UNIT-III

Syllabus:

Part-1 : Concurrency

1. Process Synchronization
2. The Critical- Section Problem
3. Synchronization Hardware
4. Semaphores
5. Classic Problems of Synchronization
6. Monitors

Part-1 : Principles of Deadlock

1. System Model
2. Deadlock Characterization
3. Deadlock Prevention, Detection and Avoidance
4. Recovery from Deadlock

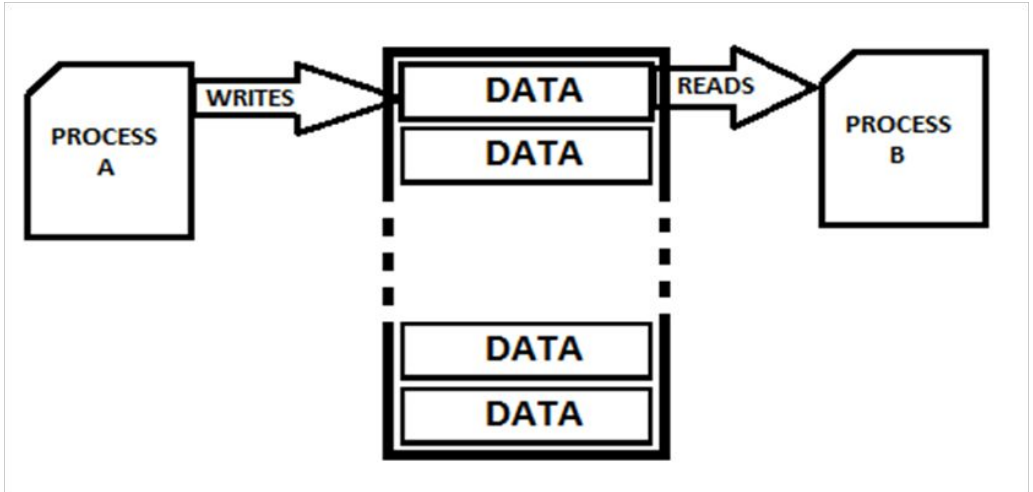
Process Synchronization

- A cooperative process is the one which can affect the execution of other process or can be affected by the execution of other process. Such processes need to be synchronized so that their order of execution can be guaranteed.
- **Process synchronization** in operating systems refers to the coordination and control of concurrent processes to ensure that they can **safely access shared resources without causing conflicts or errors.**
- When two or more process cooperates with each other, their order of execution must be preserved otherwise there can be conflicts in their execution and inappropriate outputs can be produced.
- Concurrent access to shared data may result in **data inconsistency**.

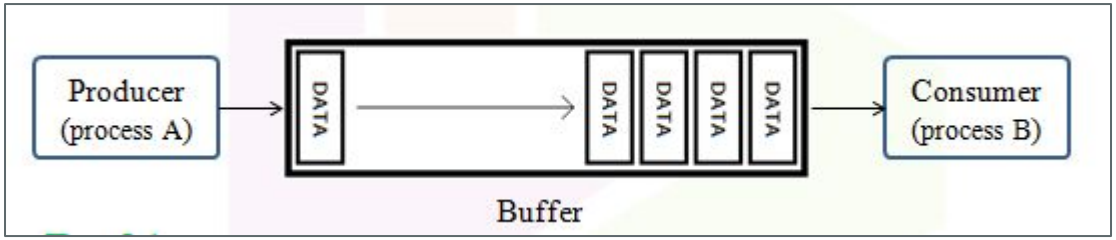
- The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other, and **to prevent the possibility of inconsistent data due to concurrent access.**
- In a multi-process environment, it is necessary to ensure that processes do not interfere with each other and that they access shared resources (such as memory or files) in a controlled and synchronized manner.
- To achieve this, various synchronization techniques such as
  a. **Critical Sections**
  b. **Semaphores**
  c. **Monitors**
  d. **Mutexes**

For Example:

- One process changing the data in a memory location where another process is trying to read the data from the same memory location. **It is possible that the data read by the second process will be erroneous.**



PRODUCER CONSUMER PROBLEM (or Bounded-Buffer Problem)



Problem:

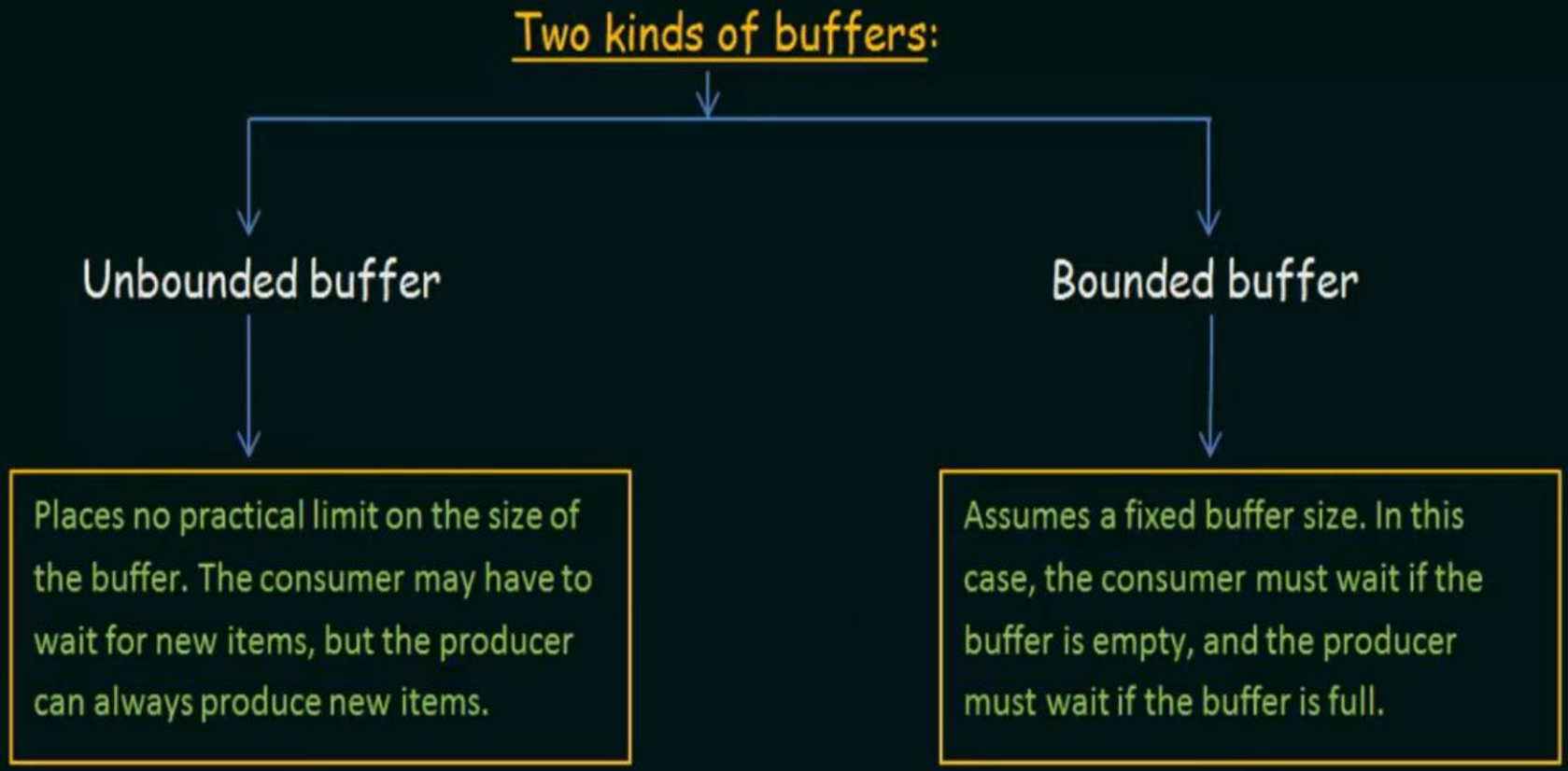
To ensure that the Producer should not add DATA **when the Buffer is full** and the Consumer should not take data **when the Buffer is empty**.

Producer Consumer Problem

A producer process produces information that is consumed by a consumer process.

For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

- One solution to the producer-consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.


Two kinds of buffers:
Unbounded buffer
Bounded buffer
Places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
Assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Shared data

```
#define BUFFER_SIZE 10

typedef struct {

. . .

} item;

item buffer[BUFFER_SIZE];

int in = 0;

int out = 0;

int counter = 0;
```

Code for the Producer Process

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Code for the Consumer Process

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

counter variable = 0

counter is incremented every time we add a new item to the buffer `counter++`

counter is decremented every time we remove one item from the buffer `counter--`

----------Example----------

- Suppose that the value of the variable counter is currently 5.
- The producer and consumer processes execute the statements "counter++" and "counter--" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!
- The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

"counter++" may be implemented in machine language (on a typical machine) as:

$register_1$ = counter
$register_1$ = $register_1$ + 1
counter = $register_1$

"counter--" may be implemented in machine language (on a typical machine) as:

$register_2$ = counter
$register_2$ = $register_2$ - 1
counter = $register_2$

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1$ + 1	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2$ - 1	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

T_0:	producer	execute	$register_1$	=	counter	{ $register_1$ = 5 }
T_1:	producer	execute	$register_1$	=	$register_1$ + 1	{ $register_1$ = 6 }
T_2:	consumer	execute	$register_2$	=	counter	{ $register_2$ = 5 }
T_3:	consumer	execute	$register_2$	=	$register_2$ - 1	{ $register_2$ = 4 }
T_4:	producer	execute	counter	=	$register_1$	{ counter = 6 }
T_5:	consumer	execute	counter	=	$register_2$	{ counter = 4 }

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

Race Condition:

Race conditions can occur in a variety of situations where multiple processes or threads attempt to access and modify shared resources simultaneously. Here are some real-time examples of race conditions:

- **Bank account transactions:** When multiple customers try to withdraw money from the same account simultaneously, race conditions can occur. If the account balance is not updated correctly, it may result in the account being overdrawn.
- **Online ticket booking:** When multiple users are trying to book the same seat on an online ticketing system, a race condition may occur where multiple users are allowed to reserve the same seat at the same time, leading to overbooking.
- **Multiplayer online games:** In a multiplayer online game, if two players attempt to pick up an object at the same time, a race condition can occur where only one of the players is able to pick it up, and the other player is left empty-handed.

- **Traffic signal systems:** In a traffic signal system, if two cars approach a traffic signal from opposite directions at the same time, and the system is not designed to handle this scenario, a race condition may occur where the signal turns green for both directions, leading to a potential collision.
- **Concurrent file access**: Multiple processes accessing a shared file simultaneously may result in a race condition where the file is overwritten or corrupted, leading to unexpected behavior.
- **Database access:** When multiple users try to update the same database record at the same time, a race condition can occur. The result may be that the data is inconsistent or that the changes made by one user are overwritten by another.
- **E-commerce websites:** When multiple users try to add the same product to their shopping cart simultaneously, a race condition can occur. This may result in incorrect product quantities or prices being displayed, leading to incorrect orders.

The Critical- Section Problem

1


TELEPHONE
←Social Distance →

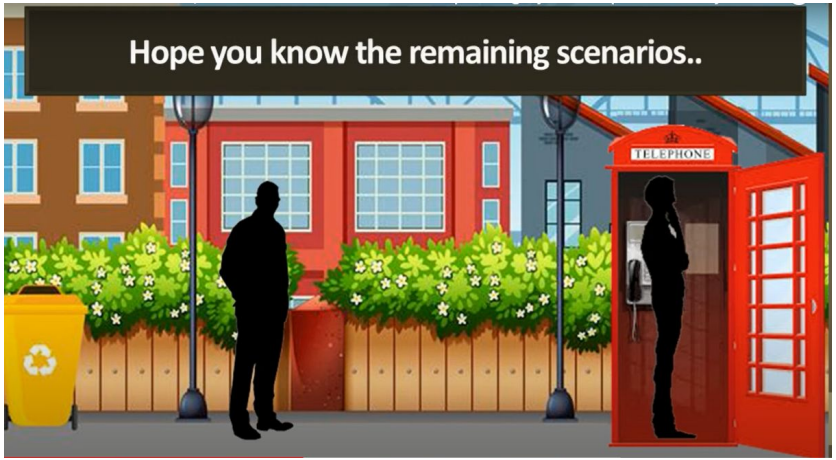
2


Since it is occupied , No other person can enter ,
Until the first person finish talking & leaves the Booth!
TELEPHONE
←Social Distance →

3


YOUTUBE
TELEPHONE

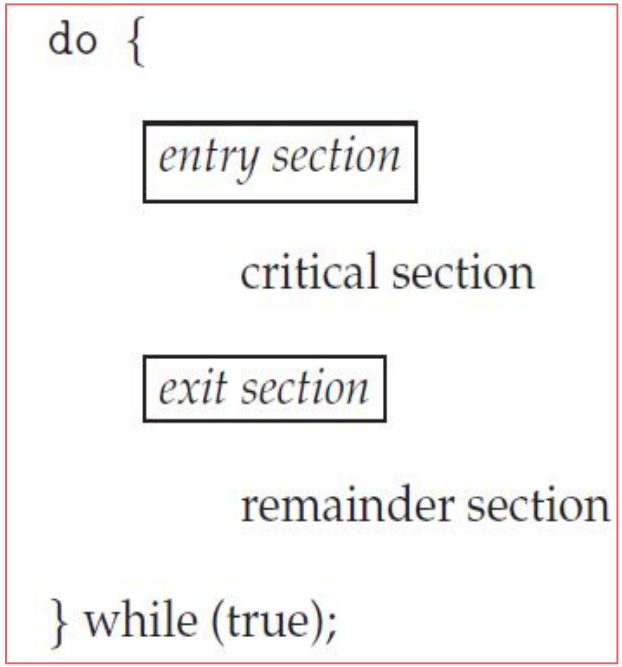
4


Hope you know the remaining scenarios..
TELEPHONE

- In multi-process program, multiple processes may access the same shared resource concurrently, leading to **race conditions** and **data inconsistency**.
- The critical section problem aims to provide a solution to ensure that **only one process or thread can access the shared resource at any given time.**
- Consider a system consisting of n processes {P0, P1, ..., Pn−1}. Each process has **a segment of code, called a critical section**,in which the process may be changing common variables, updating a table,writing a file, and so on.
- The important feature of the system is that, **when one process is executing in its critical section, no other process is allowed to execute in its critical section.**
- ***That is, no two processes are executing in their critical sections at the same time.***

- The critical-section problem is to design a protocol that the processes can use to cooperate.
- To solve this problem, various mutual exclusion algorithms have been proposed, including the use of **entry sections, exit sections, and reminder sections.**

- **Entry section:** The entry section is the part of the code where a process or thread requests permission to access the critical section. Before entering the critical section, the process or thread must acquire a lock or semaphore that ensures that no other process or thread can enter the critical section at the same time.



- **Critical section:** The critical section is the part of the code where the shared resource is accessed. Only one process or thread can be in the critical section at any given time to ensure that the resource is accessed in a mutually exclusive manner.
- **Exit section:** The exit section is the part of the code where a process or thread releases the lock or semaphore that it acquired in the entry section, indicating that it has finished accessing the critical section. This allows other processes or threads to enter the critical section if they are waiting.
- **Remainder Section:**The other parts of the Code other than Entry Section, Critical Section and Exit Section are known as Remainder Section.

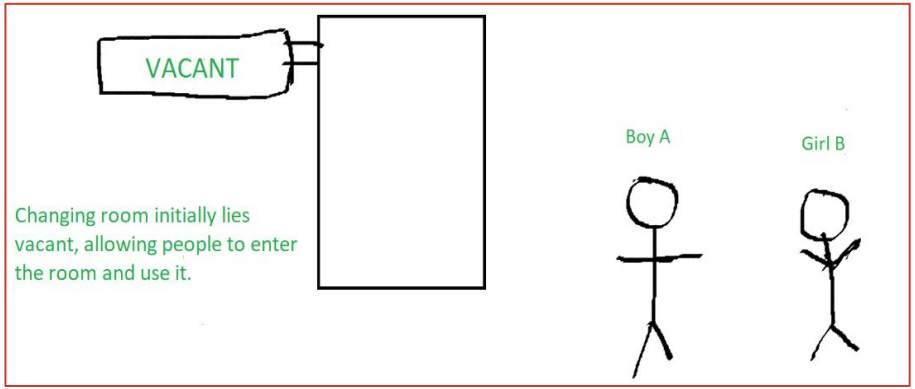
Here are some examples of critical sections:

- ★ **A bank account system:** When multiple users access their bank accounts simultaneously, they may need to withdraw or deposit money, which involves accessing a shared resource (the bank account). To prevent inconsistencies, a critical section can be used to ensure that only one user can access their account at a time.
- ★ **A printer spooler:** When multiple users send print jobs to a printer simultaneously, a critical section can be used to ensure that only one print job is executed at a time. This prevents conflicts and ensures that all print jobs are executed correctly.
- ★ **A web server:** When multiple users request access to a web server, a critical section can be used to ensure that only one request is handled at a time. This prevents conflicts and ensures that all requests are handled correctly.

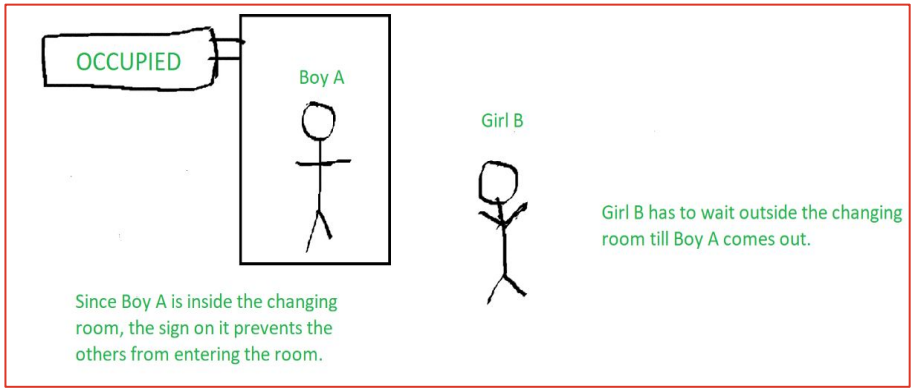
A solution to the critical-section problem must satisfy the following three requirements:

Mutual exclusion

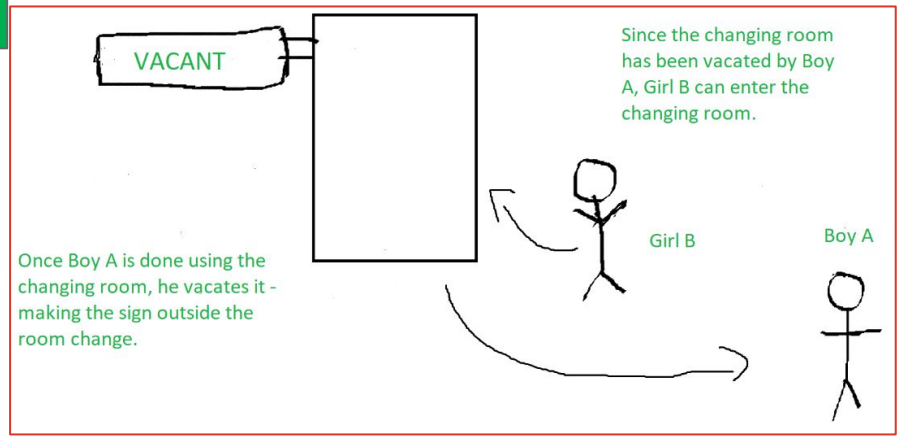
1



2



3

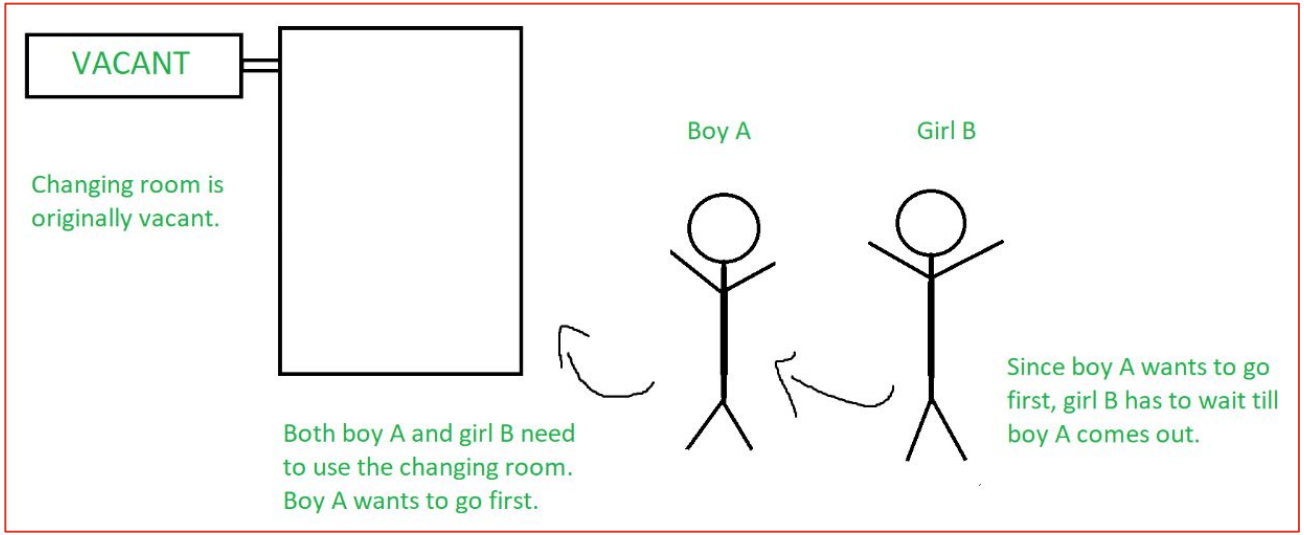


Eg:

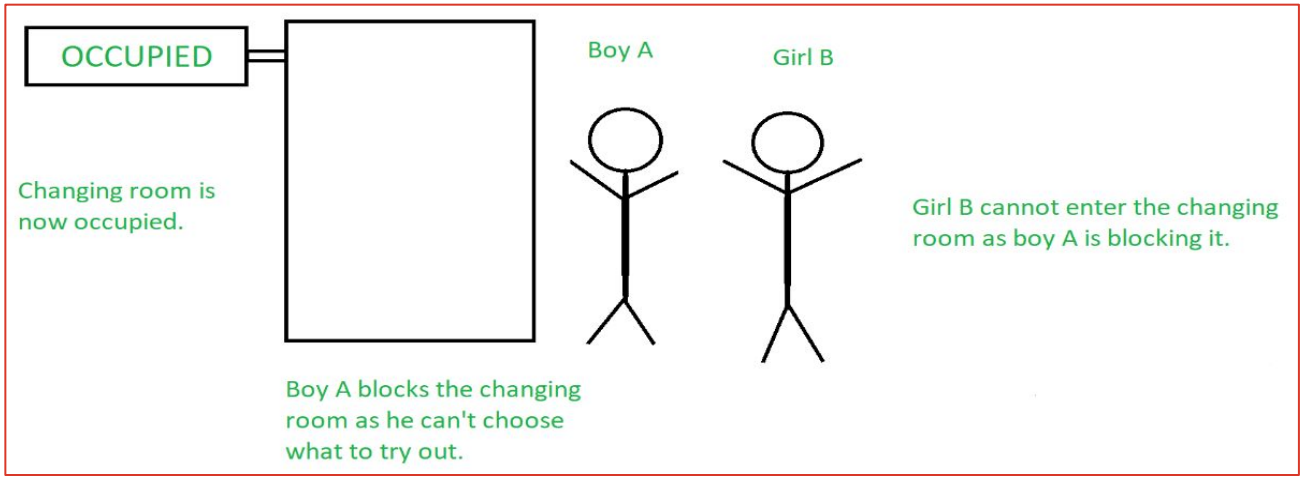
In the clothes section of a supermarket, two people are shopping for clothes.

Progress

1


VACANT
Changing room is originally vacant.
Boy A
Girl B
Since boy A wants to go first, girl B has to wait till boy A comes out.
Both boy A and girl B need to use the changing room.
Boy A wants to go first.

2


OCCUPIED
Changing room is now occupied.
Boy A
Girl B
Girl B cannot enter the changing room as boy A is blocking it.
Boy A blocks the changing room as he can't choose what to try out.

1. Mutual exclusion. If process Pi is executing in its critical section, then **no other processes** can be executing in their critical sections.

Eg:*Suppose a process P1 is executing in its critical section, then if the P2, P3 or some else process try to enter into the critical section of the P1, then all these processes need to wait until the P1 leaves the Critical Section.*

2. Progress. If **no process is currently executing in the critical section** and there are multiple processes waiting to enter, then a process that requests entry must eventually be allowed to enter and this selection **cannot be postponed indefinitely**.

This avoids situations where a process is blocked indefinitely, waiting for a resource that never becomes available.

Eg:*If a process P1 is not executing its own critical section, then P2 or some other process arrives to enter into the Critical Section of the P1, then P1 it should not stop any other process to access the Critical Section.*

3. Bounded waiting. There exists a bound, or limit, 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.

[or]

There should be a limit on the number of times a process or thread can be prevented from entering the critical section.

Eg:

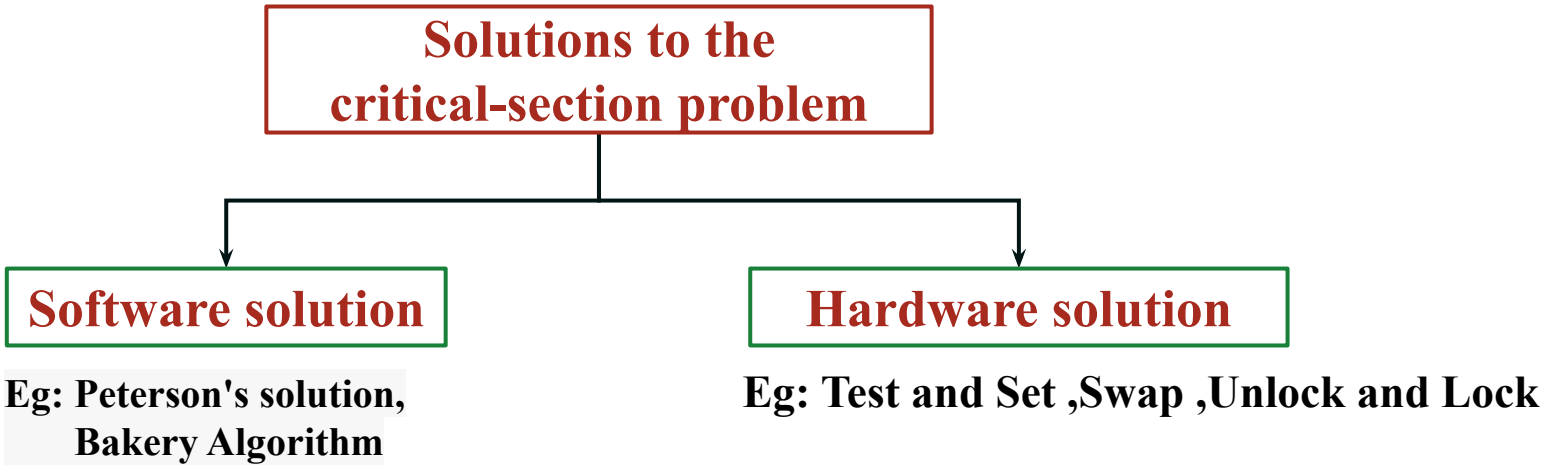
Suppose a bound(limit) can be maintained for Process P1 that if P2 or any other process wants to enter into the critical section of the P1, they can enter and leave the Critical Section of P1 ***only 1 time.***

- **Mutual Exclusion:** A traffic signal is a good example of mutual exclusion. At any given time, only one traffic signal can be green for a particular lane. If two traffic signals are green at the same time, it could cause accidents or traffic congestion. By allowing only one traffic signal to be green at a time, mutual exclusion is achieved.
- **Progress:** Imagine a ticketing system where multiple users are trying to book a ticket for a popular event. If one user gets the ticket and the others keep waiting indefinitely, it could result in frustration and negative user experience. By implementing a queue and ensuring that each user gets a turn to book a ticket, progress can be achieved.
- **Bounded Waiting:** Consider a robotic arm that performs various tasks in a manufacturing process. If multiple tasks require the robotic arm at the same time, a queue can be implemented to ensure that tasks are processed in the order they were received. However, if a task takes too long to complete, other tasks waiting in the queue may have to wait indefinitely. **To prevent this, a timeout mechanism can be added to ensure that the waiting time is bounded.**

Solutions to the Critical-Section Problem

- There are **two types** of solutions available to Critical-Section Problem.
- A software solution and a hardware solution both aim to ***ensure that only one process at a time can execute its critical section.***

Software solution	Hardware solution
software solutions rely entirely on the operating system and programming constructs to provide mutual exclusion	**a hardware solution involves using specialized hardware to enforce mutual exclusion**



Peterson's solution

Peterson's algorithm :

- Peterson's solution is a classic solution to the critical section problem.
- May not correctly work on Modern Computer Architectures.
- Peterson's solution **restricted to Two Processes Only**. The two processes **share two variables**:
  - **int turn**
  - **boolean flag[2]**

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. Let's call the processes P_i and P_j

Peterson's solution requires two data items to be shared between the two processes:

- **int turn** → Indicates whose turn it is to enter its critical section.
- **boolean flag [2]** → Used to indicate if a process is ready to enter its critical section.

The algorithm works as follows:

- ➔ Initially, both elements of the **flag array** are set to **false.**
- ➔ When a process wants to enter the critical section, it sets its **flag to true** and **sets turn to the other process.**
- ➔ **The process then enters a loop** where it checks if the other process's **flag is false or if it is its turn** to enter the critical section. If either of these conditions is true, the process enters the critical section. Otherwise, the process waits by spinning in the loop.
- ➔ When the process exits the critical section, it sets its **flag to false**, allowing the **other process to enter.**
- ➔ The algorithm continues to execute in this way, with the two processes taking turns to enter the critical section.

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Algorithm for Process P_i

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j);

            critical section

        flag[i] = false;

            remainder section

} while (true);
```

int turn

Indicates whose turn it is to enter its critical section.

boolean flag [2]

Used to indicate if a process is ready to enter its critical section.

Structure of process P_i in Peterson's solution

```
do {
    flag [i] = true ;
    turn = j ;
    while ( flag [ j ] && turn == [ j ] ) ;

        critical section

    flag [i] = false ;

        remainder section

    } while (TRUE) ;
```

Structure of process P_j in Peterson's solution

```
do {
    flag [ j ] = true ;
    turn = i ;
    while ( flag [ i ] && turn == [ i ] ) ;

        critical section

    flag [ j ] = false ;

        remainder section

    } while (TRUE) ;
```

:Case Study:
Peterson's Solution to Producer Consumer Problem

Code for producer (i)

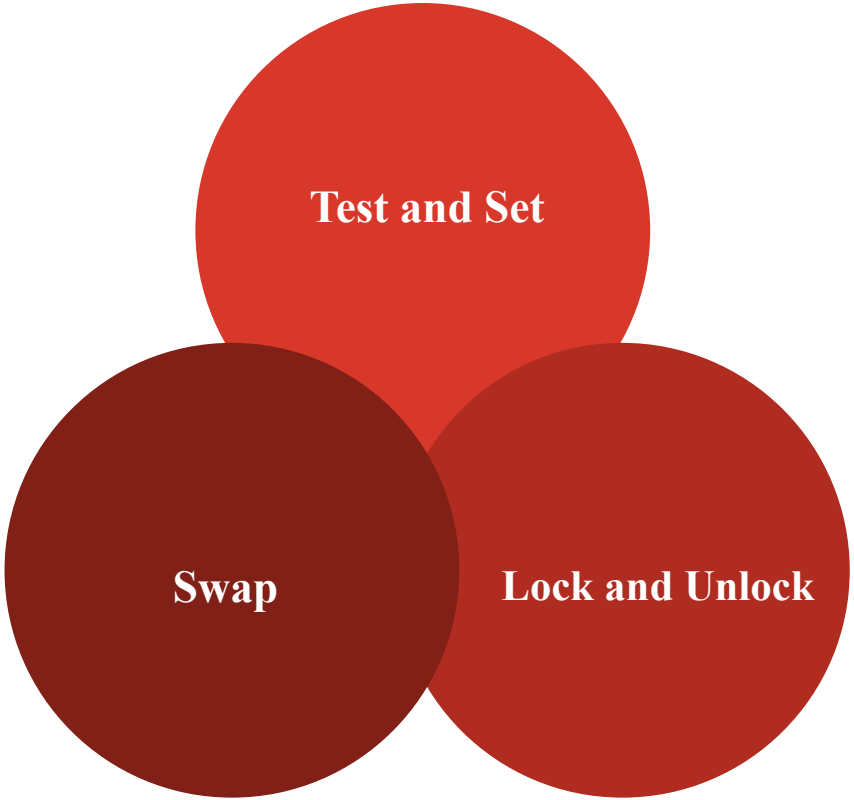
```
// producer i is ready  to produce an item

flag[i] = true;

turn = j;    // but consumer (j) can consume an item

// if consumer is ready to consume an item  and if it's consumer's turn

while (flag[j] == true && turn == j)

{     // then producer will wait

}

// otherwise producer will produce  an item and put it into buffer (Producer enter into critical Section)

flag[i] = false;      // Now, producer is out of critical section

   // end of code for producer
```

Code for Consumer (j)

```
// consumer j is ready  to consume an item

flag[j] = true;

turn = i;   // but producer (i) can produce an item

// if producer is ready to produce an item  and if it's producer's turn

while (flag[i] == true && turn == i)

{      // then consumer will wait

}

// otherwise consumer will consume  an item from buffer ( Consumer enter into critical Section)

flag[j] = false;  // Now, consumer is out of critical section

// end of code for consumer
```

1. **Mutual exclusion:** Peterson's Algorithm ensures that only one process can access the critical section at a time, thereby preventing concurrent access and potential data race conditions.
2. **Progress:** The algorithm guarantees that a process can enter the critical section if it is not currently being used by another process. This ensures that a process can make progress towards its goal, and it will eventually enter the critical section.
3. **Bounded waiting:** The algorithm ensures that a process that requests access to the critical section will eventually be granted access, provided that no other process is currently using the critical section. This prevents a process from waiting indefinitely to enter the critical section.

Hardware solution



1. Test and Set:

- software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures.
- Test and Set is a commonly used synchronization method in hardware synchronization problems.
- There is **a shared lock variable** which can take either of the two values , **0 or 1**.
- Before entering into the critical section , a process inquires about the lock.
  - If it is locked, it keeps on waiting till it becomes free.
  - If it is not locks, it takes the lock and executes the critical section.
- The important characteristic of Test and Set() instruction is that it is executed **atomically**.
- **Atomic = Non-Interruptible.**

1. The important characteristic of Test and Set() instruction is that it is executed **atomically**.
2. Thus, if two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
3. If the machine supports the TestAndSet() instruction,then we can implement **mutual exclusion by declaring a Boolean variable lock, initialized to false.**

The "Test and Set" instruction is a synchronization primitive used in concurrent programming to achieve mutual exclusion. It is typically implemented in hardware as an atomic instruction that performs ***two operations:***

Test: The instruction reads the value of a shared memory location that serves as a lock or a flag to indicate whether a critical section of code is currently being executed by another process.

Set: If the lock is not set, the instruction sets the lock and allows the current process to enter the critical section. If the lock is already set, the instruction blocks the current process until the lock is released by the other process.

Atomic Operation

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 5.3 The definition of the `test_and_set()` instruction.

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Atomic Operation

Process P1

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Atomic Operation

Process P1

** Let says that another process P2 tries to enter into Critical Section, when P1 was already executing Critical Section.

Process P2

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Semaphore

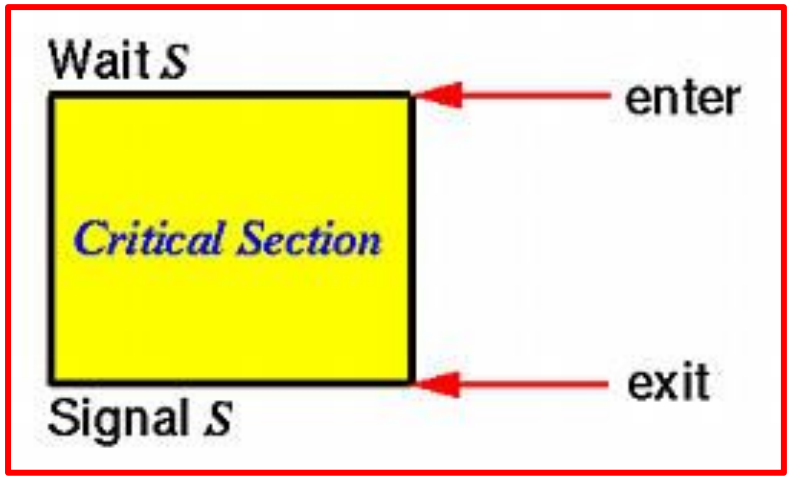
- A semaphore in an operating system is a synchronization mechanism that is used to control access to shared resources by multiple processes.
- A semaphore **S is an integer variable,** which is shared between processes.
- Semaphores are not used to exchange a large amount of data.
- Semaphores are used synchronization among Processes
- Semaphore is accessed only through two standard atomic operations:

1. Wait() **2. Signal()**

Wait() :The wait() operation was originally termed **P** (from the Dutch proberen, "**to test**")

Signal() : Signal() was originally called **V** (from verhogen, "**to increment**").

➔ Together, the wait() and signal() operations ensure that only one process or thread can access a shared resource at a time, preventing conflicts and ensuring that the resource is used correctly.




```
do
{
waiting(mutex);
   //Critical Section
signal(mutex)
 //remainder section
}while (True);
```

1. Wait( ):

➔ The wait( ) operation also called as **P( )** operation

➔ If the value of the semaphore is **greater than zero**, The wait( ) operation **decrements** the semaphore value and the process can proceed with its execution.

➔ Otherwise**, the process is blocked** until the semaphore value becomes **greater than zero**.

➔ This ensures that only one thread can access the shared resource at a time.

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

2. Signal( ):

- The signal( ) operation also called as **V( )** operation.
- **When a process is finished accessing the shared resource**, it calls **signal()** on the semaphore, which **increments** the semaphore's value.
- If there are Processes waiting on the semaphore, one of them is unblocked and allowed to proceed with its execution.

```
signal(S) {
    S++;
}
```

:Working of Semaphore:

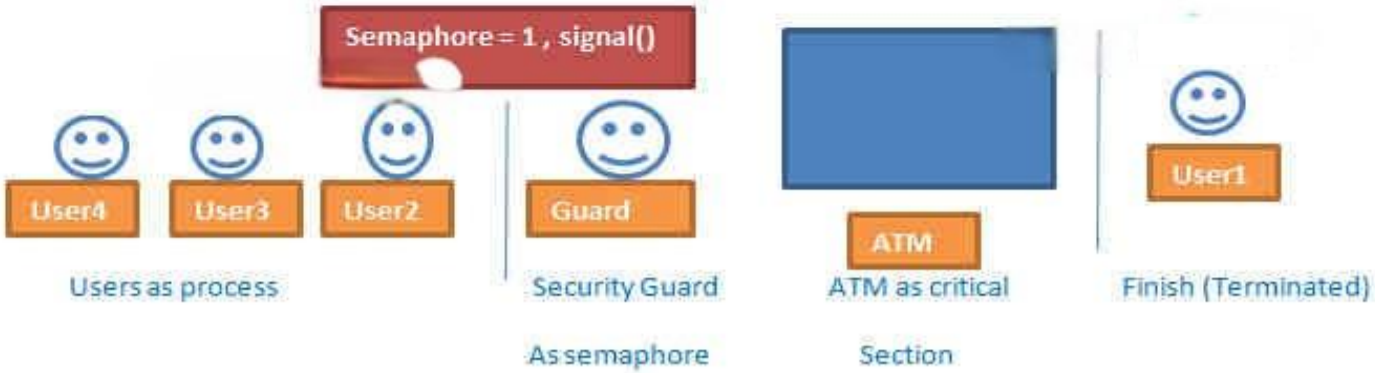
- Semaphores work by providing two basic operations, wait() and signal().
- **When a process wants to access a shared resource,** it first calls **wait()** on the semaphore.
- If the semaphore's value is **greater than zero**, the process is allowed to access the resource, and the semaphore's value is **decremented**.
- **If the semaphore's value is zero**, the process is blocked until the semaphore's value becomes greater than zero.
- **When a process is finished accessing the shared resource**, it calls **signal()** on the semaphore, which **increments** the semaphore's value and calls the blocked processes that were waiting for the resource.

:Realtime Example of Semaphore:

- ★ In this scenario, a semaphore can be used to control access to the printer so that only one process can use the printer at a time.
- ★ A semaphore is created with an initial value of 1 to represent the printer's availability.
- ★ When a process wants to print a document, it calls **wait()** on the semaphore.
- ★ If the semaphore's value is 1, it **decrements** the semaphore's value to 0 and proceeds to print the document.
- ★ If another process tries to print a document while the semaphore's **value is 0**, it is **blocked** until the semaphore's value becomes 1 again.
- ★ After the process finishes printing the document, it calls **signal()** on the semaphore to increment the semaphore's value to 1 and wake up any blocked processes waiting to use the printer.
- ★ This ensures that only one thread or process can use the printer at a time, preventing conflicts and ensuring that the printer is used efficiently.

How can processes get the critical section?

User1(Process1) is using ATM(Critical section), Guard (Semaphore) is performing wait operation and stops the user2, user3 and user4



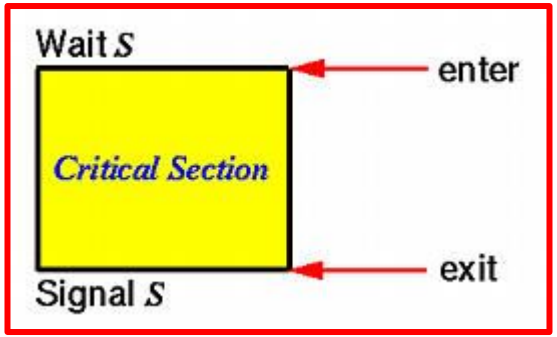
User1 finish its work and ATM (critical section) is free. Signal is given to all other process.

How can processes get the critical section?

A critical section is controlled by semaphores by following operations;

Wait:

- ★ Any process can't enter into the critical section.
- ★ The semaphore value is decremented.

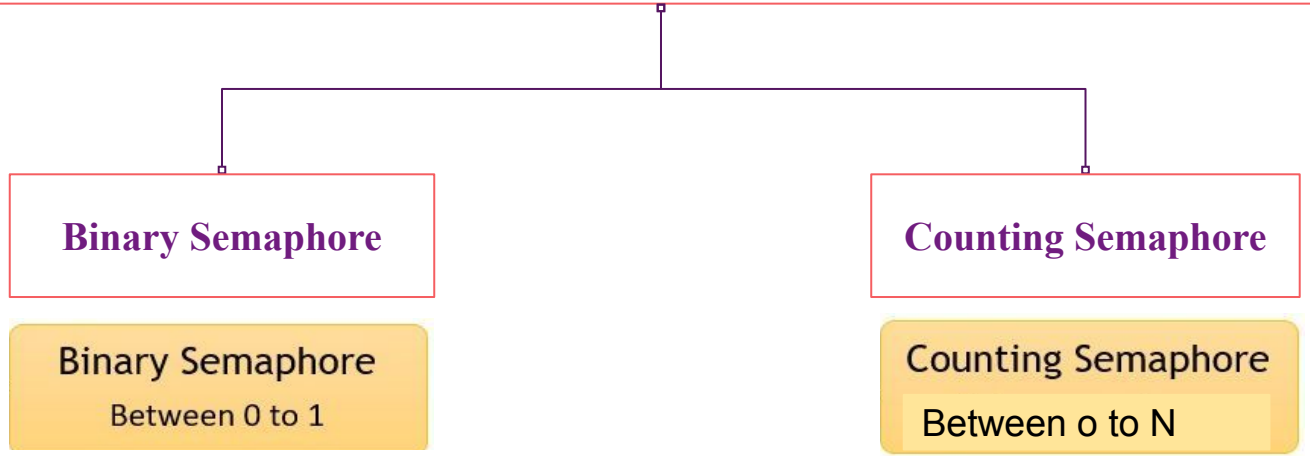


Signal:

- ★ The process can enter into the critical section.
- ★ The semaphore value is incremented.

```
do
{
waiting(mutex);
  //Critical Section
signal(mutex)
 //remainder section
}while (True);
```

Types of Semaphores



Binary Semaphores:

- ★ Also known as **mutexes**(short for mutual exclusion).
- ★ Binary semaphores are used for mutual exclusion.
- ★ They allow only one process to access a shared resource at a time.
- ★ Binary semaphores can have **two states: 0 and 1**.
- ★ When a process wants to access the resource, it checks the value of the binary semaphore.
- ★ **If the value is 1**, **the process can access the resource** and the semaphore value is set to 0 to indicate that the resource is currently in use.
- ★ **If the value is 0**, **the process must wait until the semaphore value becomes 1** again before it can access the resource.

Counting Semaphores:

- ★ Counting semaphores allow multiple processes to access a shared resource at the same time.
- ★ They maintain a count of the number of processes that can access the resource.
- ★ When a process **acquires the semaphore**, the **count is decremented.**
- ★ When the process **releases the semaphore**, the **count is incremented**.
- ★ The value of a counting semaphore can be any **non-negative integer**.

Classical Problems of Synchronization

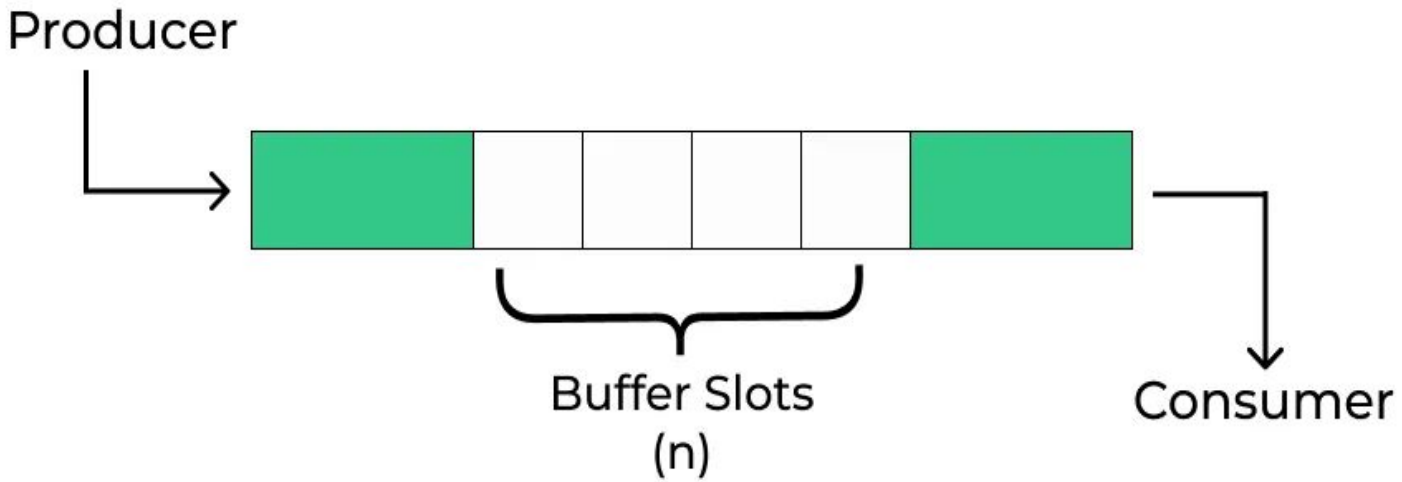
- In operating systems, synchronization is the process of coordinating the activities of concurrent processes to prevent conflicts and ensure data consistency.
- There are several classical problems of synchronization that arise in operating systems.
- Some of the most well-known problems include:

1. **Bounded-Buffer Problem or Producer-Consumer Problem**
2. **Readers-Writers Problem**
3. **Dining Philosophers Problem**

Bounded-Buffer Problem or Producer-Consumer Problem

- The Bounded-Buffer Problem is a classical synchronization problem that involves **coordinating multiple processes accessing a shared buffer with a fixed size.**
- The problem arises when one or more producer processes generate data that needs to be added to the buffer, and one or more consumer processes retrieve and process data from the buffer.
- The problem is called **"bounded"** because the buffer has **a fixed size**, and once it is full, the producers must wait for some of the data to be consumed before they can add more.
- Similarly, when the buffer is empty, the consumers must wait for the producers to add new data before they can retrieve and process it.

Bounded Buffer Problem



There is a buffer of n slots and each slot is capable of storing one unit of data.

There are two processes running, namely, **Producer** and **Consumer**, which are operating on the buffer.

- The producer tries to insert data into an empty slot of the buffer.
- The consumer tries to remove data from a filled slot in the buffer.
- The Producer must not insert data when the buffer is full.
- The Consumer must not remove data when the buffer is empty.
- The Producer and Consumer should not insert and remove data simultaneously.

Solution to the Bounded Buffer Problem using Semaphores:

We will make use of three semaphores:

1. m (mutex), a binary semaphore which is used to acquire and release the lock.
2. empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.
3. full, a counting semaphore whose initial value is 0.

:In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

★ We assume that the pool consists of **n buffers**, each capable of holding **one item**.

★ The **mutex semaphore** provides mutual exclusion for accesses to the buffer pool and is initialized to the value **1**.[**used to lock and release critical section**]

★ The **empty and full semaphores** count the number of empty and full buffers.

★ The semaphore **empty** is initialized to the value **n**.

★ The semaphore **full** is initialized to the value **0**.

The Code for The Producer Process

```
do {
    . . .
   /* produce an item in next_produced */
    . . .
   wait(empty);     // Wait until "Empty" > 0
   wait(mutex);     // Acquire a lock
    . . .
   /* add next_produced to the buffer */
    . . .
   signal(mutex); // Release a lock
   signal(full); // Increment "Full"
} while (true);
```

The Code for The Consumer Process

```
do {
   wait(full);      // Wait until "Full" > 0
   wait(mutex);     // Acquire a lock
    . . .
   /* remove an item from buffer to next_consumed */
    . . .
   signal(mutex);   // Release a lock
   signal(empty);   // Increment "Empty"
    . . .
   /* consume the item in next_consumed */
    . . .
} while (true);
```

Producer Process:

- When a producer wants to add an item to the buffer, it first waits for an empty slot (wait(empty)), then waits for exclusive access to the buffer (wait(mutex)). It then adds the item to the buffer, updates the producer index, and signals the mutex (signal(mutex)) and the full semaphore (signal(full)) to notify consumers that a new item is available.

Consumer Process:

- When a consumer wants to retrieve an item from the buffer, it first waits for a full slot (wait(full)), then waits for exclusive access to the buffer (wait(mutex)). It then retrieves the item from the buffer, updates the consumer index, and signals the mutex (signal(mutex)) and the empty semaphore (signal(empty)) to notify producers that a slot has become available.

The Readers –Writers Problem

- The Readers-Writers Problem is a classic synchronization problem that involves multiple processes trying to access a shared resource.
- In this problem, there are two types of processes: **readers and writers.**
- Suppose that a database is shared among a number of concurrent processes
    - **Readers** – only read the data set; they do not perform any updates
    - **Writers** – can both read and write.

Problem –

- If **two readers** access the shared data simultaneously, no adverse effects will result.
- However, **if a writer and some other process (either a reader or a writer)** access the database simultaneously, it results Data Inconsistency.
- To ensure that these difficulties do not arise, **Only one writer can access the resource at a time**.

Solution to Reader-Writer Problem Using Semaphore

We will make use of **Two Semaphores and an integer Value :**

1.mutex :

mutex is a semaphore **(initialized to 1)** which is used to ensure mutual exclusion when the variable **read_count** is updated. i.e. when any reader enters or exit from critical section.

2.wrt :

Wrt is a a semaphore **(initialized to 1)** common to both reader and writer process.

3.read_count :

The read count is an integer variable **(initialized to 0)** variable keeps track of how many processes are currently reading the object.

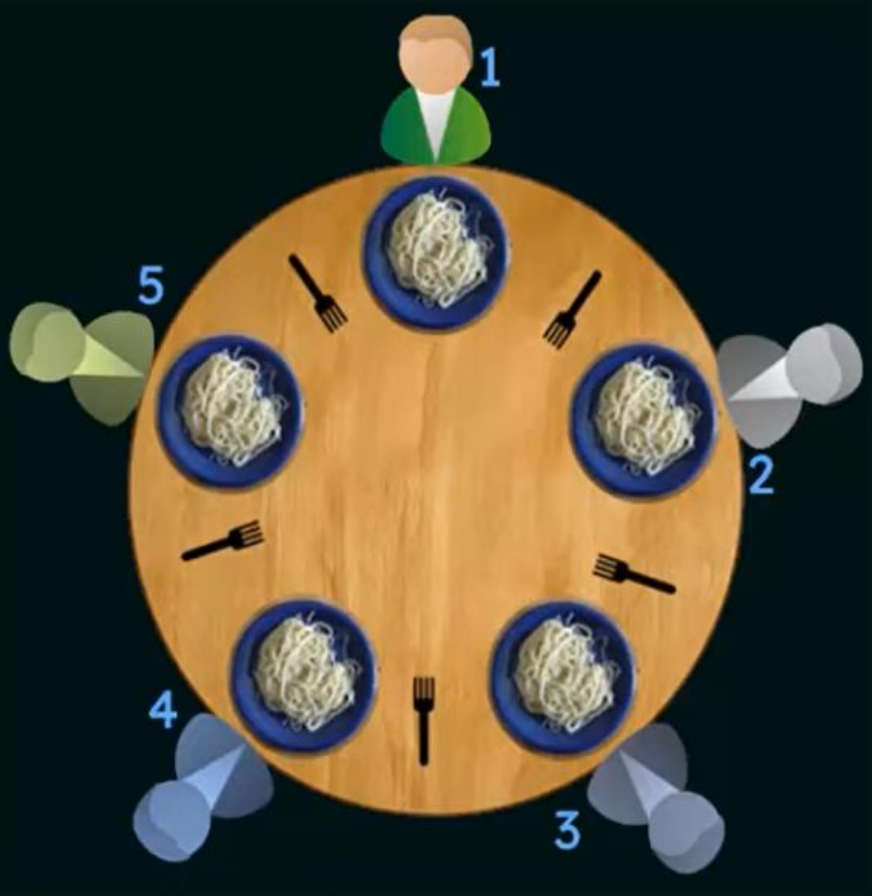
The Code for The Writer Process

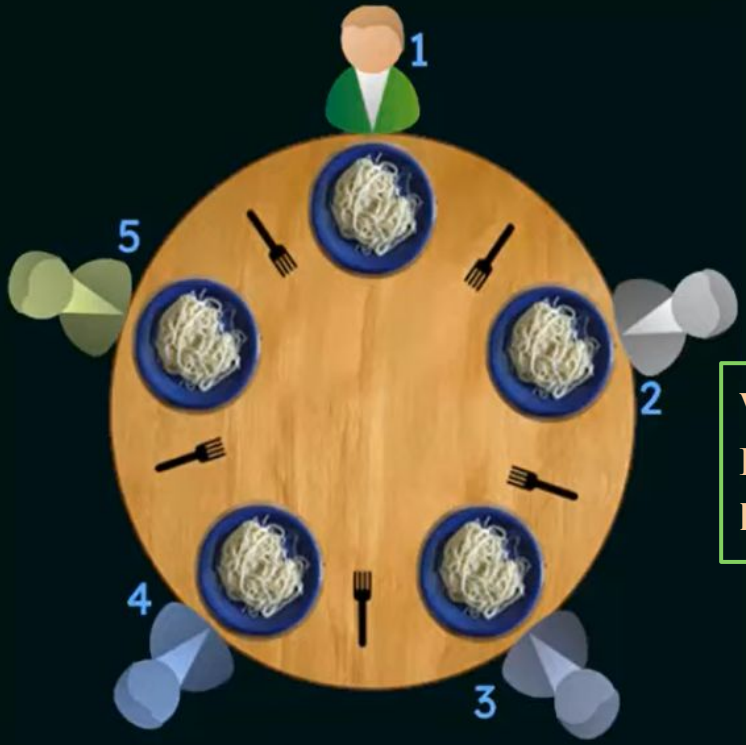
```
do {
    // Writer request for critical
    section
       wait (wrt) ;
    //    writing is performed
    // leaves the critical section
      signal (wrt) ;
    } while (true)
```

The Code for The Reader Process

```
do {
        wait(mutex);
        read_count++; // The no.of readers has now increased by 1
        if (read_count == 1)
                    wait(wrt);  // this ensure no write can enter  if there is even one reader presents
        signal(mutex); // Other readers can enter while this current reader is inside the critical section
/* Current reader performs reading here */
        wait(mutex);
        read count--;  // A reader want to leave
        if (read_count == 0)  // No reader is left in the critical section
                    signal(wrt);  // Writers can enter
        signal(mutex);      // reader leave
} while (true);
```

Dining Philosophers Problem


1
2
3
4
5



Philosopher Either

Thinks

When Philosopher thinks, He does not interact with her colleagues.

Eats

a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him (left and right).

A philosopher may pick up only one chopstick at a time.

One cannot pick up a chopstick that is already in the hand of a neighbor.

- Consider five philosophers who spend their lives **thinking and eating.**
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, He does not interact with his colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him (left and right).
- A philosopher may pick up only one chopstick at a time.
- Obviously, he cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both his chopsticks at the same time, he eats without releasing the chopsticks.
- When he is finished eating, he puts down both chopsticks and starts thinking again.

- ★ One simple solution is to represent each chopstick with **a semaphore.**
- ★ A philosopher tries to grab a chopstick by executing a **wait() operation** on that semaphore. .
- ★ He releases his chopsticks by executing the **signal() operation** on the appropriate **semaphores**.
- ★ Thus, the shared data are

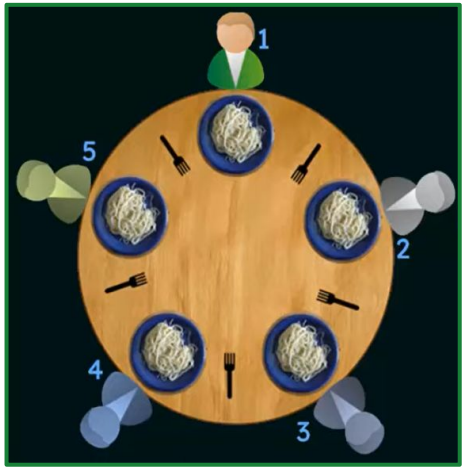
Semaphore chopstick[5];

- ★ where all the elements of chopstick are **initialized to 1.**

The structure of philosopher i

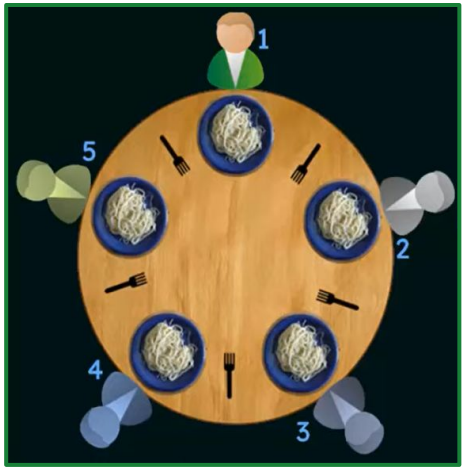
```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

- ➔ Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create ***a deadlock***.
- ➔ Suppose that **all five philosophers become hungry at the same time** and each grabs his left chopstick. All the elements of chopstick will now be **equal to 0**.
- ➔ When each philosopher tries to grab his right chopstick, he will be delayed forever.



Several possible remedies to the **deadlock** problem are replaced by:

1. Allow **at most four philosophers** to be sitting simultaneously at the table.
2. Allow a philosopher to pick up his chopsticks only **if both chopsticks are available**
3. **Use an asymmetric solution**—that is, **an odd-numbered philosopher picks up first her left chopstick and then her right chopstick,** whereas **an even numbered philosopher picks up her right chopstick and then her left chopstick.**



MONITORS

1. **Monitors and semaphores** are used for process synchronization and allow processes to access the shared resources using mutual exclusion.
2. Monitors are a synchronization construct that were created **to overcome the problems caused by semaphores** such as “timing errors and deadlocks”.
3. Monitors provide a safe and efficient way to coordinate access to shared resources in multi-process systems.
4. Monitor is an abstract data type or ADT, encapsulates **data(shared data variables) with a set of functions(procedures)** to operate on that data.

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

- ➔ Function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- ➔ Similarly, the local variables of a monitor can be accessed by only the local functions.
- ➔ **The monitor construct ensures that only one process at a time is active within the monitor.**

- ➔ Monitors has an additional mechanism to provide synchronization, i.e **Conditional Constructs.**

Conditional Constructs:

- Monitors also provide a way for processes to wait for a certain condition to become true before accessing the shared resource.
- This is achieved through the use of **condition variables**, which are associated with specific monitor procedures.
  Eg: condition x, y;
- The only operations that can be invoked on a condition variable are **wait() and signal().**

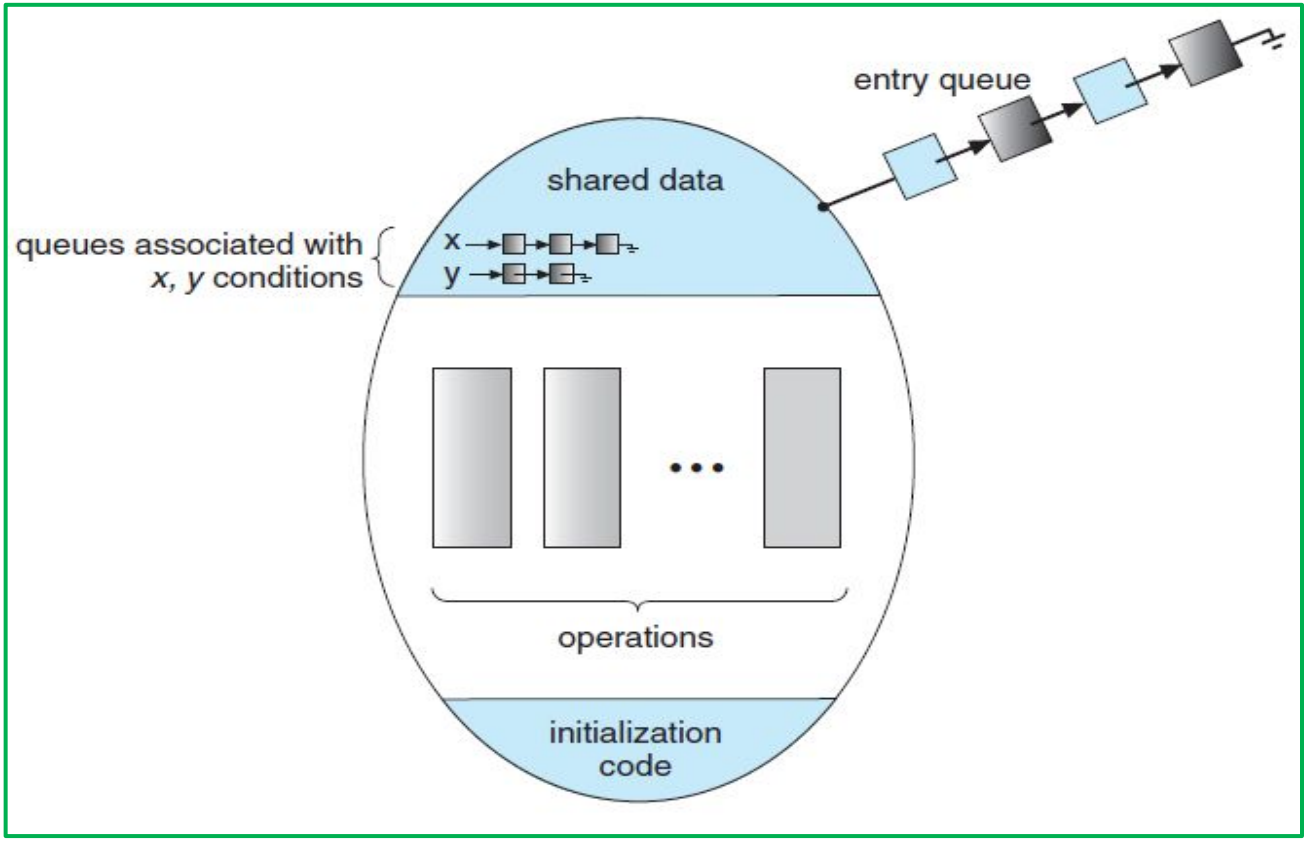
x.wait():

The process invoking x.wait() operation is suspended until another process invokes x.signal().

x.signal():

The x.signal() operation resumes exactly one suspended process.

Monitor with Conditional Variable



Schematic View of Monitor

Dining-Philosophers Solution Using Monitors

- we illustrate Monitor concepts by presenting **a deadlock-free solution** to the dining-philosophers problem.
- This solution imposes **the restriction** that a philosopher **may pick up her chopsticks only if both of them are available.**
- To code this solution, we need to distinguish among **three states** in which we may find a philosopher.
- For this purpose, we introduce the following data structure :

```
enum {THINKING, HUNGRY, EATING} state[5];
```

- Philosopher **i** can set the variable **state[i] = EATING** only if her two neighbors are not eating: **(state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING)**.
- We also need to declare **condition self[5];**
- This allows philosopher i to delay himself when he is hungry but is unable to obtain the chopsticks she needs.

```
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);
      test((i + 1) % 5);
   }

   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;
    }
}
```