INTERPROCESS COMMUNICATION

INDEX

Course Outcomes (COs)

CO	Explanation
CO1	Explain basic operating system concepts such as overall architecture, system calls, user mode and kernel mode.
CO2	Distinguish concepts related to processes, threads, process scheduling, race conditions and critical sections.
CO3	Analyze and apply CPU scheduling algorithms, deadlock detection and prevention algorithms.
CO4	Examine and categorize various memory management techniques like caching, paging, segmentation, virtual memory, and thrashing.
CO5	Design and implement file management system.
CO6	Appraise high-level operating systems concepts such as file systems, disk-scheduling algorithms and various file systems.

Some Glimpses of Previous Lecture

- Processes and its types- Independent and Cooperating Processes
- Interprocess Communication
- Methods of Interprocess Communication

i. Message Passing Model
ii. Signals
iii. Dynamic Data Exchange
iv. Object Linking and Embedding

Objective and CO

Objective	CO
Understanding the concept of race conditions, critical section, mutual exclusion and various solutions to mutual exclusion.	CO2-Distinguish concepts related to processes, threads, process scheduling, race conditions and critical sections.

Race Condition

- **Race condition-** It is a condition when several processes access and manipulate the same data at the same time.
- Race conditions occur among processes that share common storage.
- Each process can read and write on this shared common storage.
- Race conditions can occur in poorly designed systems.
- Race conditions can be extremely difficult to handle due to improper synchronization of the shared memory access.

Race conditions with threads can arise from a variety of causes, including (but not limited to):

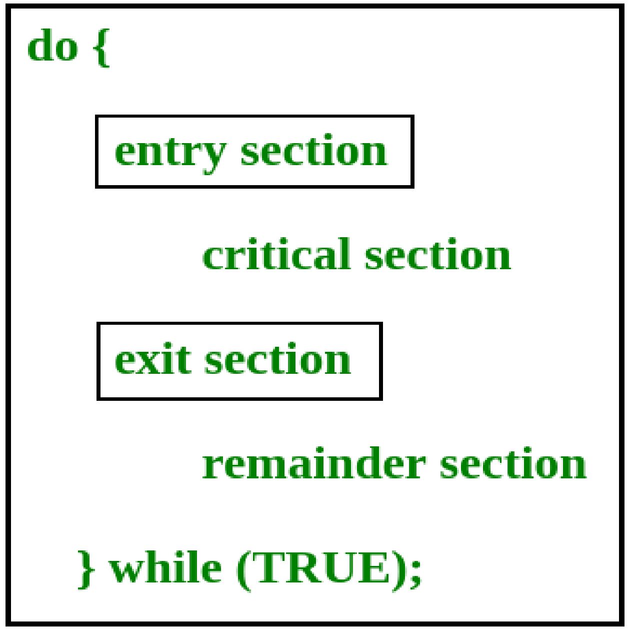
- Two threads try to modify the same global variable at the same time.
- Data exists when a thread is created but becomes invalid when the thread tries to access it later.

Critical Section

- A section of code or set of operations in which a process may be changing the shared variables, updating a common file or a table etc. is known as critical section of that process.
- The critical section problem is used to design a set of protocols which can ensure that the Race condition among the processes will never arise.
- When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.

Critical Section Problems

- Critical section is a code segment that can be accessed by only one process at a time. Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.



```
do {

    entry section

            critical section

    exit section

            remainder section

   } while (TRUE);
```

In the entry section, the process requests for entry in the **Critical Section.**

Any solution to the critical section problem must satisfy three requirements:

- **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.
- **Bounded Waiting** : A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

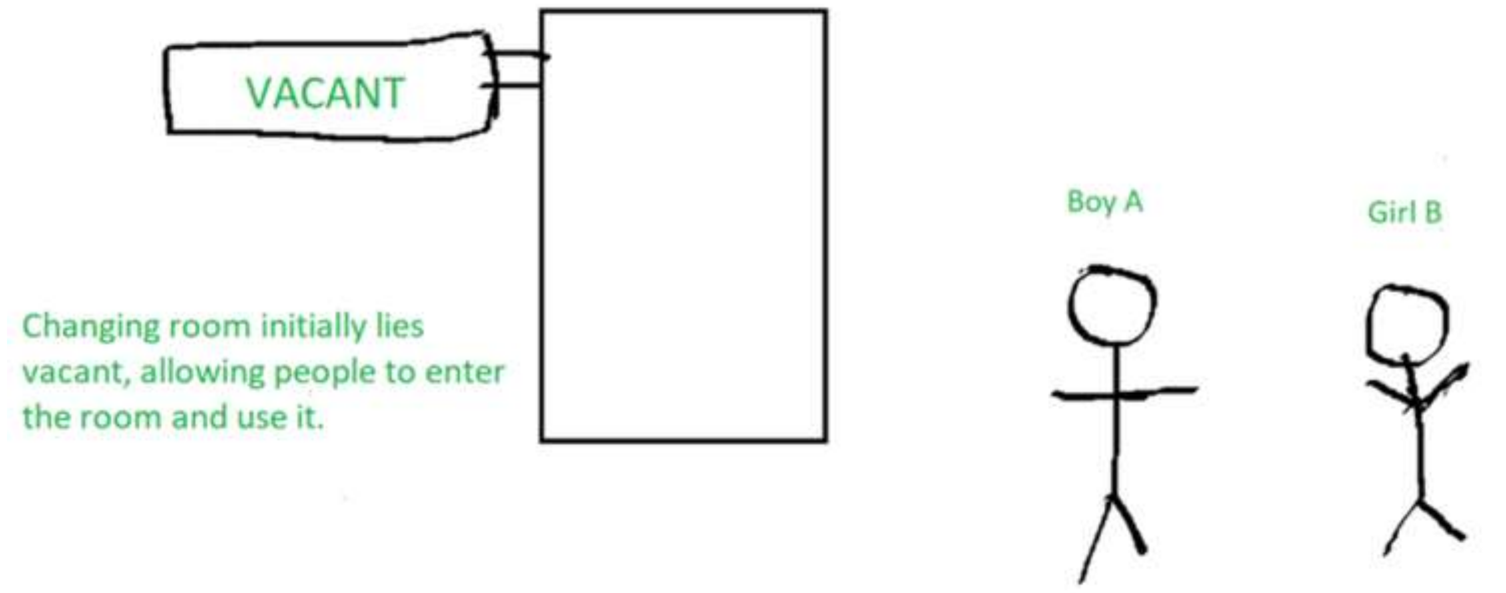
Mutual Exclusion

- **Mutual exclusion** is a property of process synchronization which states that “no two processes can exist in the critical section at any given point of time”.
- Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition.

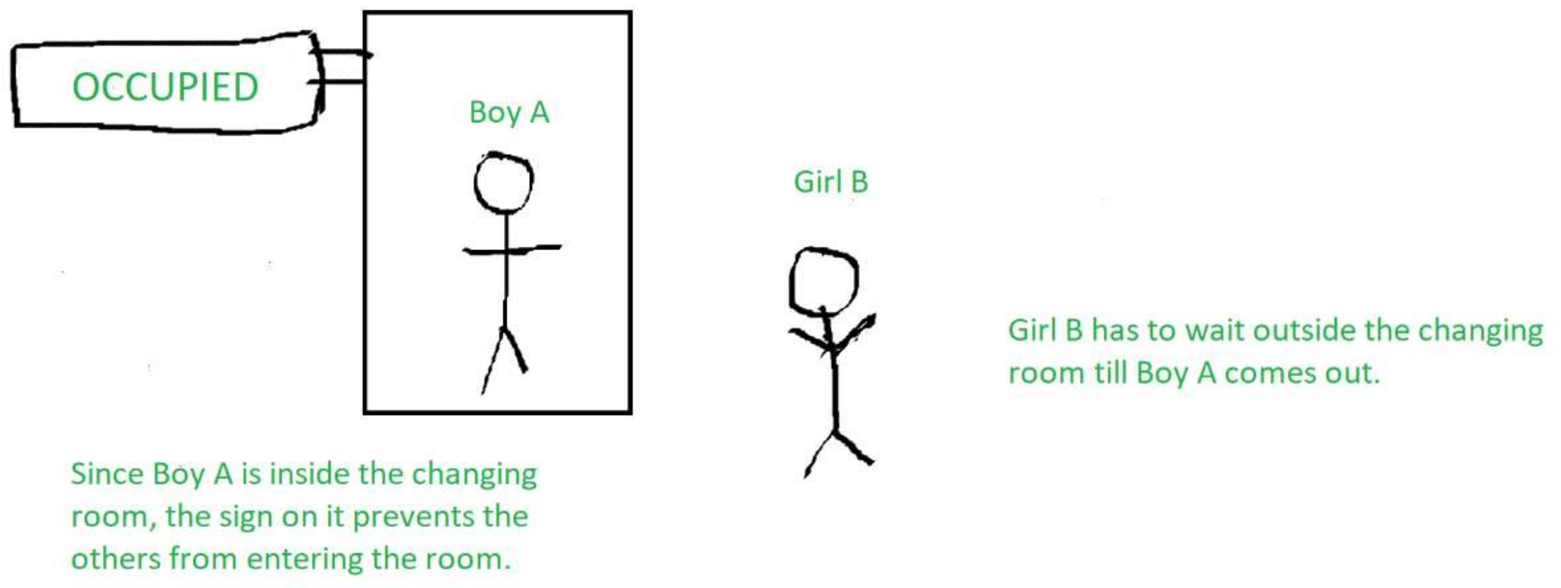
To understand mutual exclusion, let’s take an example.

Example:

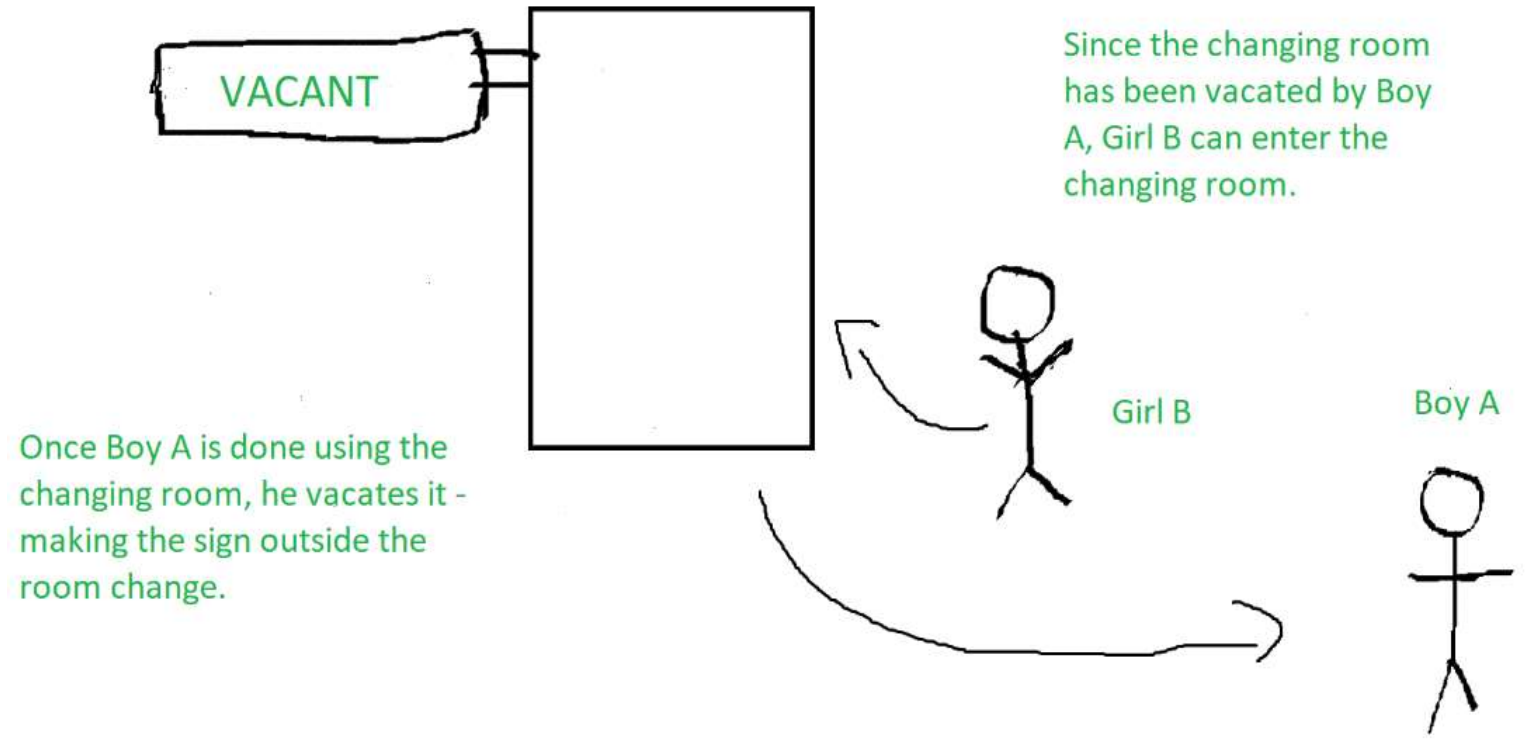
In the clothes section of a supermarket, two people are shopping for clothes


VACANT
Changing room initially lies vacant, allowing people to enter the room and use it.
Boy A
Girl B

Boy A decides upon some clothes to buy and heads to the changing room to try them out. Now, while boy A is inside the changing room, there is an ‘occupied’ sign on it – indicating that no one else can come in. Girl B also want to use the changing room so she must wait till boy A is done using the changing room.


OCCUPIED
Boy A
Girl B
Girl B has to wait outside the changing room till Boy A comes out.
Since Boy A is inside the changing room, the sign on it prevents the others from entering the room.

Once boy A comes out of the changing room, the sign on it changes from ‘occupied’ to ‘vacant’ – indicating that another person can use it. Hence, girl B proceeds to use the changing room, while the sign displays ‘occupied’ again.


VACANT
Since the changing room has been vacated by Boy A, Girl B can enter the changing room.
Girl B
Boy A
Once Boy A is done using the changing room, he vacates it - making the sign outside the room change.

The changing room is nothing but the critical section, boy A and girl B are two different processes, while the sign outside the changing room indicates the process synchronization mechanism being used.

Significance of Critical section, Race Condition and Mutual Exclusion

- Critical sections prevent thread and process migration between processors and the preemption of processes and threads by interrupts and other processes and threads.
- Race conditions are considered a common issue for multithreaded applications. They occur when two computer program processes, or threads, attempt to access the same resource at the same time and cause problems in the system.
- Mutual exclusion reduces latency and busy-waits using queuing and context switches.

Synchronization Hardware or Hardware Solutions

There are various hardware based methods that can be used to solve the critical section problem-

1) Interrupt Disabling
2) Hardware Instructions

Interrupt Disabling

- Once the process has gained the control of CPU, it can only loose that control when it invokes an operating system service or when it is interrupted.
- The best solution for mutual exclusion is to have each process disable all interrupts just entering after entering its critical section.

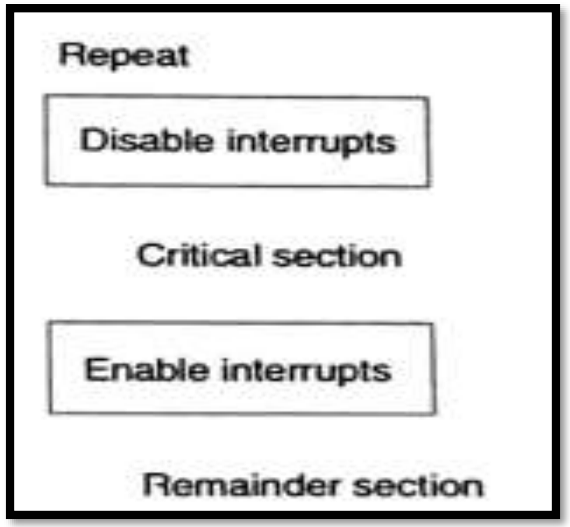


Fig.1: Disabling Interrupts

Disadvantages of Disabling Interrupts

- It works only in single processor environment.
- Performance of the system is degraded, as multiprogramming is not utilized during the execution of critical section.
- A processor waiting to enter its critical section could suffer from starvation.
- This approach will not work in multiprocessor architecture.

Hardware Instructions

- Synchronization hardware i.e., hardware-based solution for the critical section problem introduces the **hardware instructions** that can be used to resolve the critical section problem effectively.
- Many machines provides several instructions that can read, modify and store a memory word atomically.
- The most commonly used instructions are-

1) Test and Set Instruction
2) Compare and Swap
3) Exchange Instruction

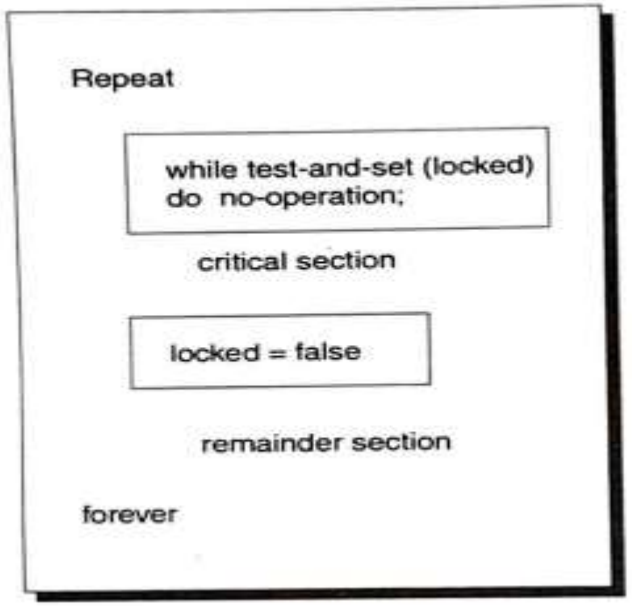
1) Test And Set Hardware Instruction

- This instruction provides the action of testing the variable and consequently setting it to a stipulated value.
- The hardware-based solution to critical section problem is based on a simple tool i.e., **lock.**
- The solution implies that before entering the critical section the process must acquire a lock and must release the lock when it exits its critical section. Using of lock also prevent the ***race condition*.**
- The Test And Set hardware instruction is **atomic** instruction. Atomic means both the test operation and set operation are executed in one machine cycle at once.

Test And Set Hardware Instruction

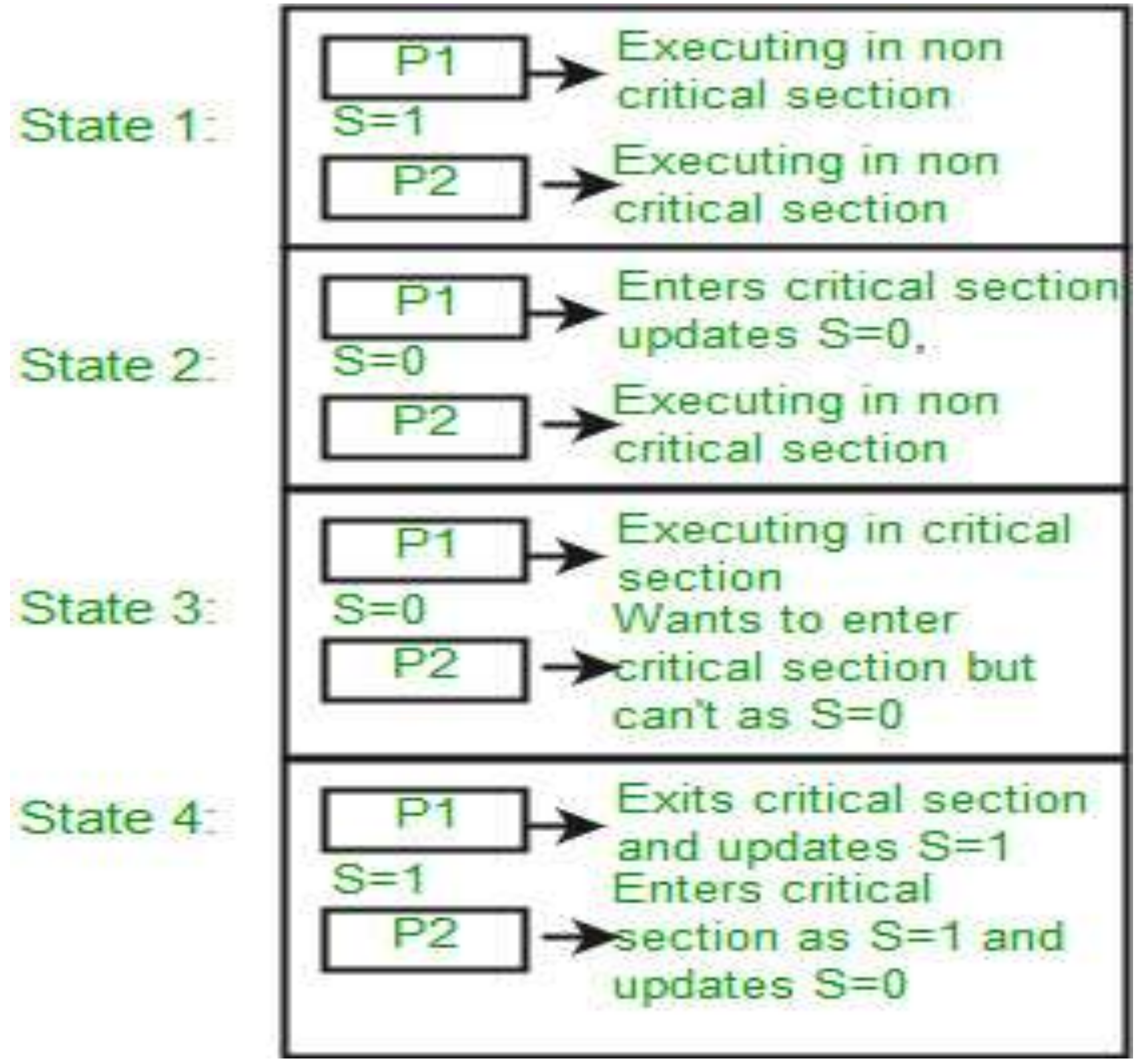
- The Test And Set instruction can be defined as in the code below:

```
do {
while (TestAndSet(&lock));
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```



```
Repeat
    while test-and-set (locked)
    do  no-operation;
        critical section
    locked = false
        remainder section
forever
```

How Mutual exclusion is achieved?

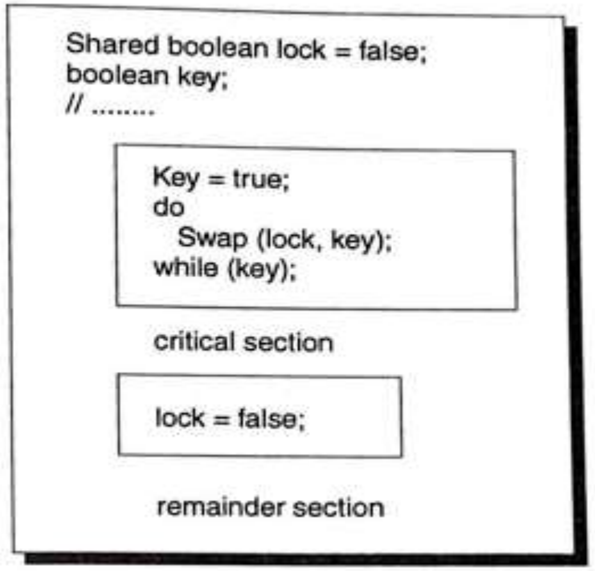

State 1:
P1
Executing in non critical section
S=1
P2
Executing in non critical section
State 2:
P1
Enters critical section updates S=0.
S=0
P2
Executing in non critical section
State 3:
P1
Executing in critical section
S=0
P2
Wants to enter critical section but can't as S=0
State 4:
P1
Exits critical section and updates S=1
S=1
P2
Enters critical section as S=1 and updates S=0

- Mutual exclusion is met because if the first process is in critical section, it sets the value of lock to true.
- Thus, the second process will have a true value for test and set instruction and therefore, it will not come out of the while loop till the lock is set to false by the first process executing its critical section.

2) Compare and Swap Instruction

- Like Test And Set instruction the swap hardware instruction is also an atomic instruction with a difference that it operates on two variables provided in its parameter.
- The structure of swap instruction is:

```
do {
key = TRUE;
while (key == TRUE)
Swap(&lock, &key);
// critical section
lock = FALSE;
// remainder section
} while (TRUE);
```



```
Shared boolean lock = false;
boolean key;
// ........

    Key = true;
    do
      Swap (lock, key);
    while (key);

    critical section

    lock = false;

    remainder section
```

3) Exchange

- This instruction exchanges the contents of a register with that of a memory location.
- In this procedure, a shared variable is used that is initialized to 0.
- The process that may enters its critical section is one that finds this shared variable equal to 0.
- It then excludes all other processes from the critical section by setting this variable to 1.
- When the process leaves its critical section it resets its shared variable to 0, allowing other process to gain access to its critical section.

Advantages of Hardware Instruction Approach

- Hardware instructions are easy to implement and improves the efficiency of the system.
- Supports any number of processes may it be on the single or multiple processor system.
- With hardware instructions, you can implement multiple critical sections each defined with a unique variable.

Disadvantages of Hardware Instruction Approach

- Processes waiting for entering their critical section consumes a lot of processors time which increases busy waiting.
- As the selection of processes to enter their critical section is arbitrary, it may happen that some processes are waiting for the indefinite time which leads to process starvation.
- Deadlock is also possible.

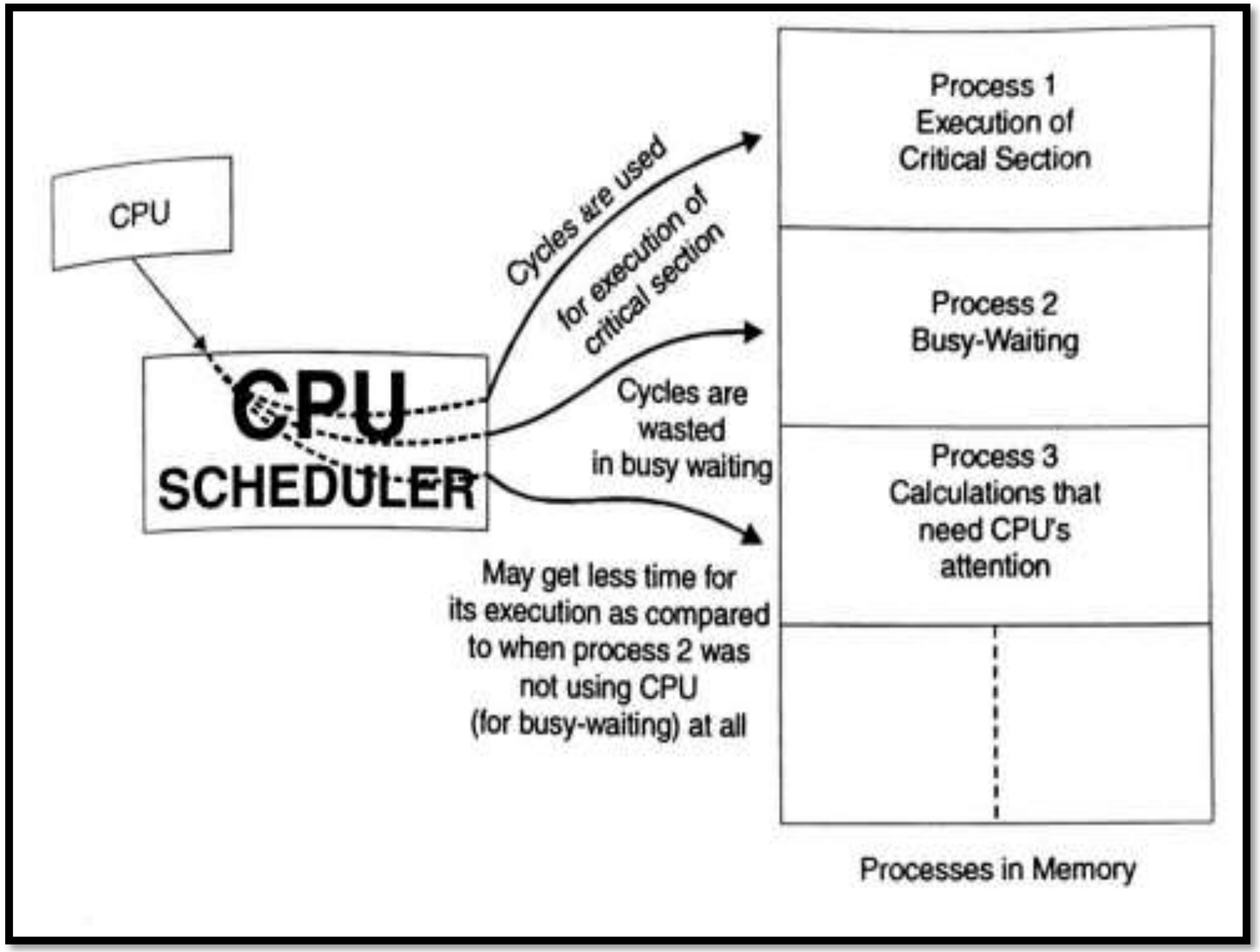

CPU
CPU SCHEDULER
Cycles are used for execution of critical section
Cycles are wasted in busy waiting
May get less time for its execution as compared to when process 2 was not using CPU (for busy-waiting) at all
Process 1 Execution of Critical Section
Process 2 Busy-Waiting
Process 3 Calculations that need CPU's attention
Processes in Memory

Fig.2: Busy waiting leads to wastage of CPU cycles

References

- Operating System Concepts by Charanjeet Singh
- https://www.geeksforgeeks.org/introduction-of-process-synchronization/
- https://archive.nptel.ac.in/courses/106/106/106106144/
- https://archive.nptel.ac.in/courses/106/106/106106144/
- https://youtu.be/qMQsd7Iy5jo

Interprocess Communication

INDEX

Objective and COs

Objective	CO	Explanation
Understanding the types of various processes and how they communicate with each other while executing in the system	CO2	Concepts related to processes, threads, process scheduling, race conditions and critical sections.

Types of Processes

The processes are classified into two categories-

- Independent Process- Its an process that does not share its data with another process.
- Cooperating process- It is a process that shares data with the other processes in the system.

Cooperating Processes- Why?

Reasons for using cooperating processes are-

- Information Sharing
- Computational Speed up

Interprocess Communication

- It is a facility provided by an operating system via which cooperating processes can communicate with each other.
- It is useful in distributed environment.
- Example- Chat program used in WWW.

Methods for Interprocess Communication

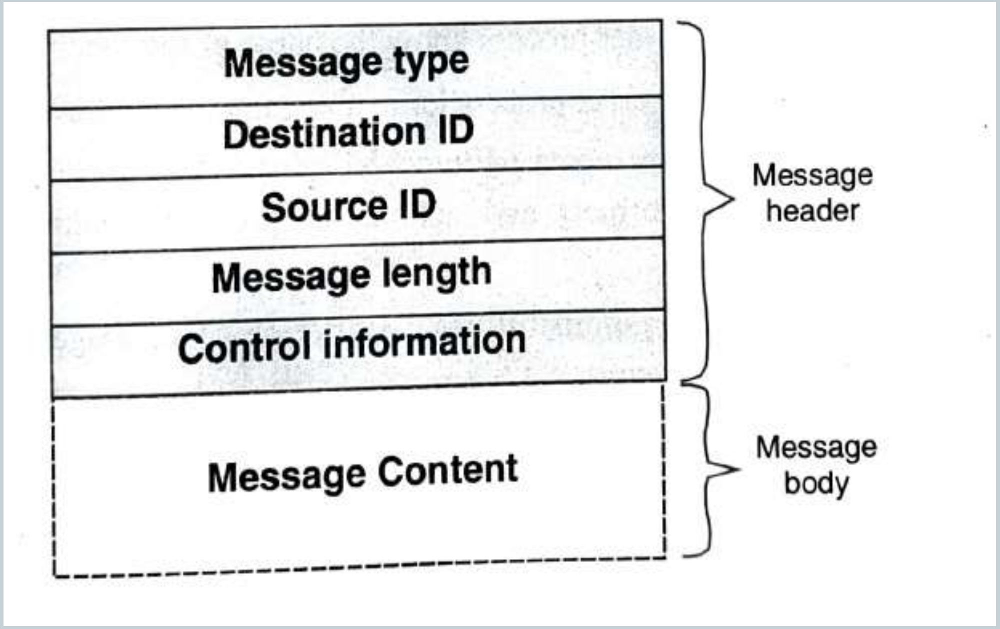
Following are the methods for establishing the interprocess communication-

- Message Passing
- Signals
- Dynamic Data Exchange (DDE)
- Object Linking and Embedding (OLE)

Methods for Interprocess Communication

1) Message Passing Model- It is a collection of information that may be exchanged between a sending and receiving process.

- Figure-1 is showing the format of the message.



Methods for Interprocess Communication

- Processes generally sends and receives the messages by using send and receive primitives.

  Send(receiver process, message);

  Receive(sender process, message);

- Two system calls are used for message transfer among processes-

  msgsnd()- It sends a message using message queue.

  msgrcv()- It receives a message using message queue.

Methods for Interprocess Communication

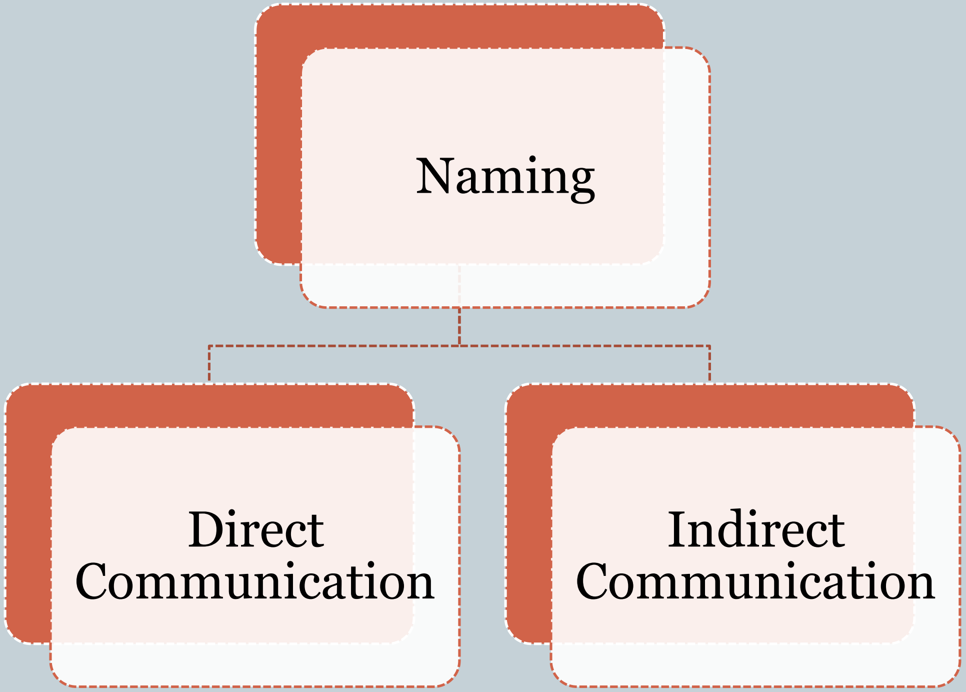
1.1 Implementation Issues in Messages- There can be two major implementation issues that are-

a) Naming of sender and receiver processes

b) Message delivery protocols

Methods for Interprocess Communication

a) **Naming-** Processes that want to communicate with each other must have a way to refer to each other. And these ways can be direct or indirect.



Methods for Interprocess Communication

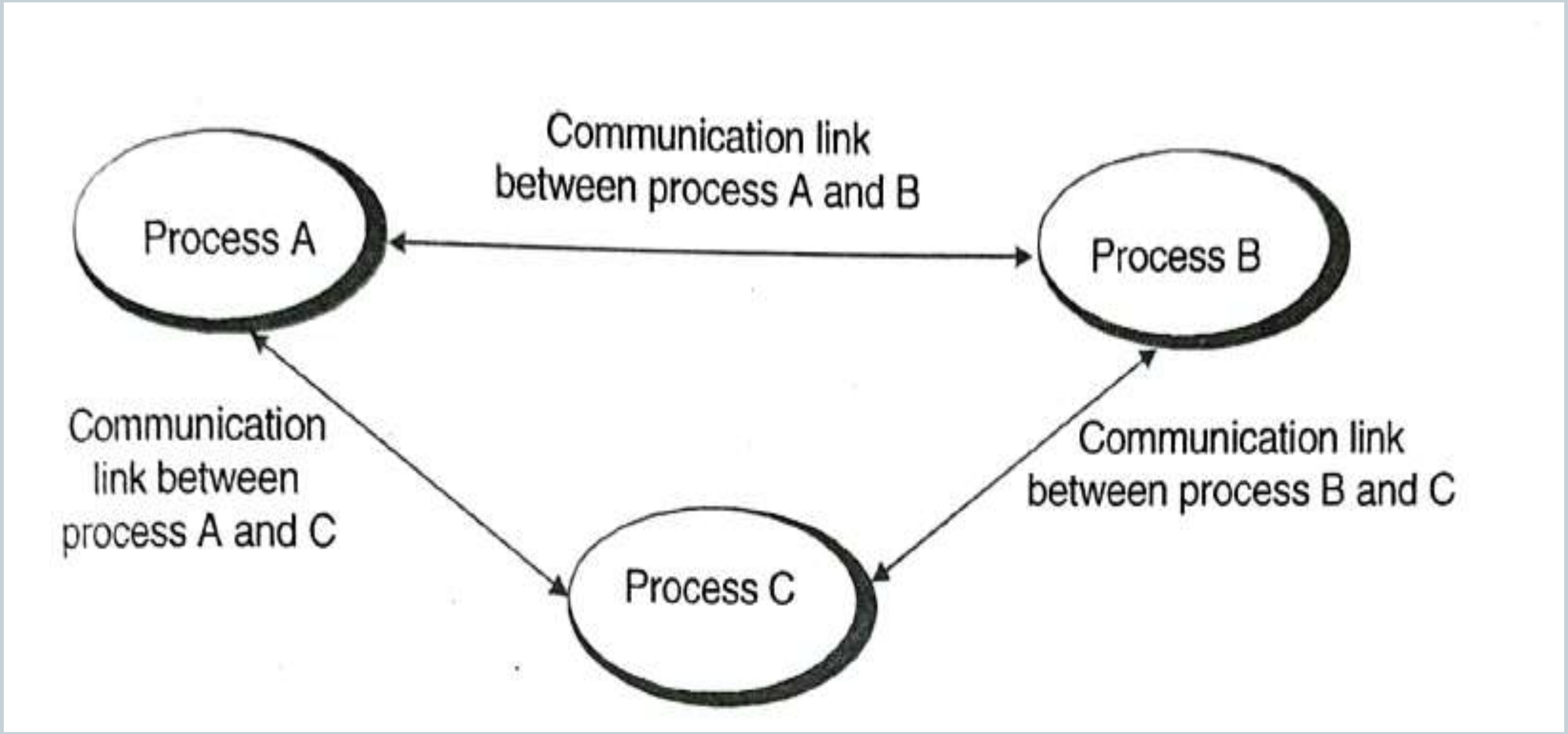

Communication link between process A and B
Process A
Process B
Communication link between process A and C
Communication link between process B and C
Process C

Fig.2: Direct Communication

Methods for Interprocess Communication

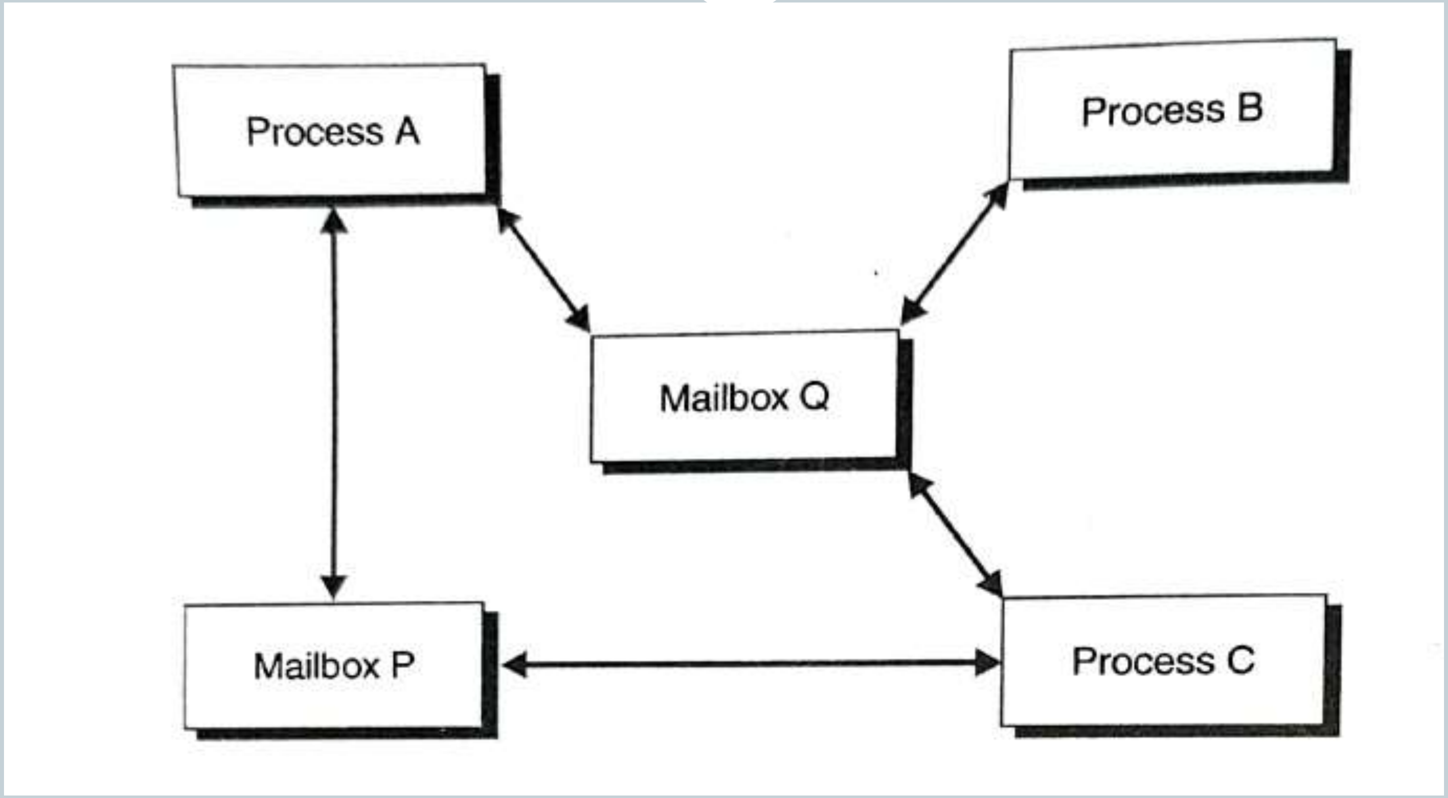
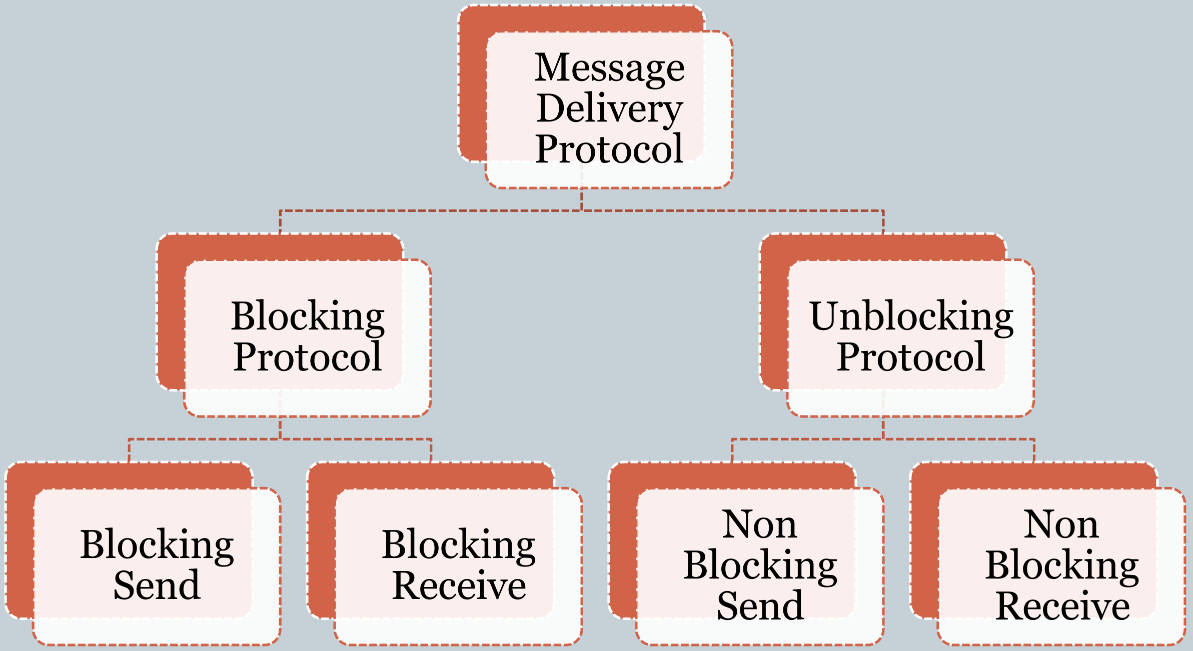


Fig.3: Indirect Communication

Methods for Interprocess Communication

b) Message Delivery Protocols-Protocols are the set of rules that determine the message data formats and actions of processes while sending and receiving.



Methods for Interprocess Communication

- **Blocking Protocol-** The sender process is blocked till the message is delivered to the receiver.

  Blocking Send -The sender process is blocked until the message is received by the receiving process or by the mailbox.

  Blocking Receiver- The receiver blocks until a message is available.

- **Non Blocking Protocol-** In it, a sender continues the execution after performing a send operation irrespective of whether the message is delivered or not.

  Non Blocking Send- The sending process sends the message and resumes the operation.

  Non Blocking Receive- The receiver receives either a valid message or a null.

Methods for Interprocess Communication

2. Signals- It is a primitive form of communication that is used to alert a process to occurrence of some event.

- For example, In UNIX, two user defined signals that are used by user processes are SIGUSR1 and SIGUSR2.

Methods for Interprocess Communication

3. Dynamic Data Exchange- It is a mean of transferring data between two window applications.

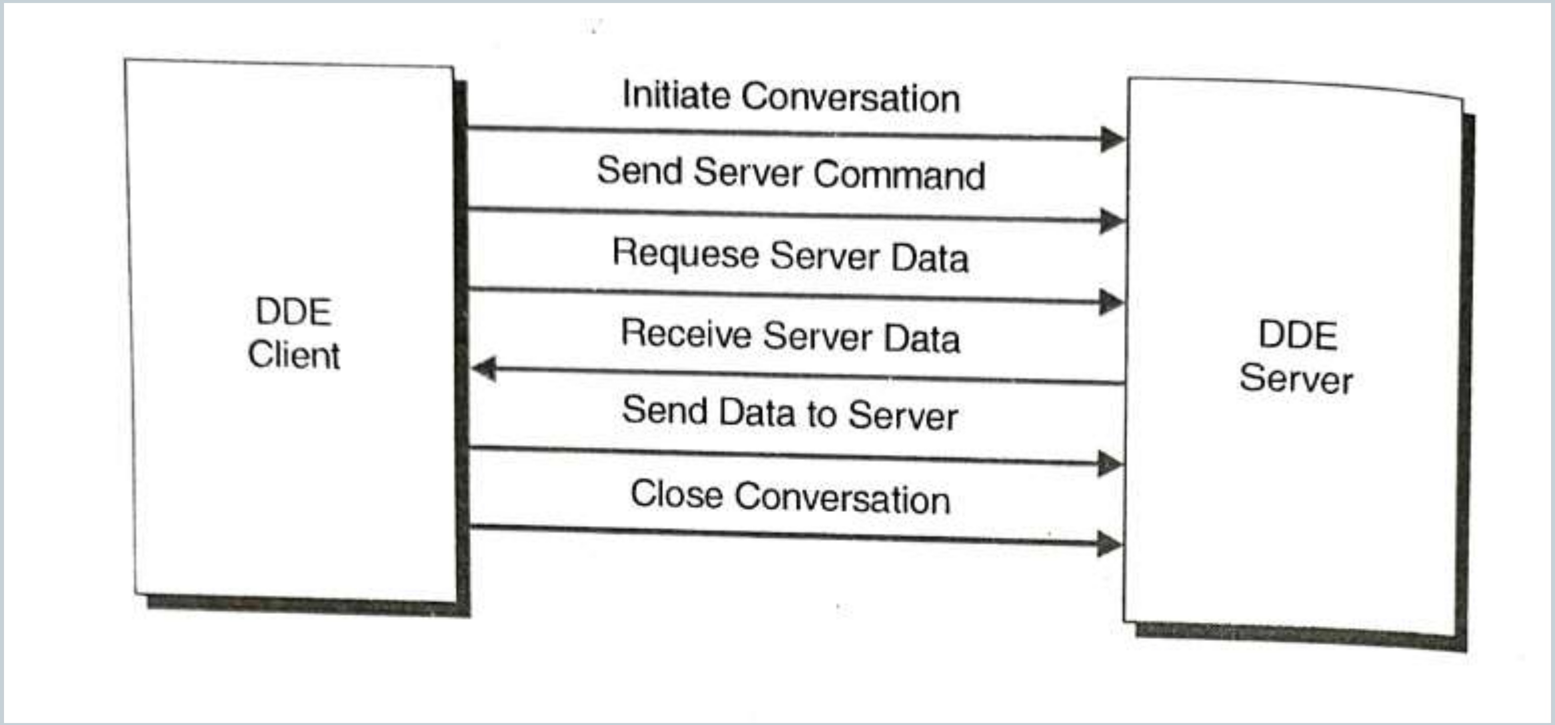


Fig.4: DDE Conversation

Methods for Interprocess Communication

4. Object Linking and Embedding(OLE)- Its purpose is to enable integration of application 'objects' from different but compliant software packages.

- With linking, the source document contains only a reference to the object.
- With embedding, the object is actually stored as a part of the data of the source document.

References

- Operating System Concepts by Charanjit Singh
- https://www.youtube.com/watch?v=Sa2-yABWEC4
- Nptel Video link- https://www.youtube.com/watch?v=lcRqHwIn5Dk

THANK YOU

Readers-Writers Problem

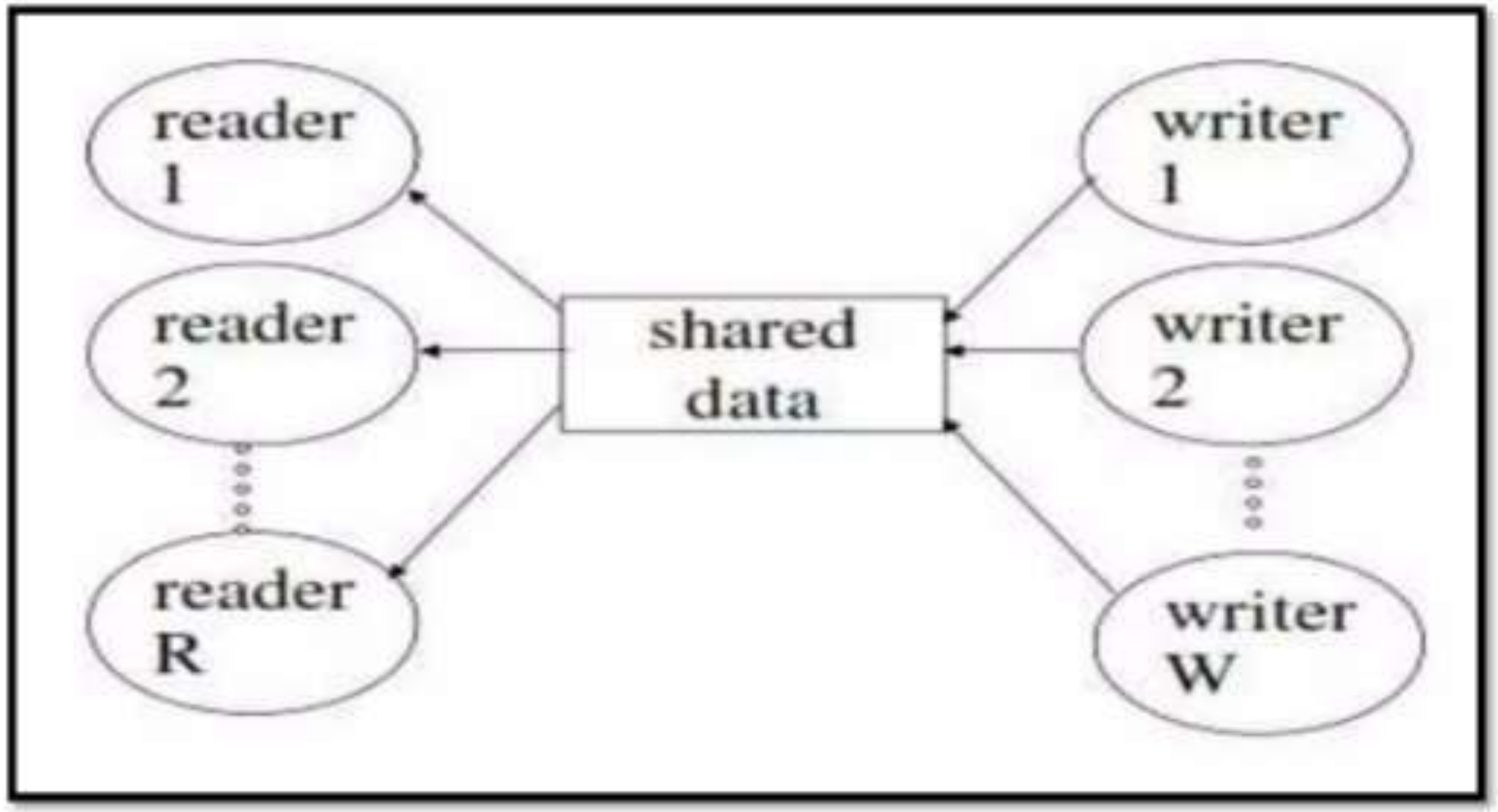
Introduction to Readers Writers Problem:

- The readers-writers problem is a classic synchronization problem with many practical applications. It is a classic problem of synchronization which is an endemic issue with multiple processes that share a common resource. It occurs when more than one process tries to access the same resource at the same time

- 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.

- For example - If two readers access the object at the same time there is no problem. However if two writers or a reader and writer access the object at the same time, there may be problems.

SHARING OF DATA

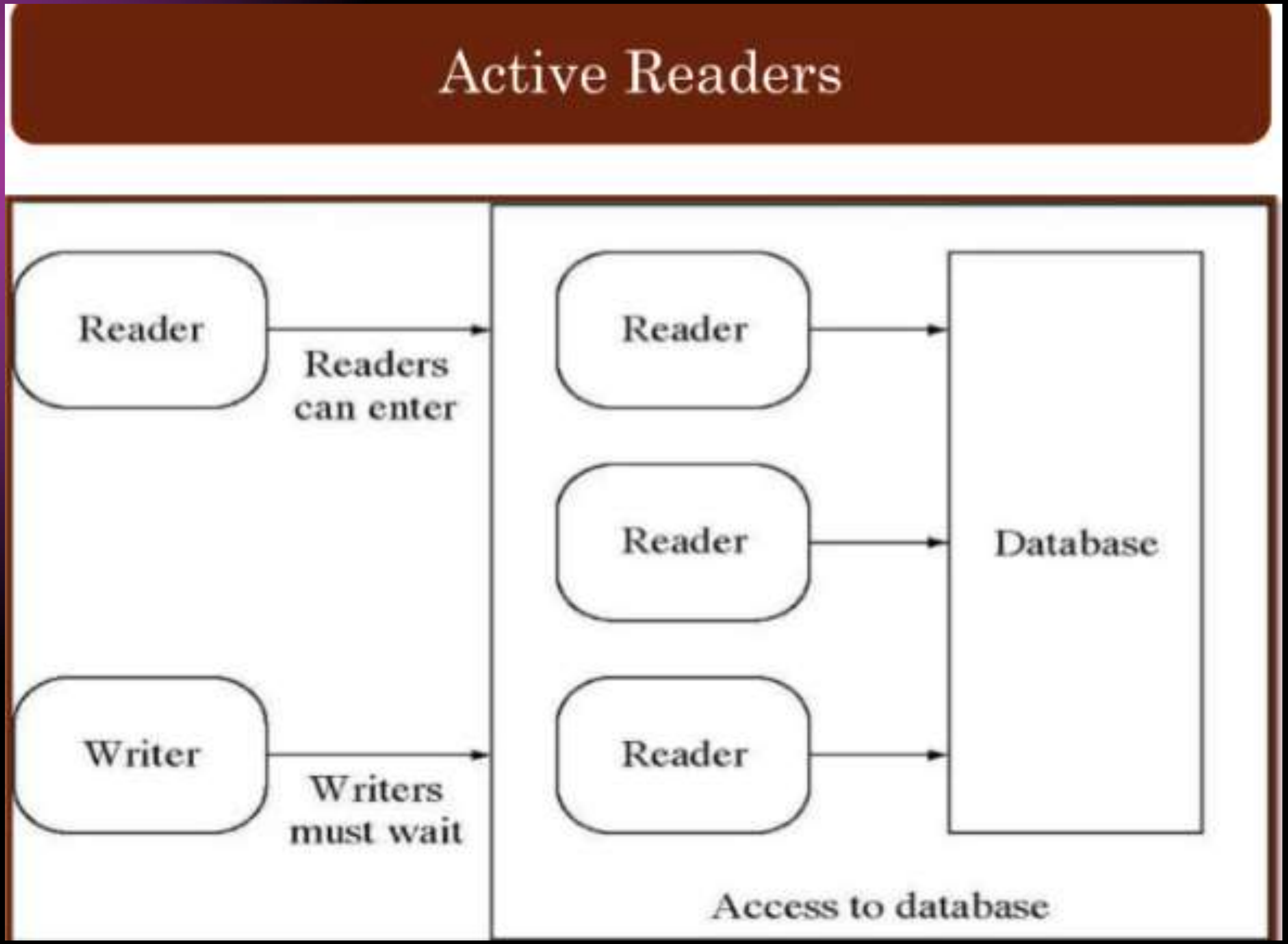


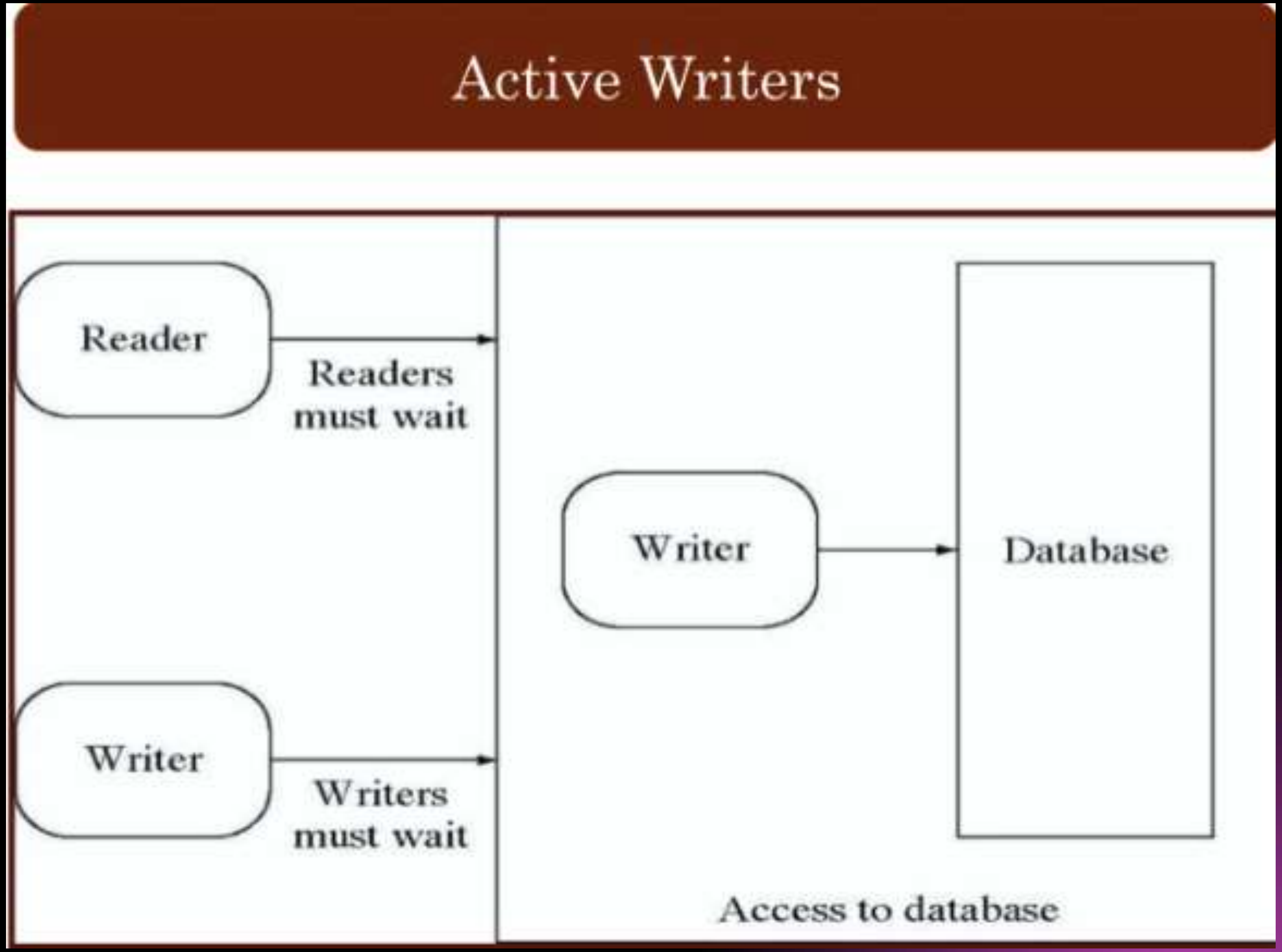
- **Readers:** only read the data set they do not perform any updates
- **Writers:** can both read and write

Constraints of readers writers problem:

- No reader can be delayed until a writer finishes its use of the resource.
- No writer should be blocked unless there are other active readers or writers.
- Once a reader entered, no other writers must remain waiting to access the shared resource and vice versa.

Flow Diagram of Active Readers & Writers:





Reader Process:

```
wait (mutex);
rc ++;
if (rc == 1)
wait (wrt);
signal(mutex);
.. READ THE OBJECT..
wait(mutex);
rc --;
if (rc == 0)
signal (wrt);
signal(mutex);
```

In this code, mutex and wrt are semaphores that are initialized to 1. Also, rc is a variable that is initialized to 0. The mutex semaphore ensures mutual exclusion and wrt handles the writing mechanism and is common to the reader and writer process code.

The variable rc denotes the number of readers accessing the object. As soon as rc becomes 1, wait operation is used on wrt. This means that a writer cannot access the object anymore. After the read operation is done, rc is decremented. When re becomes 0, signal operation is used on wrt. So a writer can access the object now.

Writer Process:

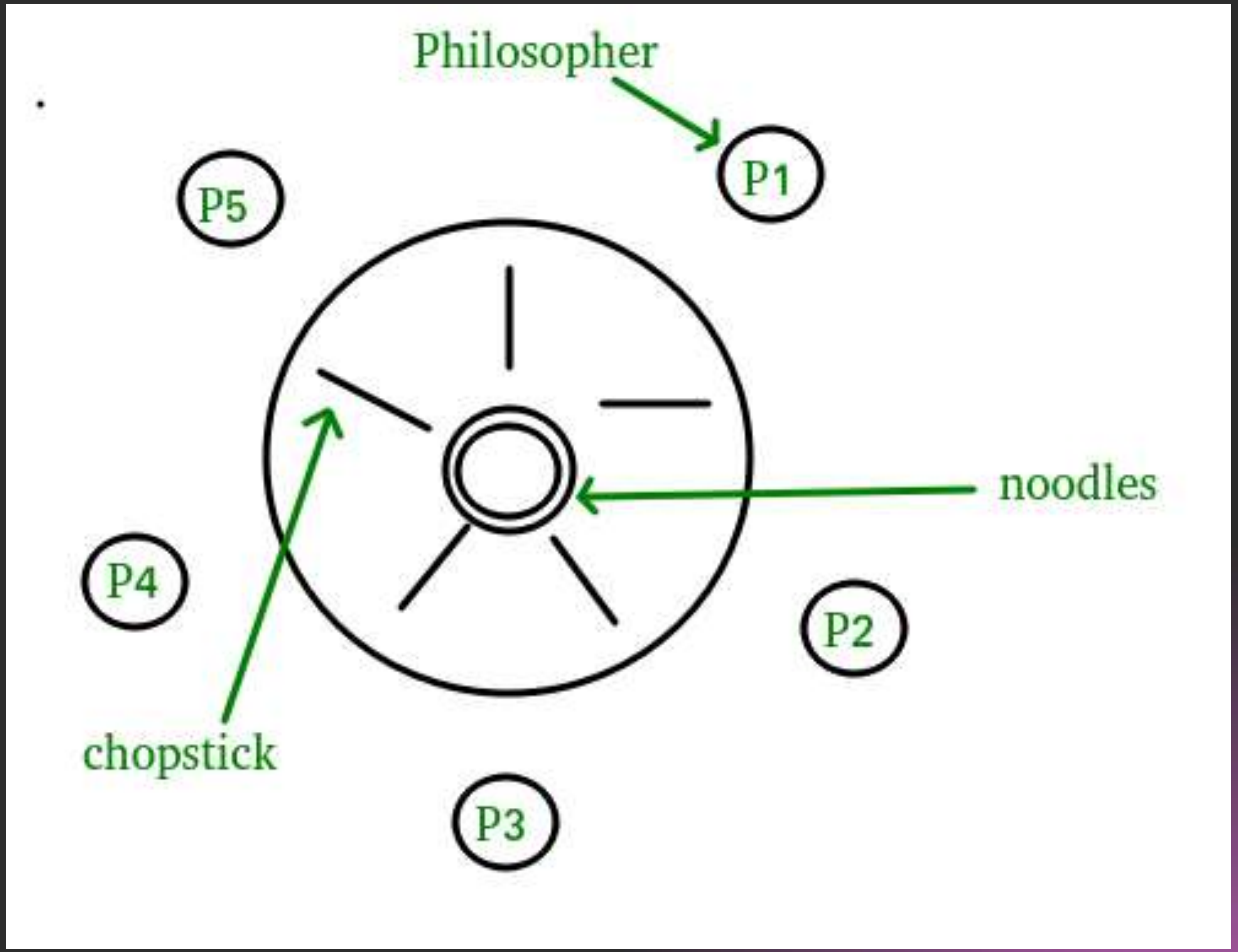
```
wait(wrt);
.. WRITE INTO THE OBJECT..
signal(wrt);
```

If a writer wants to access the object, wait operation is performed on wrt.
After that no other writer can access the object.
When a writer is done writing into the object, signal operation is performed on wrt.

Dining Philosophers Problem (DPP)

Introduction to Dining Philosophers Problem:

- The dining philosophers problem states that there are 5 philosophers sharing a circular table and they eat and think alternatively. There is a bowl of noodles for each of the philosophers and 5 chopsticks. A philosopher needs both their right and left chopstick to eat. A hungry philosopher may only eat if there are both chopsticks available.Otherwise a philosopher puts down their chopstick and begin thinking again.



Solution of Dining Philosophers Problem:

- A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick. A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.
- The solution needs to prevent deadlocks, ensure that all the philosophers get a chance to eat, and avoid starvation, where a philosopher is unable to eat indefinitely because other philosophers keep using the chopsticks they need

Structure of a random philosopher i:

```
do {
   wait( chopstick[i] );
   wait( chopstick[ (i+1) % 5] );
 .. EATING THE NOODLES..
   signal( chopstick[i] );
   signal( chopstick[ (i+1) % 5] );
.. THINKING..
   } while(1);
```

Structure of the chopstick is shown below – semaphore chopstick [5];
In the above structure, first wait operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has picked up the chopsticks on his sides. Then the eating function is performed.

After that, signal operation is performed on chopstick[i] and chopstick[(i+1) % 5]. This means that the philosopher i has eaten and put down the chopsticks on his sides. Then the philosopher goes back to thinking.

Difficulty with the solution:

- Solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock. This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

- Some of the ways to avoid deadlock are as follows –
  1. There should be at most four philosophers on the table.
  2. An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
  3. A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Semaphores, Producer consumer problem

Significance of Hardware Solution and Semaphores

- Semaphores are implemented in the machine independent code of the microkernel.
- Semaphore is a protected variable (or abstract data type) and constitutes the classic method for restricting access to shared resources
- Peterson Solution provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting. Mutual exclusion is preserved.

Semaphores

- A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that called the wait function.
- A semaphore uses two atomic operations, wait and signal for process synchronization.
- A Semaphore is an integer variable, which can be accessed only through two operations *wait()* and *signal()*.

There are two types of semaphores:

1. **Binary Semaphores**
2. **Counting Semaphores**

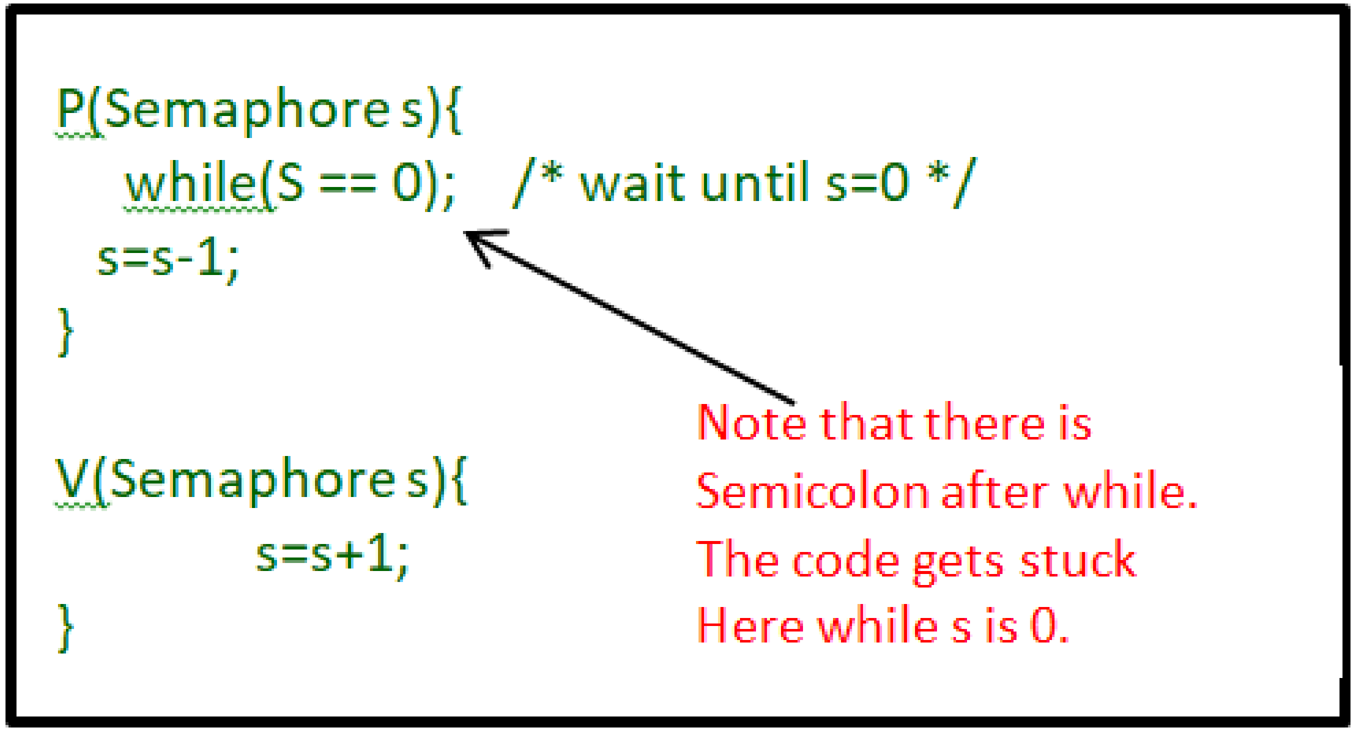
Binary Semaphores:

- They can only be either 0 or 1.
- They are also known as mutex locks, as the locks can provide mutual exclusion. All the processes can share the same mutex semaphore that is initialized to 1.
- Then, a process must wait until the lock becomes 0. Then, the process can make the mutex semaphore 1 and start its critical section.
- When it completes its critical section, it can reset the value of mutex semaphore to 0 and some other process can enter its critical section.

Counting Semaphores:

- They can have any value and are not restricted over a certain domain. They can be used to control access to a resource that has a limitation on the number of simultaneous accesses.
- The semaphore can be initialized to the number of instances of the resource.
- Whenever a process wants to use that resource, it checks if the number of remaining instances is more than zero, i.e., the process has an instance available.
- Then, the process can enter its critical section thereby decreasing the value of the counting semaphore by 1. After the process is over with the use of the instance of the resource, it can leave the critical section thereby adding 1 to the number of available instances of the resource.

Look at two operations that can be used to access and change the value of the semaphore variable:



```
P(Semaphore s){
    while(S == 0);   /* wait until s=0 */
   s=s-1;
}

V(Semaphore s){
            s=s+1;
}
```

Note that there is
Semicolon after while.
The code gets stuck
Here while s is 0.

Some point regarding P and V operation :

1. P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation.
2. Both operations are atomic, and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e., in-between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization. See the below image. The critical section of Process P is in between P and V operation.

Limitations :

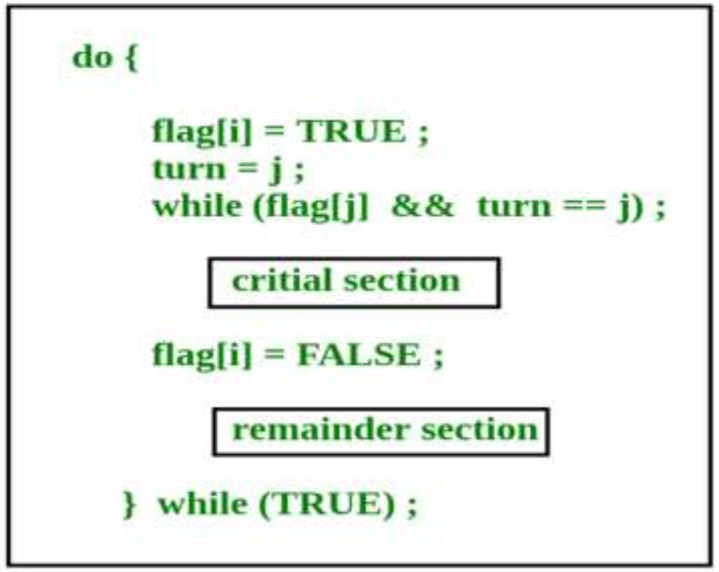
1. One of the biggest limitations of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and to signal the semaphore.

Peterson solution

Peterson's Solution is a classical software-based solution to the critical section problem.

In Peterson's solution, we have two shared variables:

- Boolean flag[i] :Initialized to FALSE, initially no one is interested in entering the critical section
- int turn : The process whose turn is to enter the critical section.



```
do {

    flag[i] = TRUE ;
    turn = j ;
    while (flag[j]  &&  turn == j) ;

        critial section

    flag[i] = FALSE ;

        remainder section

 }  while (TRUE) ;
```

Peterson’s Solution preserves all three conditions :

- Mutual Exclusion is assured as only one process can access the critical section at any time.
- Progress is also assured, as a process outside the critical section does not block other processes from entering the critical section.
- Bounded Waiting is preserved as every process gets a fair chance.

Disadvantages of Peterson’s Solution

- It involves Busy waiting
- It is limited to 2 processes.

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a spinlock

Monitors

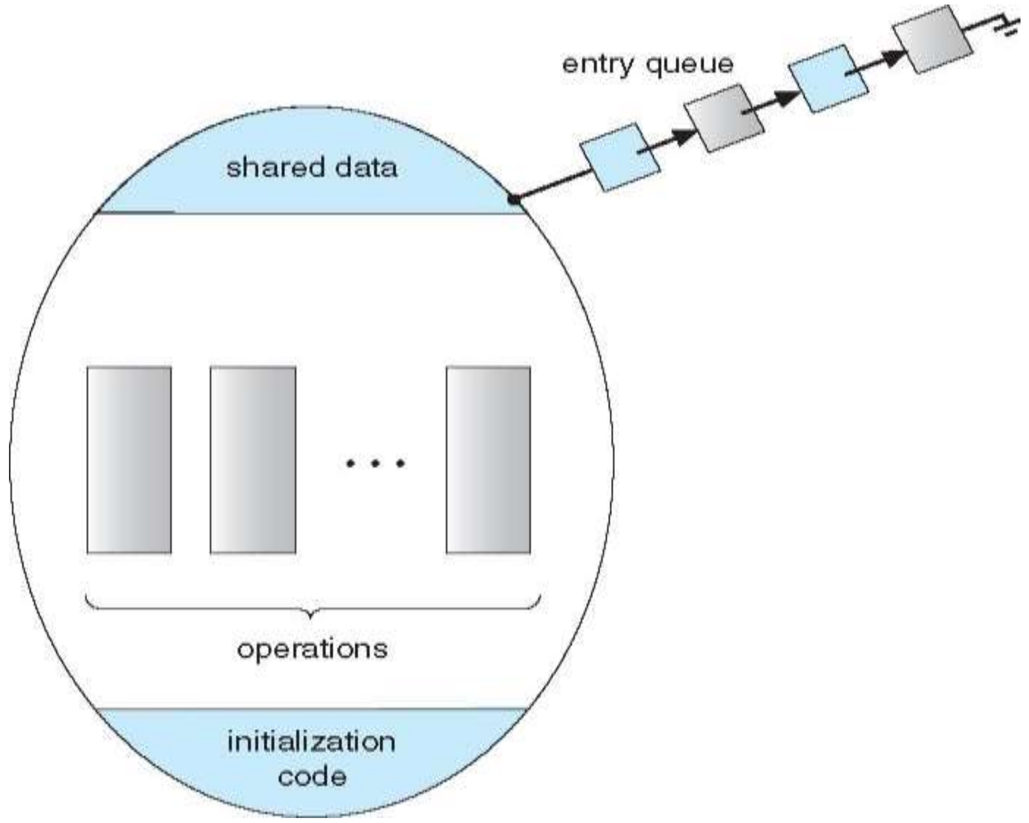
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
```

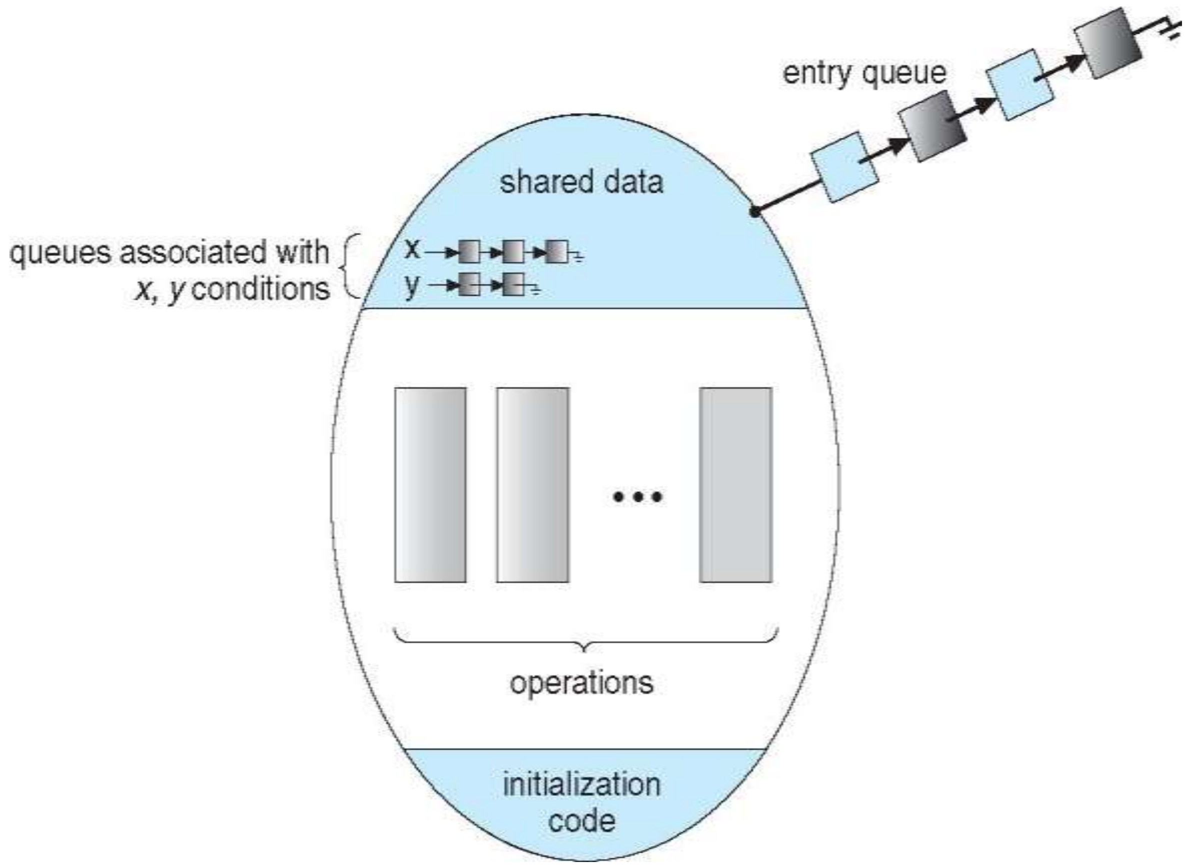
Schematic view of a Monitor



Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables

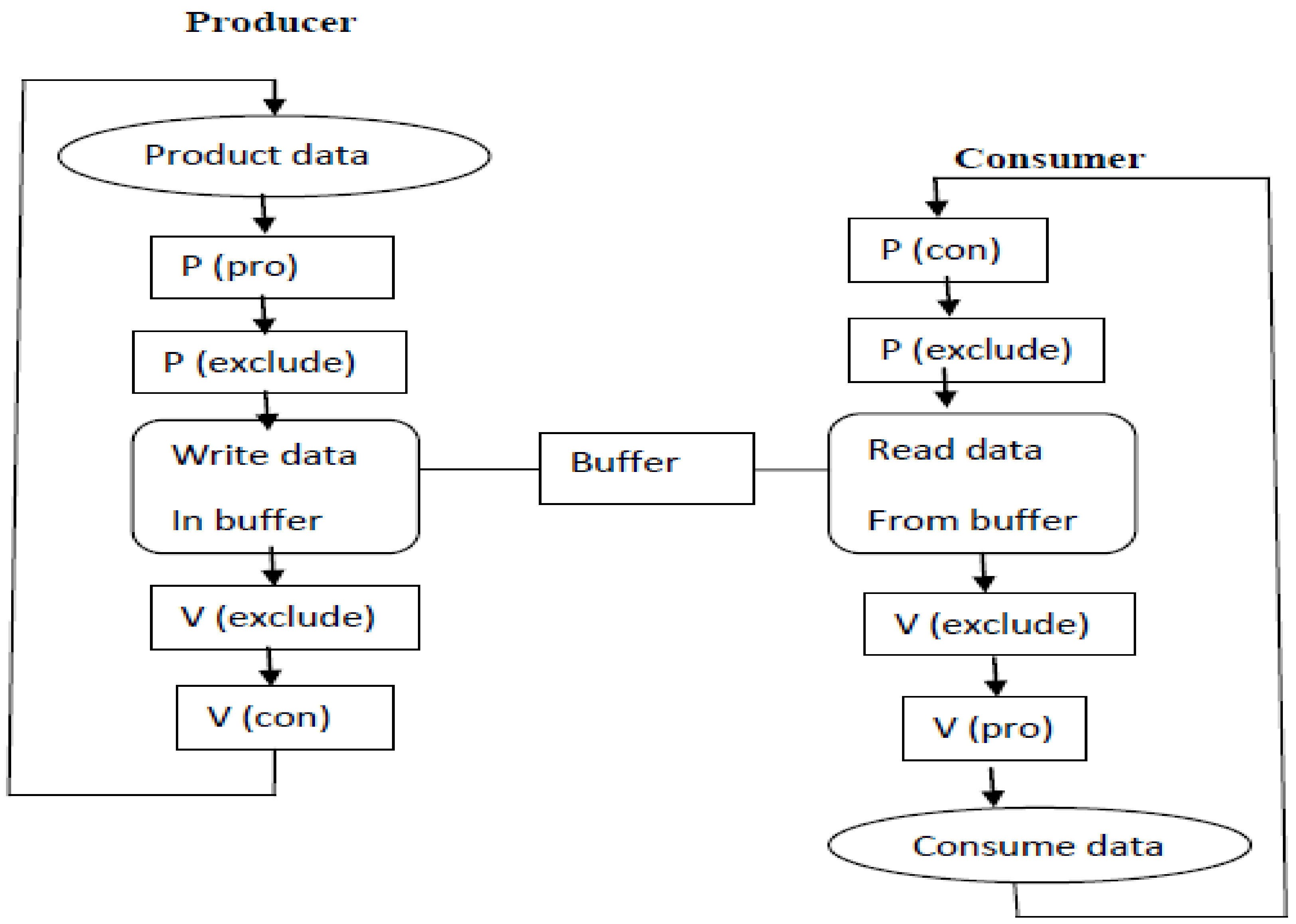


Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Producer Consumer Problem

- The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e., synchronization between more than one processes.
- In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.
- The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.


Producer
Product data
P (pro)
P (exclude)
Write data
In buffer
V (exclude)
V (con)
Buffer
Consumer
P (con)
P (exclude)
Read data
From buffer
V (exclude)
V (pro)
Consume data

What's the problem here?

The following are the problems that might occur in the Producer-Consumer:

- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

What's the solution?

The above three problems can be solved with the help of semaphores

In the producer-consumer problem, we use three semaphore variables:

1. **Semaphore S:** This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.
2. **Semaphore E:** This semaphore variable is used to define the empty space in the buffer. Initially, it is set to the whole space of the buffer i.e., "n" because the buffer is initially empty.
3. **Semaphore F:** This semaphore variable is used to define the space that is filled by the producer. Initially, it is set to "0" because there is no space filled by the producer initially.

By using the above three semaphore variables and by using the *wait()* and *signal()* function, we can solve our problem(the *wait()* function decreases the semaphore variable by 1 and the *signal()* function increases the semaphore variable by 1).

References

- www.csee.wvu.edu/~jdmooney/classes/cs550/notes/tech/mutex/Peterson.html
- https://youtu.be/iMD1Z3f9ioI
- https://youtu.be/hh9g5kKl_aE
- https://youtu.be/eoGkJWgxurQ
- https://www.researchgate.net/publication/262408664_Hardware_Implementation_of_Semaphore_Management_in_Real-Time_Operating_Systems