Introduction to System Programming

Revision

Minal Shah

Outline

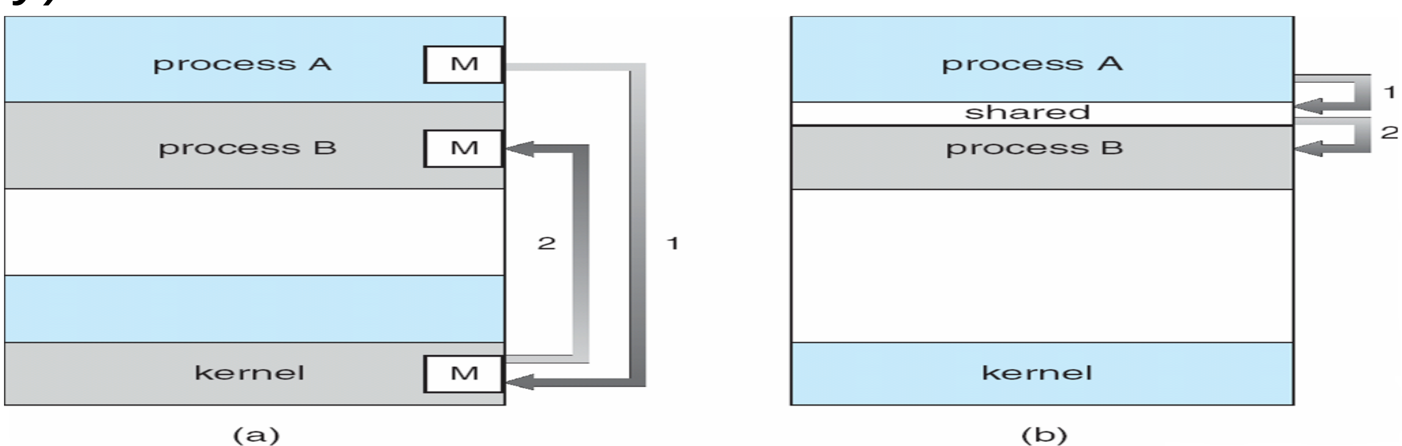
- IPC
- Race Condition
- Critical Region
- Mutual Exclusion
- Semaphores
- Monitors
- Mutex

Inter process communication (IPC)

- 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 of process cooperation
    1. Information sharing
    2. Computation speed-up
    3. Modularity
    4. Convenience

Inter process communication (IPC)

- Issues of process cooperation
  - Data corruption, deadlocks, increased complexity
  - Requires processes to synchronize their processing
- There are two models for IPC
  - a. **Message Passing** (Process A send the message to Kernel and then Kernel send that message to Process B)
  - b. **Shared Memory** (Process A put the message into Shared Memory and then Process B read that message from Shared Memory)

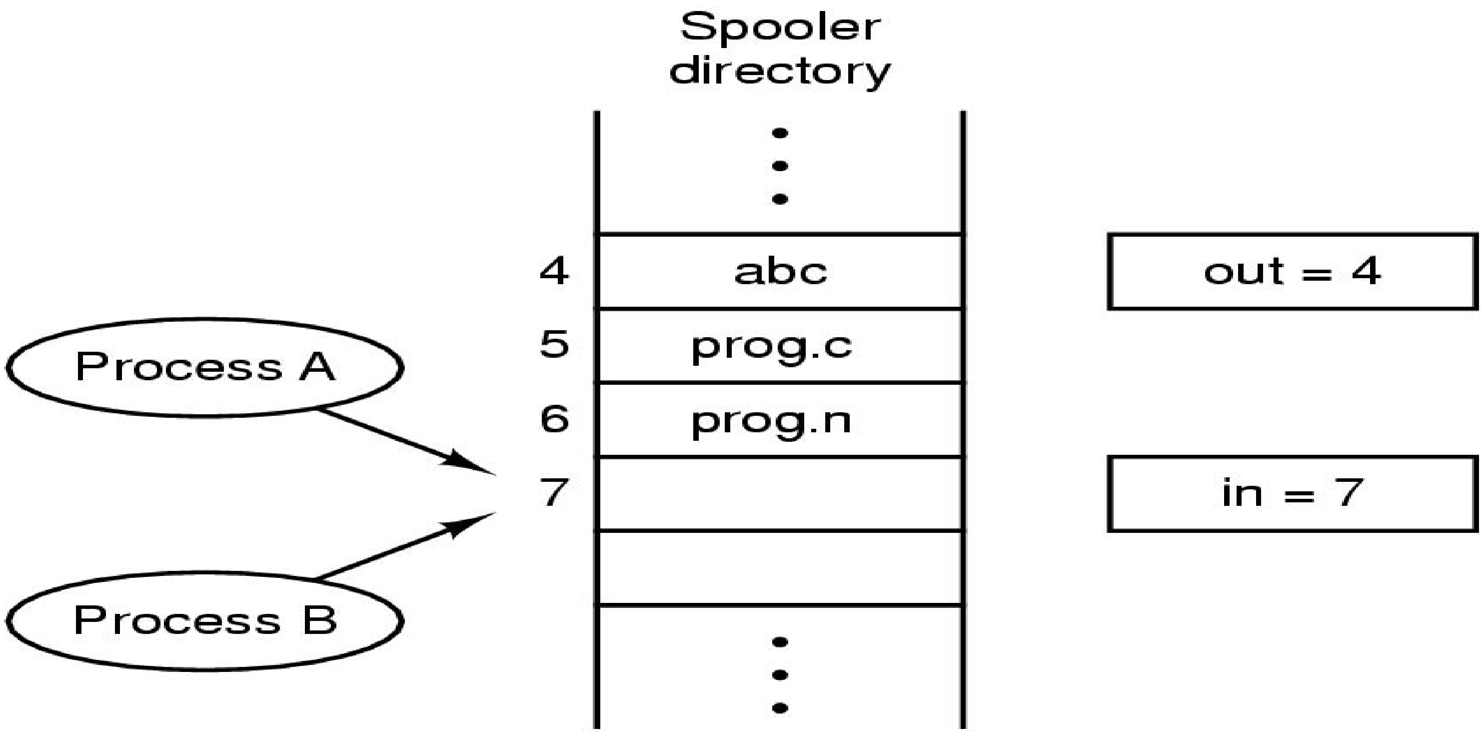


Race Condition

- **Race Condition**:
  - 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.

Example of Race Condition

- Print spooler directory example : Two processes want to access shared memory at the same time.



Example of Race Condition

- Process A

  next_free_slot = in

  Write file name at slot (7)

  next_free_slot += 1

  in = next_free_slot (8)

  Context Switch

- Process B

  next_free_slot = in

  Write file name at slot (8)

  next_free_slot += 1

  in = next_free_slot (9)

- Process A

  next_free_slot = in (7)

  Context Switch

- Process B

  next_f

  W (7)

  slot (8)

  t Switch

  Race Condition

- Process A

  Write file name at slot (7)

  next_free_slot += 1

  in = next_free_slot (8)

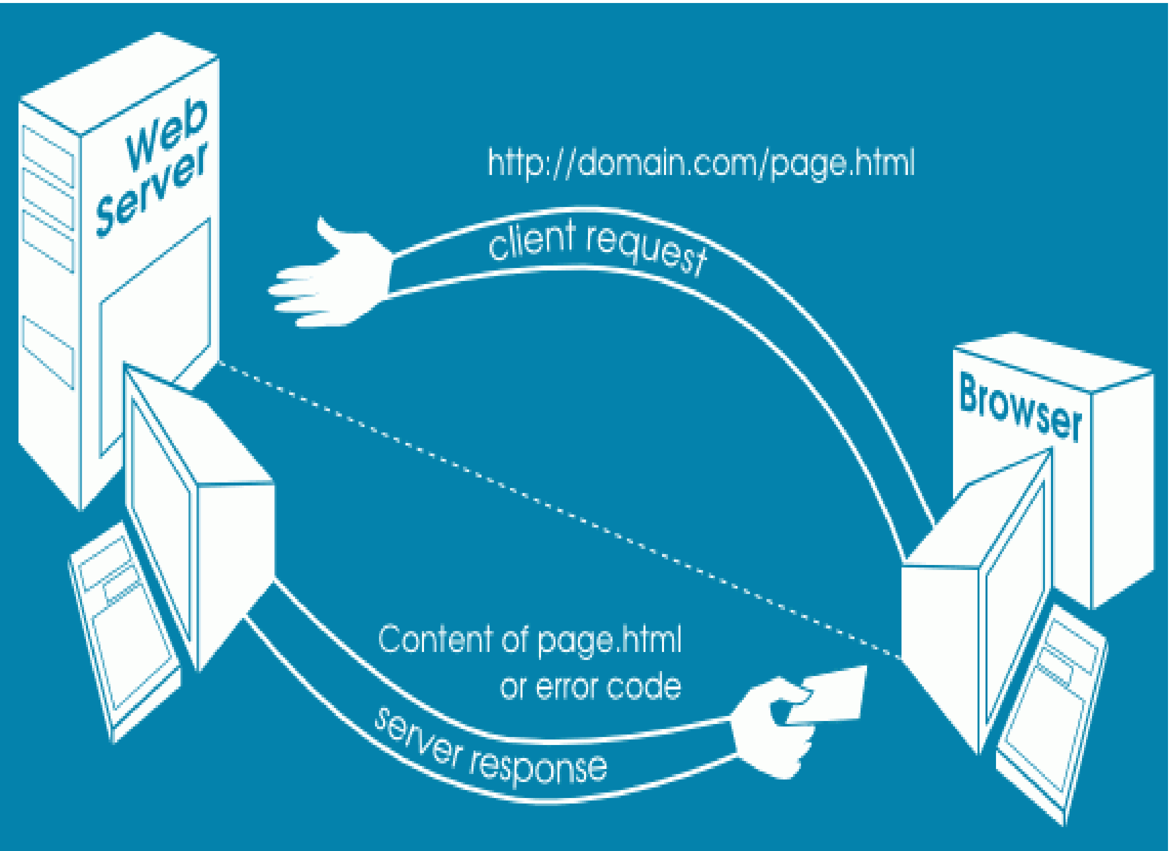
Basic Definitions

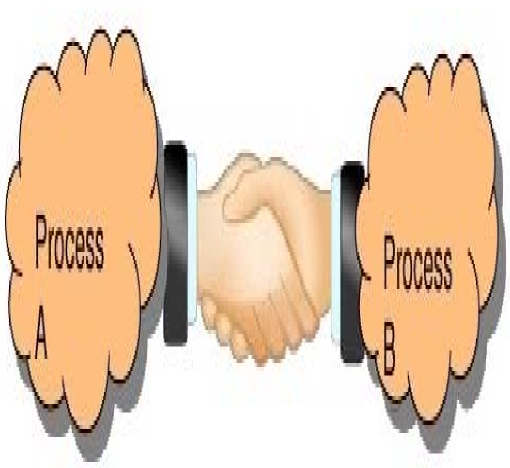
- **Race Condition**: **Situations** like this **where processes access the same data concurrently** and the **outcome of execution depends on the particular order** in which the access takes place is called a race condition. **OR**
- **Situation where two or more processes are reading or writing some shared data and the final result depends on who runs precisely when**.
- Reasons for Race Condition
  - Exact instruction execution order cannot be predicted
  - Resource (file, memory, data etc…) sharing

4_1-8

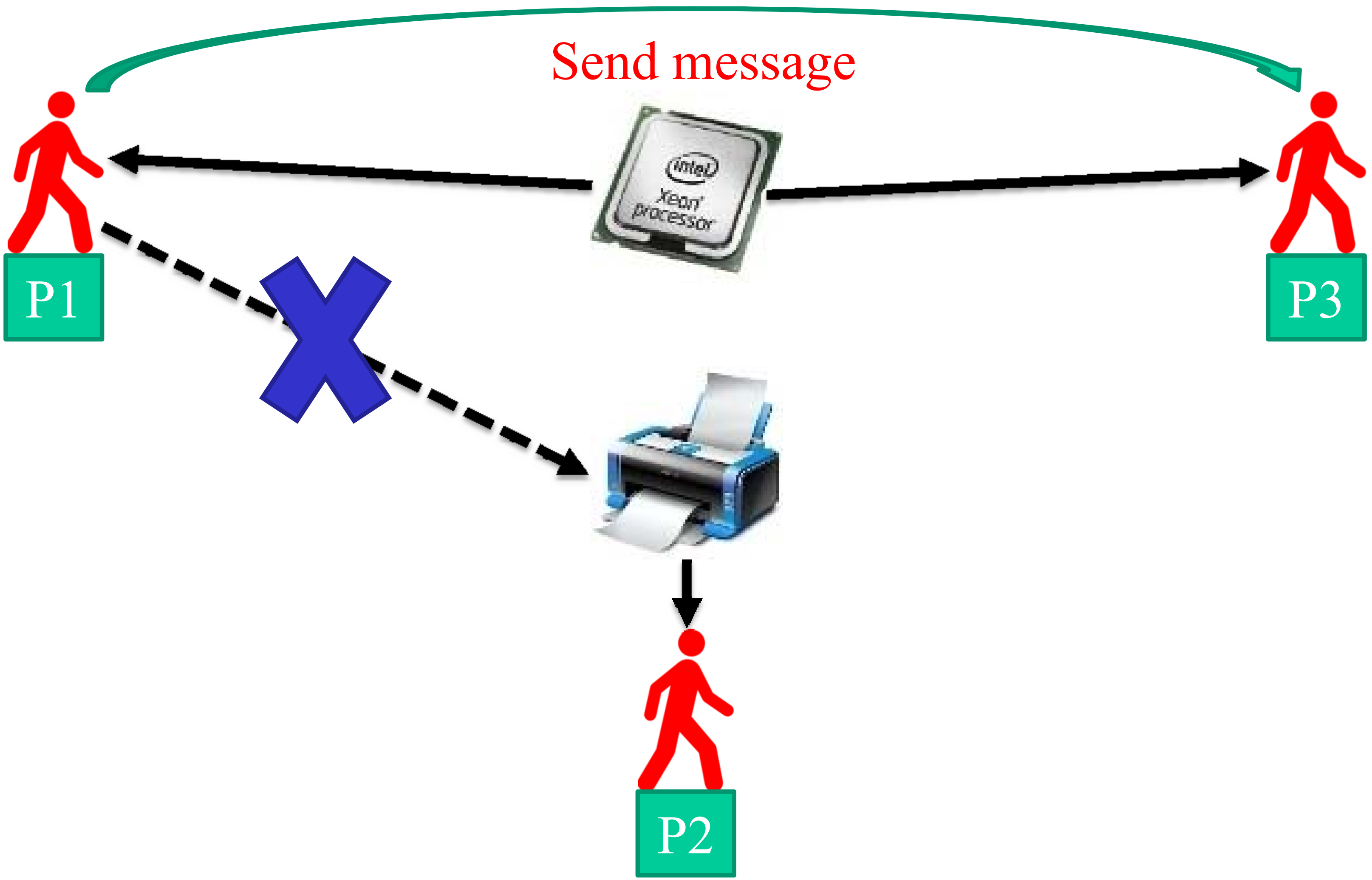
Basic Definitions

- **Inter Process Communication**: It is a **communication between two or more processes**.





Basic Definitions

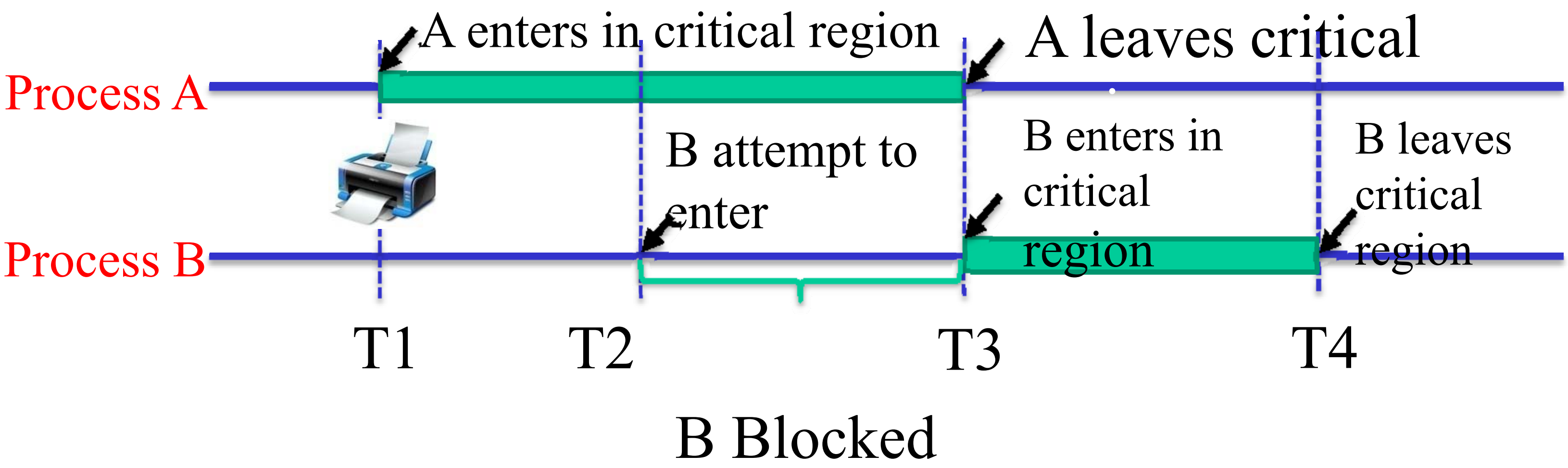


Basic Definitions

- (

Basic Definitions

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



Basic Definitions

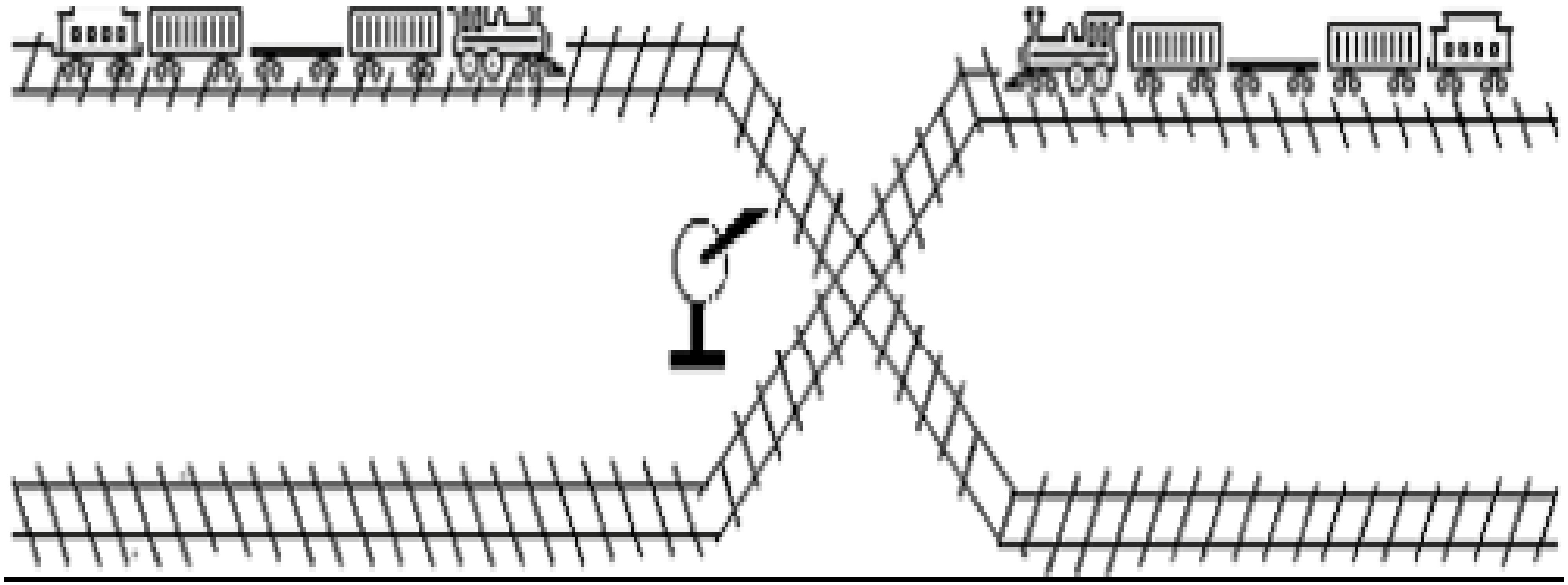
- **Mutual Exclusion**: **Way of making sure** that **if one process is using** a shared variable or file; the **other process will be excluded** (stopped) from doing the same thing.

Solving Critical-Section Problem

Any good solution to the problem must satisfy following four conditions:

1. Mutual Exclusion : **No two processes** may be **simultaneously inside** the same critical section.
2. Bounded Waiting :
   - **No process should have to wait forever** to enter a critical section.
3. Progress : **No process running outside** its critical region may **block other processes**.
4. Arbitrary Speed : **No assumption can be made** about the relative speed of different processes (though all processes have a non-zero speed).

Semaphores



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:

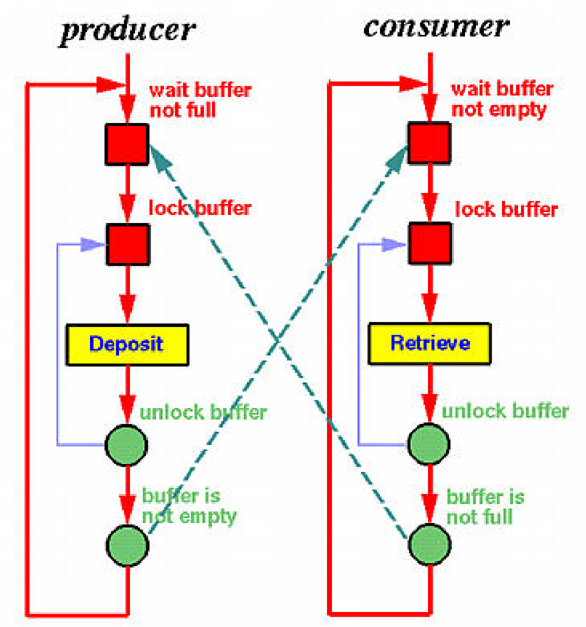
1. **Binary semaphores** :-
    - Binary semaphores can take only **2 values (0/1).**
    - Binary semaphores have 2 methods associated with it (up, down / lock, unlock).
    - They are used to acquire locks.
2. **Counting semaphores** :-
    - Counting semaphore can have possible **values more than two**.

Semaphores

- What is a **semaphore**?
  - A semaphore is an autonomous synchronous abstract data type used for controlling access by multiple processes, to a common resource in a concurrent system.
- How are semaphores **produced**?
  - Semaphores are produced by SEM_SIGNAL.
- How are semaphores **consumed**?
  - Semaphores are consumed by SEM_WAIT and SEM_TRYLOCK.
- What are the major uses of semaphores?
  - Synchronization
  - Resource allocation
  - Mutual Exclusion

Semaphores

- A **semaphore** is a protected variable whose value is accessed and altered by the operations signal (produce) and wait (consume).
  - A semaphore is used for controlling access to a common resource in a concurrent system, such as multi-threading.
  - The value of a semaphore is the number of available resources and may be used to synchronize various task activities.
- A useful way to think of a semaphore is as a record of how many units of a particular resource are available, coupled with operations to safely consume those units, and, if necessary, wait until a unit of the resource becomes available.



Semaphore

- We want functions **insert _item** and **remove_item** such that the following hold:
  1. Mutually exclusive access to buffer: **At any time only one process should be executing** (either insert_item or remove_item).
  2. 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).
  3. 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).

Semaphore

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

   4. No busy waiting.

   5. No producer starvation: A **process does not wait forever** at **insert_item()** provided the buffer repeatedly becomes full.

   6. No consumer starvation: A **process does not wait forever at remove_item()** provided the buffer repeatedly becomes empty.

Semaphores

Two operations on semaphores are defined.

1. **Down** Operation
   – The down operation on a semaphore **checks** to see if the **value is greater than 0**.
   – If so, it **decrements the value** and just continues.
   – If the **value is 0**, the process is **put to sleep** without completing the down for the moment.
   – **Checking the value, changing it, and possibly going to sleep, are all done as a single, indivisible atomic action**.
   – It is **guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked**.

Semaphores

Two operations on semaphores are defined.

2. Up Operation

– The up operation **increments the value** of the semaphore addressed.

– If **one or more processes were sleeping** on that semaphore, unable to complete an earlier down operation, **one of them is chosen by the system (e.g., at random) and is allowed to complete its down**.

– The operation of **incrementing the semaphore and waking up one process is also indivisible**.

– **No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.**

Producer Consumer problem using Semaphore

```
#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);
        }
}
```

mutex	1
empty	3
full	1
item	Item 1

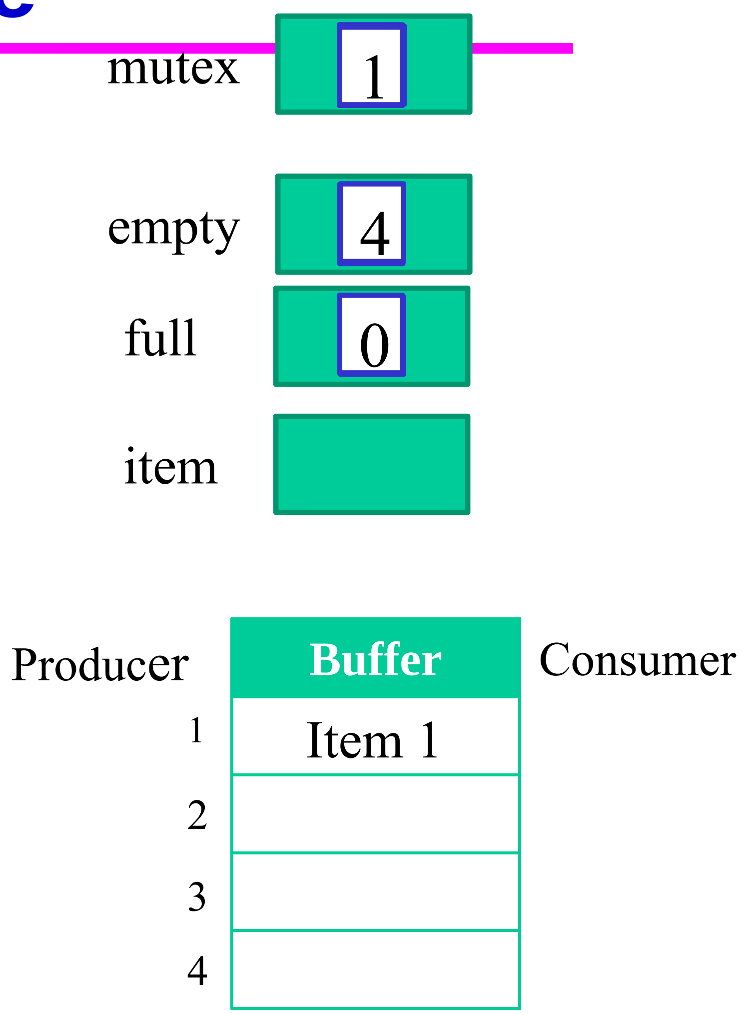
Produce

	Buffer
1	
2	
3	
4	

Consumer

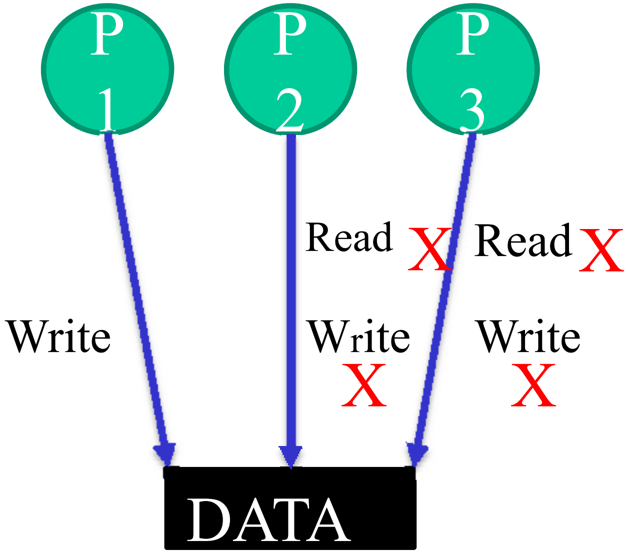
Producer Consumer problem using Semaphore

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



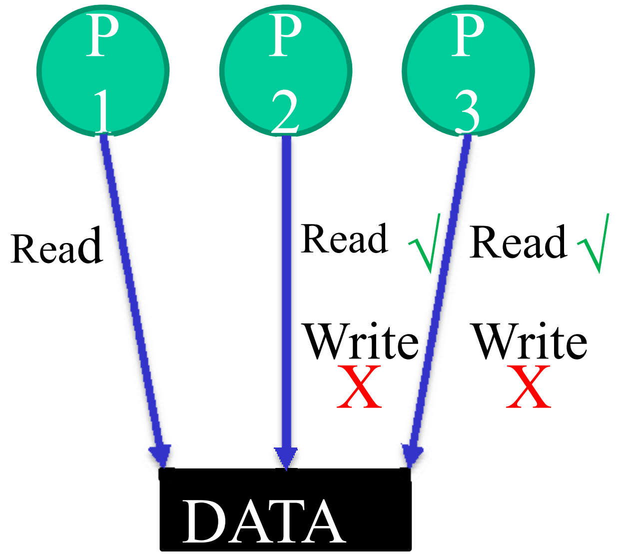
Readers Writer problem

- 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.



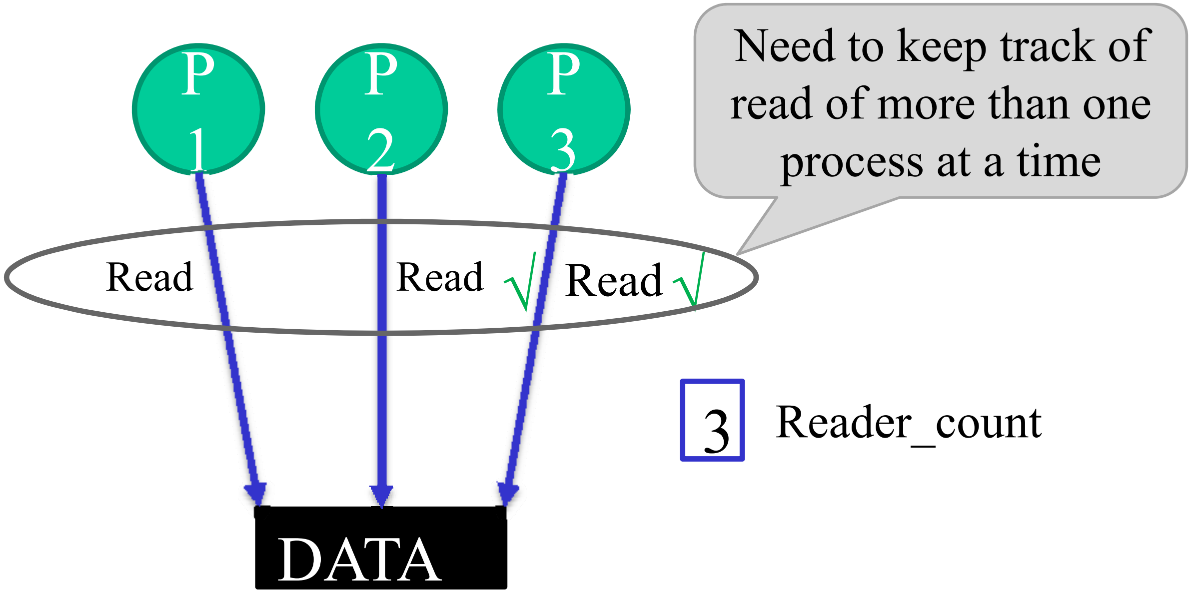
Readers Writer problem

- 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.



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.



Monitor

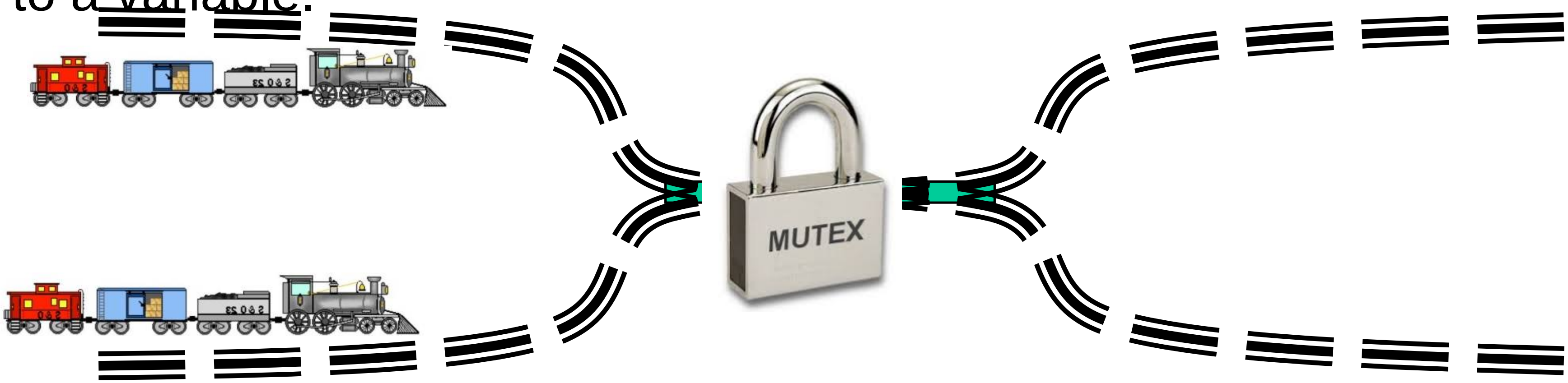
- A **higher-level synchronization primitive**.
- A monitor is a **collection of procedures, variables, and data structures** that are all **grouped together in a special kind of module or package**.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

Monitor

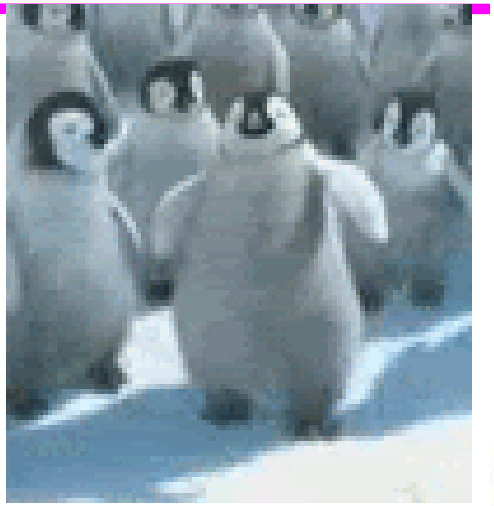
- Monitors have an **important property for achieving mutual exclusion: only one process can be active in a monitor at any instant**.
- When a **process calls a monitor procedure**, the **first few instructions of the procedure will check to see if any other process is currently active within the monitor**.
- 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.

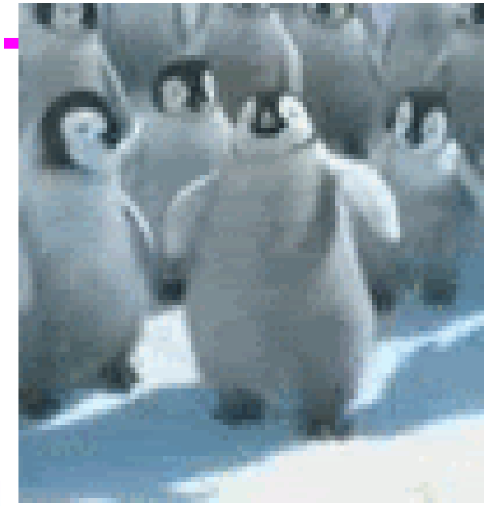
Mutexes

- A **mutex** is locking mechanism used to synchronize access to a resource (such as code). Mutex is the short form for **'Mutual Exclusion Object'**.
  - Only one task can acquire the mutex.
  - It means there is ownership associated with mutex, and only the owner can release the lock (mutex).
- A mutex is designed to protect critical data so that only one thread can access it at the same time, such as a section of code or access to a variable.










Minal Shah