GLA UNIVERSITY MATHURA

Accredited with A Grade by NAAC

Operating System

Process Synchronization

Class Presentations on Operating System

by Subhash Chand Agrawal

Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data (Variable, code, resource, memory, etc.)
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

- **Information sharing**

In the information sharing at the same time, many users may want the same piece of information(for instance, a shared file) and we try to provide that environment in which the users are allowed to concurrent access to these types of resources.

- **Computation speedup**

When we want a task that our process run faster so we break it into a subtask, and each subtask will be executing in parallel with another one. It is noticed that the speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).

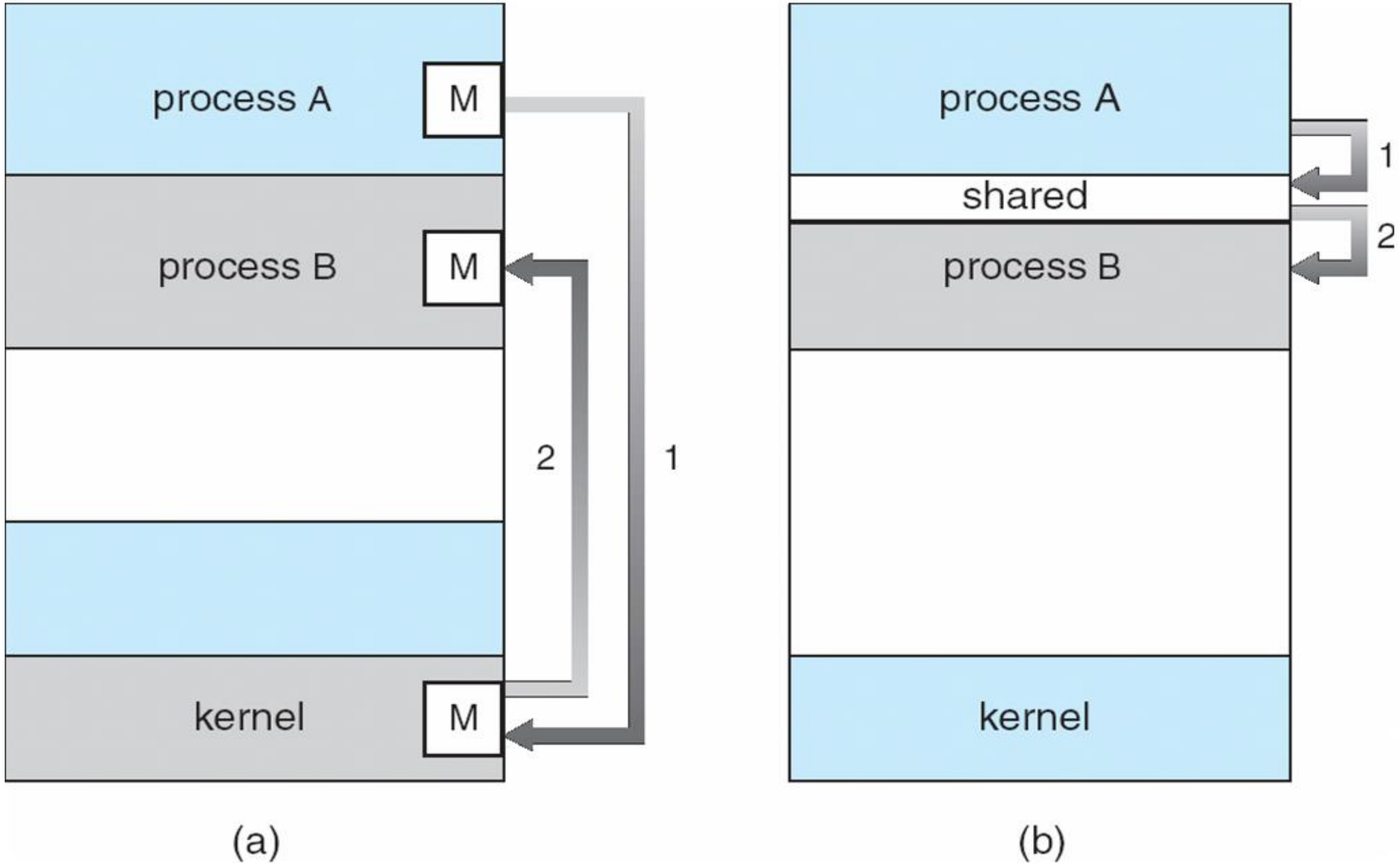
- **Modularity**

In the modularity, we are trying to construct the system in such a modular fashion, in which the system dividing its functions into separate processes.

- **Convenience**

An individual user may have many tasks to perform at the same time and the user is able to do his work like editing, printing and compiling.

Communications Models



Concurrent Processes

Two processes are said to be concurrent if they overlap in their execution.

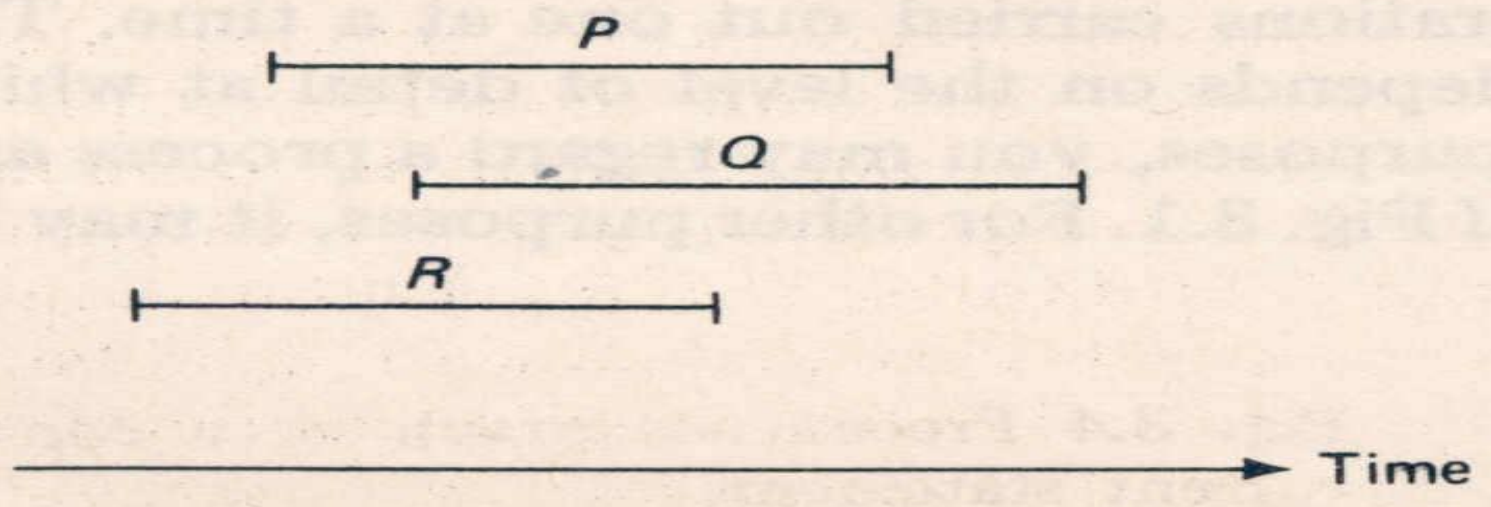


Fig. 3.2 Three concurrent processes *P*, *Q*, and *R*.

Example of inconsistent view

- bool withdraw(int withdrawal)
- {
-     if (balance >= withdrawal)
-     {
-         balance -= withdrawal;
-         return true;
-     }
-     return false;
- }
- Suppose balance = 500, and two concurrent threads make the calls withdraw(300) and withdraw(350).

Bounded-Buffer Problem

- There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer.
- The same memory buffer is shared by both producers and consumers which is of fixed-size.
- The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.
- The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
- Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
- Accessing memory buffer should not be allowed to producer and consumer at the same time

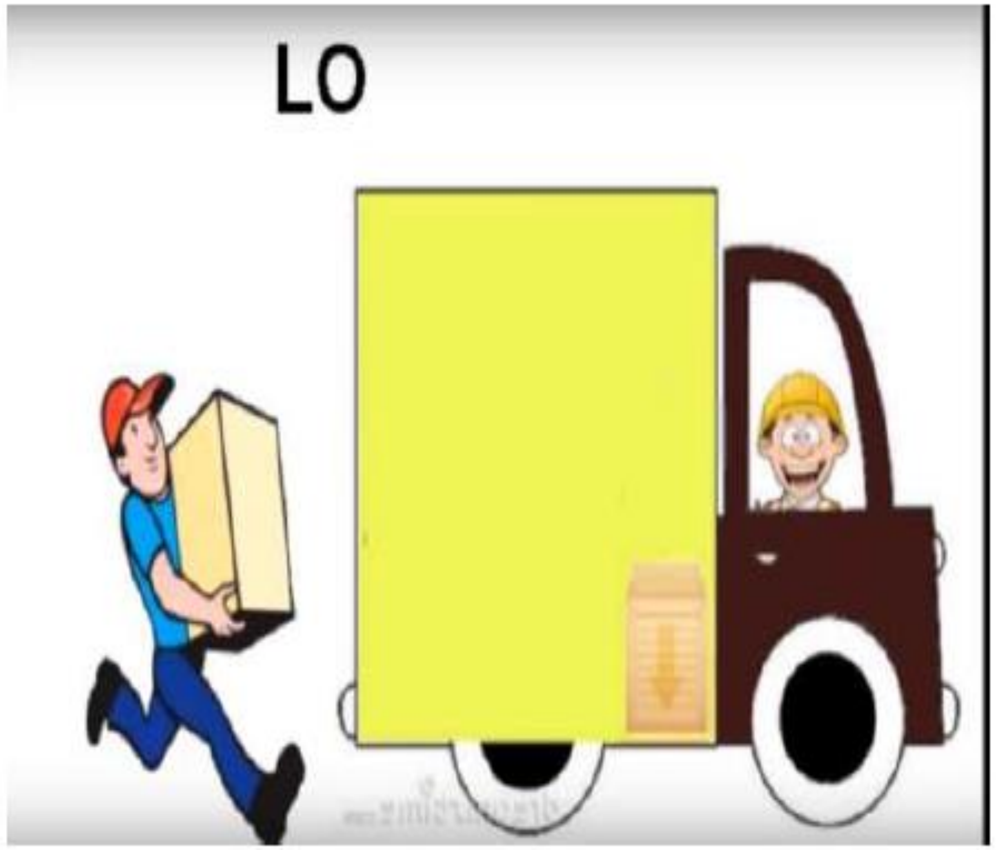
❑ Real life Example

Picture 1: Full of items

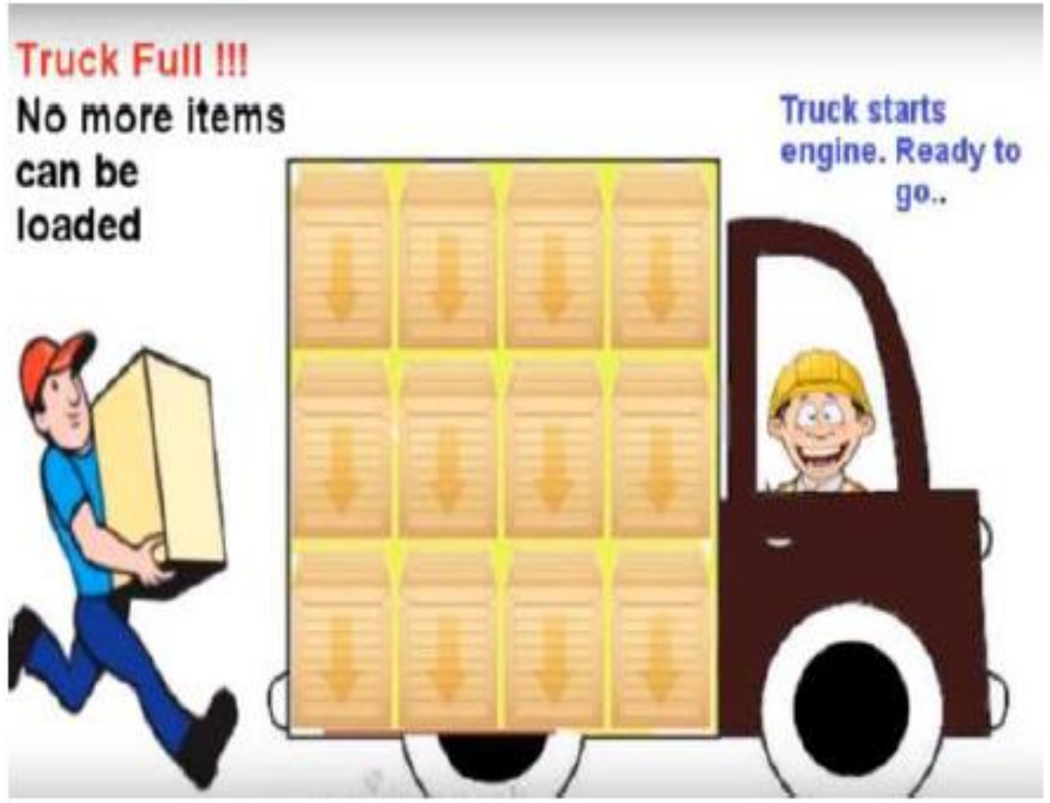
Picture 2: Lorry is empty

❑ Real life Example

Picture 3 : Working for load

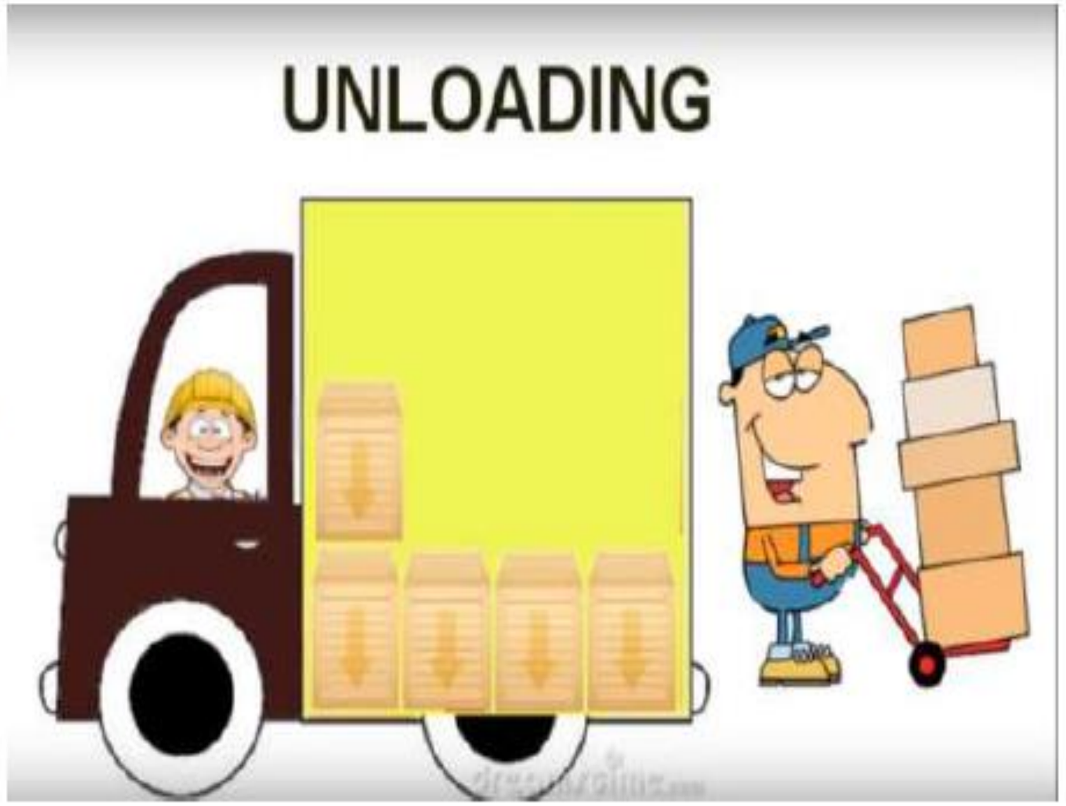


Picture 4 : Loading



Picture 5 : Truck is full

Picture 6 : Still many items

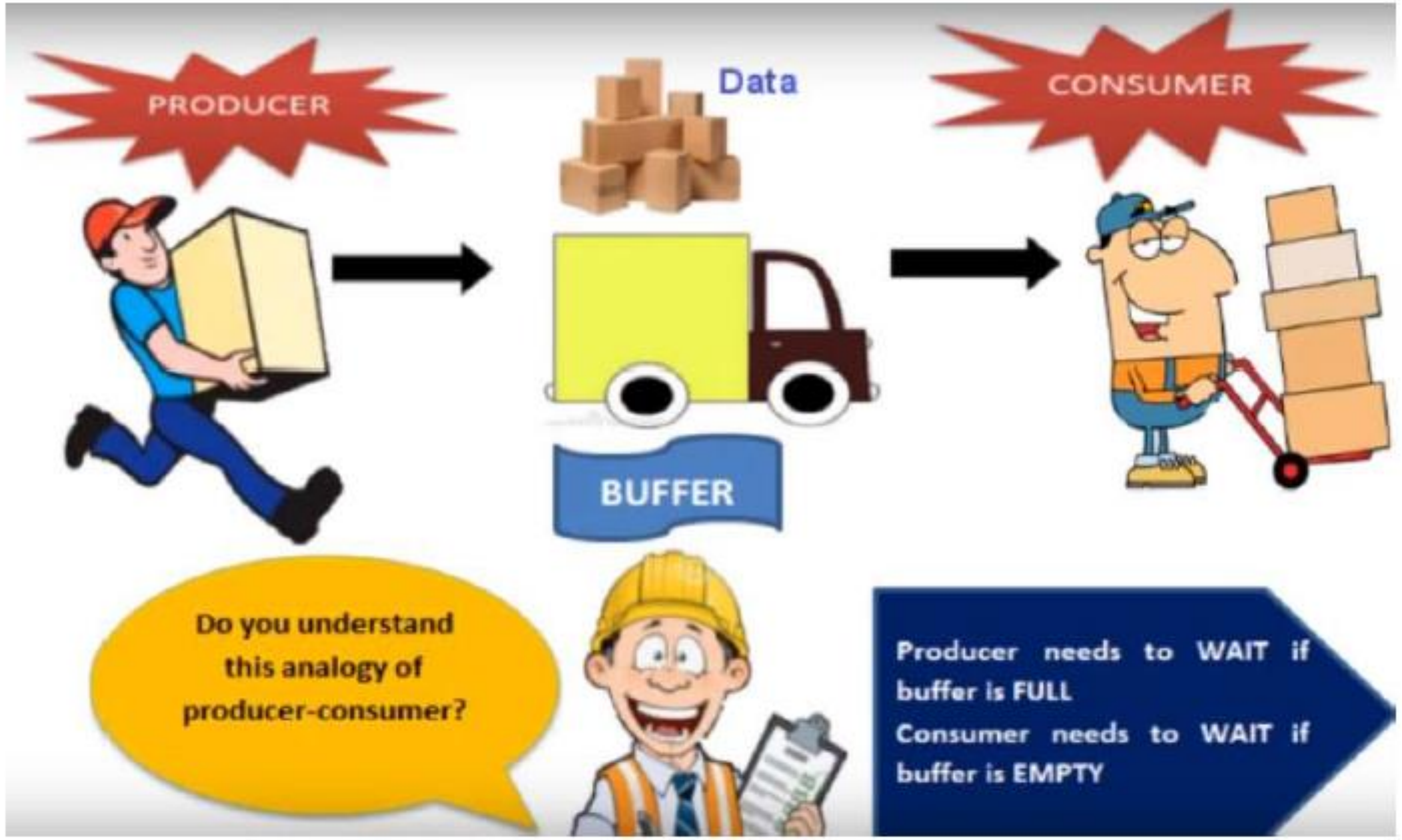


Picture 7 : Truck is full & unloaded

Picture 8 : Uloading

Picture 9 : Truck is empty

Picture 10 : Waiting



Picture: All Process

Solution to Bounded Buffer Problem :

Data Structure :

1. Circular Array (shared Pool of Buffers)
2. Two Logical Pointers : ‘in’ and ‘out’

   in points to the next free position in the buffer.

   out points to the first full position in the buffer.
3. integer variable counter initialized to 0 .

counter is incremented every time a new full buffer is added to the pool and decremented whenever one is consumed

```
Type      item= ….;
      Var buffer : array[0..n-1] of item;
      in, out : 0..n-1
      nextp, nextc  : item; in := 0, out := 0;
      Counter:=0;
```

Producer Process

- item nextProduced;
- while (1) {
- while (counter == BUFFER_SIZE); /* do nothing */
- buffer[in] = nextProduced;
- in = (in + 1) % BUFFER_SIZE;
- count++;
- }

Consumer process

- item nextConsumed;
- while (1) {
- while (counter == 0); /* do nothing */
- nextConsumed = buffer[out];
- out = (out + 1) % BUFFER_SIZE;
- count--;
- }

Bounded-Buffer – Shared Counter

- The statements

  count++;
  count--;

  must be performed *atomically*.
- Atomic/Indivisible operation means an operation that completes in its entirety without interruption.

Bounded-Buffer – Shared Counter

- The statement “**count++**” could be implemented in machine language as:

  register1 = count

  register1 = register1 + 1

  count = register1

- The statement “**count--**” could be implemented as:

  register2 = count

  register2 = register2 – 1

  count = register2

Bounded-Buffer – Shared Counter

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- The interleaving depends upon how the producer and consumer processes are scheduled.

Bounded-Buffer – Shared Counter

- Consider this execution interleaving with "count = 5" initially:

  producer: **register1 = count** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = count** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **count = register1** (*count = 6*)
  consumer: **count = register2** (*count = 4*)

- The value of **count** may be either 4 or 6, whereas the correct result should be 5.

Race condition

- A Race condition is a scenario that occurs in a multithreaded environment where multiple threads sharing the same resource or executing the same piece of code. If not handled properly, this can lead to an undesirable situation, where the output state is dependent on the order of execution of the threads.

WHY INCORRECT ?

Because we allowed processes to manipulate the variable counter concurrently.

REMEDY :

We need to ensure that only one process at a time may be manipulating the variable counter.

This observation leads us to problem of **CRITCIAL SECTION (CS)** and the part of the program code where process is executing **shared variable is known as its Critical Section.**

Problem Definition:

- Consider a system consisting of n cooperating processes { P_1, P_2,.. P_n}

- Each process has a segment of code called a Critical Section (CS)

- When one process is executing in its CS no other process is to be allowed to execute in its CS.

- Thus the execution of CS by the processes in Mutually Exclusive in time.

Critical Section Problem :

General structure may be:

……

Entry Section

……

Critical Section

…….

Exit Section

Remainder Section

- A solution to the critical-section problem must satisfy the following 3 requirements :
- **Mutual Exclusion :**

Only one process at a time in the critical section.

- **Progress:**

No process running outside the critical section should block the other interesting process from entering into a critical section when in fact the critical section is free.

Bounded Waiting :

- No process should have to wait forever to enter into the critical section. there should be a boundary on getting chances to enter into the critical section.
- If bounded waiting is not satisfied then there is a possibility of starvation.

P-1

- In which the access takes place when different processes try to access the same data concurrently and the outcome of the execution depends on the specific order, is called
- A. dynamic condition
- B. race condition
- C. essential condition
- D. critical condition
- Ans: B

P-2

- Which of the following option is suitable when a process is executing in its critical section, then no other processes can be executing in their critical section
- A. mutual exclusion
- B. critical exclusion
- C. synchronous exclusion
- D. asynchronous exclusion
- Ans: A

Algorithm 1: let the processes share a common integer variable turn initialized to 0(or 1).

Check Mutual Exclusion

Check Progress

Process P_0

```
While(1)
{
While (turn !=0 ) ;

<Critical Section>

Turn=1;
....
}
```

Process P_1

```
...
While (turn !=1 );

<Critical Section>

Turn=0;
....
```

Algorithm 2:

var flag : array[0..1] of boolean ; (initial to false)

If flag[i] is true then Pi is executing in its CS.

Process P_0

...

While (Flag[1]==true) ;

Flag[0] =True

<Critical Section>

Flag[0]=False;

....

Process P_1

...

While (Flag[0]==true);

Flag[1]=True

<Critical Section>

Flag[1]=False;

....

Analysis

ME is not Ensured :

T0 : P0 enters the while statement and finds flag[1] = false

T1 : P1 enters the while statement and finds flag[0] = false

T2 : P1 sets flag[1] and enters CS.

T3 : P0 sets flag[0] and enters CS.

Progress Satisfied

Algorithm 3:

Process P_0

...
flag[0] := true;

while (flag[1]==true) ;

critical section

flag[0] := false;

remainder section ;

....

Process P_1

...
flag[1] := true;

while (flag[0]==true) ;

critical section

flag[1] := false;

remainder section ;

....

Analysis:

The mutual–exclusion requirement is satisfied.

Unfortunately, the **progress** requirement is **not met.**

To illustrate this problem, consider the following execution sequence.

T_0: P_0 set flag [0] = true.

T_1: P_1 set flag [1] = true.

Now P_0 and P_1 are looping forever in their respective while statements

Thank You?