Parul® University
Vadodara, Gujarat

NAAC GRADE A++

Information and Communication Technology

Ch-3 Inter-process communication (IPC)

Study Guide

Assistant Prof. Sumersing Patil
CSE-AI&DS, PIET
Parul University

Ch-3 Inter-process communication (IPC)

@ Introduction To IPC

- Inter-process communication is the mechanism provided by the operating system that allows processes to communicate with each other.
- Processes in a system can be independent or cooperating.
  - Independent process cannot affect or be affected by the execution of another process.
  - Cooperating process can affect or be affected by the execution of another process.

Cooperating processes need inter process communication mechanisms.

Reasons(Need) of IPC

Information sharing: Several processes may need to access the same data (such as stored in a file.)

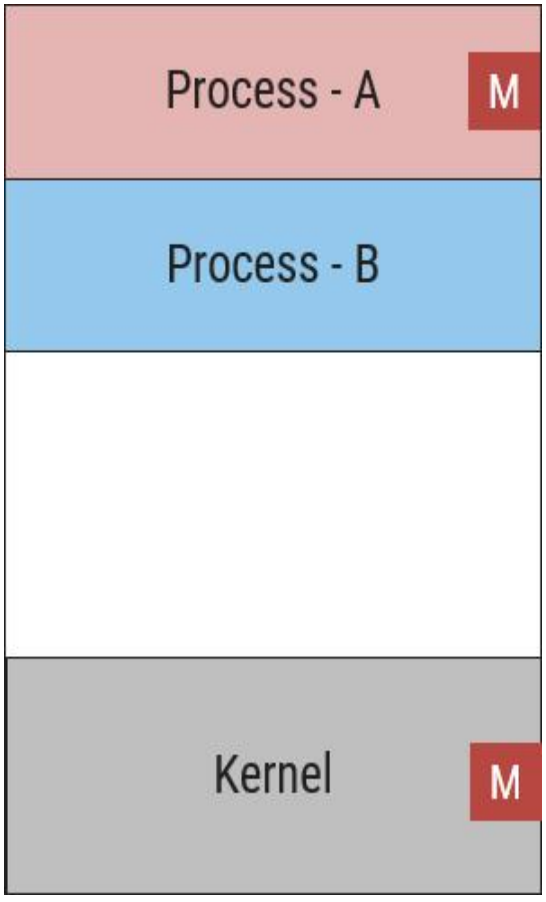
Computation speed-up: A task can often be run faster if it is broken into subtasks and distributed among different processes.

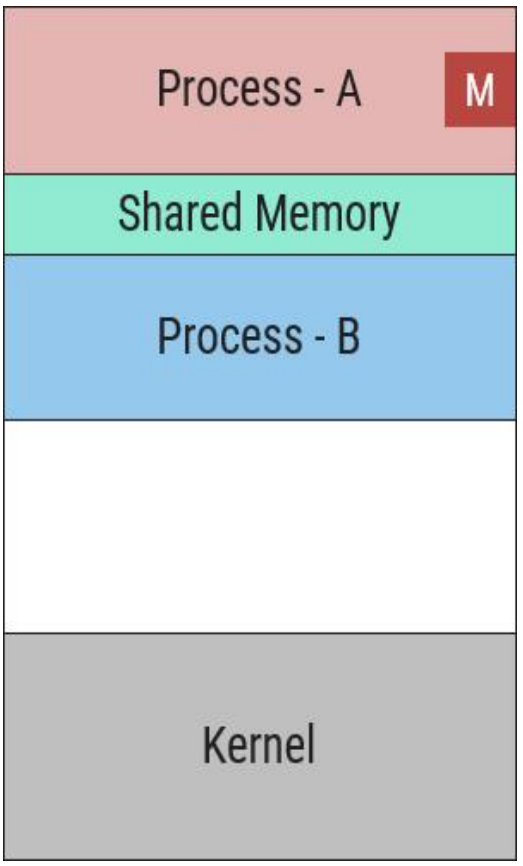
Modularity: It may be easier to organize a complex task into separate subtasks, then have different processes or threads running each subtask.

Convenience: An individual user can run several programs at the same time, to perform some task.

Models for Inter-process communication (IPC)

Message Passing	Shared Memory
Process A send the message to Kernel and then Kernel send that message to Process B	Process A put the message into Shared Memory and then Process B read that message from Shared Memory





@ Race condition

- Race condition is a situation arise due to concurrent execution of more than one processes which are accessing and manipulating the same shared data and the result of execution depends upon the specific order where the access take place.
- A race condition is an undesirable situation that occurs when a

device or system attempts to perform two or more operations at the same time.

- But, because of the nature of the device or system, the operations must be done in the **proper sequence** to be done correctly.
- To prevent race conditions, concurrent processes must be synchronized.

Assume that two threads each increment the value of a global integer variable by 1. Ideally, the following sequence of operations would take place:

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

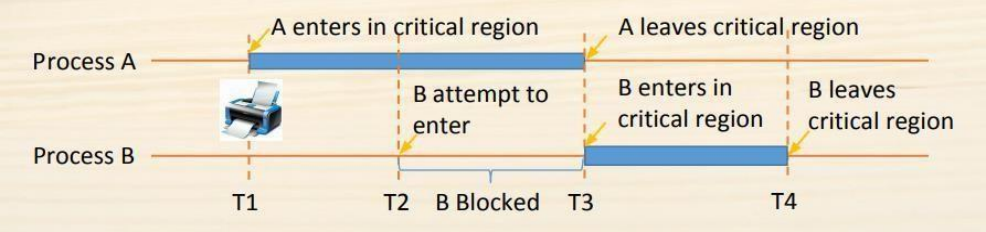
In the case shown above, the final value is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. The alternative sequence of operations below demonstrates this scenario:

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

In this case, the final value is 1 instead of the correct result of 2. This occurs because here the increment operations are not mutually exclusive. Mutually exclusive operations are those that cannot be interrupted while accessing some resource such as a memory location.

@ Critical Section

- The part of program where the shared resource is accessed is called critical section or critical region.



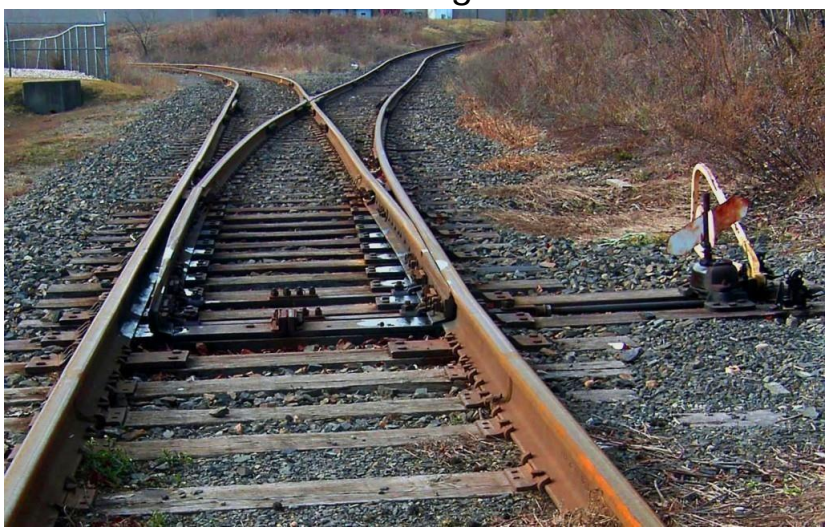
Process A:

```
// Process A
.
.
b = x + 5;          // instruction executes at time = Tx
.
```

Process B:

```
// Process B
.
.
x = 3 + z;          // instruction executes at time = Tx
.
```

- concurrent accesses to shared resources can lead to unexpected or erroneous behavior , so part of the program where the shared resource is accessed is protected. This protected section is the critical section or critical region.



To critical section: three aspects are important

1. Mutual Exclusion

- Way of making sure that if one process is using a shared variable or file; the other process will be excluded (stopped) from doing the same thing.

- Out of a group of cooperating processes, only one process can be in its critical section at a given point of time.

2. Progress:

- If no process is in its critical section, and if one or more threads want to execute their critical section then any one of these threads must be allowed to get into its critical section.
- No process running outside its critical region may block other processes

3. Bounded Waiting:

- After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section, before this process's request is granted. So after the limit is reached, system must grant the process permission to get into its critical section.
- No process should have to wait forever to enter a critical section.

4. Busy waiting : busy-waiting, busy-looping or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available

@ Solution to synchronization problem

There are some mechanisms have been introduced for synchronization problem which are as follows-

1. Strict alteration(Software solution)/ Turn variable
2. Peterson's solution (Software approach)
3. Disabling interrupts (Hardware approach)
4. Shared lock variable (Software approach)
5. TSL (Test and Set Lock) instruction (Hardware approach)

1. Strict alteration(Software solution)/ Turn variable

- **Integer variable 'turn' keeps track of whose turn is** to enter the critical section.
- **Initially turn=0. Process 0 inspects turn**, finds it to be 0, and **enters in its critical section**.
- **Process 1 also finds it to be 0 and therefore sits in a loop** continually testing 'turn' to see when it becomes 1.

- Continuously testing a variable waiting for some event to appear is called the **busy waiting**.

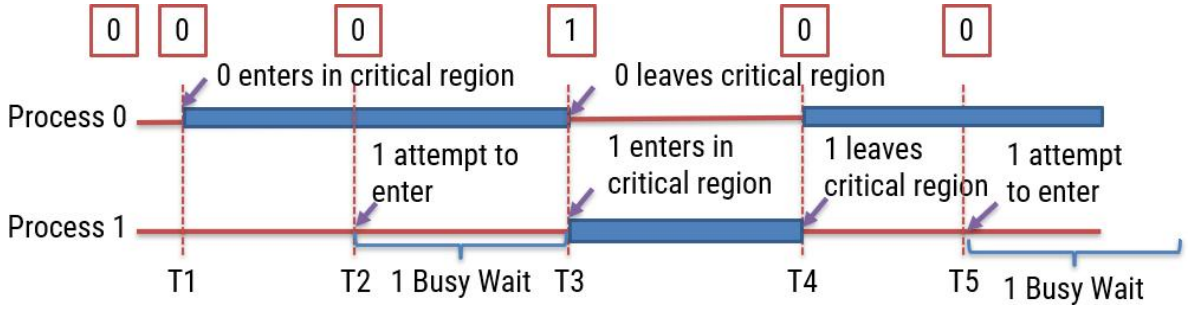
- **When process 0 exits** from critical region it **sets turn to 1** and now **process 1 can find it to be 1 and enters in to critical region**.
- In this way, **both the processes get alternate turn** to enter in critical region.

Process 0

```
while (TRUE)
{
        while (turn != 0) /* loop */ ;
        critical_region();
        turn = 1;
        noncritical_region();
}
```

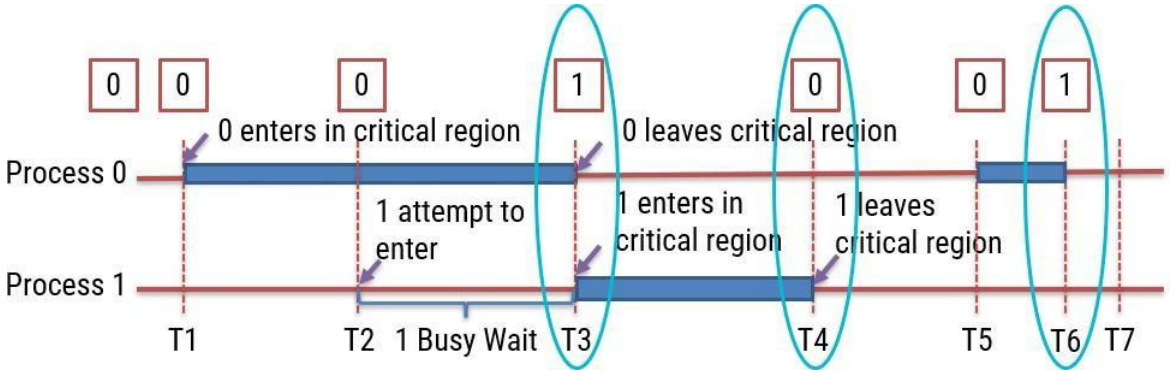
Process 1

```
while (TRUE)
{
        while (turn != 1) /* loop */ ;
        critical_region();
        turn = 0;
        noncritical_region();
}
```

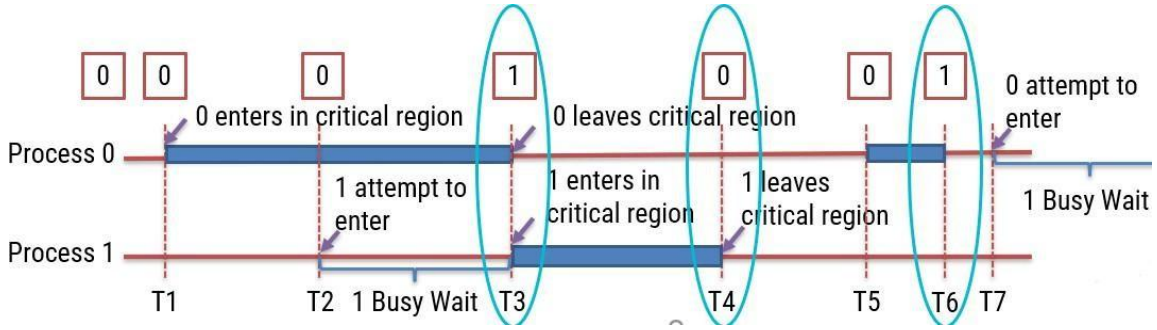


Disadvantages of Strict alteration

- Consider the following situation for two processes P0 and P1.
- P0 leaves its critical region, set turn to 1, enters non critical region.
- P1 enters and finishes its critical region, set turn to 0.
- Now both P0 and P1 in non-critical region.
- P0 finishes non critical region, enters critical region again, and leaves this region, set turn to 1.
- P0 and P1 are now in non-critical region.

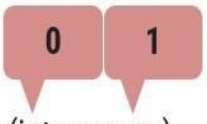


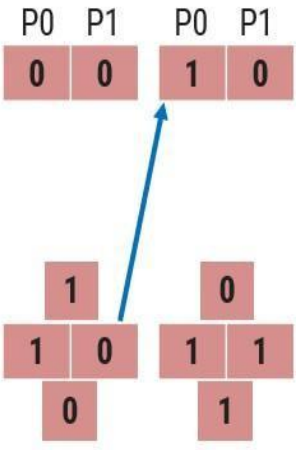
- P0 finishes non critical region but cannot enter its critical region because turn = 1 and it is turn of P1 to enter the critical section.
- Hence, P0 will be blocked by a process P1 which is not in critical region. This violates one of the conditions of mutual exclusion.
- It wastes CPU time, so we should avoid busy waiting as much as we can.



2. Peterson's solution (Software approach)

```
#define FALSE 0
#define TRUE 1
#define N 2                                  //number of processes
int turn;                                    //whose turn is it?
int interested[N];                           //all values initially 0 (FALSE)
void enter_region(int process)
{
int other;                                   // number of the other process
other = 1 - process;                         // the opposite process
interested[process] = TRUE;                  // this process is interested
turn = process;                              // set flag
while(turn == process && interested[other] == TRUE); // wait
}
void leave_region(int process)
{
interested[process] = FALSE;                 // process leaves critical region
}
```





Disadvantage of Peterson's solution Priority inversion problem

- Priority inversion means the execution of a high priority process/thread is blocked by a lower priority process/thread.
- Consider a computer with two processes, H having high priority

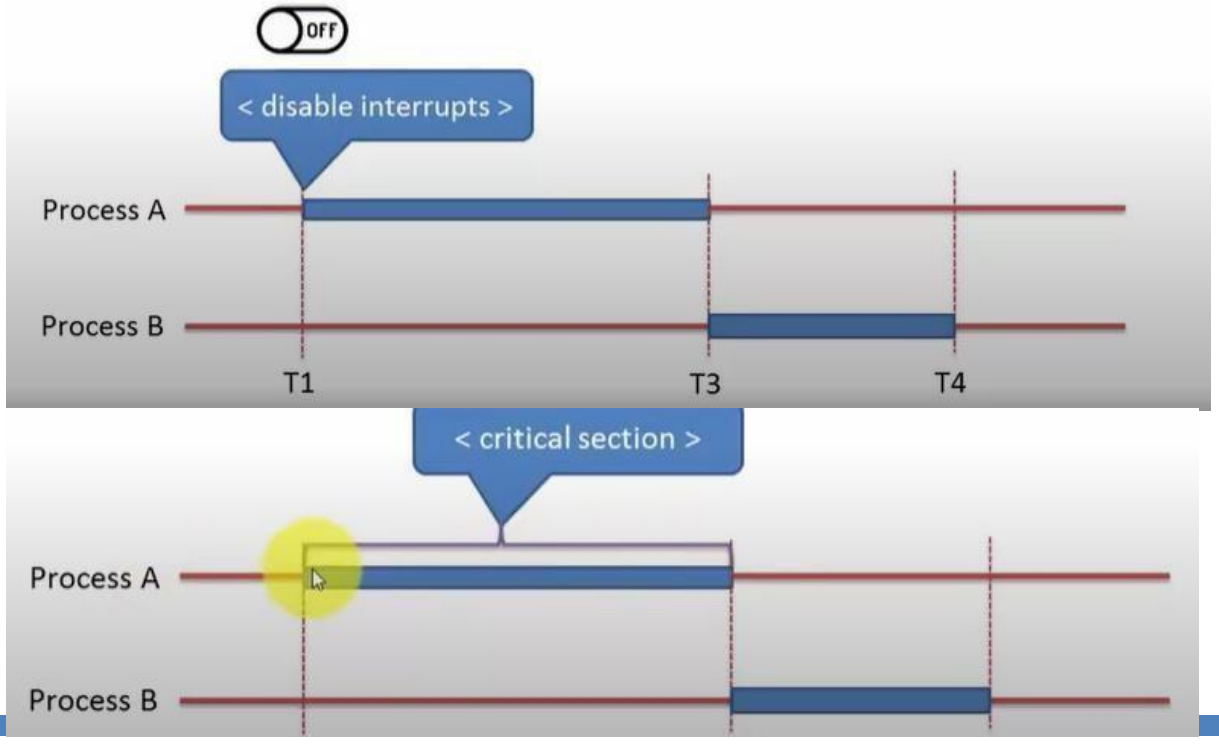
and L having low priority.

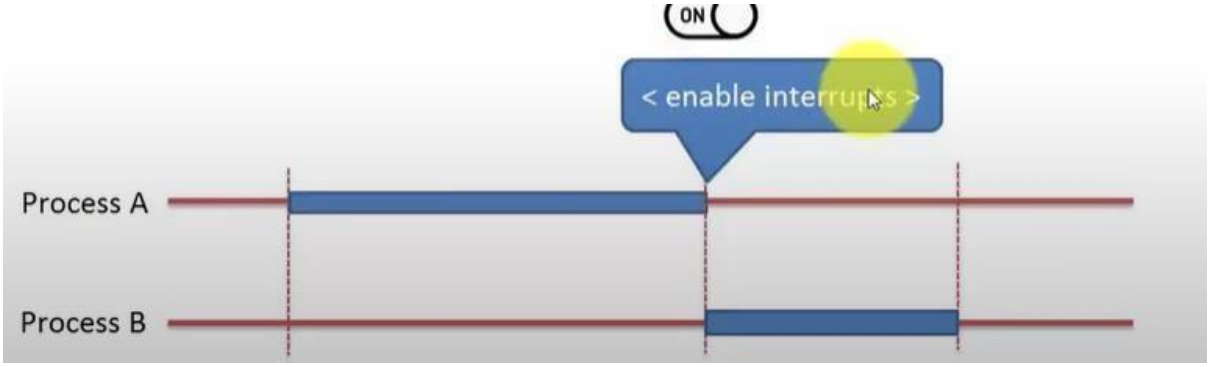
- The scheduling rules are such that H runs first then L will run.

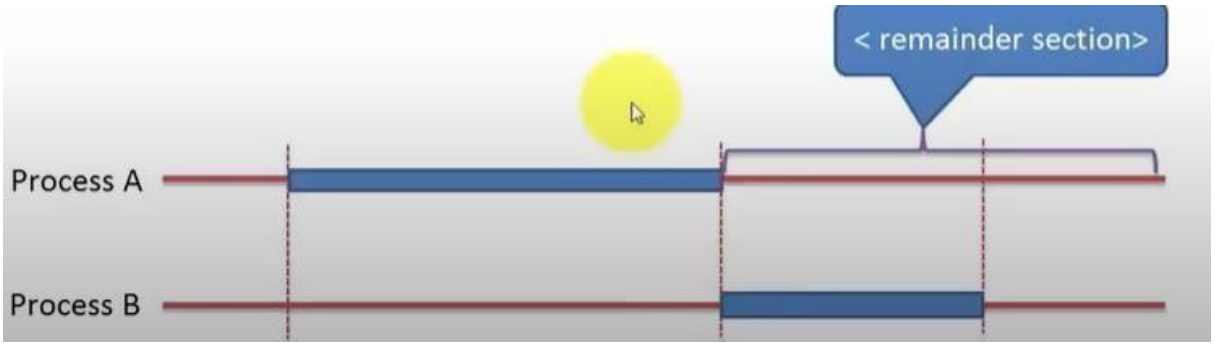
- At a certain moment, L is in critical region and H becomes ready to run (e.g. I/O operation complete).
- H now begins busy waiting and waits until L will exit from critical region.
- But H has highest priority than L so CPU is switched from L to H.
- Now L will never be scheduled (get CPU) until H is running so L will never get chance to leave the critical region so H loops forever.
- This situation is called priority inversion problem.

3. Disabling interrupts (Hardware approach)

```
while (true)
    {
        < disable interrupts >;
        < critical section >;
        < enable interrupts >;
        < remainder section >;
    }
```







Problems in Disabling interrupts (Hardware approach)

- Unattractive or unwise to give user processes the power to turn off interrupts.
- What if one of the process did it (disable interrupt) and never turned them on (enable interrupt) again? That could be the end of the system.
- If the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory

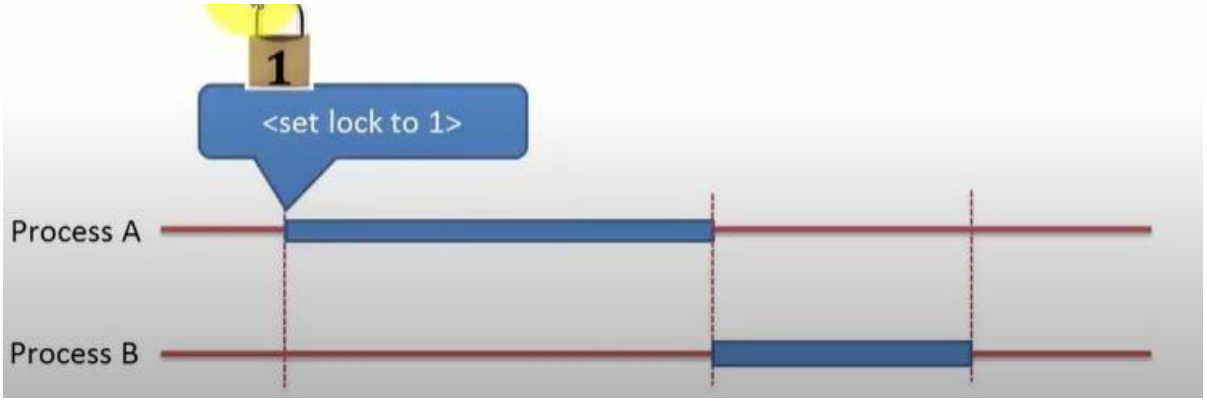
4. Shared lock variable (Software approach)

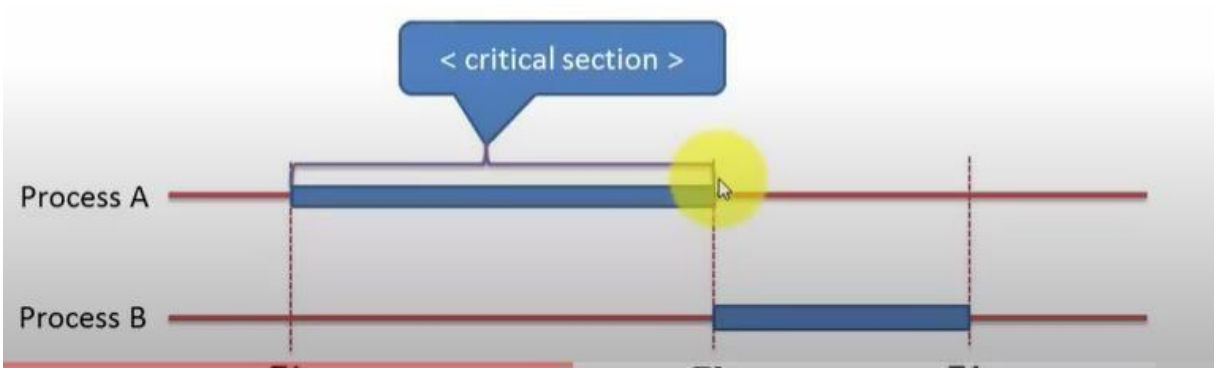
- A shared variable lock having value 0 or 1.
- Before entering into critical region a process checks a shared variable lock's value.
- If the value of lock is 0 then set it to 1 before entering the critical section and enters into critical section and set it to 0

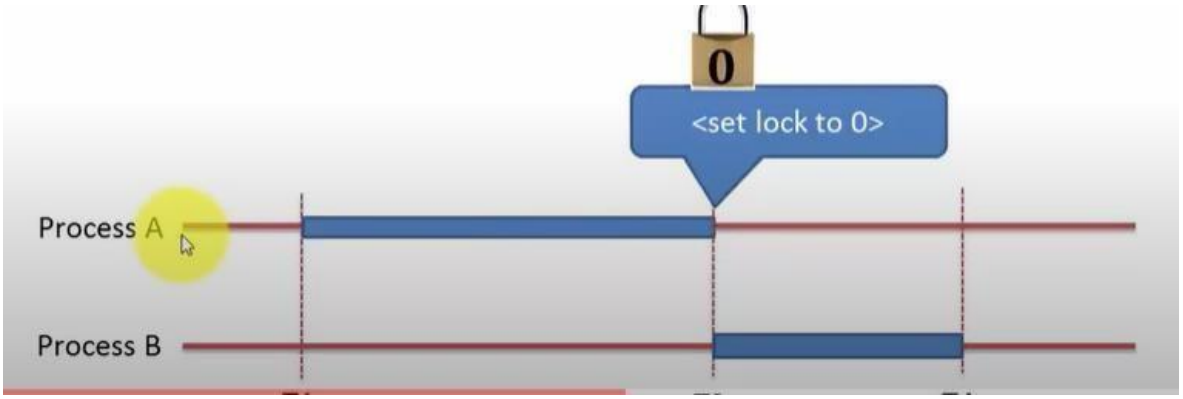
immediately after leaving the critical section.

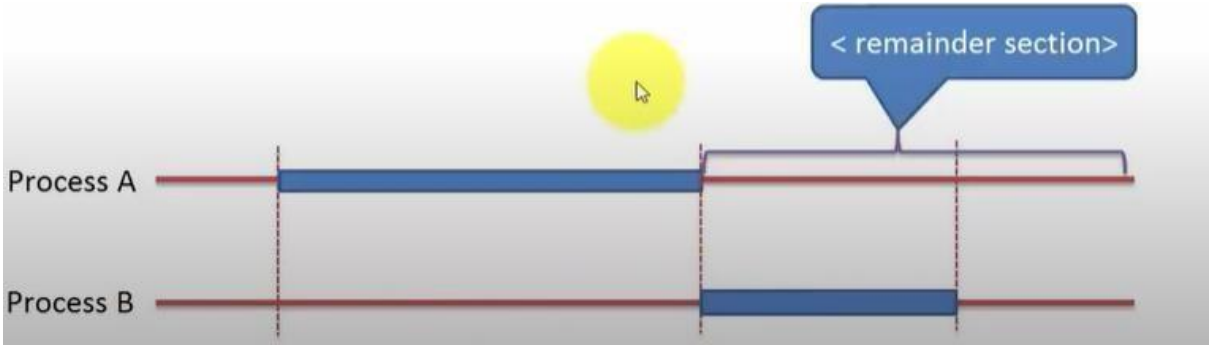
- If the value of lock is 1 then wait until it becomes 0 by some other process which is in critical section.

```
while (true)
        {       < set shared variable to 1 >;
                < critical section >;
                < set shared variable to 0 >;
                < remainder section >;          }
```





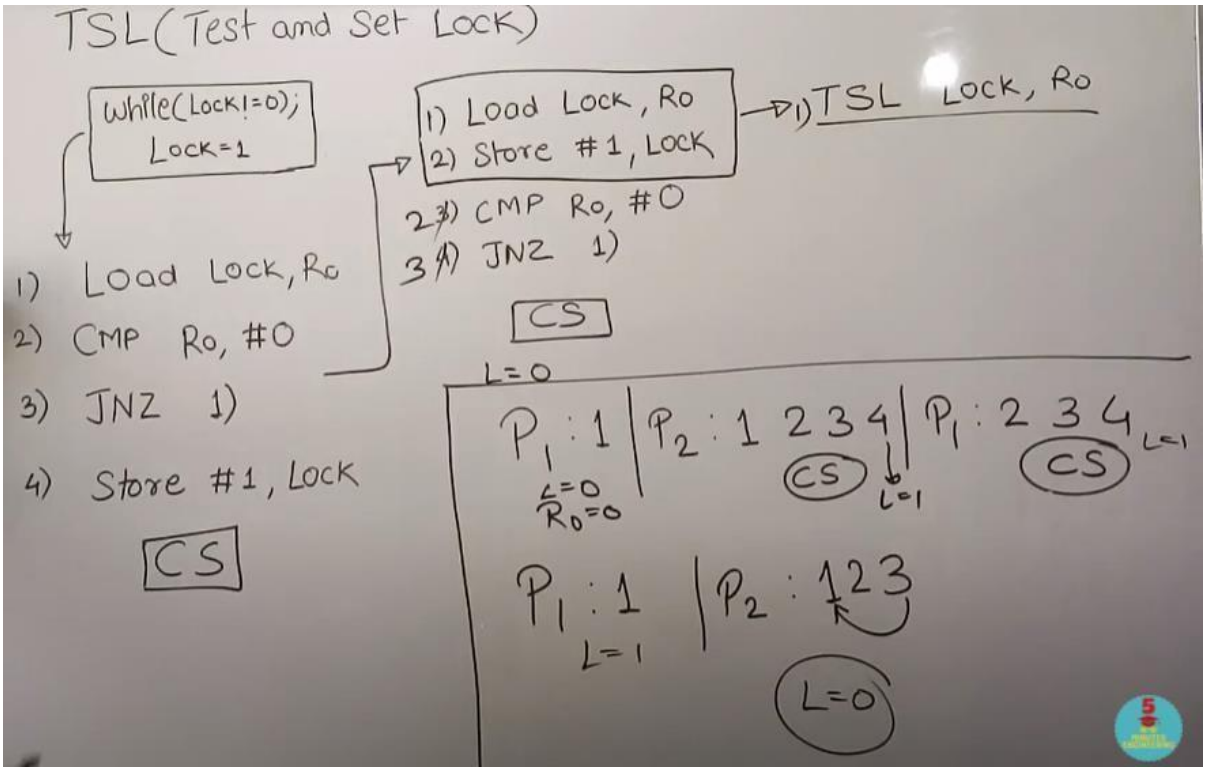




Problem:

- If process-A sees the value of lock variable 0 and before it can set it to 1 context switch occurs.
- Now process-B runs and finds value of lock variable 0, so it sets value to 1, enters critical region.
- At some point of time process-A resumes, sets the value of lock variable to 1, enters critical region.
- Now two processes are in their critical regions accessing the same shared memory, which violates the mutual exclusion

5. TSL (Test and Set Lock) instruction (Hardware approach)

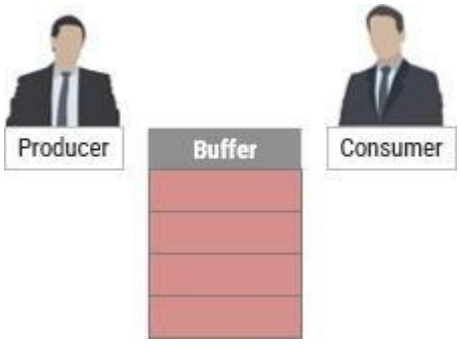


@ The Producer Consumer Problem

- It is multi-process synchronization problem.
- It is also known as bounded buffer problem.
- This problem describes two processes producer and consumer, who share common, fixed size buffer.

Producer process: Produce some information and put it into buffer

Consumer process: Consume this information (remove it from the buffer)



Buffer is empty

-Producer want to produce√

-Consumer want to consume X

Buffer is full

-Producer want to produce X

-Consumer want to consume √

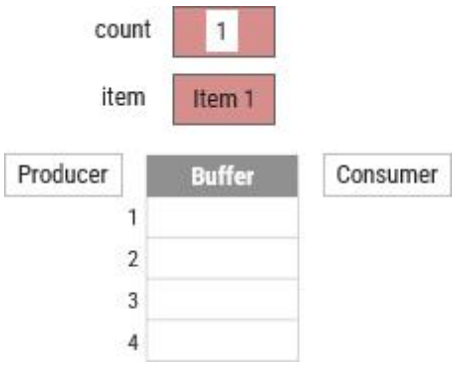
Buffer is partial filled

-Producer want to produce √

-Consumer want to consume √

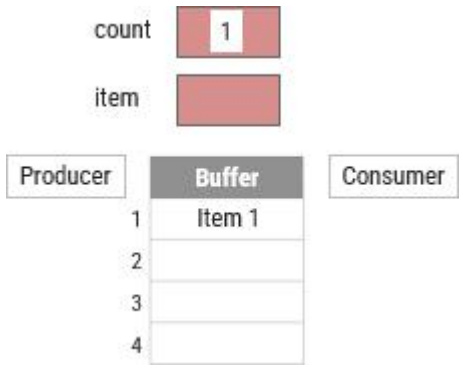
Producer Consumer problem using Sleep & Wakeup

```
#define N 4
int count=0;
void producer (void)
{       int item;
        while (true)
        {       item=produce_item();
                if (count==N) { sleep(); }
                insert_item(item);
                count=count+1;
                if(count==1) { wakeup(consumer); }
        }
}
```

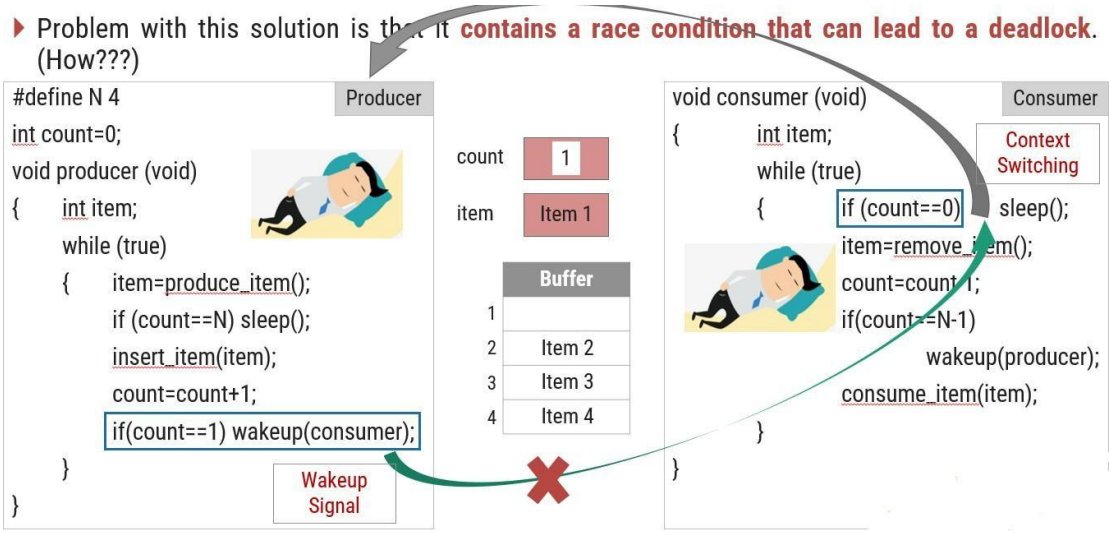


void consumer (void)

```
{       int item;
        while (true)
        {       if (count==0)
                { sleep(); }
                item=remove_item();
                count=count-1;
                if(count==N-1)
                { wakeup(producer)
                ; }
                consume_item(item)
                ;
        }
}
```



Problem in Sleep & Wakeup



- The consumer has just read the variable count, noticed it's zero and is just about to move inside the if block.
- Just before calling sleep, the consumer is suspended and the producer is resumed.
- The producer creates an item, puts it into the buffer, and increases count.
- Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
- Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost.
- When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when count is equal to 1.
- The producer will loop until the buffer is full, after which it will also go to sleep.
- Finally, both the processes will sleep forever. This solution therefore is unsatisfactory.

Semaphore

- A semaphore is a **variable that provides an abstraction for controlling the access of a shared resource** by multiple processes in a parallel programming environment.
- There are 2 types of semaphores:

Binary semaphores :-

- Binary semaphores can take only 2 values (0/1).
- Binary semaphores have 2 methods associated with it (up, down / lock, unlock).
- They are used to acquire locks.

Counting semaphores :-

- Counting semaphore can have possible values more than two.

We want functions **insert _item and remove_item** such that the following hold:

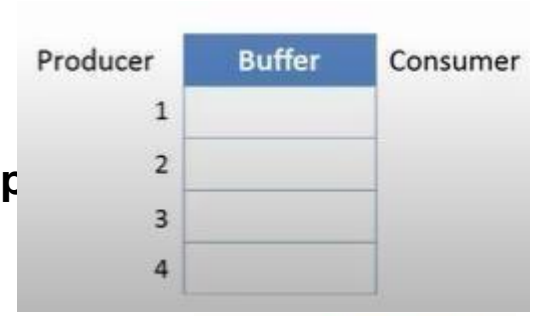
- Mutually exclusive access to buffer: At any time only one process should be executing (either insert_item or remove_item).
- No buffer overflow: A process executes insert_item only when the buffer is not full (i.e., the process is blocked if the buffer is full).
- No buffer underflow: A process executes remove_item only when the buffer is not empty (i.e., the process is blocked if the buffer is empty).
- No busy waiting.
- No producer starvation: A process does not wait forever at insert_item() provided the buffer repeatedly becomes full.
- No consumer starvation: A process does not wait forever at remove_item() provided the buffer repeatedly becomes empty.

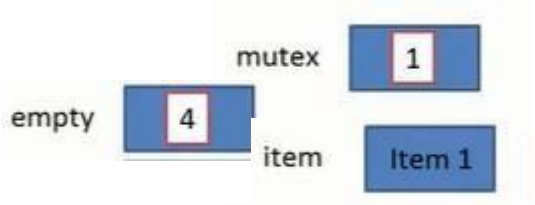
Operations on Semaphore

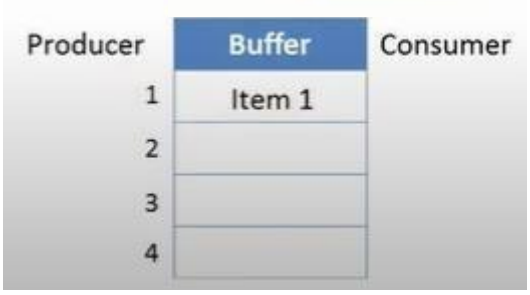
- **Wait():** a process performs a wait operation to tell the semaphore that it wants exclusive access to the shared resource. (mutex=0)
- **Signal():** a process performs a signal operation to inform the semaphore that it is finished using the shared resource.(mutex=1)

Producer Consumer problem using Semap

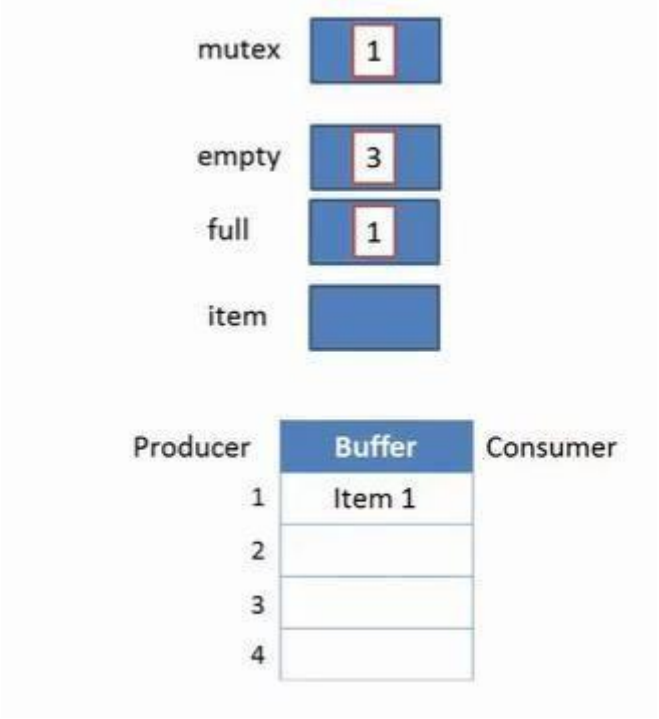
```
#define N 4
typedef int semaphore;
semaphore mutex=1;
semaphore empty=N;
semaphore full=0;
void producer (void)
{       int item;
        while (true)
        {
         item=produce_item();
         down(&empty);
         down(&mutex);
         insert_item(item);
         up(&mutex);
         up(&full);
        }
}
```

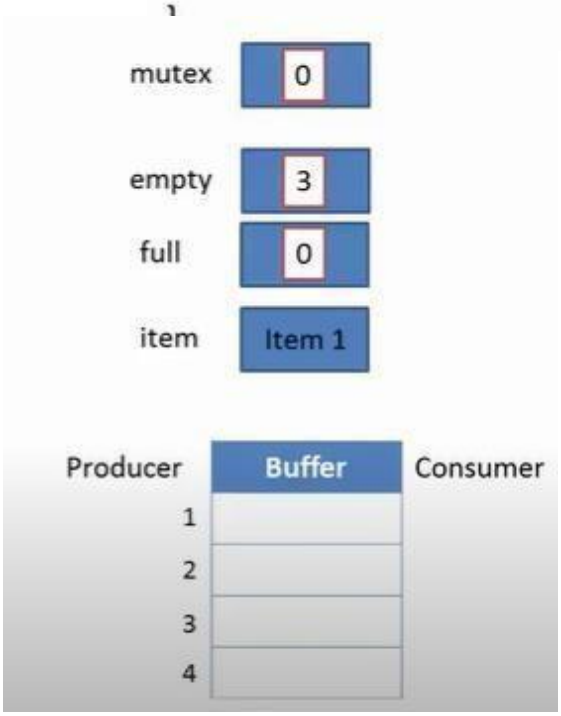


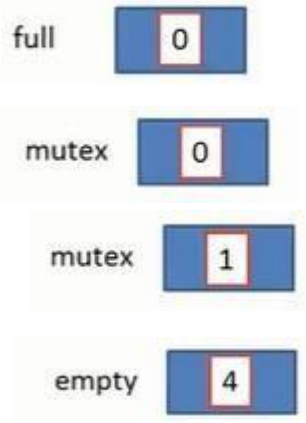




```
void consumer (void)
{    int item;
     while (true)
     {
      down(&full);
      down(&mutex);
      item=remove_item(item);
      up(&mutex);
      up(&empty);
      consume_item(item);
     }
}
```



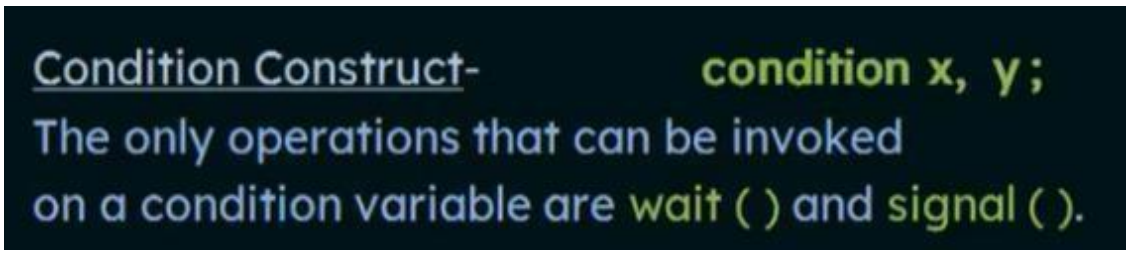




Monitor

- A higher-level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- **Processes may call the procedures** in a monitor whenever they want to, but they **cannot directly access the monitor's internal data structures** from procedures declared outside the monitor.
- only one process can be active in a monitor at any instant.
- When a process calls a monitor procedure, the **first few instructions of the procedure will check to see if any other process is currently active within the monitor**.
- If so, the **calling process will be suspended** until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

- A procedures defined within a monitor can access only those variable declared locally within the monitorand its formal parameter.



The operation x.wait( ) ; means that the process invoking this operation is suspended until another process invokes x.signal ( );
The x. signal ( ) operation resumes exactly one suspended process.

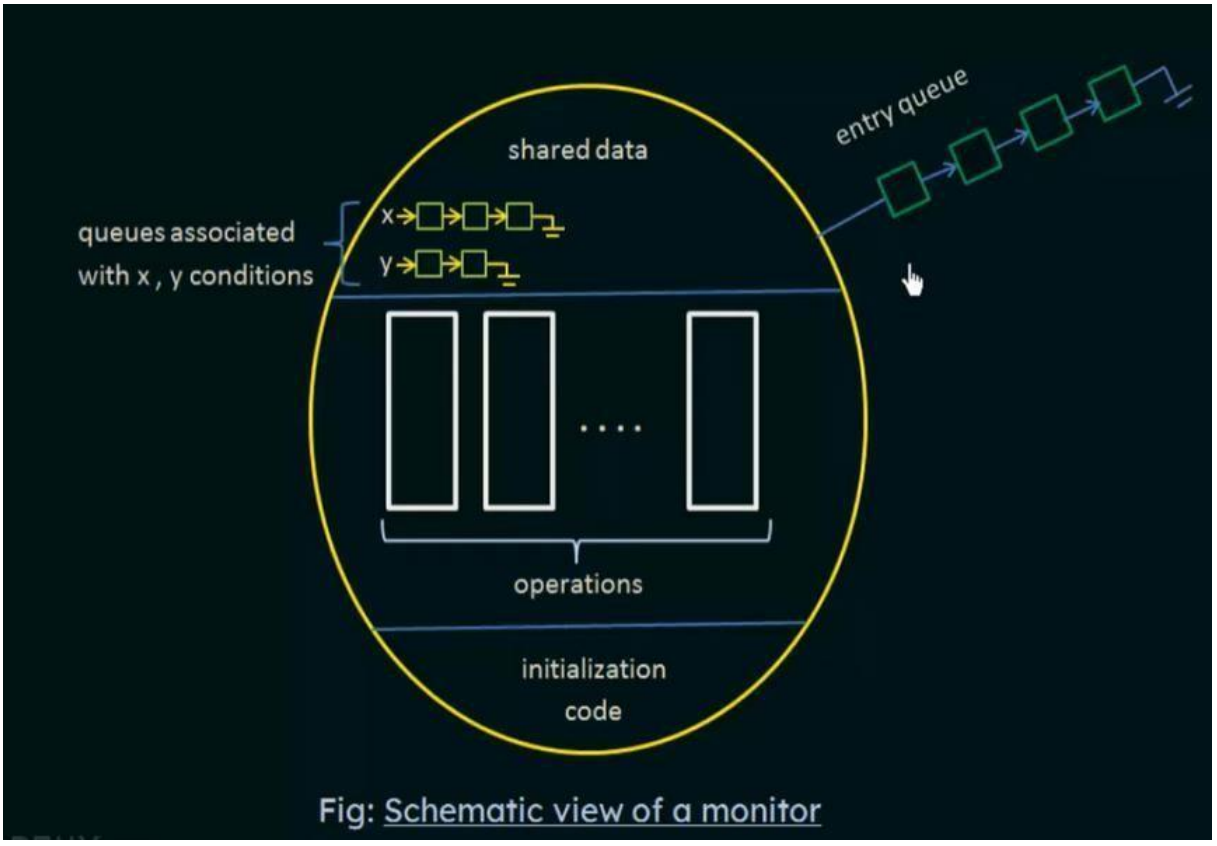


Fig: Schematic view of a monitor

Producer Consumer problem using Monitor

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
procedure producer;
begin
        while true do
        begin
                item=produce_item;
                ProducerConsumer.insert(item);
        end;
end;
```

```
procedure insert (item:integer);
        begin
                if count=N then wait (full);
                insert_item(item);
                count=count+1;
                if count=1 then signal (empty);
        end;
```

```
procedure consumer;
begin
        while true do
        begin
                item=ProducerConsumer.remove;
                Consume_insert(item);
        end;
end;
```

```
function remove:integer;
        begin
                if count=0 then wait (empty);
                remove=remove_item;
                count=count-1;
                if count=N-1 then signal (full);
        end;
        count=0;
end monitor;
```

Message Passing

- It refers to means of communication between

- Different thread within a process .

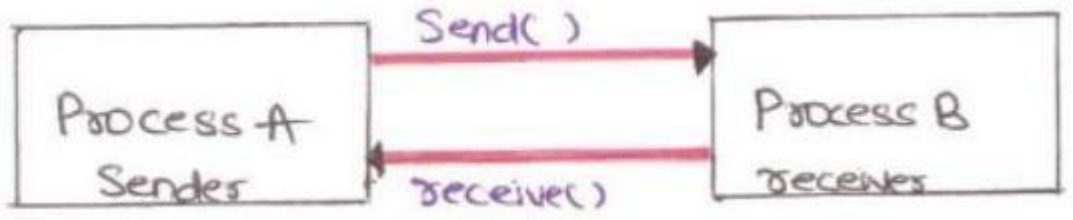
- Different processes running on same node.

- Different processes running on different node.

- In this a sender or a source process send a message to a non receiver or destination process.
- Message has a predefined structure and message passing uses two system call: Send and Receive

send(name of destination process, message);

receive(name of source process, message);



- In this calls, the sender and receiver processes address each other by names.
- Mode of communication between two process can take place through two methods

1) Direct Addressing

2) Indirect Addressing

Direct Addressing:

In this type that two processes need to name other to communicate. This become easy if they have the same parent.

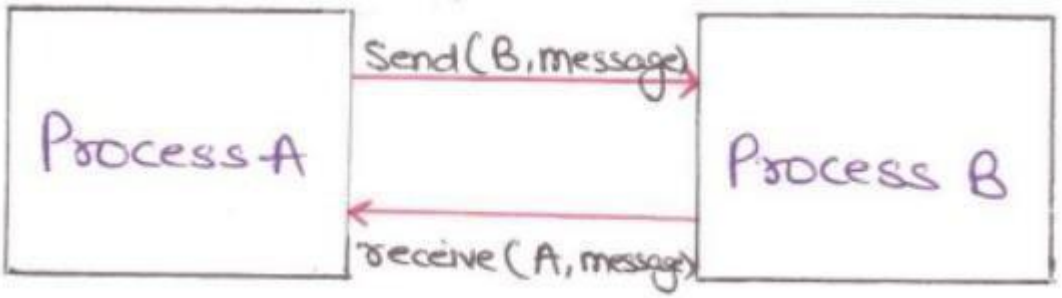
Example

If process A sends a message to process B,

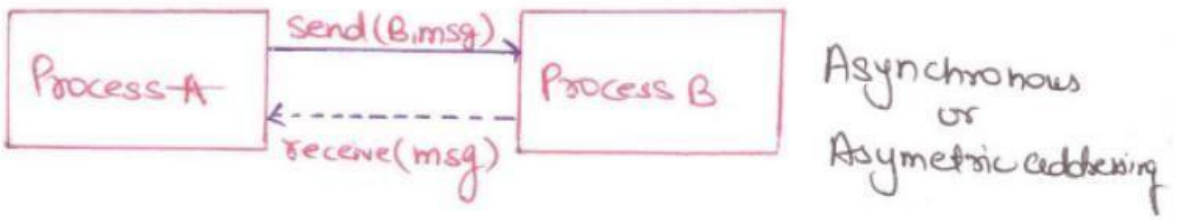
then send(B, message);

Receive(A, message);

By message passing a link is established between A and B. Here the receiver knows the Identity of sender message destination. This type of arrangement in direct communication is known as **Symmetric Addressing.**



Another type of addressing known as **asymmetric addressing** where receiver does not know the ID of the sending process in advance.



Indirect addressing:

- In this message send and receive from a mailbox. A mailbox can be abstractly viewed as an object into which messages may be placed and from which messages may be removed by processes. The sender and receiver processes should share a mailbox to communicate.

The following types of communication link are possible through mailbox.

- **One to One link**: one sender wants to communicate with one receiver. Then single link is established.

- **Many to Many link**: Multiple Sender want to communicate with single receiver.Example in client server system, there are many crying processes and one server process. The mailbox is here known as PORT.

- **One to Many link**: One sender wants to communicate with multiple receiver, that is to broadcast message.

- **Many to many**: Multiple sender want to communicate with multiple receivers.

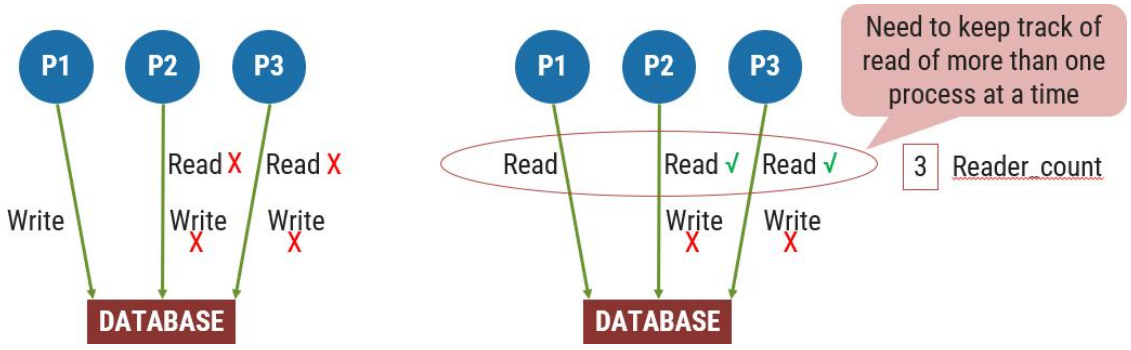
Classical IPC Problems

1. Readers & Writer Problem

2. Dinning Philosopher Problem

1. Readers & Writer Problem

- In the readers and writers problem, many competing processes are wishing to perform reading and writing operations in a database.
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers.



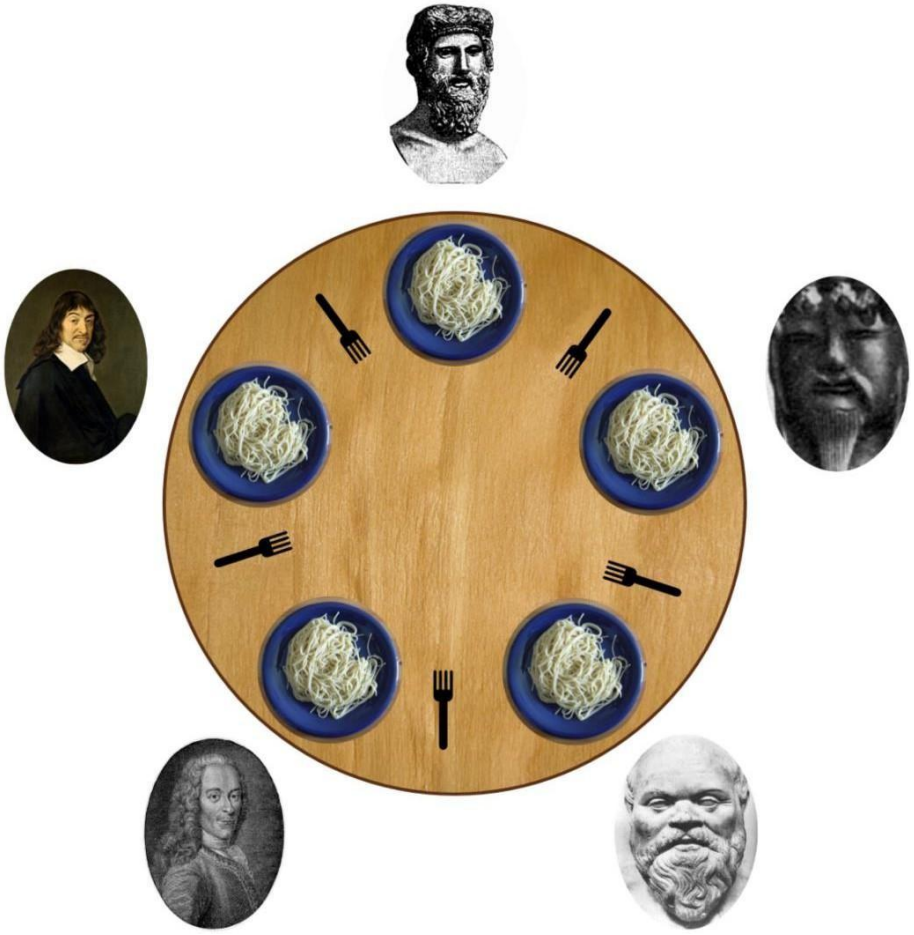

```
typedef int semaphore;
semaphore mutex=1;          //control access to reader
count semaphore db=1;       //control access to database
int reader_count=0;         //number of processes reading
database void Reader (void)
{       while (true) {
        down(&mutex);                       //gain access to reader count
        reader_count=reader_count+1;        //increment reader counter
        if(reader_count==1)                 //if this is first process to read
                                            DB
            { down(&db)   }                 //prevent writer process to access DB
        up(&mutex)                          //allow other process to access
                                            reader_count
        read_database()
        ;
        down(&mutex);                       //gain access to reader count
        reader_count=reader_count-1;        //decrement reader counter
        if(reader_count==0)                 //if this is last process to read
        DB
          {up(&db) }                        //leave the control of DB, allow writer
        process up(&mutex)                  //allow other process to access
        reader_count use_read_data(); }     //use data read from DB
        (non-critical)
}


void Writer (void)
{       while (true)
{
        create_data();      //create data to enter into DB (non-
```

```
        critical) down(&db);      //gain access to DB
        write_db();           //write information to
        DB up(&db);
}               //release exclusive access to DB
```

```
}
```

2. Dinning Philosopher Problem



- In this problem **5 philosophers sitting at a round table doing 2 things eating and thinking**.
- While **eating** they are **not thinking** and while **thinking** they are **not eating**.
- **Each philosopher has plates** that is total of **5 plates**.
- And there is a **fork place between each pair** of adjacent philosophers that is **total of 5 forks**.
- **Each philosopher needs 2 forks to eat** and each **philosopher can only use the forks on his immediate left and immediate right**.

```
#define N 5                     //no. of philosophers
#define LEFT (i+N-1)%5          //no. of i's left neighbor
#define RIGHT (i+1)%5           //no. of i's right neighbor
#define THINKING 0              //Philosopher is thinking
#define HUNGRY 1                //Philosopher is trying to get forks
#define EATING 2                //Philosopher is eating
typedef int semaphore;          //semaphore is special kind of int
int state[N];                   //array to keep track of everyone's state
semaphore mutex=1;              //mutual exclusion for critical region
semaphore s[N];                 //one semaphore per philosopher
```

Solution to Dinning Philosopher Problem

```
void philosopher (int i)        //i: philosopher no, from 0 to N-1
{
    while (true)
    {
        think();        //philosopher is thinking
        take_forks(i);  //acquire two forks or block
        eat();          //eating spaghetti
        put_forks(i);   //put both forks back on table
    }
}
```

```
void take_forks (int i)     //i: philosopher no, from 0 to N-1
{   down(&mutex);   //enter critical region
    state[i]=HUNGRY;  //record fact that

    test(i);              //try to acquire 2

void test (i)                                   //i: philosopher no,
{ if (state[i]==HUNGRY && state[LEFT]!=EATING &&
state[RIGHT]!=EATING)
        {       state[i]=EATING;
                up (&s[i]); }
}

    up(&mutex)          //exit critical

    down(&s[i]);//block if forks were not
```

```
void put_forks (int i)              //i: philosopher no, from 0 to N-1
{           down(&mutex);           //enter critical region
            state[i]=THINKING;      //philosopher has finished eating
            test(LEFT);             //see if left neighbor can now eat
             test(RIGHT);           // see if right neighbor can now eat
            up(&mutex);             // exit critical region
}
```