) program b) critical section
   c) non – critical section d)synchronizing

7. The following three conditions must be satisfied to solve the critical section problem : [ ]
   a) Mutual Exclusion b) Progress
   c) Bounded Waiting d)All of the mentioned
8. An un-interruptible unit is known as : [ ]
   a) single b) atomic
   c) static d)none of the mentioned
9. If the semaphore value is negative : [ ]
   a) its magnitude is the number of processes waiting on that semaphore
   b) it is invalid
   c) no operation can be further performed on it until the signal operation is performed on it
   d) none of the mentioned
10. The two kinds of semaphores are : [ ]
   a) mutex & counting b)binary & counting
   c) counting & decimal d)decimal & binary
11. The bounded buffer problem is also known as : [ ]
   a) Readers – Writers problem b) Dining – Philosophers problem
   c) Producer – Consumer problem d)None of the mentioned
12. In the bounded buffer problem, there are the empty and full semaphores that :
   [ ]
   a) count the number of empty and full buffers
   b) count the number of empty and full memory spaces
   c) count the number of empty and full queues
   d) none of the mentioned
13. To ensure difficulties do not arise in the readers – writers problem, _____ are given exclusive access to the shared object. [ ]
   a) readers b)writers
   c) readers and writers d)none of the mentioned

14. The dining – philosophers problem will occur in case of :
    a) 5 philosophers and 5 chopsticks [ ]
    b) 4 philosophers and 5 chopsticks
    c) 3 philosophers and 5 chopsticks
    d) 6 philosophers and 5 chopsticks
15. All processes share a semaphore variable mutex, initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward.

    Suppose a process executes in the following manner :

    signal(mutex);

    .....

    critical section

    .....

    wait(mutex);

    In this situation :

    a) a deadlock will occur [ ]
    b) processes will starve to enter critical section
    c) several processes maybe executing in their critical section
    d) all of the mentioned
16. A monitor is characterized by :
    a) a set of programmer defined operators [ ]
    b) an identifier
    c) the number of variables in it
    d) all of the mentioned
17. The monitor construct ensures that : [ ]
    a) only one process can be active at a time within the monitor
    b) n number of processes can be active at a time within the monitor (n being greater than 1)
    c) the queue has only one process in it at a time
    d) all of the mentioned
18. The operations that can be invoked on a condition variable are :
    a) wait & signal b) hold & wait [ ]

c) signal & hold d) continue & signal

19. A monitor is a module that encapsulates
    a) shared data structures [ ]
    b) procedures that operate on shared data structure
    c) synchronization between concurrent procedure invocation
    d) all of the mentioned
20. To enable a process to wait within the monitor, [ ]
    a) a condition variable must be declared as condition
    b) condition variables must be used as boolean objects
    c) semaphore must be used
    d) all of the mentioned
21. Mutual Exclusion can be provided by the [ ]
    a) Mutex Locks b) Binary Semaphores
    c) Both (a) and (b) d) None of the Mentioned
22. Process Synchronization can be done on [ ]
    a) Hardware Level b)Software Level
    c) Both (a) and (b) d) None of the mentioned

SECTION-B

Descriptive Questions

1. Prove that the Peterson's Solution for critical section problem is correct with the help of flag and turn variables.
2. Discuss hardware instructions used for process synchronization.
3. Define the instructions, test and set ( ) and swap ( )
4. Explain about Synchronization Hardware.
5. What is a semaphore? What are its operations?
6. What is a Critical Section Problem? Write any two classic problems of
   Synchronization.
7. What is Readers-Writers problem? How it can be considered as synchronization problem? Explain its solution with Mutex locks.

8. Explain in detail how monitors are used to solve the Dining-Philosopher problem.
9. How can we use Monitors in Synchronization?
10. What is a bounded-buffer problem? Explain its solution using mutex locks.
11. Explain about solution to Dining-philosophers problem using wait() and signal() operations?

SECTION-C

Previous GATE/NET questions

1. A critical section is a program segment **GATE-1996 [ ]**

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

2. A solution to the Dining Philosophers Problem which avoids deadlock is:

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 **GATE-1996 [ ]**

3. Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned.

Method Used by P1 **GATE-2010 [ ]**

Method Used by P2

while (S1 == S2) ;

Critica1 Section

S1 = S2;

while (S1 != S2) ;

Critica1 Section

S2 = not (S1);

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

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

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

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

GATE-2000 [ ]

wait (m[i]);wait (m[(i+1) mode 4]);

.....

release (m[i]); release (m[(i+1)mod 4]);

This could cause

a) Thrashing
b) Deadlock
c) Starvation, but not deadlock
d) None of the above

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

GATE-2009 [ ]

```
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:

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

7. The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0=1, S1=0, S2=0.

Process P0	Process P1	Process P2

while (true)
{
wait (S0);
print (0);
release (S1);
release (S2);
}
	wait (S1);
	Release
	(S0);
		wait (S2);
		release (S0);

How many times will process P0 print '0'? **GATE-2010 [ ]**

a) At least twice b)Exactly twice

c) Exactly thrice d)Exactly once

8. 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.

GATE-2012 [ ]

```
AcquireLock(L){
while (Fetch_And_Add(L,1))
        L = 1;
}
ReleaseLock(L){
            L = 0;
}
```

This implementation

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

9. 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 [ ]

P1	P2	P3
. . . . 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 _____. [ ]

(A) 80

(B) 130

(C) 50

(D) None of these