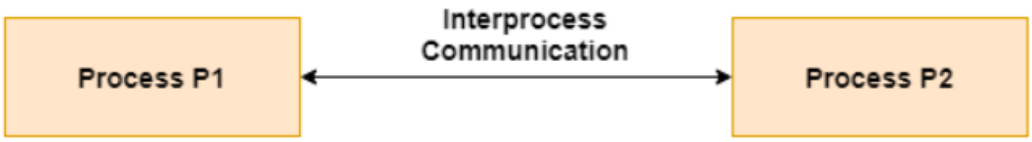
MODULE-3

Inter-process Communication:

Interposes communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.

A diagram that illustrates interprocess communication is as follows −



Synchronization in Inter process Communication

Synchronization is a necessary part of inter process communication. It is either provided by the inter process control mechanism or handled by the communicating processes. Some of the methods to provide synchronization are as follows −

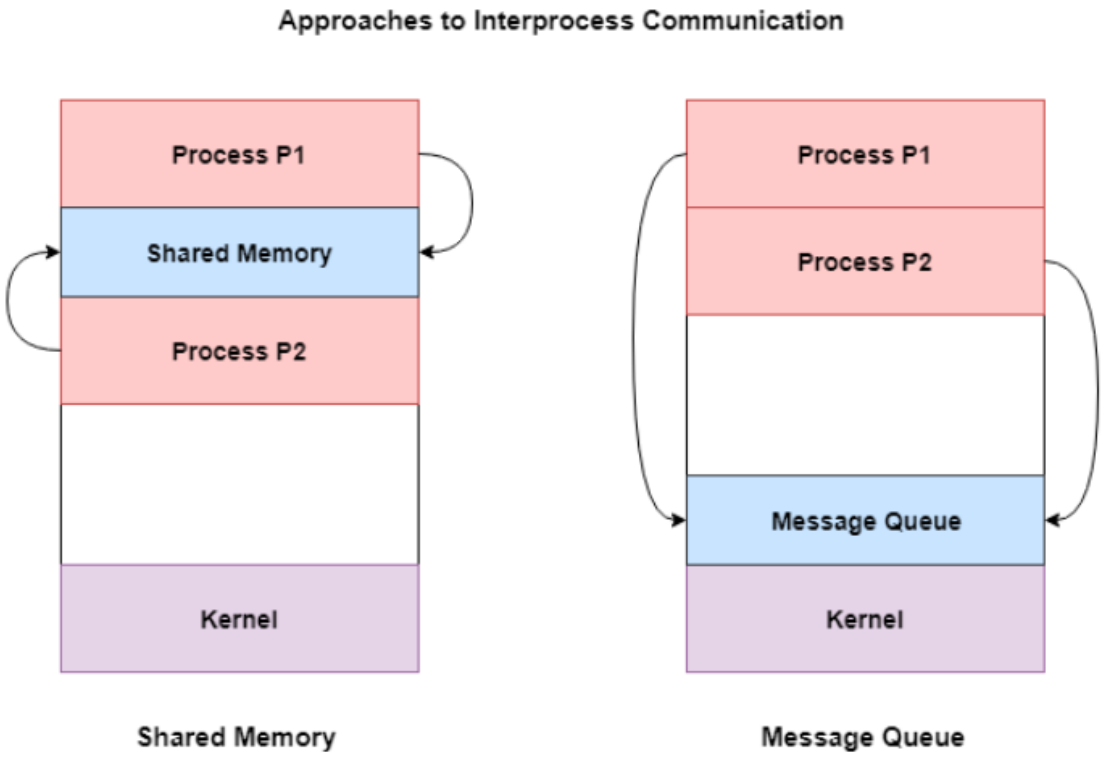
- **Semaphore**
  A semaphore is a variable that controls the access to a common resource by multiple processes. The two types of semaphores are binary semaphores and counting semaphores.
- **Mutual Exclusion**
  Mutual exclusion requires that only one process thread can enter the critical section at a time. This is useful for synchronization and also prevents race conditions.
- **Barrier**
  A barrier does not allow individual processes to proceed until all the processes reach it. Many parallel languages and collective routines impose barriers.
- **Spinlock**
  This is a type of lock. The processes trying to acquire this lock wait in a loop while checking if the lock is available or not. This is known as busy waiting because the process is not doing any useful operation even though it is active.

Approaches to Interprocess Communication

The different approaches to implement interprocess communication are given as follows −

- **Pipe**
  A pipe is a data channel that is unidirectional. Two pipes can be used to create a two-way data channel between two processes. This uses standard input and output methods. Pipes are used in all POSIX systems as well as Windows operating systems.
- **Socket**
  The socket is the endpoint for sending or receiving data in a network. This is true for data sent between processes on the same computer or data sent between different computers on the same network. Most of the operating systems use sockets for interprocess communication.
- **File**
  A file is a data record that may be stored on a disk or acquired on demand by a file server. Multiple processes can access a file as required. All operating systems use files for data storage.
- **Signal**
  Signals are useful in interprocess communication in a limited way. They are system messages that are sent from one process to another. Normally, signals are not used to transfer data but are used for remote commands between processes.
- **Shared Memory**
  Shared memory is the memory that can be simultaneously accessed by multiple processes. This is done so that the processes can communicate with each other. All POSIX systems, as well as Windows operating systems use shared memory.

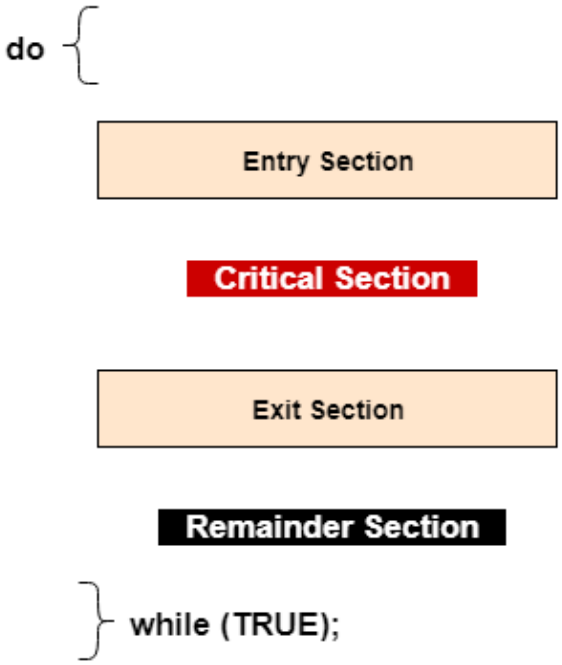
A diagram that demonstrates message queue and shared memory methods of interprocess communication is as follows –



Critical Section

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.

A diagram that demonstrates the critical section is as follows −



In the above diagram, the entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

Solution to the Critical Section Problem

The critical section problem needs a solution to synchronize the different processes. The solution to the critical section problem must satisfy the following conditions −

- **Mutual Exclusion**
  Mutual exclusion implies that only one process can be inside the critical section at any time. If any other processes require the critical section, they must wait until it is free.
- **Progress**
  Progress means that if a process is not using the critical section, then it should not stop any other process from accessing it. In other words, any process can enter a critical section if it is free.
- **Bounded Waiting**
  Bounded waiting means that each process must have a limited waiting time. It should not wait endlessly to access the critical section.

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.

Race conditions are most commonly associated with computer science and programming. 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.

Race conditions are considered a common issue for multithreaded applications.

What are examples of race conditions?

A simple example of a race condition is a light switch. In some homes, there are multiple light switches connected to a common ceiling light. When these types of circuits are used, the switch position becomes irrelevant. If the light is on, moving either switch from its current position turns the light off. Similarly, if the light is off, then moving either switch from its current position turns the light on.

With that in mind, imagine what might happen if two people tried to turn on the light using two different switches at the same time. One instruction might cancel the other or the two actions might trip the circuit breaker.

In computer memory or storage, a race condition may occur if commands to read and write a large amount of data are received at almost the same instant, and the machine attempts to overwrite some or all of the old data while that old data is still being read. The result may be one or more of the following:

- the computer crashes or identifies an illegal operation of the program

- errors reading the old data
- errors writing the new data

A race condition can also occur if instructions are processed in the incorrect order.

Suppose two processes need to perform a bit flip at a specific memory location. Under normal circumstances the operation should work as shown in Figure 1.

If a race condition occurred causing these two processes to overlap, the sequence of operations could potentially look more like what's shown in Figure 2.

Figure 2. When Process 2 is unaware that Process 1 is performing a simultaneous bit flip, the bit has an ending value of 1 when its value should be 0.

Race conditions occasionally occur in logic gates when inputs come into conflict. Because the gate output state takes a finite, nonzero amount of time to react to any change in input states, sensitive circuits or devices following the gate may be fooled by the state of the output and not operate properly.

What are the types of race conditions?

There are a few types of race conditions. Two categories that define the impact of the race condition on a system are referred to as critical and noncritical:

- A **critical** race condition will cause the end state of the device, system or program to change. For example, if flipping two light switches connected to a common light at the same time blows the circuit, it is considered a critical race condition. In software, a critical race condition is when a situation results in a bug with unpredictable or undefined behavior.
- A **noncritical** race condition does not directly affect the end state of the system, device or program. In the light example, if the light is off and flipping both switches simultaneously turns the light on and has the same effect as flipping one switch, then it is a noncritical race condition. In software, a noncritical race condition does not result in a bug.

Critical and noncritical race conditions aren't limited to electronics or programming. They can occur in many types of systems where race conditions happen.

In programming, two main types of race conditions occur in a critical section of code, which is a section of code executed by multiple threads. When multiple threads try to read a variable and then each acts on it, one of the following situations can occur:

- **Read-modify-write.** This kind of race condition happens when two processes read a value in a program and write back a new value. It often causes a software bug. Like the example above, the expectation is that the two processes will happen sequentially -- the first process produces its value and then the second process reads that value and returns a different one.

  For example, if checks against a checking account are processed sequentially, the system will make sure there are enough funds in the account to process check A first and then look again to see if there are enough funds to process check B after processing check A. However, if the two checks are processed at the same time, the system may read the same account balance value for both processes and give an incorrect account balance value, causing the account to be overdrawn.

What security vulnerabilities do race conditions cause

A program that is designed to handle tasks in a specific sequence can experience security issues if it is asked to perform two or more operations simultaneously. A threat actor can take advantage of the time lapse between when the service is initiated and when a security control takes effect in order to create a deadlock or thread block situation.

A deadlock vulnerability is a severe form of a denial-of-service vulnerability. It can be made to occur when two or more threads must wait for one another to acquire or release a lock in a circular chain. This situation results in deadlock, where the entire software system comes to a halt because such locks can never be acquired or released if the chain is circular.

Thread block can also dramatically impact application performance. In this type of concurrency defect, one thread calls a long-running operation while holding a lock and preventing the progress of other threads.

How to identify race conditions

Detecting and identifying race conditions is considered difficult. They are a semantic problem that can arise from many possible flaws in code. It's best to design code in a way that prevents these problems from the start.

Programmers use dynamic and static analysis tools to identify race conditions. Static testing tools scan a program without running it. However, they produce many false reports. Dynamic analysis tools have fewer false reports, but they may not catch race conditions that aren't executed directly within the program.

Race conditions are sometimes produced by data races, which occur when two threads concurrently target the same memory location and at least one is a write operation. Data races are easier to detect than race conditions because specific conditions are required for them to occur. Tools, such as the Go Project's Data Race

Detector, monitor for data race situations. Race conditions are more closely tied to application semantics and pose broader problems.

How do you prevent race conditions?

Two ways programmers can prevent race conditions in operating systems and other software include:

- **Avoid shared states.** This means reviewing code to ensure when shared resources are part of a system or process, atomic operations are in place that run independently of other processes and locking is used to enforce the atomic operation of critical sections of code. Immutable objects can also be used that cannot be altered once created.
- **Use thread synchronization.** Here, a given part of the program can only execute one thread at a time.

Preventing race conditions with other types of technology is also possible:

Storage and memory

The serialization of memory or storage access will also prevent race conditions. This means if read and write commands are received close together, the read command is executed and completed first by default.

Networking

In a network, a race condition may occur if two users try to access a channel at the same instant and neither computer receives notification the channel is occupied before the system grants access. Statistically, this kind of situation occurs mostly in networks with long lag times, such as those that use geostationary satellites.

To prevent such a race condition, a priority scheme must be devised to give one user exclusive access. For example, the subscriber whose username or number begins with

the earlier letter of the alphabet or the lower numeral may get priority when two subscribers attempt to access the system within a prescribed increment of time.

The takeaway

Race conditions show up in several ways in software, storage, memory and networking. Proactively monitoring for them and preventing them is a critical part of software and technology design and development.

Preventing race conditions is particularly important because hackers can take advantage of race-condition vulnerabilities to gain unauthorized access to networks. One notable example of a race condition-based exploit is called Dirty Cow, which exploits a flaw in a Linux kernel's memory subsystem to create a race condition where the attacker gains write privileges for read-only memory mappings.

Mutual Exclusion

Only one process can execute its critical sections at any time. No other processes can be executed in their critical sections. If a process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection can’t be postponed indefinitely. This process is called Mutual Exclusion.

Requirements for Mutual Exclusion:
1. Mutual Exclusion must be enforced, only one process at a time is allowed into its critical section among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its non-critical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely, no deadlock or starvation can be allowed.
4. When no process is in a critical section, any process that requires entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or the number of processes.
6. A process remains inside its critical section for a finite time only.

Strict Alternation:-

Peterson's formula was designed to work with just two processes but has since been generalized to accommodate multiple processes.
It would be possible to avoid race conditions if no two processes were ever at a critical stage simultaneously. To solve the critical section problem of mutual exclusion, we need four conditions to be true.

1. During critical sections, two processes cannot run simultaneously.
2. It is not assumed that processes are running at the same speed or multiple CPUs.
3. There should be no process blocking another process.
4. It is important not to act arbitrarily long before entering a critical section in a process.

A critical section is a sequence of instructions that cannot be interleaved among more than one thread. The simplest case of a critical section occurs when two threads use the same global variable var and change its value simultaneously using var++.

How does the Peterson's algorithm work

Mutex, or mutual exclusion, is an object in a program that simultaneously prevents multiple people from accessing the same resource.

- During concurrent programming, critical sections are used to access a shared resource by processes or threads.
- Only one thread can own the mutex during program startup, so it is given a unique name.
- If a thread holds a resource, it must lock the mutex from other threads so that other threads cannot access it concurrently. The thread unlocks the mutex after releasing the resource.
- Multithreading comes into play when two threads work on the same data simultaneously. As a synchronization tool, it acts as a lock.
- If a mutex is available, a thread can acquire it; otherwise, it is set to sleep.

It is a software-based solution for critical section problems. However, modern computer architecture may not be compatible with it. Despite this, it is a valuable description of solving the critical-section problem. It illustrates some of the complexities involved when designing software that addresses mutual exclusion, progress, and the bounded waiting period of requirements.

Peterson's solution is limited to two processes running alternatively between critical sections. We will call these processes Pi and Pj.
Peterson's solution needs two data items to be shared between the two processes:

1. int turn: It indicates its turn to enter its critical section.
2. boolean flag[2]: It indicates if a process is ready to enter its critical section. It gives results as true or false.

Algorithm for Peterson's solution

Structure of Process Pi.

```
do {
  // Critical Section
```

```
   flag[i] = true; // It means process pi is ready to enter its critical section
   turn = j; // It means thatif pj wants to enter than allow it to enter and pi will wait

   // Condition to check if the flag of pj is true and turn is equal to pj, this will only break when
one of the conditions gets false.
   while (flag[j] && turn == [j]);

   // Remainder Section
   // It  sets the flag of pi to false because it has completed its critical section.
   flag[i] = false;
} while (true);
```

Similarly, Structure of Process Pj will be.

```
do
{
   // Critical Section
   // It means process pi is ready to enter its critical section
   flag[j] = true;

   turn = i; // It means thatif pi wants to enter than allow it to enter and pj will wait

   // Condition to check if the flag of pi is true and turn is equal to pi, this will only break when
one of the conditions gets false.
   while (flag[i] && turn == [i]);

   // Remainder Section
   //it  sets the flag of pj to false because it has completed its critical section.
   flag[j] = false;
} while (TRUE);
```

Implementation of Peterson's Algorithm

Steps for implementing Peterson's Algorithm are:

1. We are creating a class to implement Peterson's algorithm.
2. Inside the class, declare to variables flag and turn.
3. After that, implement two functions, one for locking the flag value as 1 and the other for unlocking the flag value.
4. In the lock function, change the thread's priority, wait for the thread to get executed, and change the flag value.
5. In the unlock function, mark the thread that no longer has to be executed in the critical section.
6. Now create a function that will reserve the value for which each thread will try lock-in Peterson's algorithm, increment an integer value entered by the user, and then unlock the thread.
7. The above process will run for 2 hundred million times because it is very time-independent and may take a few iterations to seal this problem.
8. In the main function, we have initialized an integer value to 0, created a Peterson object p, created two threads, and waited for them to join and print the final value.

The process of ensuring mutual exclusion can be performed by two processes simultaneously. Both processes create and initialize shared variables before starting. Neither of the processes is currently interested in the critical section, so flags [0] and [1] are set to FALSE. In this instance, the turn is set either to 0 or 1 randomly (or it can always be set to 0).

Disadvantage

- Peterson's solution works for two processes, but this solution is best scheme in user mode for critical section.
- This solution is also a busy waiting solution so CPU time is wasted. So that **"SPIN LOCK"** problem can come. And this problem can come in any of the busy waiting solution.

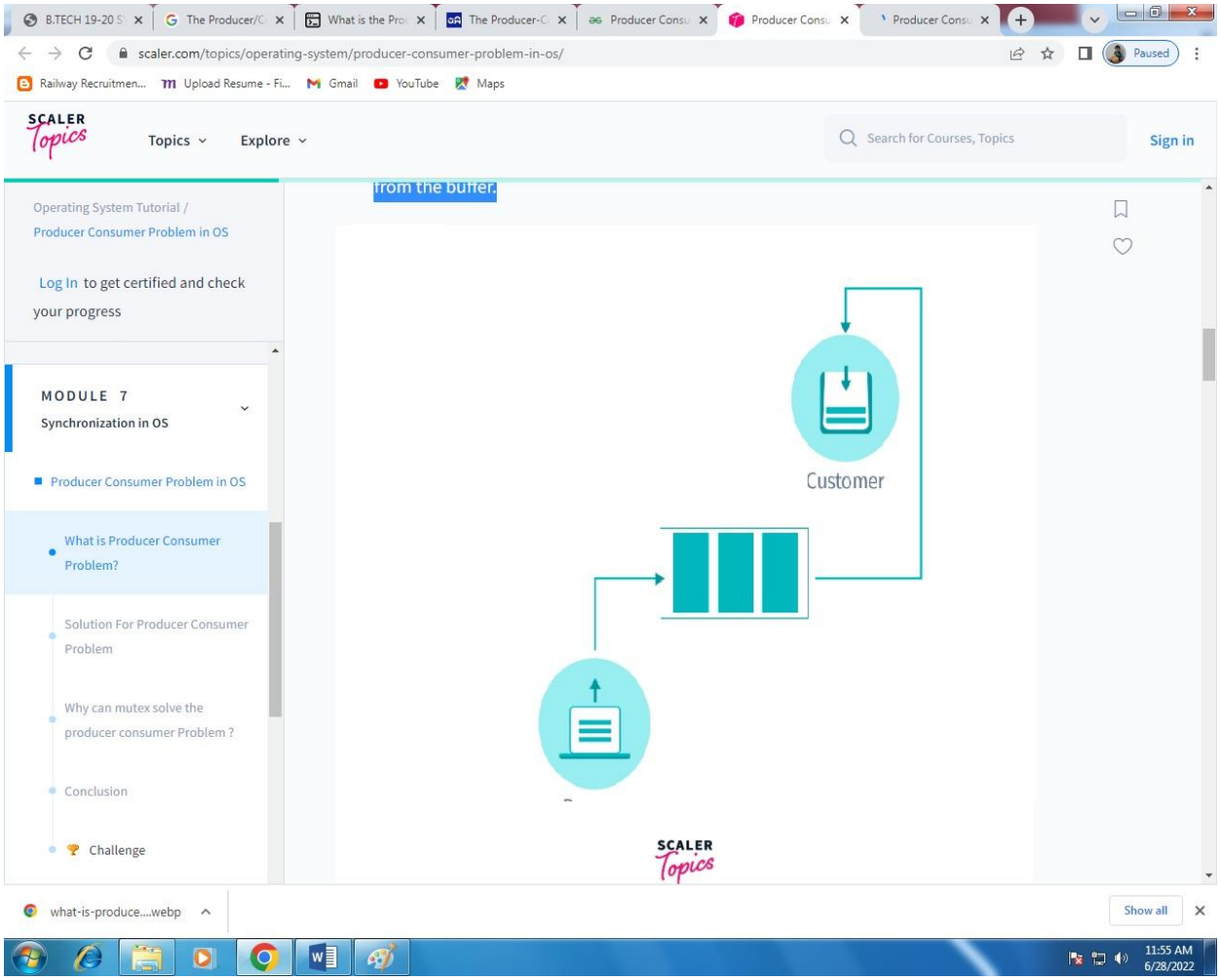
Producer-Consumer Problem

Producer-Consumer problem is a classical synchronization problem in the operating system. With the presence of more than one process and limited resources in the system the synchronization problem arises. If one resource is shared between more than one process at the same time then it can lead to data inconsistency. In the producer-consumer problem, the producer produces an item and the consumer consumes the item produced by the producer.

What is Producer Consumer Problem?

Before knowing what is Producer-Consumer Problem we have to know what are Producer and Consumer.

- In operating System **Producer is a process which is able to produce data/item**.
- Consumer is a **Process that is able to consume the data/item produced by the Producer**.
- Both Producer and Consumer share a common memory buffer. This buffer is a space of a certain size in the memory of the system which is used for storage. The producer produces the data into the buffer and the consumer consumes the data from the buffer.



So, what are the Producer-Consumer Problems?

1. Producer Process should not produce any data when the shared buffer is full.
2. Consumer Process should not consume any data when the shared buffer is empty.
3. The access to the shared buffer should be mutually exclusive i.e at a time only one process should be able to access the shared buffer and make changes to it.

Solution For Producer Consumer Problem

To solve the Producer-Consumer problem three **semaphores variable** are used :

Semaphores are variables used to indicate the number of resources available in the system at a particular time. semaphore variables are used to achieve `Process Synchronization.

Full

The full variable is used to track the space filled in the buffer by the Producer process. It is initialized to 0 initially as initially no space is filled by the Producer process.

Empty

The Empty variable is used to track the empty space in the buffer. The Empty variable is initially initialized to the **BUFFER-SIZE** as initially, the whole buffer is empty.

Mutex

Mutex is used to achieve mutual exclusion. mutex ensures that at any particular time only the producer or the consumer is accessing the buffer.

Mutex - mutex is a binary semaphore variable that has a value of 0 or 1.

We will use the Signal() and wait() operation in the above-mentioned semaphores to arrive at a solution to the Producer-Consumer problem.

Signal() - The signal function increases the semaphore value by 1.

Wait() - The wait operation decreases the semaphore value by 1.

Let's understand the above Producer process code :

- **wait(Empty)** - Before producing items, the producer process checks for the empty space in the buffer. If the buffer is full producer process waits for the consumer process to consume items from the buffer. so, the producer process executes wait(Empty) before producing any item.
- **wait(mutex)** - Only one process can access the buffer at a time. So, once the producer process enters into the critical section of the code it decreases the value of mutex by executing wait(mutex) so that no other process can access the buffer at the same time.
- **add()** - This method adds the item to the buffer produced by the Producer process. once the Producer process reaches add function in the code, it is guaranteed that no other process will be able to access the shared buffer concurrently which helps in data consistency.
- **signal(mutex)** - Now, once the Producer process added the item into the buffer it increases the mutex value by 1 so that other processes which were in a busy-waiting state can access the critical section.
- **signal(Full)** - when the producer process adds an item into the buffer spaces is filled by one item so it increases the Full semaphore so that it indicates the filled spaces in the buffer correctly.

Let's understand the above Consumer process code :

- **wait(Full)** - Before the consumer process starts consuming any item from the buffer it checks if the buffer is empty or has some item in it. So, the consumer process creates one more empty space in the buffer and this is indicated by the full variable. The value of the full variable decreases by one when the wait(Full) is executed. If the Full variable is already zero i.e the buffer is empty then the consumer process cannot consume any item from the buffer and it goes in the busy-waiting state.
- **wait(mutex)** - It does the same as explained in the producer process. It decreases the mutex by 1 and restricts another process to enter the critical section until the consumer process increases the value of mutex by 1.
- **consume()** - This function consumes an item from the buffer. when code reaches the consuming () function it will not allow any other process to access the critical section which maintains the data consistency.
- **signal(mutex)** - After consuming the item it increases the mutex value by 1 so that other processes which are in a busy-waiting state can access the critical section now.
- **signal(Empty)** - when a consumer process consumes an item it increases the value of the Empty variable indicating that the empty space in the buffer is increased by 1.

Semaphore?

Semaphore is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread. It uses two atomic operations, 1) Wait, and 2) Signal for the process synchronization.

Characteristic of Semaphore

Here, are characteristic of a semaphore:

- It is a mechanism that can be used to provide synchronization of tasks.
- It is a low-level synchronization mechanism.
- Semaphore will always hold a non-negative integer value.
- Semaphore can be implemented using test operations and interrupts, which should be executed using file descriptors.

Types of Semaphores

The two common kinds of semaphores are

- Counting semaphores
- Binary semaphores.

Counting Semaphores

This type of Semaphore uses a count that helps task to be acquired or released numerous times. If the initial count = 0, the counting semaphore should be created in the unavailable state.

Binary Semaphores

The binary semaphores are quite similar to counting semaphores, but their value is restricted to 0 and 1. In this type of semaphore, the wait operation works only if semaphore = 1, and the signal operation succeeds when semaphore= 0. It is easy to implement than counting semaphores.

Example of Semaphore

The below-given program is a step by step implementation, which involves usage and declaration of semaphore.

```
Shared var mutex: semaphore = 1;
Process i
    begin
    .
    .
    P(mutex);
    execute CS;
    V(mutex);
    .
    .
    End;
```

Wait and Signal Operations in Semaphores

Both of these operations are used to implement process synchronization. The goal of this semaphore operation is to get mutual exclusion.

Wait for Operation

This type of semaphore operation helps you to control the entry of a task into the critical section. However, If the value of wait is positive, then the value of the wait argument X is decremented. In the case of negative or zero value, no operation is executed. It is also called P(S) operation.

After the semaphore value is decreased, which becomes negative, the command is held up until the required conditions are satisfied.

```
Copy CodeP(S)
```

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

Signal operation

This type of Semaphore operation is used to control the exit of a task from a critical section. It helps to increase the value of the argument by 1, which is denoted as V(S).

```
Copy CodeP(S)
{
    while (S>=0);
    S++;
}
```

Counting Semaphore vs. Binary Semaphore

Here, are some major differences between counting and binary semaphore:

Counting Semaphore	Binary Semaphore
No mutual exclusion	Mutual exclusion
Any integer value	Value only 0 and 1
More than one slot	Only one slot
Provide a set of Processes	It has a mutual exclusion mechanism.

Difference between Semaphore vs. Mutex

Parameters	Semaphore	Mutex
Mechanism	It is a type of signaling mechanism.	It is a locking mechanism.
Data Type	Semaphore is an integer variable.	Mutex is just an object.
Modification	The wait and signal operations can modify a semaphore.	It is modified only by the process that may request or release a resource.
Resource	If no resource is free, then the	If it is locked, the process has

management	process requires a resource that should execute wait operation. It should wait until the count of the semaphore is greater than 0.	to wait. The process should be kept in a queue. This needs to be accessed only when the mutex is unlocked.
Thread	You can have multiple program threads.	You can have multiple program threads in mutex but not simultaneously.
Ownership	Value can be changed by any process releasing or obtaining the resource.	Object lock is released only by the process, which has obtained the lock on it.
Types	Types of Semaphore are counting semaphore and binary semaphore and	Mutex has no subtypes.
Operation	Semaphore value is modified using wait () and signal () operation.	Mutex object is locked or unlocked.
Resources Occupancy	It is occupied if all resources are being used and the process requesting for resource performs wait () operation and blocks itself until semaphore count becomes >1.	In case if the object is already locked, the process requesting resources waits and is queued by the system before lock is released.

Event Counters

Found on many Data Acquisition devices, Event Counters count the number of times a digital signal changes state. A simple example of a digital signal is where one state is below 0.8 volts (Low) and the other is above 2.0 volts (High). When the state changes from Low to High the Event Counter adds one (+1) to its count. Event Counter inputs generally have a high input count rate some as high as 20MHz or more.

How do they work?

The Event Counter has a clock input and a gating mechanism. The gating mechanism determine when and when not to count – like an on/off switch. Depending on the mode of operation, the clock source can be the input to the counter or an on device frequency source (clock).

The Event Counter counts transitions of the clock source and the gating mechanism is simply an enable switch telling it to either count or ignore the signal. Upon initialization, the count is set to zero counts and increments plus one on each clock transition. In the simplest form it counts and can be reset to zero. Some Event Counters allow pre setting values with countdown functions. Like the internal clock, this feature is device dependant.

Monitors

Monitors in Operating System

Monitors are used for process synchronization. With the help of programming languages, we can use a monitor to achieve mutual exclusion among the processes. **Example of monitors: *Java Synchronized methods such as Java offers notify() and wait() constructs.***

In other words, monitors are defined as the construct of programming language, which helps in controlling shared data access.

The Monitor is a module or package which encapsulates shared data structure, procedures, and the synchronization between the concurrent procedure invocations.

Characteristics of Monitors.

1. Inside the monitors, we can only execute one process at a time.
2. Monitors are the group of procedures, and condition variables that are merged together in a special type of module.

3. If the process is running outside the monitor, then it cannot access the monitor's internal variable. But a process can call the procedures of the monitor.
4. Monitors offer high-level of synchronization
5. Monitors were derived to simplify the complexity of synchronization problems.
6. There is only one process that can be active at a time inside the monitor.

Components of Monitor

There are four main components of the monitor:

1. Initialization
2. Private data
3. Monitor procedure
4. Monitor entry queue

Initialization: - Initialization comprises the code, and when the monitors are created, we use this code exactly once.
Private Data: - Private data is another component of the monitor. It comprises all the private data, and the private data contains private procedures that can only be used within the monitor. So, outside the monitor, private data is not visible.
Monitor Procedure: - Monitors Procedures are those procedures that can be called from outside the monitor.
Monitor Entry Queue: - Monitor entry queue is another essential component of the monitor that includes all the threads, which are called procedures.
Syntax of monitor

Condition Variables
There are two types of operations that we can perform on the condition variables of the monitor:

1. Wait
2. Signal

```
Suppose there are two condition variables

condition a, b // Declaring variable
```

Wait Operation
a.wait(): - The process that performs wait operation on the condition variables are suspended and locate the suspended process in a block queue of that condition variable.
Signal Operation
a.signal() : - If a signal operation is performed by the process on the condition variable, then a chance is provided to one of the blocked processes.

Advantages of Monitor

It makes the parallel programming easy, and if monitors are used, then there is less error-prone as compared to the semaphore.
Difference between Monitors and Semaphore

Monitors	Semaphore
We can use condition variables only in the monitors.	In semaphore, we can use condition variables anywhere in the program, but we cannot use conditions variables in a semaphore.

In monitors, wait always block the caller.	In semaphore, wait does not always block the caller.
The monitors are comprised of the shared variables and the procedures which operate the shared variable.	The semaphore S value means the number of shared resources that are present in the system.
Condition variables are present in the monitor.	Condition variables are not present in the semaphore.

Advantages and Disadvantages of Monitor

Various advantages and disadvantages of the monitor are as follows:

Advantages

1. Mutual exclusion is automatic in monitors.
2. Monitors are less difficult to implement than semaphores.
3. Monitors may overcome the timing errors that occur when semaphores are used.
4. Monitors are a collection of procedures and condition variables that are combined in a special type of module.

Disadvantages

1. Monitors must be implemented into the programming language.
2. The compiler should generate code for them.
3. It gives the compiler the additional burden of knowing what operating system features is available for controlling access to crucial sections in concurrent processes.

Head-to-head comparison between the Semaphore and Monitor

Various head-to-head comparisons between the semaphore and monitor are as follows:

Features	Semaphore	Monitor
Definition	A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS.	It is a synchronization process that enables threads to have mutual exclusion and the wait() for a given condition to become true.

Syntax	// Wait Operation
	wait(Semaphore S) {
	while (S<=0);
	S--;
	}
	// Signal Operation
	signal(Semaphore S) {
	S++;
	}
		monitor {
		//shared variable declarations
		data variables;
		Procedure P1() { ... }
		Procedure P2() { ... }
		.
		.
		.
		Procedure Pn() { ... }
		}

Basic	Integer variable	Abstract data type
Access	When a process uses shared resources, it calls the wait() method on S, and when it releases them, it uses the signal() method on S.	When a process uses shared resources in the monitor, it has to access them via procedures.
Action	The semaphore's value shows the number of shared resources available in the system.	The Monitor type includes shared variables as well as a set of procedures that operate on them.
Condition Variable	No condition variables.	It has condition variables.

Reader–Writer Problem :-

In an Operating System, we deal with various processes and these processes may use files that are present in the system. Basically, we perform two operations on a file i.e. read and write. All these processes can perform these two operations. But the problem that arises here is that:

- If a process is writing something on a file and another process also starts writing on the same file at the same time, then the system will go into the inconsistent state. Only one process should be allowed to change the value of the data present in the file at a particular instant of time.
- Another problem is that if a process is reading the file and another process is writing on the same file at the same time, then this may lead to dirty-read because the process writing on the file will change the value of the file, but the process reading that file will read the old value present in the file. So, this should be avoided.

The solution

We can solve the above two problems by using the semaphore variable(learn more about semaphore from here). The following is the proposed solution:

- If a process is performing some write operation, then no other process should be allowed to perform the read or the write operation i.e. no other process should be allowed to enter into the critical section(learn more about critical section from here).
- If a process is performing some read operation only, then another process that is demanding for reading operation should be allowed to read the file and get into the critical section

because the read operation doesn't change anything in the file. So, more than one reads are allowed. But if a process is reading a file and another process is demanding for the write operation, then it should not be allowed.

So, we will use the above two concepts and solve the reader-writer problem with the help of semaphore variables. The following semaphore variables will be used in our solution:

- **Semaphore "writer":** This semaphore is used to achieve the mutual exclusion property. It is used by the process that is writing in the file and it ensures that no other process should enter the critical section at that instant of time. Initially, it will be set to "1".
- **Semaphore "mutex":** This semaphore is used to achieve mutual exclusion during changing the variable that is storing the count of the processes that are reading a particular file. Initially, it will be set to "1".

Apart from these two semaphore variables, we have one variable *readerCount* that will have the count of the processes that are reading a particular file. The *readerCount* variable is initially initialized to 0.

We will also use two function *wait()* and *signal()*. The wait() function is used to reduce the value of a semaphore variable by one and the *signal()* function is used to increase the value of a semaphore variable by one.

The following is the pseudo-code for the process that is writing something in the file:

```
wait(writer)
  ...
  write operation
  ...
signal(writer)
```

The above code can be summarized as:

- The *wait(writer)* function is called so that it achieves the mutual exclusion. The wait() function will reduce the *writer* value to "0" and this will block other processes to enter into the critical section.
- The write operation will be carried and finally, the *signal(writer)* function will be called and the value of the *writer* will be again set to "1" and now other processes will be allowed to enter into the critical section.

This is how we can deal with the process of doing the write operation. Now, let's look at the reader problem.

The following is the pseudo-code for the process that is reading something from the file:

```
wait(mutex)                 -----
readerCount++                 |
if(readerCount == 1)          |--- changing the readerCount
    wait(writer)              |
signal(mutex)               -----

    ...
    read operation

wait(mutex)                 -----
readerCount--                 |
if(readerCount == 0)          |--- changing the readerCount
    signal(writer)            |
signal(mutex)               -----
```

The above code can be summarized as:

- We are using the *mutex* variable to change something in the *readerCount* variable. This is done because if some process is changing something in the *readerCount* variable, then no

other process should be allowed to use that variable. So, to achieve mutual exclusion, we are using the mutex variable.

- Initially, we are calling the *wait(mutex)* function and this will reduce the value of the mutex by one. After that, the *readerCount* value will be increased by one.
- If the *readerCount* variable is equal to "1" i.e. the reader process is the first process, in this case, no other process demanding for write operation will be allowed to enter into the critical section. So, the *wait(writer)* will be called and the value of the writer variable will be decreased to "0" and no other process demanding for write operation will be allowed.
- After changing the *readerCount* variable, the value of the *mutex* variable will be increased by one, so that other processes should be allowed to change the value of the *readerCount* value.
- The read operation by various processes will be continued and after that when the read operation is done, then again we have to change the count the value of the *readerCount* and decrease the value by one.
- If the *readerCount* becomes "0", then we have to increase the value of the *writer* variable by one by calling the *signal(writer)* function. This is done because if the *readerCount* is "0" then other writer processes should be allowed to enter into the critical section to write the data in the file.

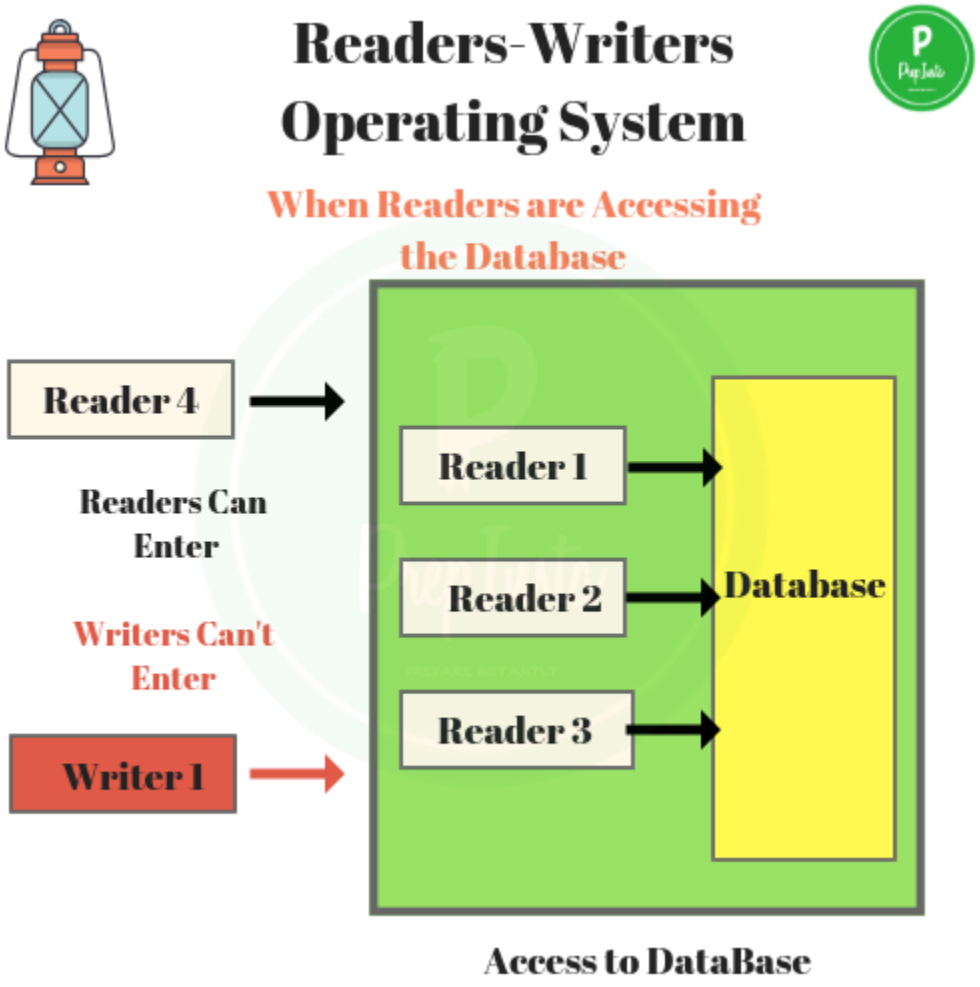
Highlights

- If writer W1 has begun writing process then
  - No additional writer can perform write function
  - No reader is allowed to read
- If 1 or more readers are reading then
  - Other readers may read as well
  - No writer may perform write function until all readers have finished reading

Explanations

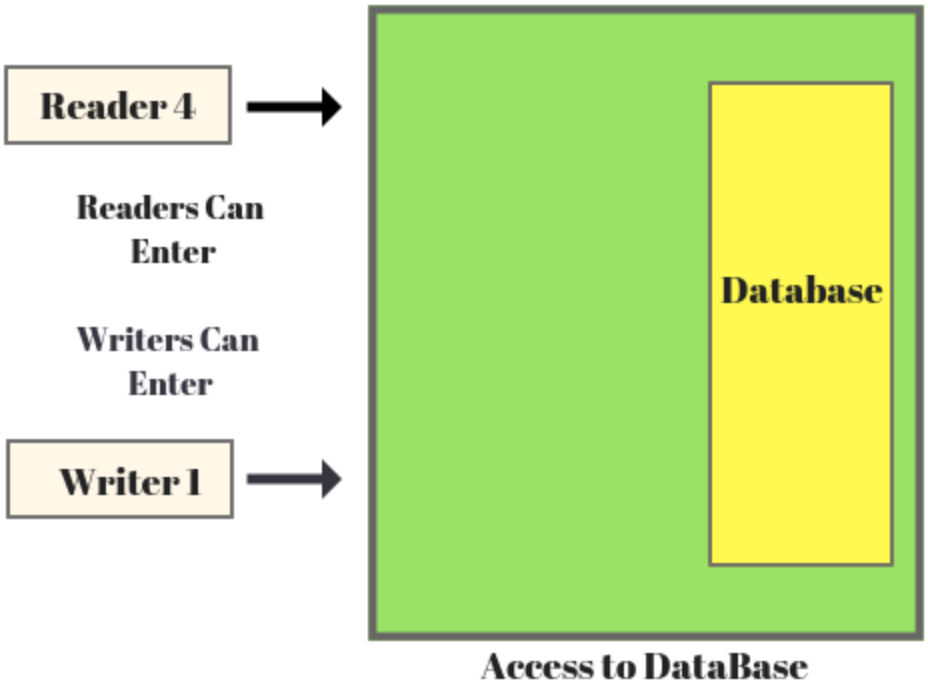
In simple terms understand this as unlimited number of readers can read simultaneously. Only one writer can write at a time.

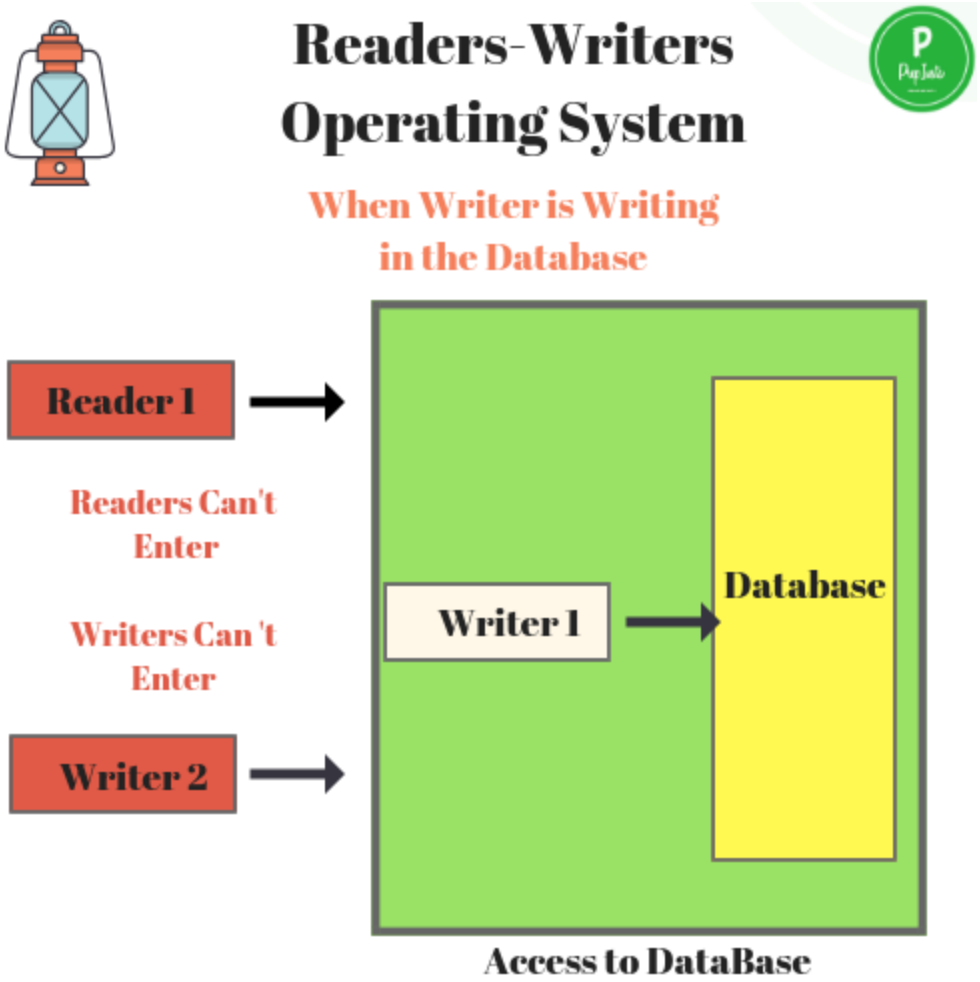
When a writer is writing no other writer can write to the file. A writer can not write when there are one or more than one readers reading. That is writer can only write when there is no readers or no writers accessing the resource.



Readers-Writers Operating System

When No one is Accessing the Database





Solution

Variables used –

1. Mutex – mutex (used for mutual exclusion, when readcount is changed)
    1. initialised as 1
2. Semaphore – wrt (used by both readers and writers)
    1. initialised as 1
3. readers_count – Counter of number of people reading the file
    1. initialised as 0

Functions –

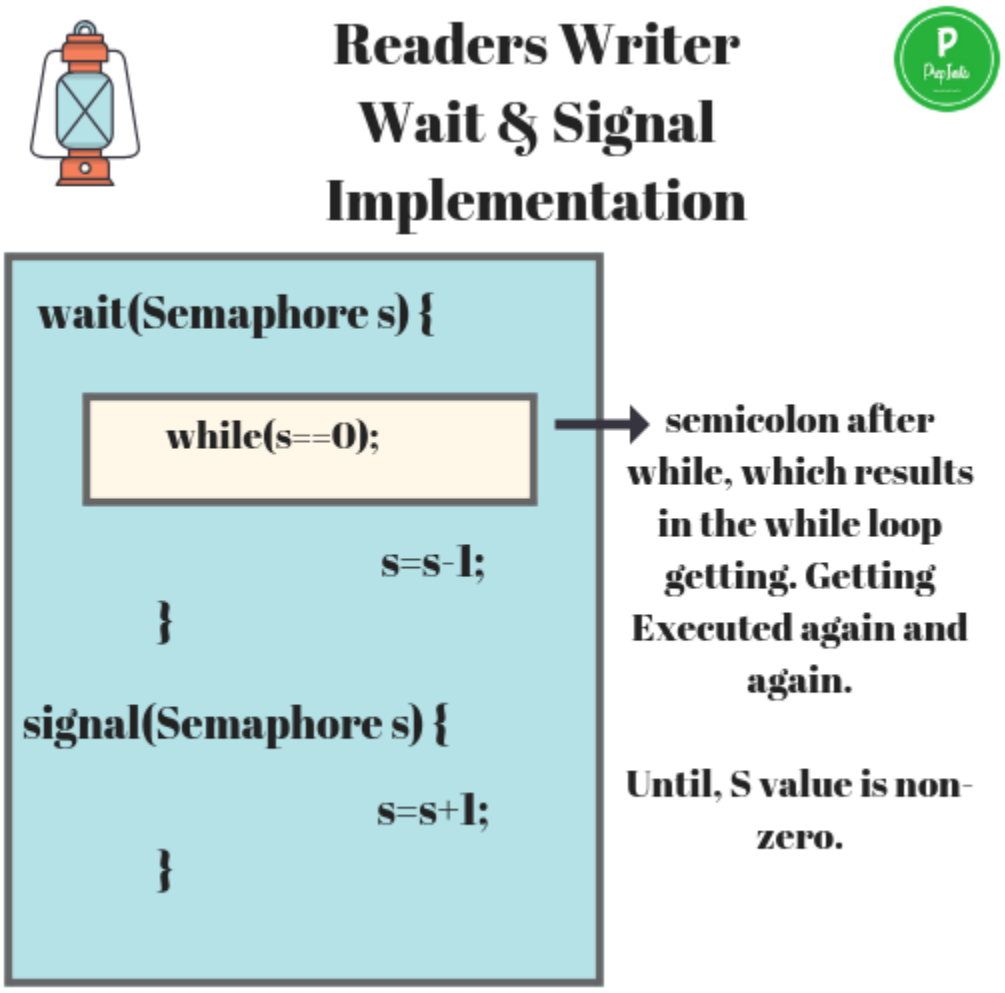
There are two functions –

1. wait() – performs as –, which basically decrements value of semaphore

2. signal() – performs as ++. which basically increments value of semaphore

How does Wait and Signal work

To understand how readers and writers work we first need to understand that how atomic operations wait and signal work



Dinning Philosopher Problem:-

Overview

Dining Philosophers Problem in OS is a classical synchronization problem in the operating system. With the presence of more than one process and limited resources in the system the synchronization problem arises. If one resource is shared between more than one process at the same time then it can lead to data inconsistency.

Scope

- The article discusses the need of synchronization in operating system, the use of semaphores and critical section.
- We also discuss the Dining Philosophers in OS Problem and approach to solve it, algorithm and code explanation.
- We also consider drawbacks of the solution of Dining Philosophers Problem, and how it can be improved.

Dining Philosophers Problem in OS

Consider two processes P1 and P2 executing simultaneously, while trying to access the same resource R1, this raises the question who will get the resource and when? This problem is solved using process synchronisation.

The act of synchronising process execution such that no two processes have access to the same associated data and resources is referred as process synchronisation in operating systems.

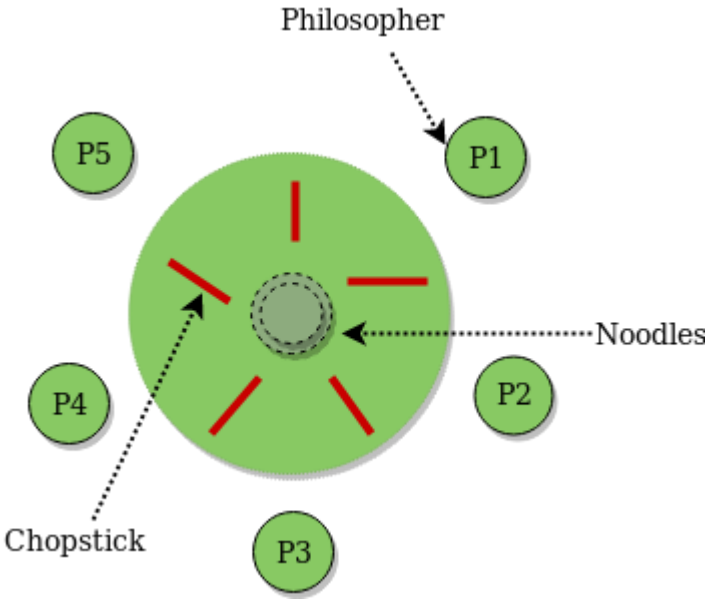
It's particularly critical in a multi-process system where multiple processes are executing at the same time and trying to access the very same shared resource or data.

This could lead to discrepancies in data sharing. As a result, modifications implemented by one process may or may not be reflected when the other processes access the same shared data. The processes must be synchronised with one another to avoid data inconsistency.

And the Dining Philosophers Problem is a typical example of limitations in process synchronisation in systems with multiple processes and limited resource. According to the Dining Philosopher Problem, assume there are K philosophers seated around a circular table, each with one chopstick between them. This means, that a philosopher can eat only if he/she can pick up both the chopsticks next to him/her. One of the adjacent followers may take up one of the chopsticks, but not both.

For example, let's consider P0, P1, P2, P3, and P4 as the philosophers or processes and C0, C1, C2, C3, and C4 as the 5 chopsticks or resources between each

philosopher. Now if P0 wants to eat, both resources/chopstick C0 and C1 must be free, which would leave in P1 and P4 void of the resource and the process wouldn't be executed, which indicates there are limited resources(C0,C1..) for multiple processes(P0, P1..), and this problem is known as the Dining Philosopher Problem.



The Solution of the Dining Philosophers Problem

The solution to the process synchronization problem is Semaphores, A semaphore is an integer used in solving critical sections.

The critical section is a segment of the program that allows you to access the shared variables or resources. In a critical section, an atomic action (independently running process) is needed, which means that only single process can run in that section at a time.

Semaphore has 2 atomic operations: wait() and signal(). If the value of its input S is positive, the wait() operation decrements, it is used to acquire resource while entry. No operation is done if S is negative or zero. The value of the signal() operation's parameter S is increased, it used to release the reource once critical section is executed at exit.

Here's a simple explanation of the solution:

```
void Philosopher
{
  while(1)
  {
    //  Section where the philosopher is using chopstick
    wait(use_resource[x]);
    wait(use_resource[(x + 1) % 5]);
    //  Section where the philosopher is thinking
    signal(free_resource[x]);
    signal(free_resource[(x + 1) % 5]);
  }
}
```

Explanation:

- The wait() operation is implemented when the philosopher is using the resources while the others are thinking. Here, the threads use_resource[x] and use_resource[(x + 1) % 5] are being executed.
- After using the resource, the signal() operation signifies the philosopher using no resources and thinking. Here, the threads free_resource[x] and free_resource[(x + 1) % 5] are being executed.

To model the Dining Philosophers Problem in a C program we will create an array of philosophers (processes) and an array of chopsticks (resources). We will initialise the array of chopsticks with locks to ensure mutual exclusion is satisfied inside the critical section.

We will run the array of philosophers in parallel to execute the critical section (dine ()), the critical section consists of thinking, acquiring two chopsticks, eating and then releasing the chopsticks.