# 5...

# Process Synchronization

## Objectives...

- To introduce concept of Process synchronization.
- To learn about Critical Section Problem.
- To get knowledge of Semaphores.
- To study about Classical Problems of Synchronization.

## 5.1 INTRODUCTION

- Process synchronization means sharing resources by process in such a way that no two processes can share data and resources at the same time.
- It is specially used in multi process system when multiple processes wants to acquire same shared resource or data at the same time.
- When two or more processes wants to acquire shared region data it can lead to inconsistency of shared data, processes need to be synchronized with each other.
- The basic technique used to implement synchronization is to block a process until an appropriate condition is fulfilled.
- There are two kinds of synchronization one is control synchronization and another is data access synchronization.
    - **Control synchronization:** In this synchronization the processes may wish to co-ordinate their activities with respect to one another such that a process performs an action only when some other processes reach specific points in their execution.
    - **Data access synchronization:** This synchronization ensures that shared data do not lose consistency when they are updated by several processes. It is implemented by ensuring that accesses to shared data are performed in a mutually exclusive manner.
- Process Synchronization was introduced to handle problems that arise while multiple process executions. Some of the problems are discussed below.

## 5.2 CRITICAL SECTION PROBLEM [S-18, 19, 22, 23]

- 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.

- Consider a system consisting of n processes $\{P_0, P_1 \ldots P_{n-1}\}$. Each process has a segment of code, called a **Critical Section,** in which the process may be changing common variables, updating a table, writing a file and so on.
- In a sense, updating of a shared variable may be regarded as a critical section. The critical section is a sequence of instructions with a clearly marked beginning and end. It usually safeguards of updating of one or more shared variables.
- When one process is executing in critical section, no other process is allowed to enter in critical section. Each process must request to enter into the critical section.
- There are three sections which can be categorized as Entry Section, Critical Section, and Exit Section. The process has to make request in first section that is entry section.
- When a process enters a critical section, it must complete all instructions there in before any other process is allowed to enter the same critical section. Only the process executing the critical section is allowed to access the shared variable.
- Thus, is often referred to as **mutual exclusion,** in which a single process temporarily excludes all other from using a shared resource in order to ensure the system's integrity.
- To be acceptable as a general tool, a solution to mutual exclusion problem should:
  - Ensure mutual exclusion between processes accessing the protected shared resources.
  - Make no assumption about relative speeds and priorities of contending processes.
  - Guarantee that crashing of process outside of its critical section does not affect the ability of other contending processes to access the shared resource.
  - When more than one process wishes to enter critical section, grant entrance to one of them in finite time.

```
do
  {
     entry section
     critical section
     exit section
     remainder section
  }  while(1);
```

**Fig. 5.1: General Structure of a Typical Process Pi**

- A solution to critical section problem must satisfy the following conditions:
  1. **Mutual Exclusion:** If process P1 is executing in the critical section, no other process can be executing in the critical section.
  2. **Progress:** When 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.

3. **Bounded waiting:** There exists a bound 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.

## 5.3 SEMAPHORES [S-18, 19, 23; W-22]

### 5.3.1 Concept

- It is a method or tool which is used to prevent race condition. When co operating processes try to gain access to shared region data at that time race condition can occur so semaphore is one of the tools which are used to apply synchronization between processes and to avoid race conditions and deadlocks.
- A semaphore is a shared integer variable with non-negative values which can only be subjected to following two operations.
  1. Initialization,
  2. Invisible operations.
- The first major advance in dealing with the problems of concurrent processes came in 1965 with **Dijkstra's** solution. Dijkstra's proposal for mutual exclusion among an arbitrary number of processes is called **Semaphore.**
- A semaphore mechanism basically consists of two primitive operations SIGNAL and WAIT, (originally defined as P and V by Dijkstra), which operate on a special type of semaphore variables s.
- The semaphore variable can assume integer values, and except possibly for initialization may be accessed and manipulated only by means of the SIGNAL and WAIT operations.
- The two primitives take one argument each the semaphore variable and may be defined as follows.
  - **Wait(s):** Decrements the value of its argument semaphore, s, as soon as it would become non-negative. Completion of WAIT operation, once the decision is made to decrement its argument semaphore, must be indivisible.
  - **Signal(s):** Increments the value of its argument semaphore, s, as an indivisible operation. The logic of busy wait versions of the WAIT and SIGNAL operations is given in Fig. 5.2.

```
struct semaphore
   { int count;
    queue Type queue;
   };
void wait(semaphore s)
{ s.count--;
   if(s.count<0)
   {
   place a process P in s.queue;
    block this process.
   }
}
```

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

**Fig. 5.2: A Busy – wait Implementation of WAIT and SIGNAL**

## 5.3.2 Implementation

- The main disadvantage of semaphore is that it requires busy waiting. It waste CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called as Spinlock because the process spins while waiting for the clock. To overcome the need of busy waiting, we can modify the definition of wait() and signal() semaphore operations.
- When a process executes the wait() operation and finds that semaphore value is 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 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 wakeup() operation, which changes process from waiting to ready state. The process is then placed in ready queue.
- To implement semaphore under this definition, we have define semaphore C Struct as:

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

- Each semaphore is an integer value and a list of processes list. When a process must wait on semaphore, it is added to the list of processes.

**Semaphore Operations:**

- Wait() operation can be defined as:

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

- A signal() operation removes one process from the list of waiting processes and awakens the process.
- signal() 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() operation resumes the execution of blocked process p.

## 5.3.3 Types of Semaphores [S-19, W-22]

- Semaphores can be implemented with two types: counting semaphores and binary semaphores.

**1. Binary Semaphore:**

- In binary semaphores, the value of semaphore can range only between 0 and 1. Initially the value is set to 1 and if some process wants to use resource then the wait() function is called and the value of the semaphore is changed from 0 to 1.
- The process uses the resource and when it releases the resource then signal() function is called and the value of semaphore variable is increased to 1. If at a particular instant time the value of semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of resource by previous process.
- For both semaphores a queue is used to hold processes waiting on semaphore. The process that has been blocked the longest is released from the queue first. A semaphore whose definition includes this policy is called a Strong Semaphore.
- A semaphore that does not specify the order in which processes are removed from the queue is a Weak Semaphore.

**Counting Semaphore:**

- Counting semaphore can take an integer value ranging between 0 and an arbitrarily large number. The Initial value represents the number of units of the critical resources that are available. This is also known as general semaphore.
- A counting semaphore is helpful to coordinate the resource access, which includes multiple instances.
- In counting semaphore, there is no mutual exclusion:
- In counting semaphores, firstly, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the wait() function is called and the value of the semaphore variable is decreased by one.
- The process then uses the resource and after using the resource, the signal() function is called and the value of the semaphore variable is increased by one. So, when the value of the semaphore variable goes to 0 that is all the resources are taken by the process and there is no resource left to be used, then if some other process wants to use resources then that process has to wait for its turn. In this way, we achieve the process synchronization.

## 5.3.4 Deadlocks and Starvation [W-22]

- **Deadlocks:** The implementation of a semaphore with a waiting queue may result in a situation where more than two 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() operation.
- When such a state is reached, these processes are said to be deadlocked.
- To illustrate this approach, we consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

| P0 | P1 |
|---|---|
| wait(S); | wait(Q); |
| wait(Q); | wait(S); |
| signal(S); | signal(Q); |
| signal(Q); | signal(S); |

1. Suppose that P0 executes wait(S) and then P1 executes wait(Q).
2. When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S).
3. Since, these signal() operations cannot be executed, P0 and P1 are deadlocked.
4. 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.
5. The events with which we are mainly concerned here are resource acquisition and release.

- **Starvation:** Another problem related to deadlocks is indefinite blocking, or starvation, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (Last-In, First-Out) order.

## 5.4 CLASSICAL PROBLEMS OF SYNCHRONIZATION

- Semaphore can also be used to solve various synchronization problems. In this section, we present some classical problems of synchronization and use semaphores for synchronization in the solutions of these problems.

### 5.4.1 Bounded Buffer Problem

- Bounded buffer problem is commonly used to illustrate the power of synchronization primitives.
- In general, the producer/consumer problem may be stated as follows:
  1. Given a set of co-operating processes, some of which 'produce' data item (producers) to be consumed by others (consumers), with possible disparity between production and consumption rates.
  2. Device a synchronization protocol that allows both producers and consumers to operate concurrently as their respective service rates in such a way that produced items are consumed in the exact order in which they are produced (FIFO).

**(i) Producer and Consumer with an unbounded buffer:**

- Here, we assume that buffer is of unbounded capacity. After the system is initialized, a producer must be the first process to run in order to provide the first item. From that point on, a consumer process may run whenever there is more than one item in the buffer produced but not yet consumed. Given the unbounded buffer, producers may run at any time without restrictions. We also assume that all items produced and subsequently consumed have identical, but unspecified structure. The buffer may be implemented as an array, a linked list, or any other collection of data items. The first solution is as given in Fig. 5.3.
- Since, producer and consumer both accesses same buffer, buffer manipulation procedure must be placed within a critical section protected by a semaphore.

```
/* Program producer - consumer */
int n;
binary semaphore mutex = 1;
general_semaphore produced = 0;
void producer( )
{
     while(true)
     {
        produce( );
        wait(mutex);
```

```
            place_in_buffer;
            signal (mutex);
            signal (produced);
        }
    }
    void consumer( )
    {
        while(true)
        {
            wait(produce);
            wait(mutex);
            take_from_buffer;
            signal(mutex);
            consume;
        }
    }
    void main( )
    {
        initiate producer, consumer;
    }
```

*Fig. 5.3: Producer/Consumer Bounded Buffer*

- Consequently, a number of producers and consumers may execute concurrently. The initial value of producer is set to 0. When an item is produced, it is placed in the buffer, and the fact is signaled by means of the general semaphore PRODUCED. The nature of the problem implies that the consumer can never get ahead of the producer, since consumer is waiting on PRODUCED semaphore. When item is produced then only consumer can enter in critical section by applying wait on mutex semaphore.
- Adding two semaphores must be handled with care because two semaphores may interact in an undesirable manner. Consider for e.g. reversing the order of WAIT operations in consumer process. As a consequence, waiting on PRODUCED semaphore is moved into critical section controlled by MUTEX. This in turn, may deadlock the system from very start. For instance assume that where a producer is busy preparing in first item, a consumer process becomes scheduled. MUTEX is initially FREE and consumer enters in critical section. It has no item to consume and it is forced to wait on PRODUCED semaphore. However no producer can ever succeed in placing its item in buffer since MUTEX is busy. Consequently consumer remains in critical section forever and system is deadlocked. This tells that although semaphores are powerful tool, their use by no means automatically solve all timing problems.

**(ii) Producers and Consumers with a bounded buffer:**

- The unbounded - buffer assumption simplifies analysis of producer - consumer problem by allowing unrestricted execution of producers. However this assumption unrealistic since, computer system, which have finite memory capacity.
- The main difference imposed by bounded buffer is that both consumer and producer may be halted under certain circumstances. At any particular time the shared global buffer may be empty, partially filled or full. A producer process may run in either of two former cases, but all producers must be kept waiting when buffer is full similarly when buffer is empty consumer must wait.
- Let icount be the number of items produced but not yet consumed so,

  icount = produced - consumed.

  If we have finite capacity then,

  0 < icount < capacity.

  Since, producer may run only when there are some empty slots in buffer it can be said,

  mayproduce: icount < capacity.

  Consumers can execute only when there is at least one item produced but not yet consumed i.e.

  mayconsume: icount>0.
- In practice, buffers are usually implemented in circular fashion, using linked list. Two indices in and out, point to next slot available for a produced item and to the place where the next item is to be consumed from respectively.

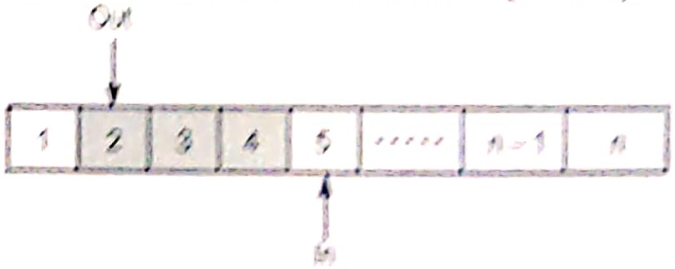


**Fig. 5.4: Producer/Consumer Buffer**

- **Fig. 5.5** shows a solution to bounded buffer producer/consumer problem. General semaphores mayproduce and mayconsume represent two conditions introduced earlier to control execution of producer and consumer. Two binary semaphores pmutex and cmutex protect buffer and index manipulations of producers and consumers.

```
buffer → array [1 - capacity];
Semaphore mayproduce, mayconsumer; /* general */
binary semaphore pmutex, cmutex;
void producer( )
{
   item pitem;
```

```
    while(true)
    {
        wait(mayproduce);
        pitem=produce;
        wait(pmutex);
        buffer[in] = pitem;
        In=(in mod capacity) +1;
        Signal(pmutex);
        Signal(mayconsume);
    }
}
void consumer( )
{
    item citem;
    {
    while(true)
    {
    wait(mayconsume);
    wait(cmutex);
    citem = buffer[out];
    out=(out mod capacity) +1;
    signal(cmutex);
    signal(mayproduce);
    consume (item);
    }
    }
void paraent process( )
{
    in = 1;
    out = 1;
    signal(pmutex);
    signal(cmutex);
    mayconsume = 0;
    for(i=1 to capacity)
    {signal(mayproduce);
}
}
```

**Fig. 5.5: Producer / Consumer Bounded Buffer**

- Initially *mayproduce* is set to capacity to indicate producer can produce that many item. Whenever consumer completes its cycle, it implies a slot by removing the consumed item from buffer and signals the fact via the *mayproduce*. The *mayconsume* semaphore indicates availability of produced items, and it functions much the same way as in unbounded buffer.

## 5.4.2 Readers and Writers Problem [S-18, 19]

- Readers and writers is another classical problem in concurrent programming. It basically resolves around a number of processes using a shared global data structure.
- The processes are categorized depending on their usage of the resource, as either readers or writers.
- A reader never modifies the shared data structure, whereas a writer may both read it and write into it. A number of readers may use the shared data structure concurrently because no matter how they are interleaved, they cannot possibly compromise its consistency.
- Writers, on the other hand, must be granted exclusive access to data.
- The problem may be stated as follows:
  - Given a universe of readers that read a common data structure and a universe of writers that modify the same common data structure.
  - Device a synchronization protocol among the readers and writers that ensure consistency of common data while maintaining as high a degree of concurrency as possible.
- One approach to solve readers/writers problem is as shown in Fig. 5.6. The writer process waits on binary semaphore WRITE to grant the permission to enter the critical section and to use the shared resources.
- A reader, on the other hand, goes through two critical sections, one before and one after using the resource. Their purpose is to allow a large number of readers to execute concurrently while making sure that readers are kept away when writers are active.
- An integer, READERCOUNT is used to keep track of the number of readers actively using the resource.
- In Fig. 5.6, first reader passes through MUTEX, increments the number of readers and waits on writers if any. While reader is reading data, semaphore MUTEX is free and WRITE is busy to allow multiple readers only.
- If there are writers, waiting they are prevented by busy WRITE semaphore. When READERCOUNT reaches to zero writers can enter in critical section.
- Even if this solution has high degree of concurrency it faces starvation of writer by postponing them indefinitely when readers are active.
- A strategy proposed by Hoare (1974) holds promise for both readers and writers to complete in finite time. It suggests that,
  - A new reader should not start if there is writer waiting, (prevent starvation of writer).
  - All readers waiting at the end of a write should have priority over next writer, (prevents starvation of readers).

```
/* Program readers_ writers*/
int readercount;
binary_semaphore mutex, write;
void reader( )
{
     while(true)
     {
        wait(mutex);
        readercount ++;
        If(readercount==1)
        wait(write)
        signal(mutex);
        /* read data */
        wait(mutex)
        readercount - -;
        if(readercount==0)
        signal(write);
     }
  }
  void writer()
  {
     while(true)
     {
        wait (write);
        ..............
        /* write data */
        signal(write)
     }
}
void parentprocess()
{
     readercount=0;
     signal(mutex);
     signal(write);
     initiate readers, writers
  }
```

**Fig. 5.6: Readers/Writers**

## 5.4.3 Dining Philosophers Problem [W-18, 22]

- To understand the Dining Philosopher's problem, Consider five philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of table is a bowl of rice, and the table is laid with five single chopsticks (Fig. 5.7).

- 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 neighbor).
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already on the hand of a neighbour.
- 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 start thinking again.

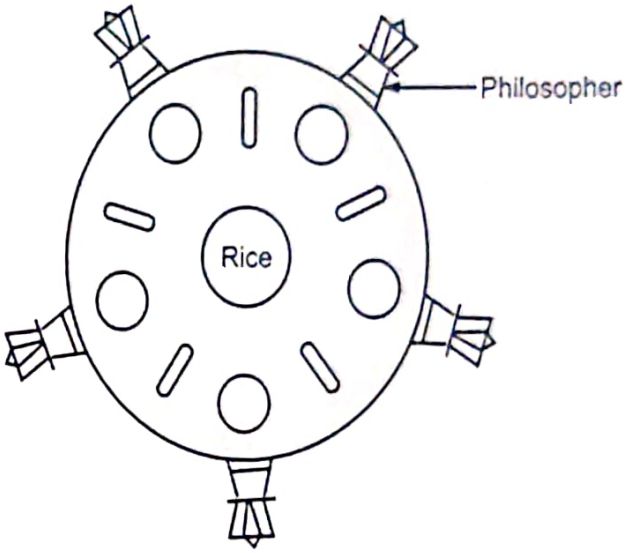


**Fig. 5.7: The Situation of Dining Philosophers**

- The Dining Philosopher problem is considered a classic synchronization problem.
- One simple solution is to represent each chopstick of a semaphore.
- A philosopher tries to grab the chopstick by executing a wait operation on that semaphore. She releases her chopsticks by executing the signal operation on the appropriate semaphores.
- Thus, the shared data are,

```
Semaphore chopsticks[5];
```

- Where all the elements of chopsticks are initialized to 1. Structure of philosopher i is shown in Fig. 5.8.

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

**Fig. 5.8: Structure of Philosopher**

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick.
- All elements of chopstick will now be equal to 0. When each tries to grabs right chopstick she will be delayed forever. We can solve this by allow to pick chopsticks only if both are available.

## Summary

- Process synchronization means sharing resources by process in such a way that no two processes can share data and resources at the same time. There are two kinds of synchronization: Control synchronization and Data access synchronization.
- 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.
- Critical section is the part of program which tries to access shared resource, that resource may be any resource like memory location, data structure, CPU or any I/O device.
- The critical section cannot be executed more than one process at same time.
- The critical section problem is used to design a set of protocols which can ensure that the race condition among the processes will never arise.
- Semaphore is defined as a variable that is non-negative and shared between threads. It is a mechanism that can be used to provide synchronization of tasks.
- Counting semaphore uses a count that helps task to be acquired or released numerous times.
- The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1. Wait operation helps you to control the entry of a task into the critical section.
- Signal semaphore operation is used to control the exit of a task from a critical section. Counting Semaphore has no mutual exclusion whereas Binary Semaphore has Mutual exclusion Semaphore means a signaling mechanism whereas Mutex is a locking mechanism.
- Semaphore allows more than one thread to access the critical section. One of the biggest limitations of a semaphore is priority inversion.
- Classical problems of synchronization we had discusses are bounded buffer problem, readers-writers problem and dining philosopher problems.

## Check Your Understanding

1. Which of the following processes can get affected by other process executing in the system?
   (a) child process (b) parent process
   (c) independent process (d) co operating process

2. When process is executing in critical section, no other process can execute in critical section, this condition is called?
   (a) critical section (b) logical section
   (c) mutual exclusion (d) non mutual exclusion
3. Amongst the following which can be considered as synchronization tool?
   (a) socket (b) mutex
   (c) semaphore (d) thread
4. Select the correct statement regarding mutex lock to prevent race condition.
   I. A process must acquire the lock before entering critical section.
   II. A process need to acquire the lock before entering critical section.
   III. It releases the lock when it exits the critical section.
   IV. A process must acquire the lock when it exits critical section
   (a) 1, 4 (b) 2, 4
   (c) 2, 3 (d) 1, 3
5. In the spinlocks _____.
   (a) no context switch is required when a process must wait on a lock
   (b) locks are expected to be held for short times
   (c) employed on multiprocessor systems
   (d) all of above
6. Indefinite blocking or starvation is problem related with _____.
   (a) mutex locks (b) deadlocks
   (c) spinlocks (d) none of the above
7. Which of the following is not classical problem of synchronization?
   (a) reader writer problem (b) bounder buffer problem
   (c) dining philosopher problem (d) customer employer problem
8. A semaphore is a shared integer variable that can not _____.
   (a) go beyond zero (b) more than zero
   (c) it can not more than 1 (d) none of above
9. Process synchronization can be done with following levels.
   (a) hardware (b) software
   (c) both (d) none of above
10. Deadlock can be avoided by _____.
   (a) resource allocation must be done at once
   (b) there must be fixed number of resources to allocate
   (c) all deadlocked processes must be aborted
   (d) inversion technique can be used
11. Which is not part of critical section?
   (a) entry section (b) critical section
   (c) exit section (d) waiting section

## Answers

| 1. (d) | 2. (c) | 3. (c) | 4. (d) | 5. (d) | 6. (b) | 7. (d) | 8. (a) | 9. (c) | 10. (b) | 11. (d) |
|---|---|---|---|---|---|---|---|---|---|---|

## Practice Questions

**Q.I Answer the following questions in short:**

1. What is process synchronization?
2. What is mutual exclusion?
3. What is meant by binary semaphore?
4. What is meant by deadlock and starvation?
5. List all types of classical problems of synchronization.

**Q.II Answer the following questions:**

1. Describe Critical Section problem in detail.
2. Explain Readers and Writers problem in detail.
3. With suitable example describe bounded buffer problem in detail.
4. What is semaphore? Explain with its types.
5. Explain bounded buffer problem of synchronization in detail.
6. Explain in detail Dining Philosopher problem.

**Q.III Define terms:**

1. Strong semaphore
2. Weak semaphore
3. Synchronization
4. Deadlocks
5. Starvation

## Previous Exam Questions

### Summer 2018

**1.** What is Semaphores? **[2 M]**

**Ans.** Refer to Section 5.3.

**2.** Describe solution for critical section problem. **[4 M]**

**Ans.** Refer to Section 5.2.

**3.** Explain the readers and writes problem which is a classical problem of synchronization. **[4 M]**

**Ans.** Refer to Section 5.4.2.

### Winter 2018

**1.** Describe in detail the 'Dinning Philosopher Problem' synchronization problem. **[4 M]**

**Ans.** Refer to Section 5.4.3

### Summer 2019

**1.** Define critical section problem and list its solutions. **[2 M]**

**Ans.** Refer to Section 5.2.

**2.** Explain Reader's writer's problems. **[4 M]**

**Ans.** Refer to Section 5.4.2.

**3.** Explain semaphores and its types in detail. **[4 M]**

**Ans.** Refer to Sections 5.3.1 and 5.3.3.

**4.** Explain WAIT and SIGNAL semaphore operations. **[4 M]**

**Ans.** Refer to Section 5.3.1.