Marwadi University

Unit 3
Inter Process Communication (Part-2)

Prepared By: Prof. Foram Chovatiya
Computer Engineering Dept.

Producer Consumer Problem

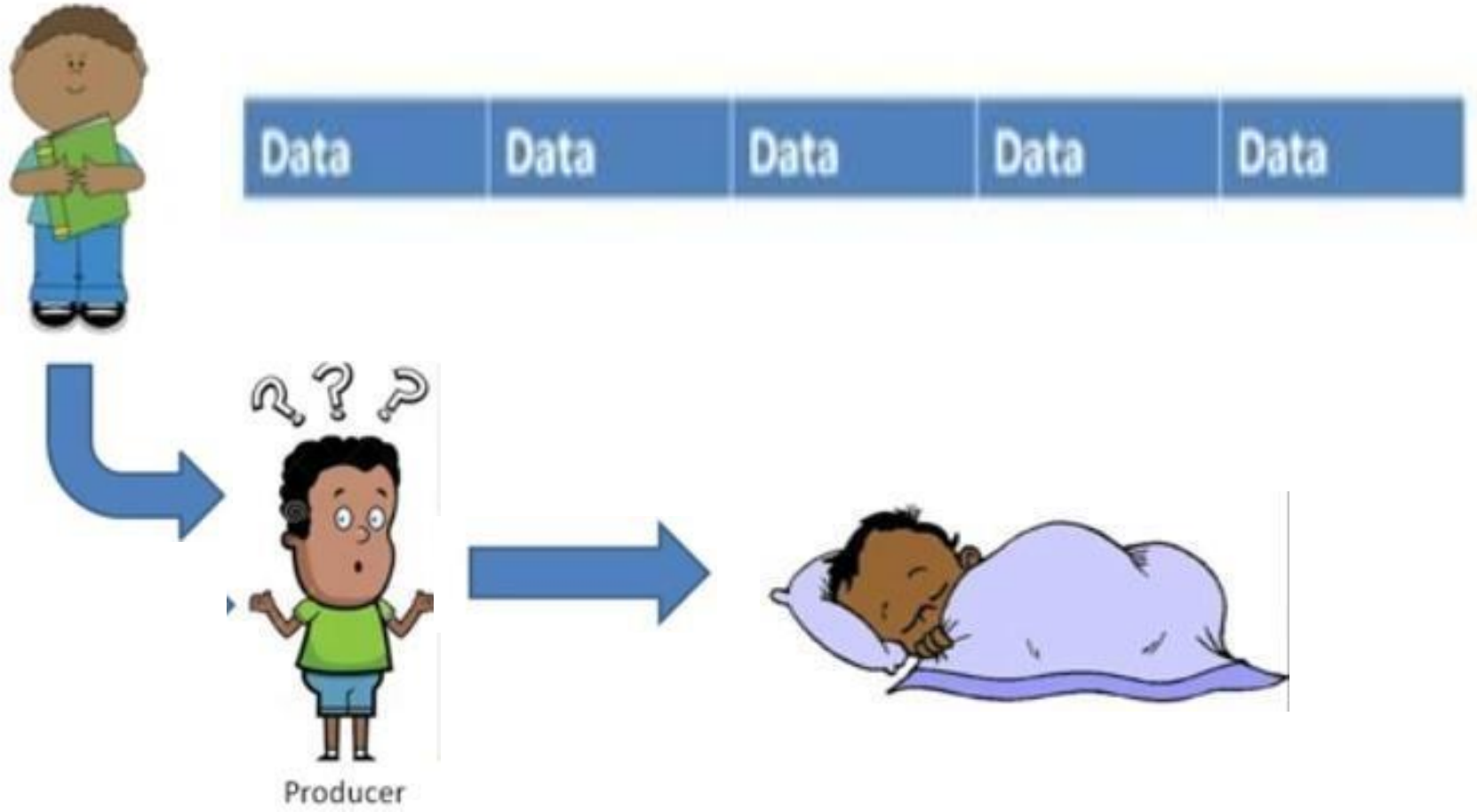
Producer Consumer Problem

- Two processes share a common, fixed-sized buffer
- The producer puts information into the buffer
- The consumer takes it out

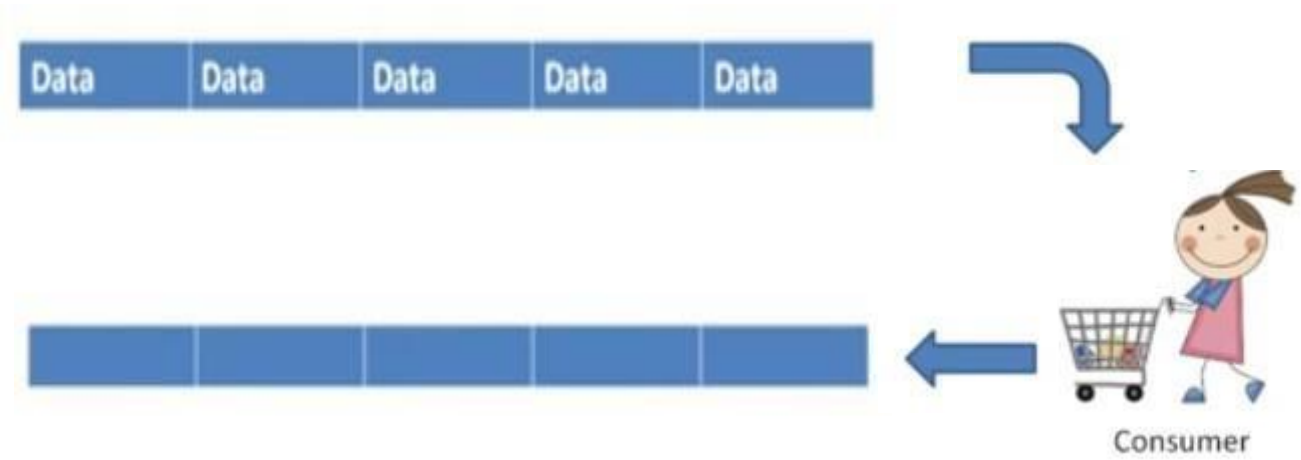
❑ Problem arises when-

- Producer wants to put information into a buffer that is full
- Consumer wants to get information from a buffer that is empty

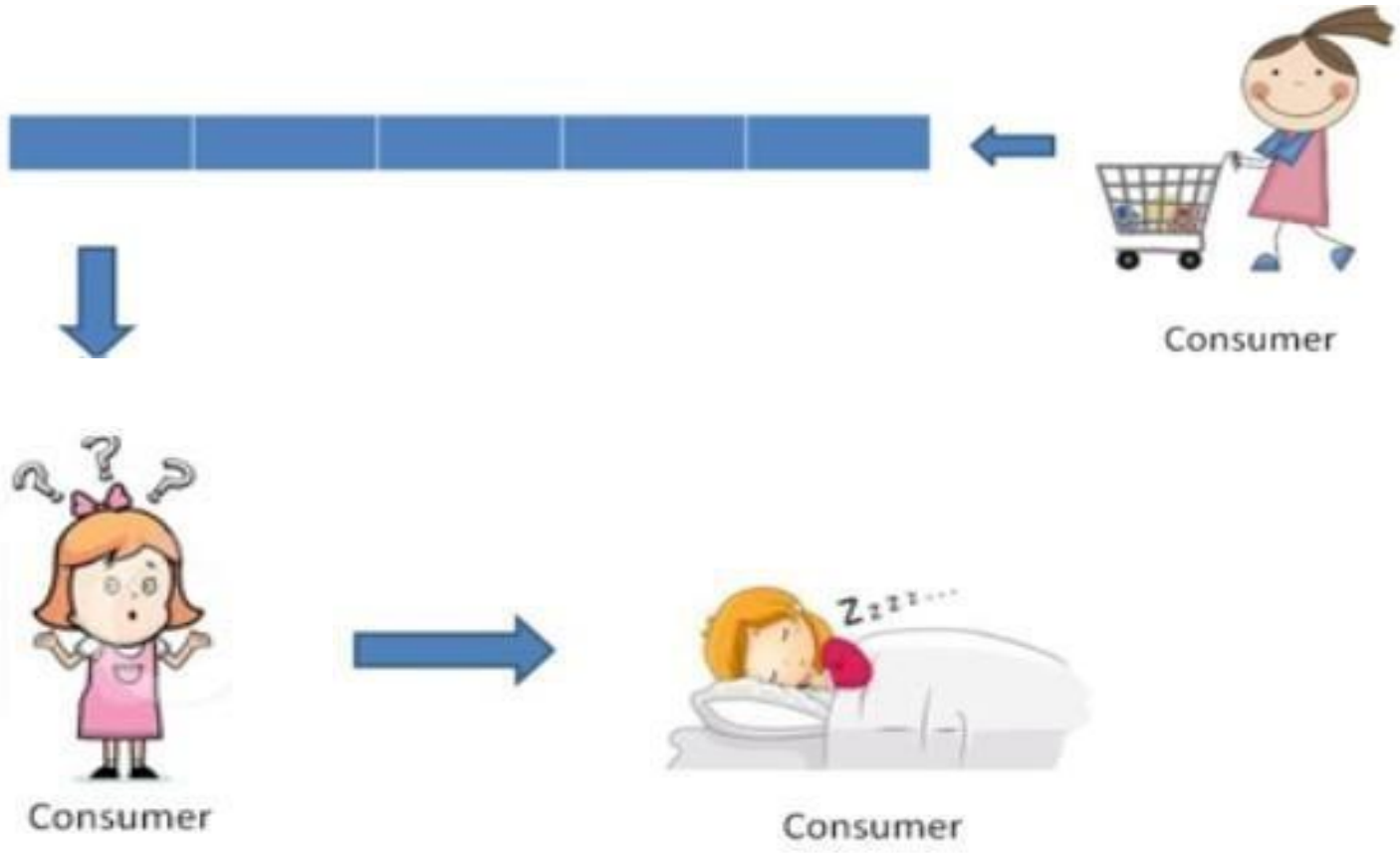
Producer-Consumer Problem



Producer-Consumer Problem

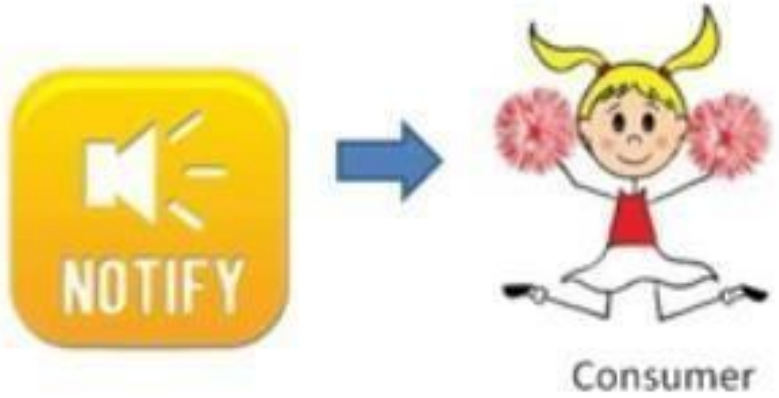


Producer-Consumer Problem



Producer-Consumer Problem

Producer



Consumer

Producer-Consumer Problem


```
#define MAX SIZE 100
int count = 0;

Producer()
{
while (TRUE)
produce item();
if (count == MAX SIZE)
sleep();
enter item();
count = count + 1;
if (count == 1)
wakeup(Consumer);
}
```

```
Consumer()
{
while(TRUE)
if (count == 0)
sleep();
remove item();
count = count - 1;
if (count == MAXSIZE -1 )
wakeup(Producer);
consume item();
}
```

Producer-Consumer Problem

What happens when more than one producer wants to access shared buffer?

Solution : Mutual Exclusion

Race Condition in Problem

- The buffer is empty and the consumer has just read *count* to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer.
- The producer inserts an item in the buffer, increments *count , and* notices that it is now 1. Reasoning that *count was just 0, and thus the consumer must be sleeping,* the producer calls *wakeup to wake the consumer up.*

Race Condition in Problem

- Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost.
- When the consumer next runs, it will test the value of ***count it previously read, find it to be 0, and go to sleep.***
- Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

Race Condition in Problem

- One problem with implementing a Sleep and Wakeup policy is the potential for losing **Wakeups**.
- Semaphores solve the problem of lost wakeups. In the Producer-Consumer problem, semaphores are used for two purposes:
  - Mutual exclusion
  - Synchronization.

Semaphore

Semaphore

- Semaphore is a simply a variable. This variable is used to solve critical section problem and to achieve process synchronization in the multi-processing environment.
- Semaphores are of two types –
  - Binary semaphore
  - Counting semaphore
- Binary semaphore can take the value 0 & 1 only.
- Counting semaphore can take nonnegative integer values.

Semaphore

Marwadi University

Two Standard operation of Semaphore

- Wait
- Signal

- Entry to the critical section is controlled by the **wait** operation and exit from a critical section is taken care by **signal** operation.

Semaphore

The wait, signal operations are also called P and V operations.

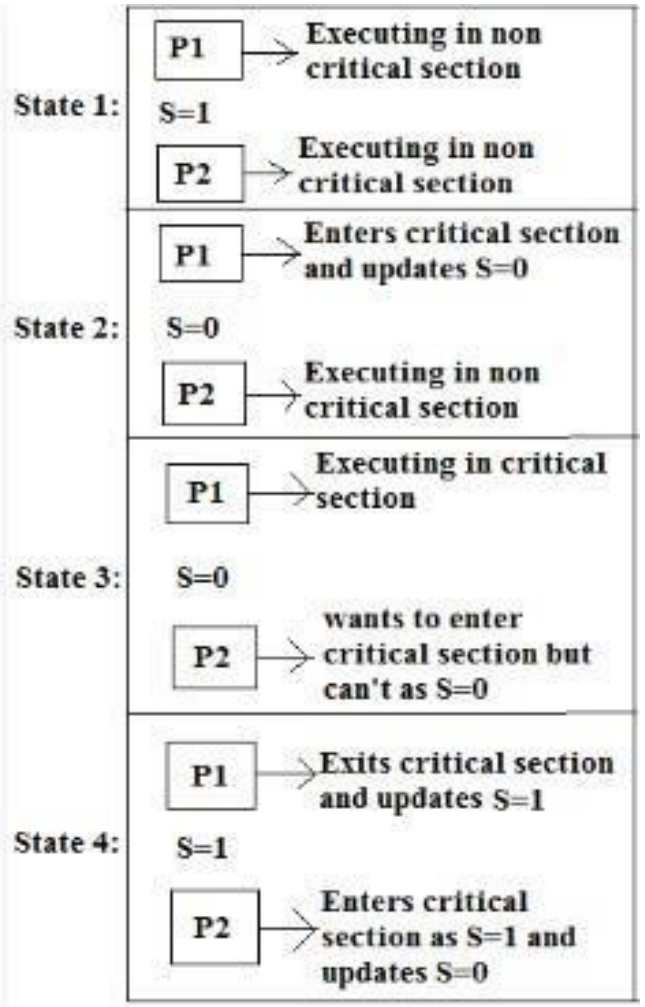
- P also called wait, sleep or down operation.
- V also called signal, wake up or up operation.

Process P

```
//Some code
P(s);
    // critical section
V(s);
    // remainder section
```

Semaphore

- Two processes P1 and P2 and a semaphores are initialized as 1.
- Now if suppose P1 enters in its critical section then the value of semaphore's becomes 0.
- Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphores.



Semaphore


```
P(Semaphore s)
{
   s = s - 1;
   if (s < 0) {

      // add process to queue
      block();
   }
}

V(Semaphore s)
{
   s = s + 1;
   if (s >= 0) {

      // remove process p from queue
      wakeup(p);
   }
}
```

Semaphore

3 semaphores: full, empty and mutex

- ***Full*** counts full slots (initially 0)
- ***Empty*** counts empty slots (initially N)
- ***Mutex*** protects variable which contains the items produced and consumed. (Binary Semaphore)

Producer-Consumer Problem

Marwadi University

```
procedure producer() {
while (true) {
    item = produceItem();
    down(emptyCount);
    down(buffer_mutex);
    putItemIntoBuffer(item);
    up(buffer_mutex);
    up(fillCount);
}
}
```

```
procedure consumer() {
while (true) {
    down(fillCount);
    down(buffer_mutex);
    item = removeItemFromBuffer();
    up(buffer_mutex);
    up(emptyCount);
    consumeItem(item);
}
}
```

Producer-Consumer Problem

```
semaphore mutex = 1;
Producer()
{
int item;
while (TRUE)
{
produce item( item);
wait( mutex);
enter item(item);
signal( mutex);
}
}
```

```
Consumer()
{
int item;
while (TRUE)
{
wait( mutex);
remove item( item);
signal( mutex);
Consume item( );
}
}
```

Monitor

Monitor

- A Monitor is used to provide control access of shared resources in concurrent process.
- Monitor is supported by programming languages to achieve mutual exclusion between processes.
- For example ➔ Java Synchronized methods.

  Java provides wait() and notify() constructs.

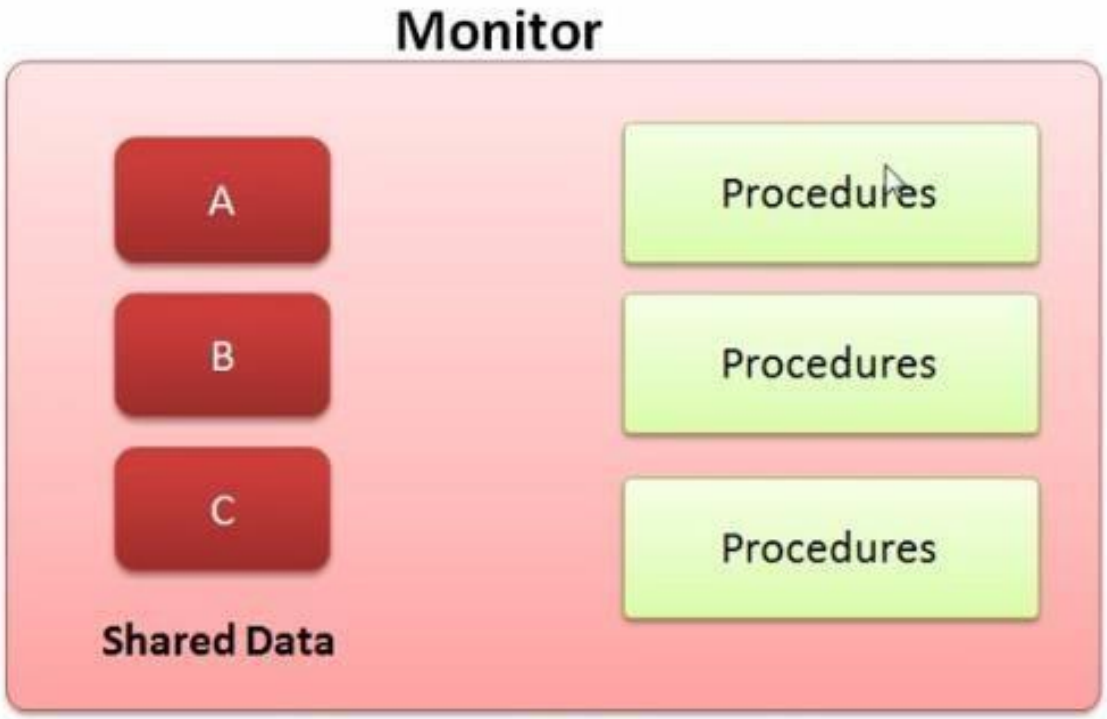
Monitor

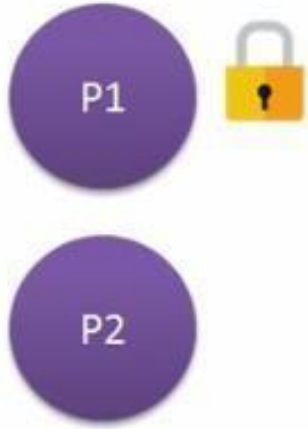
- It is the collection of condition variables and procedures combined together in a special kind of module or a package.

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

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


}
```

Syntax of Monitor

Monitor - Example





Only one process can enter at a time

Classical IPC Problems

Classical IPC Problems

1. Bounded Buffer Problem
2. The Readers and Writers Problem
3. Dining Philosophers Problem

Bounded Buffer Problem

Bounded Buffer Problem (Producer Consumer Problem)

- This problem is generalized in terms of the Producer Consumer problem, where a finite buffer pool is used to exchange messages between producer and consumer processes.
- Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem.**

Solution using Monitor

Procedures:

Get_item, Put_item

Variable:

Count

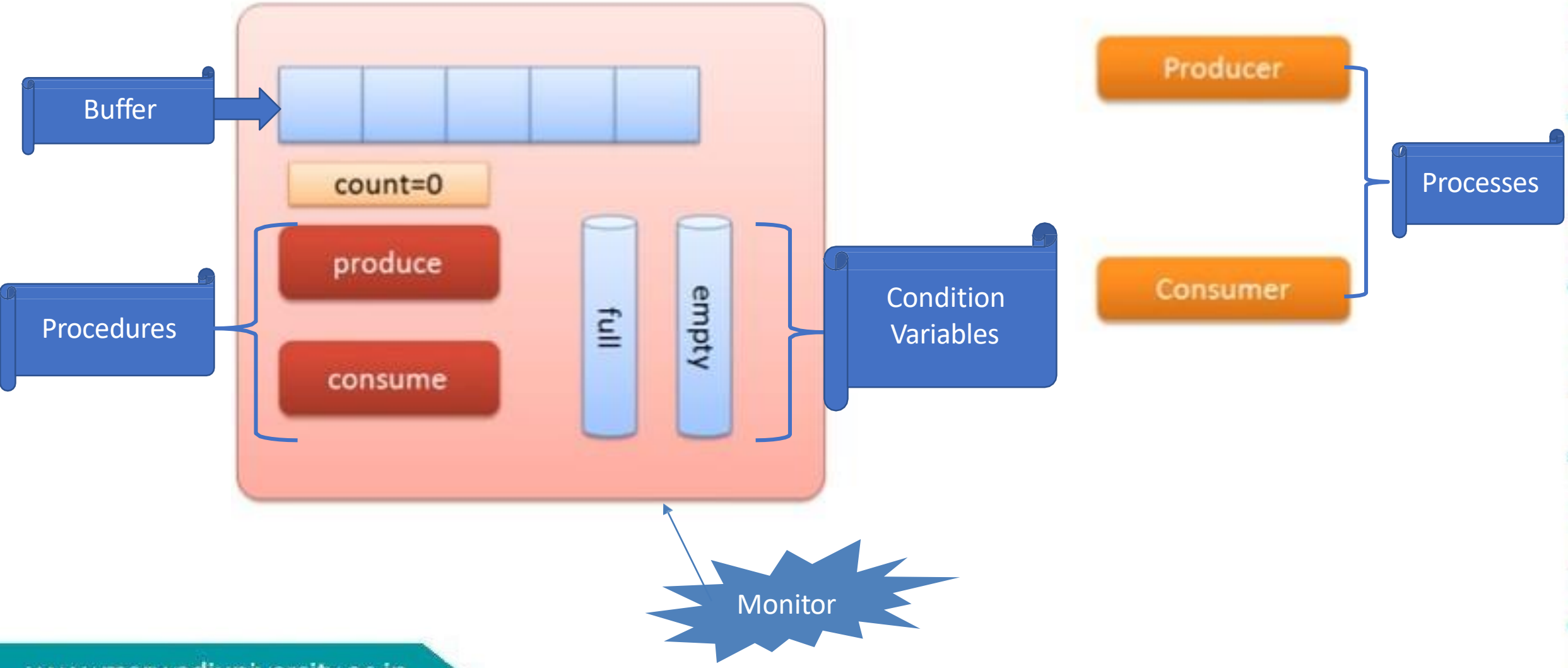
Condition Variables:

Empty, Full

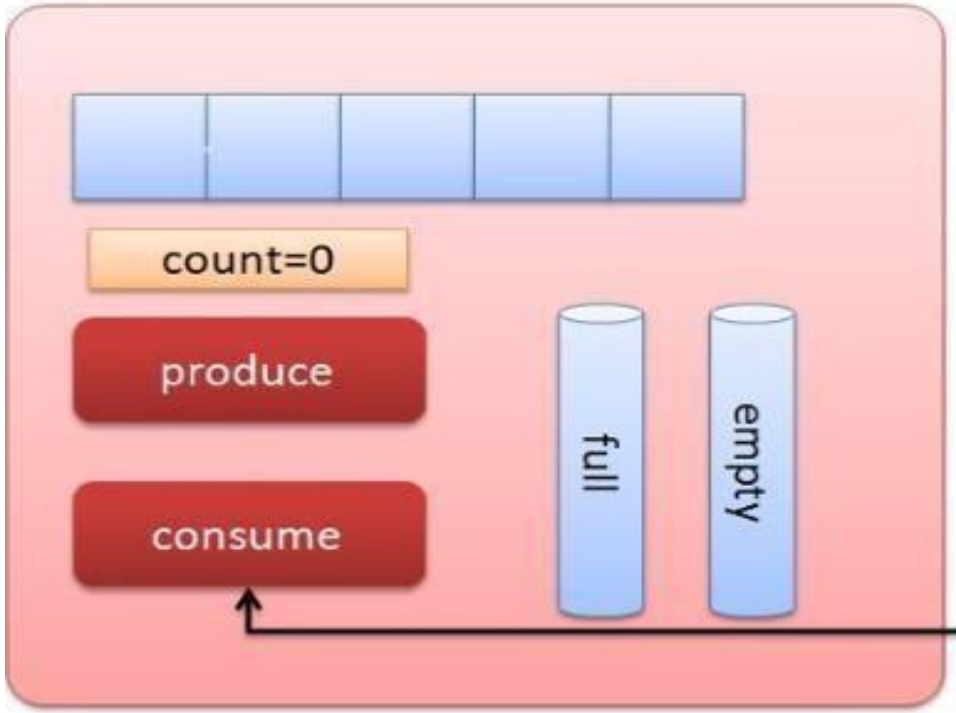
- Two different operations are performed on the condition variables of the monitor. ➔Wait, Signal
- Lets say we have condition variable x.

  Then x.wait() x.signal()

Solution using Monitor



Solution - Consumer



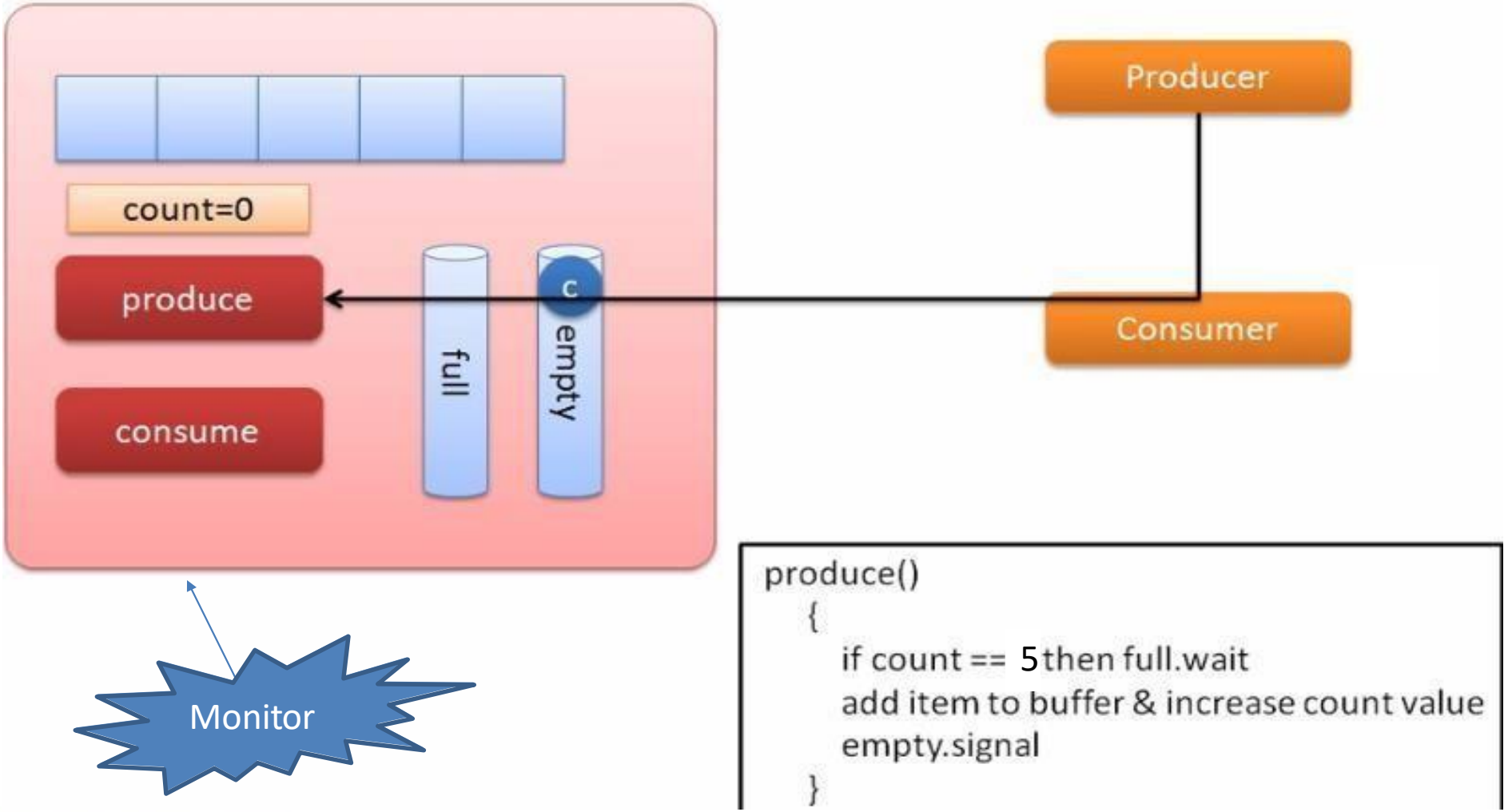
Producer

Consumer

```
counsume()
   {
      if count == 0 then empty.wait
      access item from buffer and decrease
value of count
            Full.  signal
   }}
```

Monitor

Solution - Producer




```
produce()
   {
      if count == 5 then full.wait
      add item to buffer & increase count value
      empty.signal
   }
```

Solution using Monitor

Marwadi University

```
monitor ProducerConsumer
    condition full, empty;
    int count;

procedure_insert();
{
    if (count == N) wait(full);        // if buffer is full, block
    insert_item(item);                 // put item in buffer
    count = count + 1;                 // increment count of full slots
    if (count == 1)  signal(empty);    // if buffer was empty, wake consumer
}
```

Solution using Monitor

```
    procedure_remove();
    {
      if (count == 0)  wait(empty);      // if buffer is empty, block
      remove_item(item);                 // remove item from buffer
      count = count - 1;                 // decrement count of full slots
      if (count == N-1)  signal(full);   // if buffer was full, wake producer
    }
end monitor;
```

The Readers- Writers Problem

Reader Writer Problem

- The readers-writers problem relates to an object such as a file that is shared between multiple processes.
- Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.
- The Readers-Writers problem defines the situation where database access is required.

Reader Writer Problem

- If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state. Only one process should be allowed to change the value of the data present in the file at a particular instant of time.

Reader Writer Problem

- Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read because the process writing on the file will change the value of the file, but the process reading that file will read the old value present in the file. So, this should be avoided.

Reader Writer Problem

Reader

Multiple readers can read simultaneously

At time of reading, writing is not allowed.

Writer

Only one writer allowed to write at a time.

At time of writing, reading is not allowed.

Reader Writer Problem - Criteria

- One set of data is shared among a number of processes
- Once a writer is ready, it performs its write. Only one writer may write at a time
- If a process is writing, no other process can read it
- If at least one reader is reading, no other process can write

Solution – using Mutex Semaphore

Marwadi University

Writer process:

- Writer requests the entry to critical section.
- If allowed i.e. wait() gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
- It exits the critical section.

Solution-**Mutex Semaphore for Writer**

Marwadi University

```
while(TRUE)
{
        wait(wrt);
                /*perform the write operation */
        signal(wrt);
}
```

Solution – using Mutex Semaphore

Reader process:

❑If allowed:

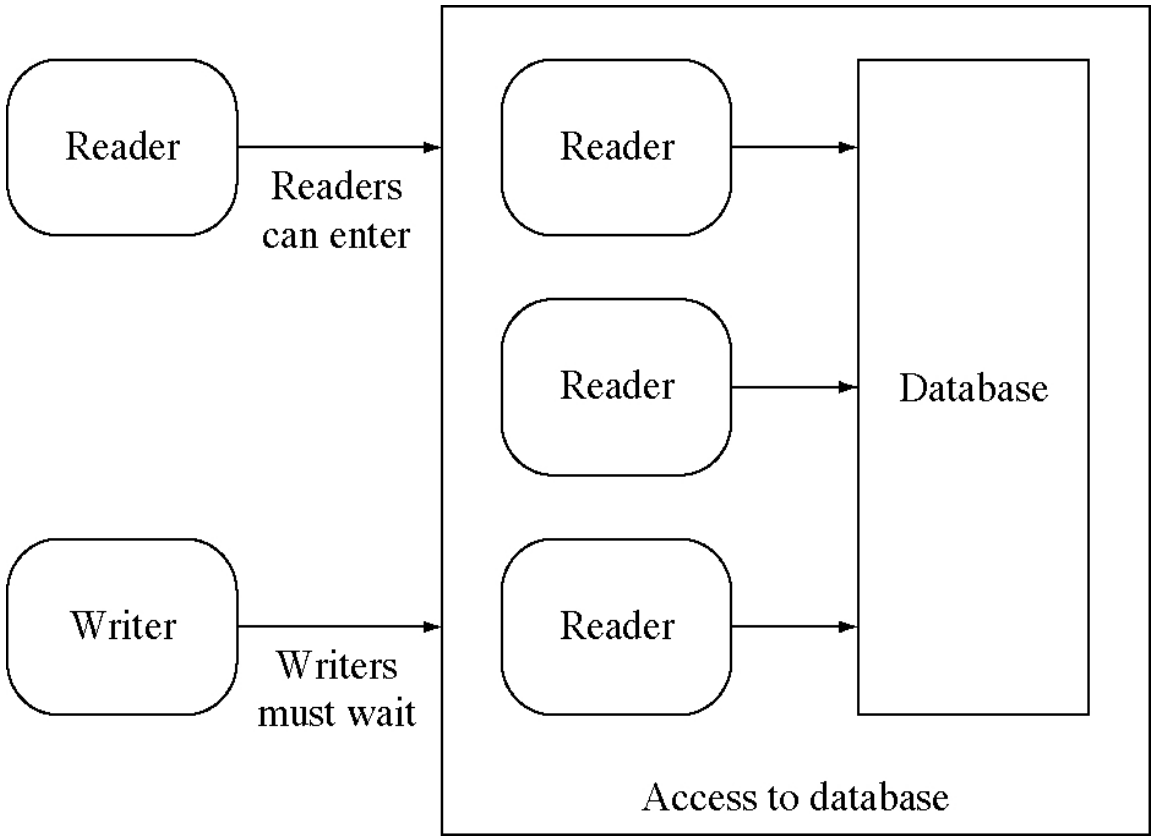
- it increments the count of number of readers inside the critical section. If this reader is the first reader entering, it locks the wrt semaphore to restrict the entry of writers if any reader is inside.
- It then, signals mutex as any other reader is allowed to enter while others are already reading.
- After performing reading, it exits the critical section. When exiting, it checks if no more reader is inside, it signals the semaphore “wrt” as now, writer can enter the critical section.

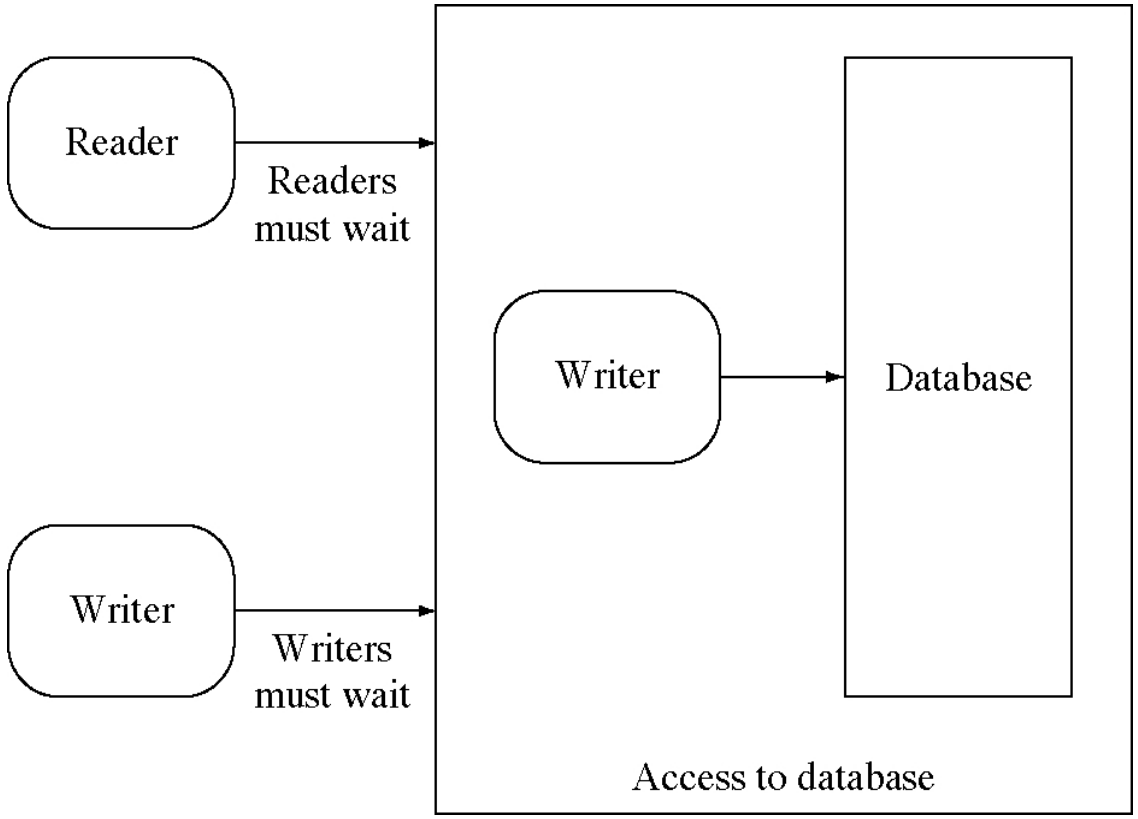
❑If not allowed, it keeps on waiting.

Solution - **Mutex Semaphore for Reader**

```
read_count=0
while(TRUE)
{
    wait(m) //acquire  a lock
    read_count++;
    If (read_count==1)
        wait(wrt);          //this ensure no writer can enter if there is even one reader
        signal(m);      //other readers can enter while this current reader is inside one reader
         /* perform the reading operation */
                                wait(m); // a reader wants to leave, acquire a lock
                                    read_count--;

                                if(read_count == 0)

                                    signal(wrt);          // writers can enter ,release lock
                                signal(m); // reader leaves
}
```

Reader Writer

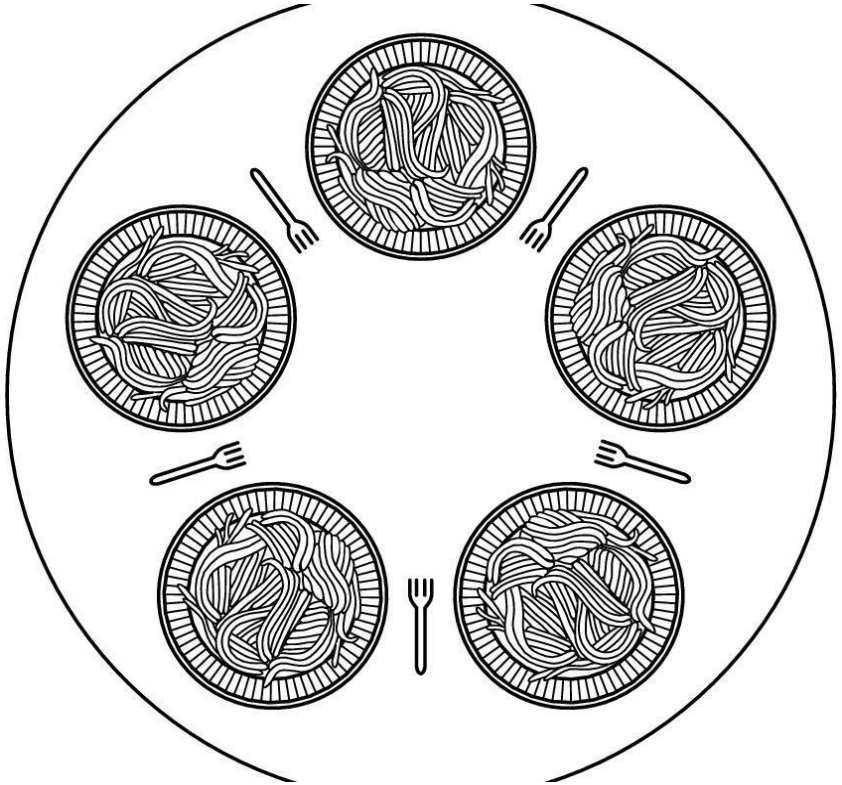




Dining Philosophers Problem

Dining Philosopher

- Dining philosopher problem defines the situation where processes complete for limited resources
- Five Philosophers seated around the circular table
- Each has a plate of Spaghetti
- Each needs two forks to eat it
- Between each pair of plates there is one fork



Dining Philosopher

At any instant, a philosopher is

- either eating or thinking.
- When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right.
- When a philosopher wants to think, he keeps down both chopsticks at their original place.

Dining Philosopher

Marwadi University

```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Li);
      put_fork(Ri);
  }
}
```

Dining Philosopher

There are three states of philosopher

THINKING

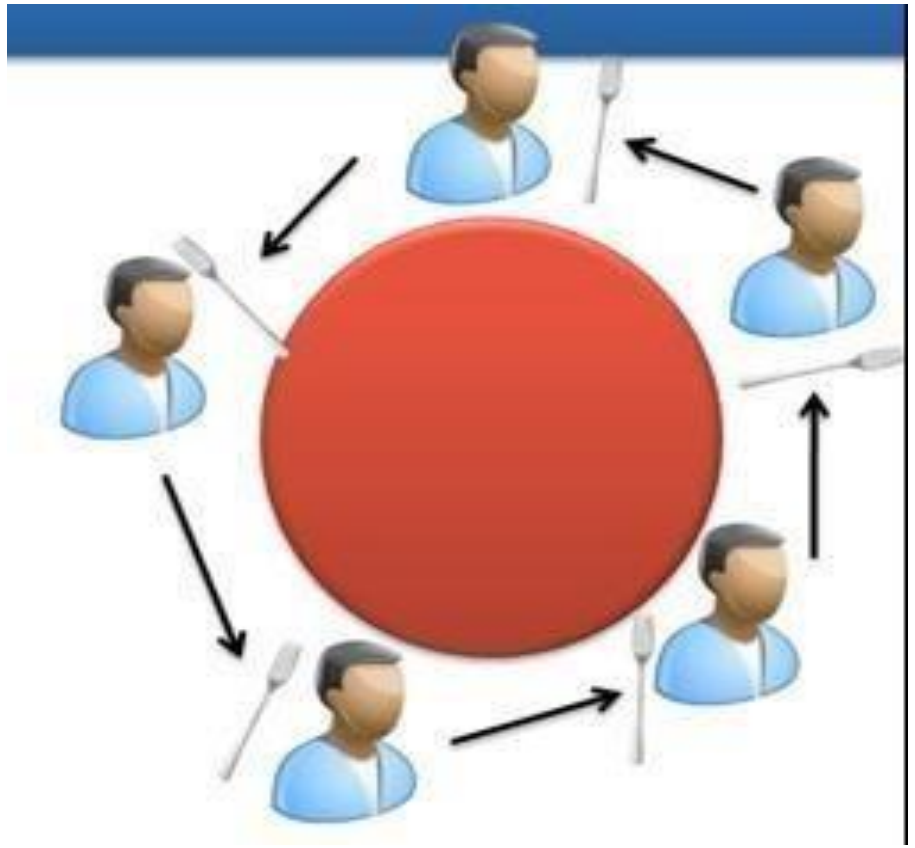
HUNGRY

EATING.

```
process P[i]

while (true ) do

 { THINK;
   PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
   EAT;
   PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
 }
```

Problem

- But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.



Solution

- After taking the right fork, philosopher will check if its left fork is available
- If it's not, philosopher puts back her right fork, wait for some times and proceeds again in similar way

```
#define N 5

void philosopher(int i){
 while(TRUE){
     think();
     take_fork(Ri);
     if (available(Li){
       take_fork(Li);
       eat();
       put_fork(Ri);
       put_fork(Li);
     }else{
        put_fork(Ri);
        sleep(T)
```

Problem

Marwadi University

❑ Again Problem:

- What if all philosophers start simultaneously?
- They will never find their left fork available ➔causing - **starvation**

Solution

- Well, there is a solution that will stop **deadlock** and **starvation**
- No Two neighbor Philosopher should be allowed to access chopstick at same time.
- **Solution:- using Monitor**

Solution using Monitor

	P1	P2	P3	P4	P5
state	T	T	T	T	T
semaphore	0	0	0	0	0

	P1	P2	P3	P4	P5
state	T	T	H	T	T
semaphore	0	0	0	0	0

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	1	0	0

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

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

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

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

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

Figure 5.18 A monitor solution to the dining-philosopher problem.

Marwadi University

For “i” process
LEFT – (i + 4) % 5
RIGHT - (i + 1) % 5

```
process  P[i]
while (true )
do
{
THINK;
PICKUP(P[i])
EAT;
PUTDOWN(P[i])
}
```

Thank you!!