Operating System
KCS – 401

CONCURRENT PROCESSES AND CLASSICAL PROBLEM IN CONCURRENCY

Dr. Pankaj Kumar
Associate Professor – CSE
SRMGPC Lucknow

Outline of the Lecture

Process Concept

Process in memory

Process state

Principle of Concurrency

Process Concept

An operating system executes a variety of programs:

- Batch system – **jobs**
- Time-shared systems – **user programs** or **tasks**

Textbook uses the terms ***job*** and ***process*** almost interchangeably

Process – a program in execution; process execution must progress in sequential fashion

Multiple parts

Program is ***passive*** entity stored on disk (**executable file**), process is ***active***

- Program becomes process when executable file loaded into memory

Execution of program started via GUI mouse clicks, command line entry of its name, etc

One program can be several processes

- Consider multiple users executing the same program

Process in Memory

Multi Parts of OS

The program code, also called **text section**

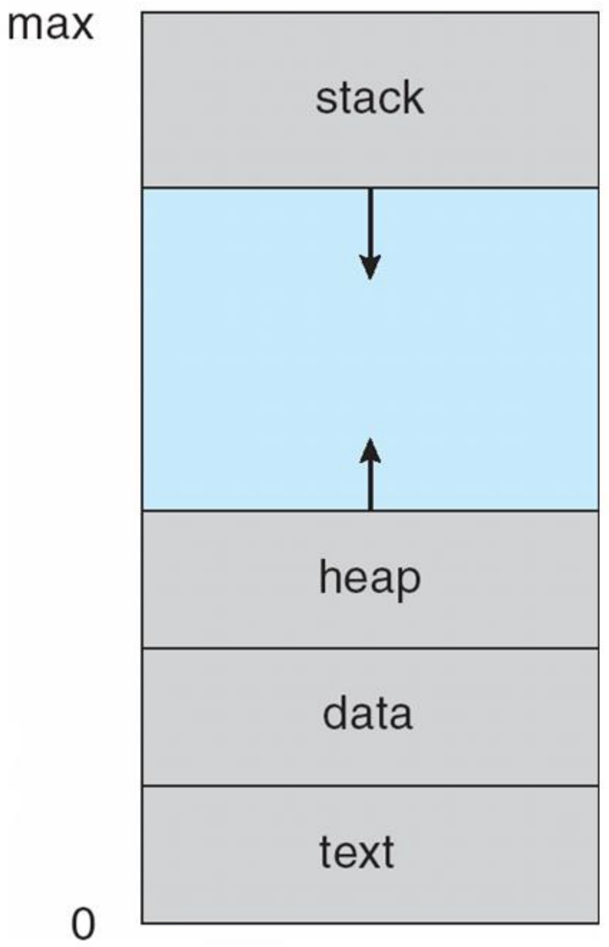
Current activity including **program counter**, processor registers

Stack containing temporary data

Function parameters, return addresses, local variables

Data section containing global variables

Heap containing memory dynamically allocated during run time

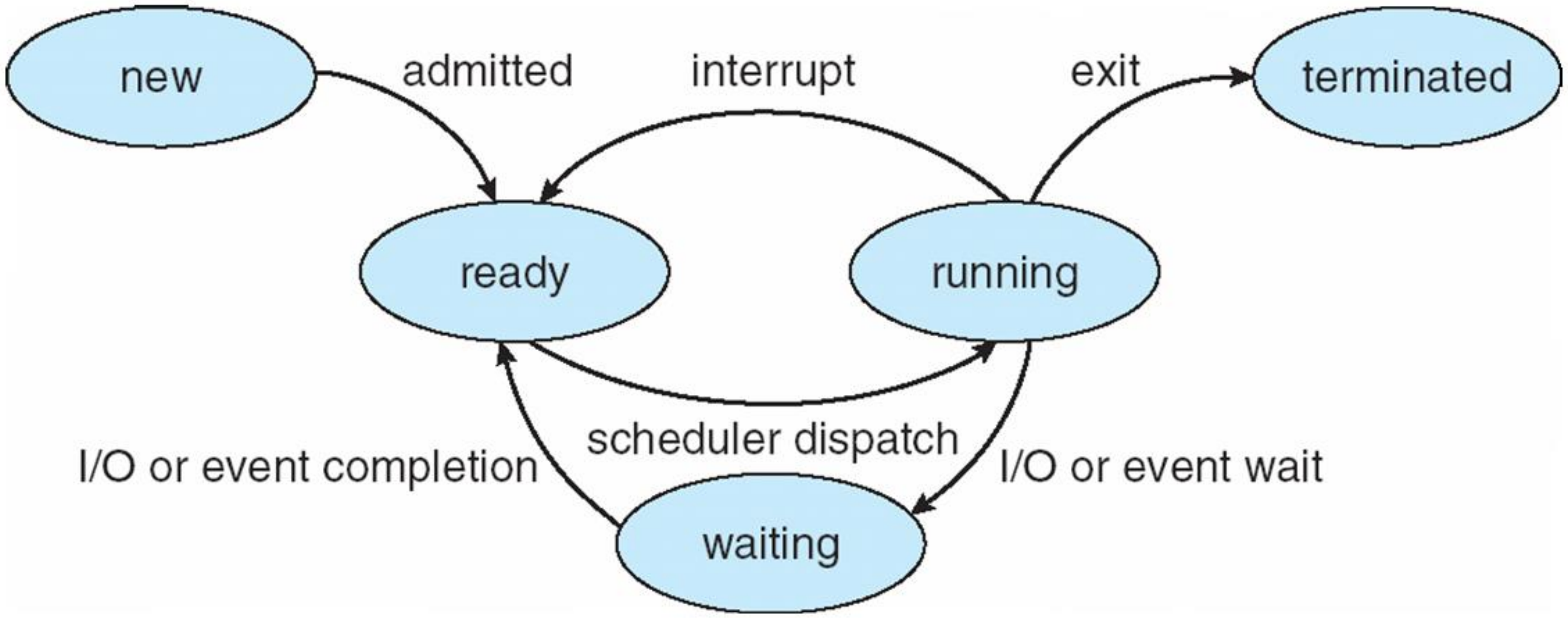


Process State

As a process executes, it changes **state**

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution

Process State



Concurrency

Concurrency is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel. The running process threads always communicate with each other through shared memory or message passing. Concurrency results in sharing of resources result in problems like deadlocks and resources starvation.

It helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput.

Principles of Concurrency :

Both interleaved and overlapped processes can be viewed as examples of concurrent processes, they both present the same problems.

The relative speed of execution cannot be predicted. It depends on the following:

- The activities of other processes
- The way operating system handles interrupts
- The scheduling policies of the operating system

Process Synchronisation

Process Synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources.

It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time.

This can lead to the inconsistency of shared data. So the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other.

Process Synchronisation

On the basis of synchronization, processes are categorized as one of the following two types:

Independent Process : Execution of one process does not affects the execution of other processes.

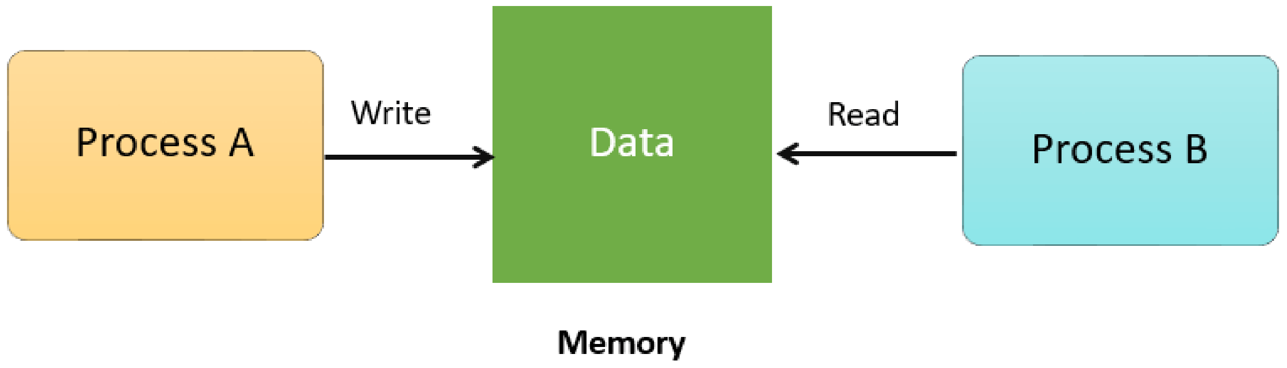
Cooperative Process : Execution of one process affects the execution of other processes.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.

Process Synchronisation

How Process Synchronization Works?

For Example, process A changing the data in a memory location while another process B is trying to read the data from the **same** memory location. There is a high probability that data read by the second process will be erroneous.



Process Synchronisation

Race Condition

When more than one processes are executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition. Several processes access and process the manipulations over the same data concurrently, then the outcome depends on the particular order in which the access takes place.

Producer-Consumer Problem

In this problem, we have one producer and one consumer. The producer is the one who produces something, and the consumer is the one who consumes something, produced by the producer. The producer and consumer both share the common memory buffer, and the memory buffer is of fixed-size.

The task performed by the producer is to generate the data, and when the data gets generated, and then it put the data into the buffer and again generates the data. The task performed by the consumer is to consume the data which is present in the memory buffer.

Problem

1. At the same time, the producer and consumer cannot access the buffer.
2. The producer cannot produce the data if the memory buffer is full. It means when the memory buffer is not full, then only the producer can produce the data.
3. The consumer can only consume the data if the memory buffer is not vacant. In a condition where memory buffer is empty, the consumer is not allowed to take data from the memory buffer.

Process Synchronisation

Sections of a Program

Here, are four essential elements of the critical section:

- **Entry Section:** It is part of the process which decides the entry of a particular process.

- **Critical Section:** This part allows one process to enter and modify the shared variable.

- **Exit Section:** Exit section allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It also checks that a process that finished its execution should be removed through this Section.

- **Remainder Section:** All other parts of the Code, which is not in Critical, Entry, and Exit Section, are known as the Remainder Section.

Critical Section Problem

What is Critical Section Problem?

A critical section is a segment of code which can be accessed by a signal process at a specific point of time. The section consists of shared data resources that required to be accessed by other processes.

The entry to the critical section is handled by the wait() function

The exit from a critical section is controlled by the signal() function

In the critical section, only a single process can be executed. Other processes, waiting to execute their critical section, need to wait until the current process completes its execution.

```
do {

    entry section

        critical section

    exit section

        remainder section

  } while (TRUE);
```

Critical Section Problem

Rules/Solution for Critical Section

The critical section need to must enforce all three rules:

Mutual Exclusion: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.

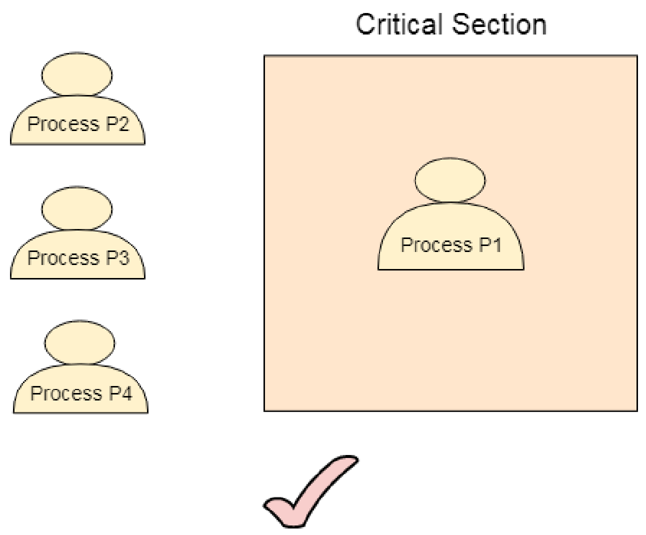
Progress: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.

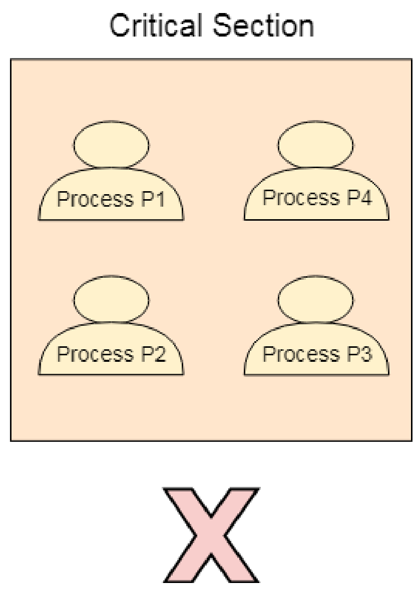
Bound Waiting: When a process makes a request for getting into critical section, there is a specific limit about number of processes can get into their critical section. So, when the limit is reached, the system must allow request to the process to get into its critical section.

Critical Section Problem

Mutual Exclusion:

if one process is executing inside critical section then the other process must not enter in the critical section.





Critical Section Problem

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Critical Section Problem

Solution

Two Process Solution

Algorithm 1:

For process Pi

```
do
{
   while turn !=i;
     critical section
    turn=j;
    remainder section
}
while(1)
```

The variable **turn** indicates whose turn it is to enter the critical section.

The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] =** ***true*** implies that process P_i ***is ready!***

Algorithm 2:

For process Pi

```
do
{
   flag[i]=true
       while(flag[j]);
     critical section
      flag[i]=false
    remainder section
}
while(1)
```

Critical Section Problem

Peterson's Solution

Two Process Solution

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either **flag[j] = false** or **turn = i**

2. Progress requirement may be satisfied
3. Bounded-waiting requirement is met

For process Pi

```
do
{
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);
                critical section
        flag[i] = false;
                remainder section
}
while (true);
```

Critical Section Problem

Dekker's Solution

Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   P$_i$ enters CS only if:

   either **flag[j] = false** or **turn = i**

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

For process Pi

```
repeat

    flag[i] := true;
    while flag[j] do
        if turn = j then
        begin
            flag[i] := false;
            while turn = j do no-op;
            flag[i] := true;
        end;

        critical section

    turn := j;
    flag[i] := false;

        remainder section

until false;
```

Critical Section Problem

Producer and Consumer

Producer

```
while (true) {
/* produce an item in next produced */

while (counter == BUFFER_SIZE) ;
/* do nothing */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

Consumer

```
while (true) {
while (counter == 0)
;  /* do nothing */
next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
/* consume the item in next consumed */
}
```

Outline of the Lecture

Semaphore – Introduction

Characteristic of Semaphore

Type of Semaphore

Semaphore

A **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems in a concurrent system such as a multitasking operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.

A semaphore is a variable that indicates the number of resources that are available in a system at a particular time and this semaphore variable is generally used to achieve the process synchronization. It is generally denoted by "S".

Semaphore

A semaphore uses two functions i.e. ***wait()*** and ***signal()***. Both these functions are used to change the value of the semaphore but the value can be changed by only one process at a particular time and no other process should change the value simultaneously.

The ***wait()*** function is used to decrement the value of the semaphore variable "**S**" by one if the value of the semaphore variable is positive. If the value of the semaphore variable is 0, then no operation will be performed.

```
wait(S) {
while (S == 0); //there is ";" sign here S--;
 }
```

The ***signal()*** function is used to increment the value of the semaphore variable by one.

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

Semaphore

Characteristic of Semaphore

- It is a mechanism that can be used to provide synchronization of tasks.
- It is a low-level synchronization mechanism.
- Semaphore will always hold a non-negative integer value.
- Semaphore can be implemented using test operations and interrupts, which should be executed using file descriptors.

Semaphore

Types of Semaphore

The two common kinds

- Counting semaphores
- Binary semaphores

Counting Semaphores:

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 i.e 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.

Semaphore

Types of Semaphore

The two common kinds

- Counting semaphores
- Binary semaphores

Binary Semaphores: In Binary semaphores, the value of the semaphore variable will be 0 or 1. Initially, the value of semaphore variable is set to 1 and if some process wants to use some resource then the *wait()* function is called and the value of the semaphore is changed to 0 from 1. The process then uses the resource and when it releases the resource then the *signal()* function is called and the value of the semaphore variable is increased to 1.

If at a particular instant of time, the value of the semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of the resource by the previous process. In this way, process synchronization can be achieved.

Test and Set Lock

Test and Set Lock (TSL) is a synchronization mechanism.

It uses a test and set instruction to provide the synchronization among the processes executing concurrently.

It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.

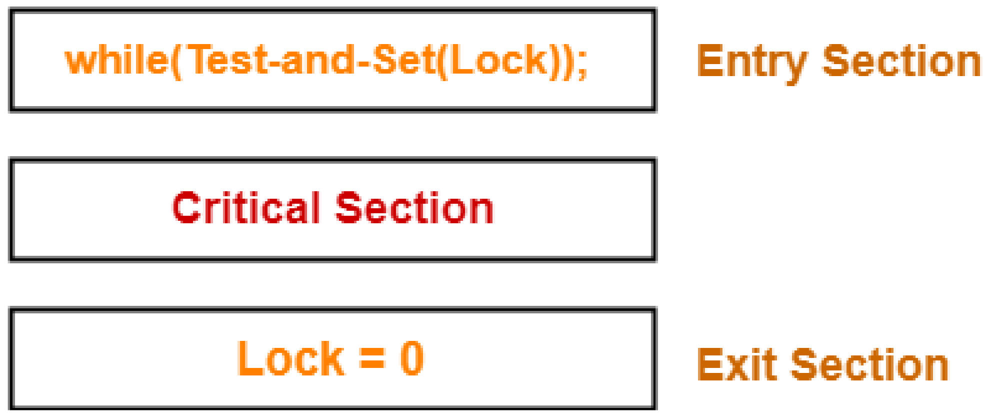
If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

Test and Set Lock

Initially, lock value is set to 0.

● Lock value = 0 means the critical section is currently vacant and no process is present inside it.

● Lock value = 1 means the critical section is currently occupied and a process is present inside it.



```
while(Test-and-Set(Lock));
Critical Section
Lock = 0
```

Semaphore Usage

Consider ***P***$_1$ and ***P***$_2$ that require ***S***$_1$ to happen before ***S***$_2$

Create a semaphore "**synch**" initialized to 0

```
P1:
   S1;
             signal(synch);
do {
              wait(mutex):
             critical Section
             signal(mutex);
Remainder section
}while(1)
P2:
   wait(synch);
   S2;
```

Dining-Philosophers Problem

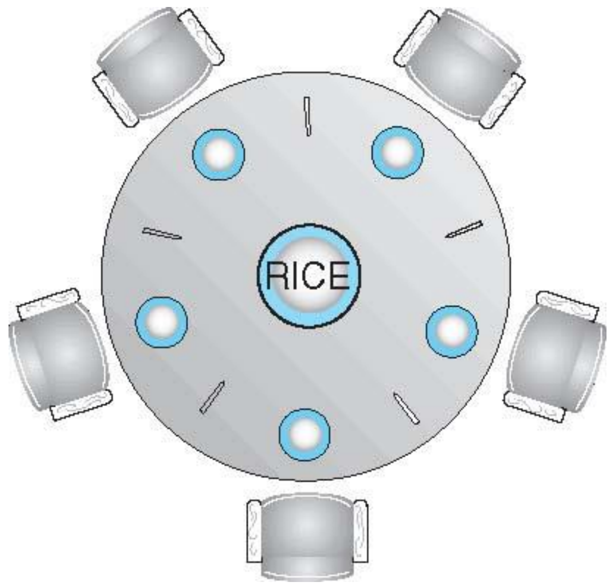
Philosophers spend their lives alternating thinking and eating

Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl

- Need both to eat, then release both when done

In the case of 5 philosophers

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem

The structure of Philosopher *i*:

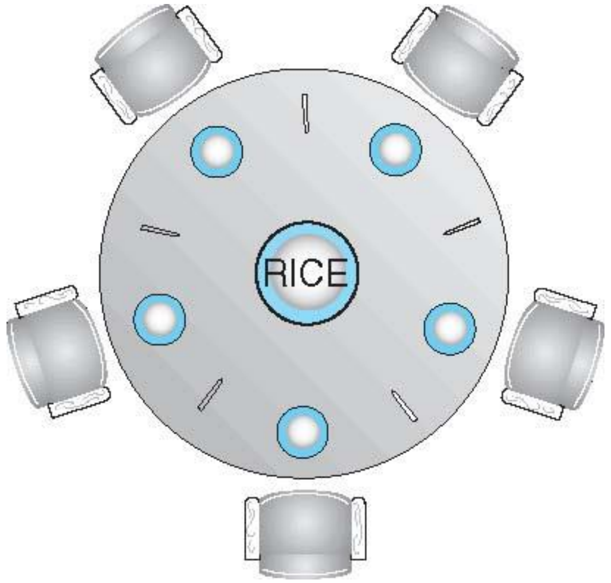
```
do {
    wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );

                 //  eat

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

                //  think

} while (TRUE);
```



Bounded buffer Problem

The structure of the producer process

```
do {
    ...
   /* produce an item in next_produced */
    ...
   wait(empty);
   wait(mutex);
     ...
   /* add next produced to the buffer */
     ...
   signal(mutex);
   signal(full);
} while (true);
```

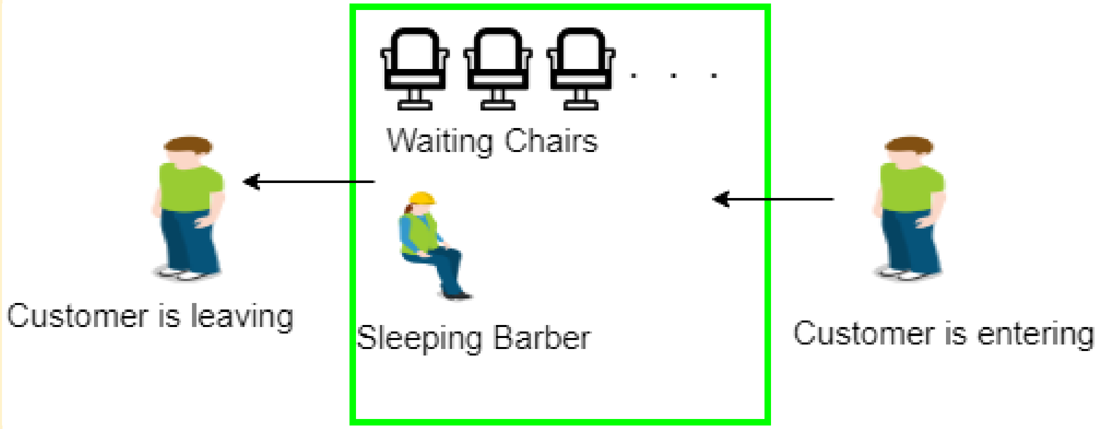
Sleeping Barber problem

Problem : The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

•If there is no customer, then the barber sleeps in his own chair.

•When a customer arrives, he has to wake up the barber.

•If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.



Sleeping Barber problem

```
Customer {
while(true) {
sem_wait(accessseats);
if(FreeSeats > 0) {
FreeSeats--; /* sitting down.*/
sem_post(Customers); /* notify the barber. */
sem_post(accesseats); /* release the lock */
sem_wait(Barber); /* wait in the waiting room if barber is
busy. */
// customer is having hair cut
} else {
srm_post(accessSeats); /* release the lock */
// customer leaves } }}
```

Variables: shared data

Semaphore Customers = 0;

Semaphore Barber = 0; // sleeping

Access Mutex Seats = 1;

int FreeSeats = N;

Sleeping Barber problem

```
Barber {
while(true) {
Wait (customers); /* waits for a customer (sleeps). */
wait (mutex); // whenever wait(1) is executed it decrement value 0 ie. mutex to protect the
number of available seats
Numberoffreeseat++ // a chair get free
sem_post(barber) /* bring customer for haircut.*/
sem_post(mutex) /* release the mutex on the chair.*/
/* barber is cutting hair.*/
}
}
```

Problem in Semaphore

1. One of the biggest limitations of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and to signal the semaphore.

several process may active simultaneously

```
Signal(mutex);
          ……
          Critical section
          ……..
Wait(mutex);
```

Deadlock may occur

```
wait(mutex);
          ……
          Critical section
          ……..
wait(mutex);
```

Critical Region

One of the main problem in semaphore is that semaphore are not syntactically related to the shared resources that they guard. In particular there are no declaration that could alert the compiler that a specific data structure is shared and that its accessing needs to be controlled.

Acritical region protects a shared data structure by making its known to it known to the compiler which can then generate code that maintains mutual exclusive access to the related data. The declaration of shared variable has the following format:

V: shared : T;

Where the keyword shared informs the compiler that the variable mutes of user deifned type T, is shared by several person. Process may acess a guarede variable by means of the region construct as follows:

Region V when B do S;

Statement “do” is executed as a critical section.

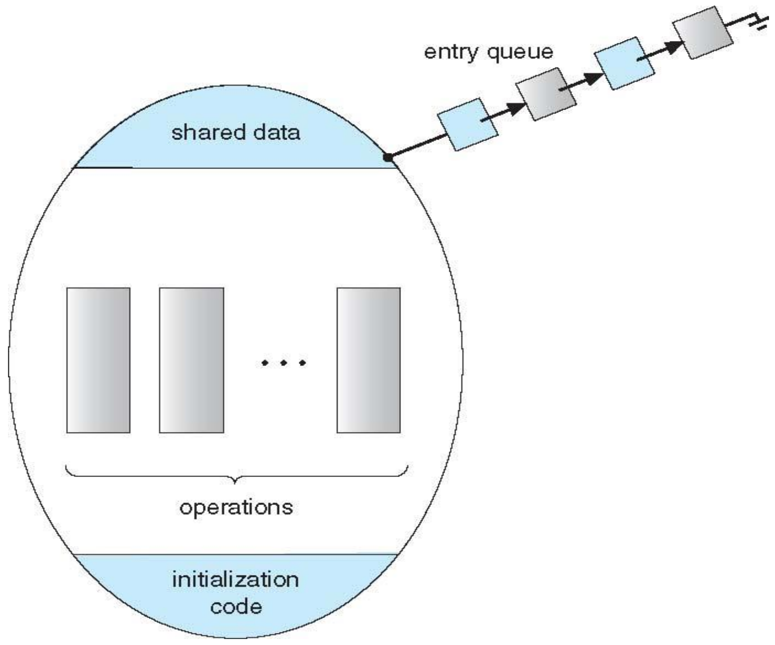
Monitor

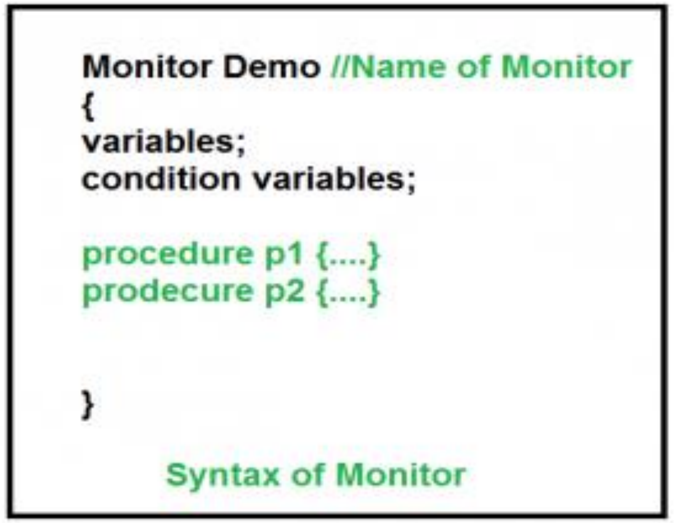
The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example Java Synchronized methods. Java provides wait() and notify() constructs.

1.It is the collection of condition variables and procedures combined together in a special kind of module or a package.

2.The processes running outside the monitor can't access the internal variable of the monitor but can call procedures of the monitor.

3.Only one process at a time can execute code inside monitors.





```
Monitor Demo //Name of Monitor
{
variables;
condition variables;

procedure p1 {....}
prodecure p2 {....}


}
```

Syntax of Monitor

Monitor

Key Differences Between Semaphore and Monitor

1. The basic difference between semaphore and monitor is that the **semaphore** is an **integer variable S** which indicate the number of resources available in the system whereas, the **monitor** is the **abstract data type** which allows only one process to execute in critical section at a time.
2. The value of semaphore can be modified by **wait()** and **signal()** operation only. On the other hand, a monitor has the shared variables and the procedures only through which shared variables can be accessed by the processes.
3. In Semaphore when a process wants to access shared resources the process performs **wait**() operation and block the resources and when it release the resources it performs **signal**() operation. In monitors when a process needs to access shared resources, it has to access them through procedures in monitor.
4. Monitor type has **condition variables** which semaphore does not have.

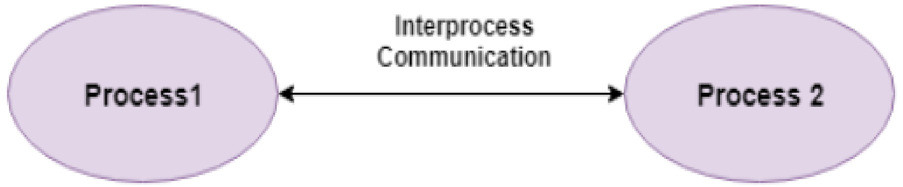
Interprocess Communication

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

Why IPC?

Here, are the reasons for using the interprocess communication protocol for information sharing:

- It helps to speedup modularity
- Computational
- Privilege separation
- Convenience
- Helps operating system to communicate with each other and synchronize their actions.



Different Models of Interprocess Communication

Shared Memory Model

Message Passing Model

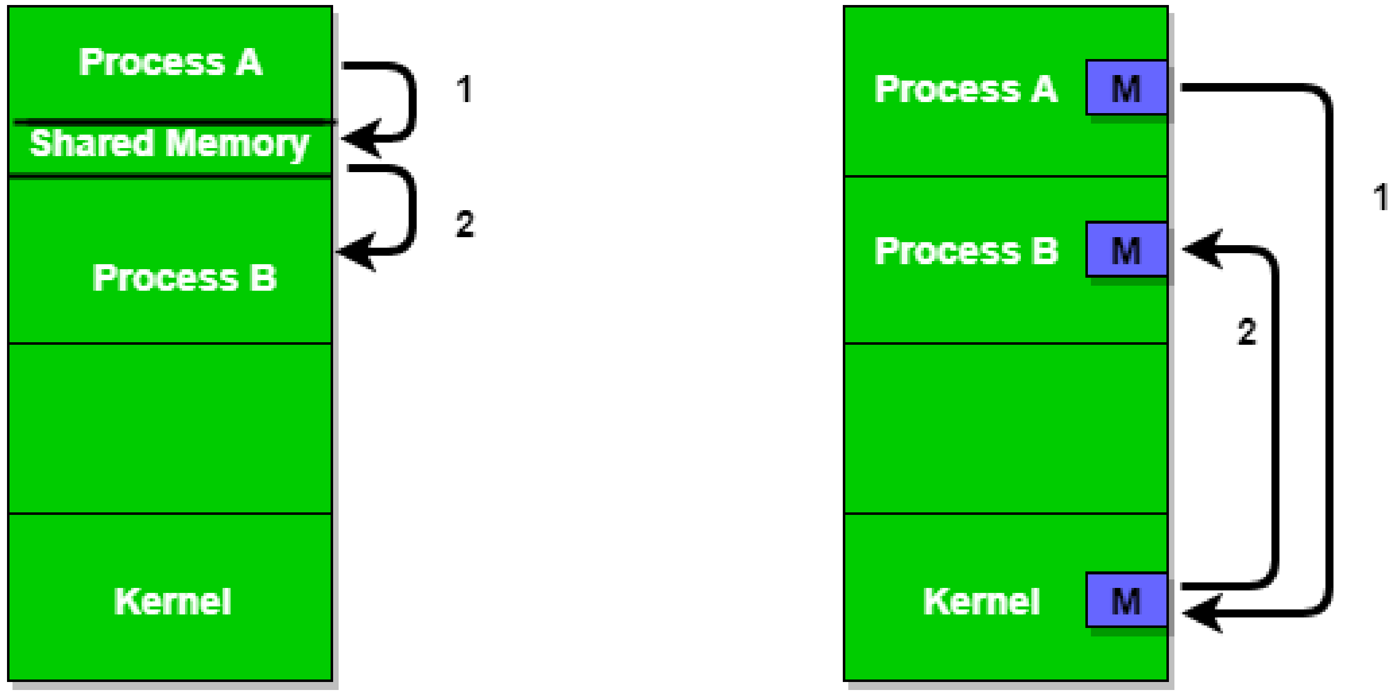
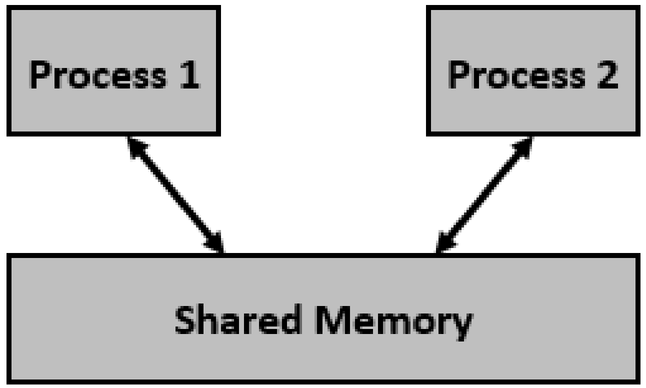


Figure 1 - Shared Memory and Message Passing

Shared Memory Model

Shared Memory Model

- Shared memory is a memory shared between two or more processes. An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.



Shared Memory Model

Advantage of Shared Memory Model

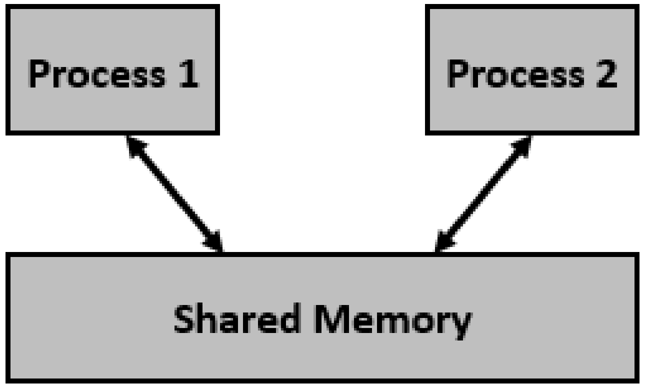
Memory communication is faster on the shared memory model as compared to the message passing model on the same machine.

Disadvantages of Shared Memory Model

Some of the disadvantages of shared memory model are as follows −

● All the processes that use the shared memory model need to make sure that they are not writing to the same memory location.

● Shared memory model may create problems such as synchronization and memory protection that need to be addressed.



Message Passing Model

Multiple processes can read and write data to the message queue without being connected to each other. Messages are stored on the queue until their recipient Retrieves them. Message queues are quite useful for interprocess communication and are used by most operating systems.

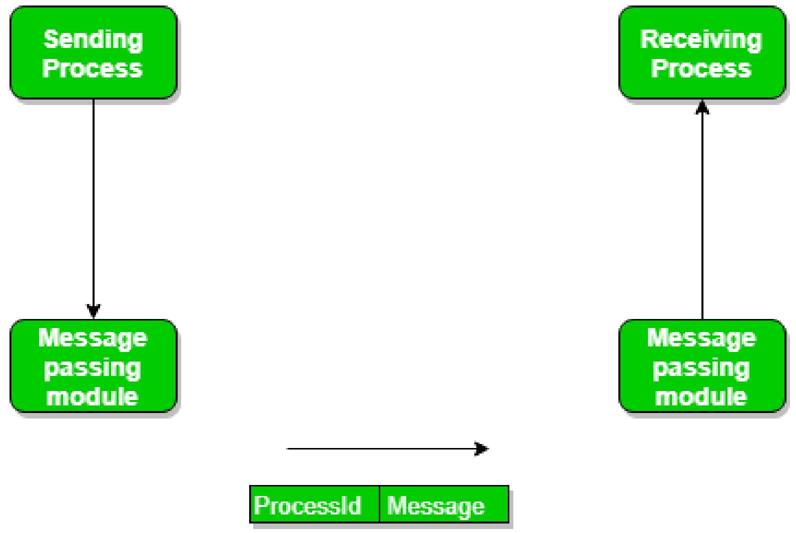
Mechanism for processes to communicate and to synchronize their actions

Message system – processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

- **send**(*message*)
- **receive**(*message*)

The *message* size is either fixed or variable


Sending Process
Receiving Process
Message passing module
Message passing module
ProcessId
Message