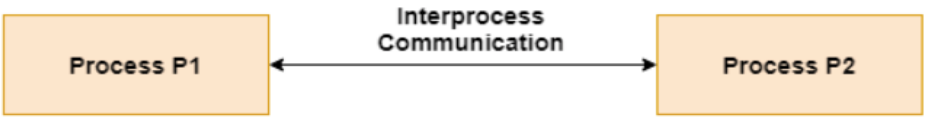
Unit 3

Interprocess Communication , Synchronization and Deadlock

3.1 Interprocess communication :

- It is the mechanism provided by the operating system that allows processes to communicate with each other synchronize their actions.
- This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

A diagram that illustrates interprocess communication is as follows −



In multiprogramming environment , multiple processes run simultaneously

3.1.1 Types of Process

process can be of two types:

- Independent process.
- Co-operating process.

Independent process

- An independent process is a process which executes independently without interference of any other process.
- It is not affected by the execution of other processes.

Co-operating process

- Cooperative processes share some logical or physical resources amongst them.
- Co-operating process can be affected by other executing processes.

- Cooperating processes can either directly share a logical address space (that is, both code and data) Eg. Sharing of variable, memory, file, --- logical addresses and resources such as CPU, Printer etc.
- **For Cooperative Processes Synchronization is very important.**

3.1.2 Types of Communication between cooperate processes

Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

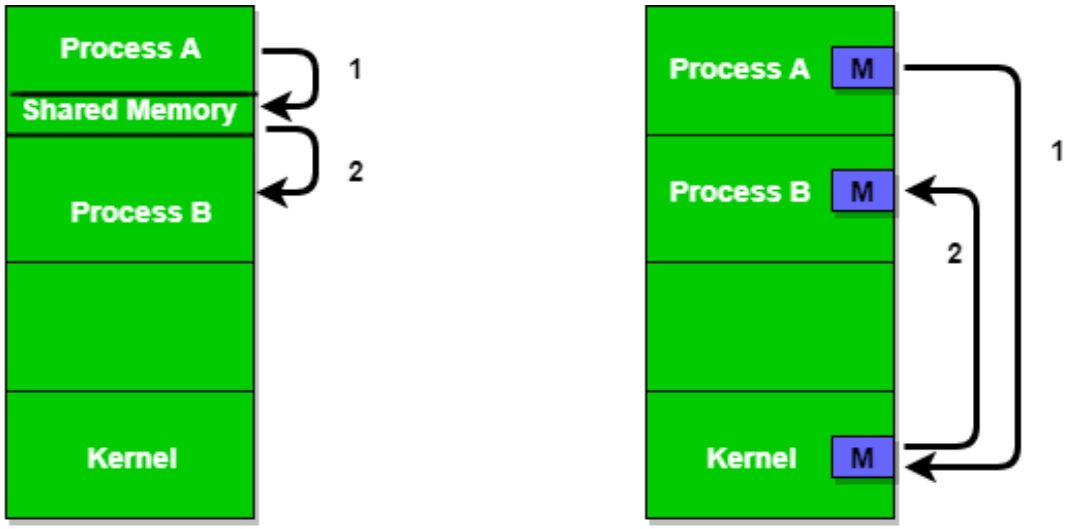


Figure 1 - Shared Memory and Message Passing

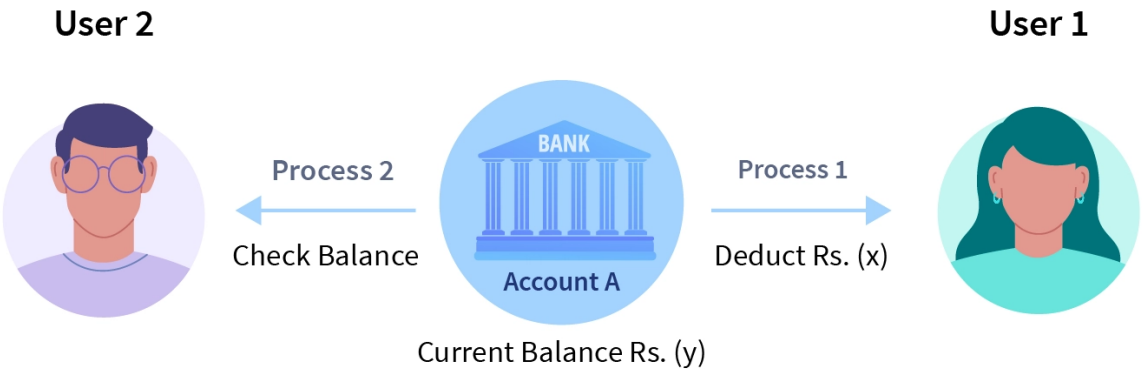
An operating system can implement both methods of communication.

Shared Memory :

Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

Let’s discuss an example of communication between processes using the shared memory method.

For ex. 1.



If a process2 is trying to read the data present in a memory location while another process1 is trying to change the data present at the same location, there is a high chance that the data read by the process2 will be incorrect.

2.

P1	**P2**
Int X_shared =5;	**Int X_shared =5;**
A = X_shared	**B = X_shared;**
A = A + 1;	**B = B - 1**
Sleep(1)	**Sleep(1)**
X_shared =A	**X_shared=B**

i) Global (Standard)Example Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. The producer produces some items and the Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed. There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then the producer will start producing items. If the total produced

item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, Consumer will consume them.

(Note : Write Example of Counter variable and register here from OS , Silberschatz Book, Already explain in class)

Message passing : Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages. A message-passing facility provides at least two operations: send(message) and receive(message)

In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

Establish a communication link (if a link already exists, no need to establish it again.)
Start exchanging messages using basic primitives.
We need at least two primitives:
– **send**(message, destination) or **send**(message)
– **receive**(message, host) or **receive**(message)

The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: **header and body.**

The **header part** is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Message Passing through Communication Link.
Direct and Indirect Communication link .

In Direct message passing, The process which wants to communicate must explicitly name the recipient or sender of the communication.
e.g. **send(p1, message)** means send the message to p1.
Similarly, **receive(p2, message)** means to receive the message from p2.
In this method of communication, the communication link gets established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of links.
Symmetry and asymmetry between sending and receiving can also be implemented i.e. either both processes will name each other for sending and receiving the messages or only the sender will name the receiver for sending the message and there is no need for the receiver for naming the sender for

receiving the message. The problem with this method of communication is that if the name of one process changes, this method will not work.

In Indirect message passing, processes use mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these links may be unidirectional or bi-directional. Suppose two processes want to communicate through Indirect message passing, the required operations are: create a mailbox, use this mailbox for sending and receiving messages, then destroy the mailbox. The standard primitives used are: **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**. There is a problem with this mailbox implementation. Suppose there are more than two processes sharing the same mailbox and suppose the process p1 sends a message to the mailbox, which process will be the receiver? This can be solved by either enforcing that only two processes can share a single mailbox or enforcing that only one process is allowed to execute the receive at a given time or select any process randomly and notify the sender about the receiver. A mailbox can be made private to a single sender/receiver pair and can also be shared between multiple sender/receiver pairs. Port is an implementation of such mailbox that can have multiple senders and a single receiver. It is used in client/server applications (in this case the server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver processor when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion.

Synchronous and Asynchronous Message Passing:

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that for a sender it is more natural to be non-blocking after message passing as there may be a need to send the message to different processes. However, the sender expects acknowledgment from the receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution. At the same time, if the message send keep on failing, the receiver will have to wait indefinitely. That is why we also consider the other possibility of message passing. There are basically three preferred combinations:

3.2 Problem Arises in Cooperative Process

3.2.1 Race Condition

When more than one process is either running the same code or modifying the same memory or any shared data, there is a risk that the result or value of the shared data may be incorrect because all processes try to access and modify this shared resource. Thus, all the processes race to say that my result is correct. This condition is called the **race condition.**

To avoid this, process need to be synchronized.

3.2.2 Process Synchronization

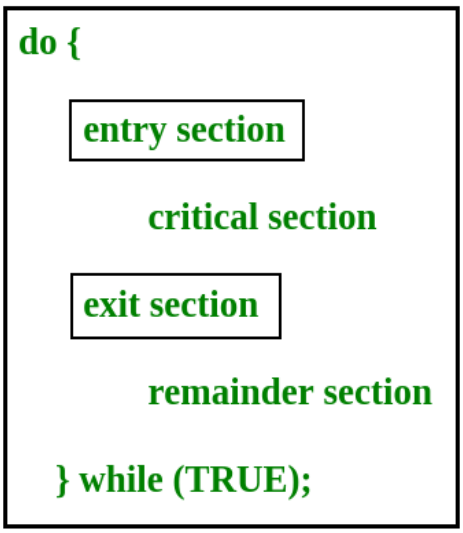
- It is the mechanism to coordinate the execution of processes such that no two processes access the same shared resources and data.
- It is required in a multi-process system where multiple processes run together, and more than one process tries to gain access to the same shared resource or data at the same time.
- Changes made in one process aren't reflected when another process accesses the same shared data.
- It is necessary that processes are synchronized with each other as it helps avoid the inconsistency of shared data.

3.2.3 Critical Section Problem

- A segment of code that a single process can access at a particular point of time is known as the critical section.
- This means that when a lot of programs want to access and change a single shared data, only one process will be allowed to change at any given moment.
- The other processes have to wait until the data is free to be used.

So the critical section problem means designing a way for cooperative processes to access shared resources without creating data inconsistencies.

Basic Structure of Critical section is given as follows:



Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

different elements/sections:

- **Entry Section:** The entry Section decides the entry of a process.
- **Critical Section:** Critical section allows and makes sure that only one process is modifying the shared data.
- **Exit Section:** The entry of other processes in the shared data after the execution of one process is handled by the Exit section.
- **Remainder Section:** The remaining part of the code which is not categorized as above is contained in the Remainder section.

A solution to the critical-section problem or for synchronization must satisfy the following three requirements:

1. Mutual exclusion. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. 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.

3. 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.

4.No Assumption related to hardware or speed.

Code to Explain critical Section(Executed on Ubuntu OS)

1. Without critical section block

```
#include<stdio.h>
#include<omp.h>

void main()
{
int x=0;

#pragma omp parallel num_threads(300)
{
x=x+1;
}

printf("x=%d\n",x);
}
```

2. With critical section Block.

```
#include<stdio.h>
#include<omp.h>

void main()
{
int x=0;

#pragma omp parallel num_threads(300)
{
#pragma omp critical
{
x=x+1;
```

```
}

}

printf("x=%d\n",x);
}
```

Output :Here, we got "x=300" because at one instance only one thread was incrementing the value of x.

3.3 Peterson's Solution(Software Solution to Critical Section(for Process synchronization)

1. It is a classical software-based solution.
2. It provides a good algorithmic solution to solve the critical section problem and illustrate some of the complexities that need to address such as mutual exclusion, progress and bound wait in designing the software.

Peterson solution restricted to the processes that alternate execution between their critical section and remainder section.

Petersons solution required to be shared two data items to be shared between the processes.

Eg.

Int turn -→ Indicates whose turn it is to enter the critical section

Boolean flag[2]→ used to indicate if a process is ready to enter into critical section.

For process i

```
int turn;
boolean flag[2]
while (true)
 {
 flag[i] = true;
turn = j;
while (flag[j] && turn == j) ;
/* critical section */
flag[i] = false;
```

For process j

```
int turn;
boolean flag[2]
while (true) {
 flag[j] = true;
turn =i;
while (flag[i] && turn == i) ;
/* critical section */
flag[i] = false;
 /*remainder section */ }
```

3.3 Hardware Support for Synchronization

(Following text in blue only for your information and knowledge ,not imp exam point of view)

1. As discussed, software-based solutions are not guaranteed to work on modern computer architectures.

2. three hardware instructions can be used that provide support for solving the critical-section problem.

3. These primitive operations can be used directly as synchronization tools, or they can be used to form the foundation of more abstract synchronization mechanisms.

4. Memory Barriers

a system may reorder instructions, a policy that can lead to unreliable data states.

In general, a memory model falls into one of two categories:

1. Strongly ordered, where a memory modification on one processor is immediately visible to all other processors.

2. Weakly ordered, where modifications to memory on one processor may not be immediately visible to other processors. Memory models vary by process

Memory models vary by processor type, so kernel developers cannot make any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor. To address this issue, computer architectures provide instructions that can force any changes in memory to be propagated to all other processors, thereby ensuring that memory modifications are visible to threads running on other processors. Such instructions are known as memory **barriers or memory fences.**

When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed. Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

use a memory barrier to ensure that we obtain the expected output. If we add a memory barrier operation to

Thread 1

while (!flag)

memory barrier();

print x;

we guarantee that the value of flag is loaded before the value of x. Similarly, if we place a memory barrier between the assignments performed by

Thread 2

x = 100;

memory barrier();

flag = true;

we ensure that the assignment to x occurs before the assignment to flag.

memory barriers are considered very low-level operations and are typically only used by kernel developers when writing specialized code that ensures mutual exclusion.

3.4 Hardware Solution

Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically— that is, as one uninterruptible unit.

Test and Set lock

1.It is hardware solution to synchronization problem.

2.There is a shared lock variable which can take either of the two values, 0 or 1

shared lock variable =0 means unlocked.

3.Before entering into the critical section, a process inquires about the lock.

4. If it is not locked , it takes the lock and executes the critical section.

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

Target ==lock variable

Process 1

```
do
{
while (test and set(&lock)) ;
```

```
/* do nothing */
/* critical section */
lock = false;
 /* remainder section */
} while (true);
```

Process 2

```
do
{
while (test and set(&lock)) ;
/* do nothing */
/* critical section */
lock = false;
 /* remainder section */
} while (true);
```

Intial value of lock = 0;

(NOTE:Explain the execution for both of these processes in your words)

3.5 Mutex Locks

- **Mutex word comes from Mutual Exclusion.**
- operating-system designers build higher-level software tools to solve the critical-section problem.
- The simplest of these tools is the mutex lock.
- We use the mutex lock to protect critical sections and thus prevent race conditions.
- That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.
- The acquire()function acquires the lock, and the release() function releases the lock.

For Eg.

while (true)

{

acquire lock

critical section

release lock

remainder section

}

A mutex lock has a boolean variable available whose value indicates if the lock is available or not.

Boolean variable available is shared between the processes and initially it is true.

The definition of `acquire()` is as follows:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}
```

The definition of `release()` is as follows:

```
release() {
   available = true;
}
```

A mutex lock has a boolean variable available whose value indicates if the lock is available or not.

If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable.

A process that attempts to acquire an unavailable lock is blocked until the lock is released.

Acquire and Release both are atomic functions, atomic functions means instructions of both of these function execute only once without any context switch or preemption .

Mutex Locks Execution for Two Processes

Initially available = true;

Process P1	Process P2

acquire()
{
while (!available) ; /* busy wait
*/
 	
available = false;
}
 	
release()
{
available = true;
 	
}
	acquire()
	{
	while (!available) ; /* busy wait
	*/
	
	available = false;
	}
	
	release()
	{
	available = true;
	
	}

(NOTE:Explain the execution for both of these processes in your words)

Disadvantages of Mutex Lock

- The main disadvantage is busy waiting.
- When one process is in critical section , another process tries to enter in the critical section must loop continuously in the call to acquire( ) function.
- This type of mutex lock is also called as spin lock, because the process "spins" while waiting to become the process available.
- This continuous looping is really a problem in multiprogramming system where the single CPU is shared among many processes.
- Busy waiting wastes the CPU cycle while other processes might be able to use it productively.

Advantages of Mutex Locks

- Spinlocks do have an advantage,
- however, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.
- In certain circumstances on multicore systems, spinlocks are in fact the preferable choice for locking.
- If a lock is to be held for a short duration, one thread can "spin" on one processing core while another thread performs its critical section on another core.
- On modern multicore computing systems, spinlocks are widely used in many operating systems.

3.6 Semaphores

- A more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().
- Semaphores were introduced by the Dutch computer scientist Edsger Dijkstra, and such, the wait() operation was originally termed P (from the Dut proberen, "to test"); signal() was originally called V (from verhogen, "to increment").
- A semaphore is an integer maintained by the kernel whose value is restricted to being greater than or equal to 0.

- The definition of wait() is as follows:

```
wait(S)
 {
while (S <= 0) ;
// busy-wait
```

```
 S--;
}
```

The definition of signal() is as follows:

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

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Types of Semaphore

1.Binary Semaphore: The value of a binary semaphore can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

2.Counting Semaphore: The value of a counting semaphore can range over an unrestricted domain.

3. Counting semaphores can be used to control access to a given resource consisting of a finite number of instances

Semaphore Implementation

Implementation of mutex locks suffers from busy waiting.

The definitions of the wait() and signal() semaphore operations just described present the same problem.

To overcome this problem, 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 suspend itself.

The suspend 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 suspended, 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. When a process must wait on a semaphore, it is added to the list of processes.

A signal() operation removes one process from the list of waiting processes and awakens that process.

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

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

and the signal() semaphore operation can be defined as

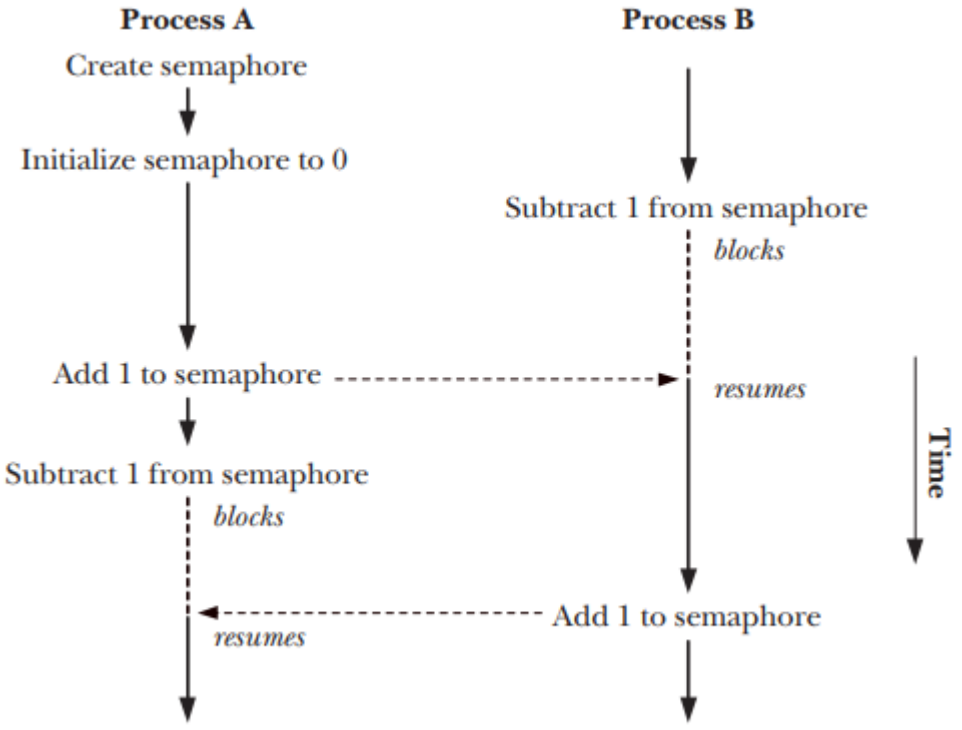
```
signal(semaphore *S)
{
S->value++;
if (S->value <= 0)
{
remove a process P from S->list;
wakeup(P);
 }
}
```

(NOTE:Explain the execution for multiples processes in your words)

There are different operations that can be performed on a semaphore. They are :

- setting the semaphore to an absolute value.
- adding a number to the current value of the semaphore.
- subtracting a number from the current value of the semaphore.
- waiting for the semaphore value to be equal to 0.

Here is good flowchart that describes the synchronization of two processes using semaphores.



There are two API's for implementing the semaphores. They are System V semaphores and POSIX semaphores. The System V semaphores are the older versions and the POSIX implementation is the newer one.

(NOTE: Following text in blue is for information and knowledge , not imp exam point of view)

System V model :

The general steps for using a semaphore of System V model are:

- create or open a semaphore set using **semget()** system call.
- initialize the semaphores in the set using the **semctl()** system call.
- perform operations on semaphore values using **semop()** system call.
- when all processes have finished using the semaphore set, remove the set using the **semctl()**.
-

POSIX model :

There are two types of **POSIX** semaphores.

- **Named** : this type of semaphore has a name. Unrelated processes can access the semaphores. To work with the named semaphores use the following calls.
  - **sem_open()** function opens or creates a semaphore and initializes it.
  - **sem_post(sem)** and **sem_wait(sem)** functions increment and decrements a semaphores value.
  - **sem_getvalue()** function retrieves the current value of the semaphore.
  - **sem_close()** function removes the calling process's association with a semaphore that it previously opened.
  - **sem_unlink()** function removes a semaphore name and marks the semaphore for deletion when all processes have closed it.
- **Unnamed** : this type of semaphore doesn't have a name and it exists at an agreed location in memory. The operations on unnamed semaphores use the same functions that used to operate on named semaphores. However, there are two additional functions that are required :
  - **sem_init()** function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process.
  - **sem_destroy(sem)** function destroys a semaphore.

Steps to use semaphore

1. Include semaphore.h
2. Compile the code by linking with -lpthread -lrt

To lock a semaphore or wait we can use the **sem_wait** function:
int sem_wait(sem_t *sem);

To release or signal a semaphore, we use the **sem_post** function:
int sem_post(sem_t *sem);

A semaphore is initialised by using **sem_init**(for processes or threads) or **sem_open** (for IPC).
sem_init(sem_t *sem, int pshared, unsigned int value);

Question based on this topic

1. Define Interprocess communication

2. What are the various types of Processes?

3. Explain the types of Communication between processes.

4. Explain process communication through shared memeory.

5.Expalin Process communication through message passing.

6. Discuss Producer and Consume problem with example.

7. Discuss the problem arises in cooperative processes.

8. What is race condition?

9. What is the solution to race condition?

10.What is critical Section ? Why it is needed? Explain with example.

11. Explain Process synchronization

12. What are the requirements need to be stasified to implement solution to critical section problem?

13. Explain Peterson's Solution to critical Section (or Process Synchronization)

14.Discuss hardware solution to implement critical section problem?

15.Explain TSL (TEST and SET LOCK) instruction for critical section problem.

16. What are various software solutions to provide solution to critical section problem?

(Ans:PTerson's, Mutex Locks, Semaphore Explain these in detail, if question ask for more marks(Ans depends on marks)

17. Describe the working of Mutex Lock. What are the advantages and disadvantages of Mutex lock?

18.Explain Semaphore in detail with example. What are the types of Semaphore.