

Inter Process Communication (IPC)

Processes need to communicate with each other in many situations, for example, to count occurrences of a word in text file, output of grep command needs to be given to wc command, something like `grep -o -i <word> <file> | wc -l`. **Inter-Process Communication or IPC** is a mechanism that allows processes to communicate. It helps processes synchronize their activities, share information, and avoid conflicts while accessing shared resources.

Types of Process

Let us first talk about types of processes.

- **Independent process:** An independent process is not affected by the execution of other processes. Independent processes are processes that do not share any data or resources with other processes. No inter-process communication required here.
- **Co-operating process:** Interact with each other and share data or resources. A co-operating process can be affected by other executing processes. Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of cooperation between them.

Inter Process Communication

Inter process communication (IPC) allows different programs or processes running on a computer to share information with each other. IPC allows processes to communicate by using different techniques like sharing memory, sending messages, or using files. It ensures that processes can work together without interfering with each other. [Cooperating processes](#) require an Inter Process Communication (IPC) mechanism that will allow them to exchange data and information.

The two fundamental models of Inter Process Communication are:

- Shared Memory
- Message Passing

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication

using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

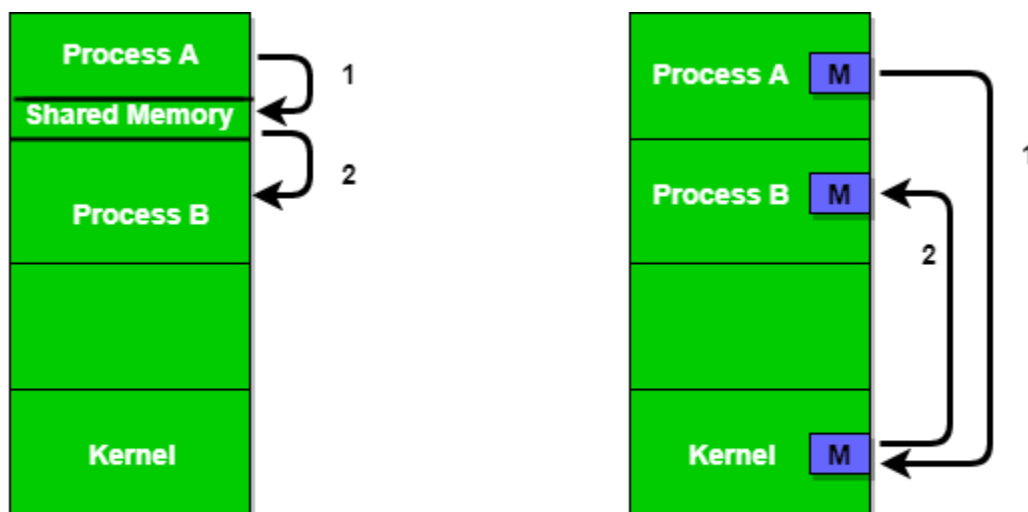


Figure 1 - Shared Memory and Message Passing

Let's discuss an example of communication between processes using the shared memory method.

Methods in Inter process Communication

Inter-Process Communication refers to the techniques and methods that allow processes to exchange data and coordinate their activities. Since processes typically operate independently in a multitasking environment, IPC is essential for them to communicate effectively without interfering with one another. There are several methods of IPC, each designed to suit different scenarios and requirements. These methods include shared memory, message passing, semaphores, and signals, etc.

To read more refer – [methods of Inter Process Communication](#)

Role of Synchronization in IPC

In IPC, synchronization is essential for controlling access to shared resources and guaranteeing that processes do not conflict with one another. Data consistency is ensured and problems like race situations are avoided with proper synchronization.

Advantages of IPC

- Enables processes to communicate with each other and share resources, leading to increased efficiency and flexibility.
- Facilitates coordination between multiple processes, leading to better overall system performance.
- Allows for the creation of distributed systems that can span multiple computers or networks.
- Can be used to implement various [synchronization](#) and communication protocols, such as semaphores, pipes, and sockets.

Disadvantages of IPC

- Increases system complexity, making it harder to design, implement, and debug.
- Can introduce security vulnerabilities, as processes may be able to access or modify data belonging to other processes.
- Requires careful management of system resources, such as memory and [CPU](#) time, to ensure that IPC operations do not degrade overall system performance.
Can lead to data inconsistencies if multiple processes try to access or modify the same data at the same time.
- Overall, the advantages of IPC outweigh the disadvantages, as it is a necessary mechanism for modern operating systems and enables processes to work together and share resources in a flexible and efficient manner. However, care must be taken to design and implement IPC systems carefully, in order to avoid potential security vulnerabilities and performance issues.

Introduction of Process Synchronization

Process Synchronization is used in a computer system to ensure that multiple processes or threads can run concurrently without interfering with each other.

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. To achieve this, various synchronization techniques such as semaphores, monitors, and critical sections are used.

In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems. Process synchronization is an important aspect of modern operating systems, and it plays a crucial role in ensuring the correct and efficient functioning of multi-process systems.

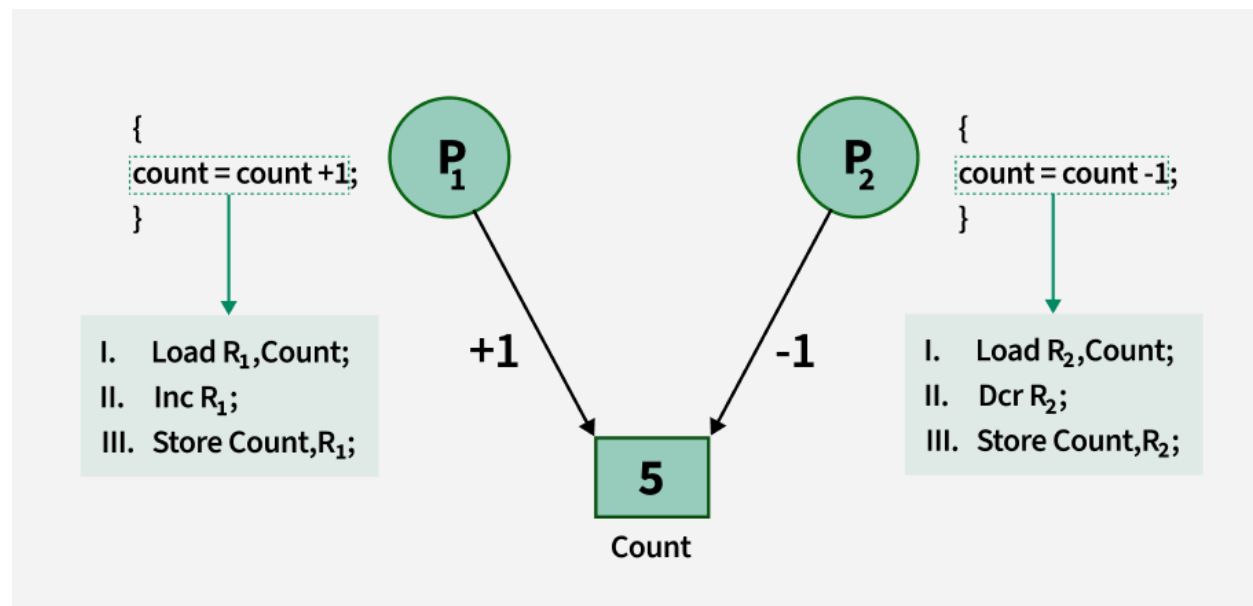
On the basis of synchronization, [processes](#) are categorized as one of the following two types:

- **Independent Process:** The execution of one process does not affect the execution of other processes.
- **Cooperative Process:** A process that can affect or be affected by other processes executing in the system.

Process synchronization problem arises in the case of Cooperative processes also because resources are shared in Cooperative processes.

Process Synchronization

Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner. It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system.



Lack of Synchronization in [Inter Process Communication](#) Environment leads to following problems:

1. **Inconsistency:** When two or more processes access shared data at the same time without proper synchronization. This can lead to conflicting changes, where one

process's update is overwritten by another, causing the data to become unreliable and incorrect.

2. **Loss of Data:** Loss of data occurs when multiple processes try to write or modify the same shared resource without coordination. If one process overwrites the data before another process finishes, important information can be lost, leading to incomplete or corrupted data.
3. **Deadlock:** Lack of Synchronization leads to [Deadlock](#) which means that two or more processes get stuck, each waiting for the other to release a resource. Because none of the processes can continue, the system becomes unresponsive and none of the processes can complete their tasks.

Types of Process Synchronization

The two primary type of process Synchronization in an Operating System are:

1. **Competitive:** Two or more processes are said to be in Competitive Synchronization if and only if they compete for the accessibility of a shared resource.
Lack of Synchronization among Competing process may lead to either Inconsistency or Data loss.
2. **Cooperative:** Two or more processes are said to be in Cooperative Synchronization if and only if they get affected by each other i.e. execution of one process affects the other process.
Lack of Synchronization among Cooperating process may lead to Deadlock.

Example:

Let consider a Linux code:

```
>>ps/grep "chrome"/wc
```

- ps command produces list of processes running in linux.
- grep command find/count the lines form the output of the ps command.
- wc command counts how many words are in the output.

Therefore, three processes are created which are ps, grep and wc. grep takes input from ps and wc takes input from grep.

From this example, we can understand the concept of cooperative processes, where some processes produce and others consume, and thus work together. This type of problem must be handled by the operating system, as it is the manager.

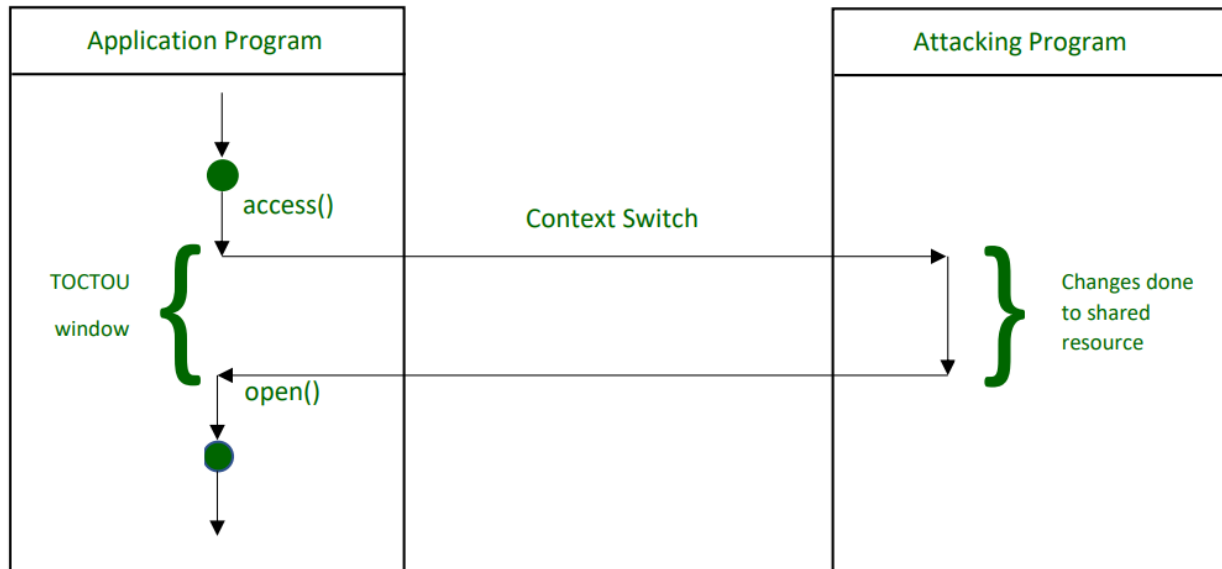
Conditions That Require Process Synchronization

1. **Critical Section:** It is that part of the program where shared resources are accessed. Only one process can execute the critical section at a given point of time. If there are no shared resources, then no need of synchronization mechanisms.
2. **Race Condition:** It is a situation wherein processes are trying to access the critical section and the final result depends on the order in which they finish their update. Process Synchronization mechanism need to ensure that instructions are being executed in a required order only.
3. **Pre Emption:** Preemption is when the operating system stops a running process to give the CPU to another process. This allows the system to make sure that important tasks get enough CPU time. This is important as mainly issues arise when a process has not finished its job on shared resource and got preempted. The other process might end up reading an inconsistent value if process synchronization is not done.

What is Race Condition?

A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in a critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions.

This is a major security vulnerability [CWE-362], and by manipulating the timing of actions anomalous results might appear. This vulnerability arises during a TOCTOU (time-of-check, time-of-use) window.



Flow of File Access during its TOCTOU Window

Flow of File Access during its TOCTOU Window So, what if we lock the file during this TOCTOU window itself?

1. **General Misconception** – A trivial cure to this vulnerability could be locking the file itself during this check-and-use window because then no other process can use the file during the time window. Seems easy, then why isn't this practical? Why can't we use this approach to solve the race condition problem? The answer is simply that such a vulnerability could not be prevented by just locking the file.
2. **Problems while locking the file** – A file is locked out for other processes only if it is already in the open state. This process is called the check-and-open process and during this time it is impossible to lock a file. Any locks created can be ignored by the attacking or the malicious process. What happens is that the call to Open() does not block an attack on a locked file. When the file is available for a check-and-open process, the file is open to any access/ change. So it's impossible to lock a file at this point in time. This makes any kind of lock virtually non-existent to the malicious processes. Internally it is using the sleep_time which doubles at every attempt. More commonly this is referred to as a spinlock or the busy form of waiting. Also, there is always a possibility of the file getting locked indefinitely i.e. danger of getting stuck in a deadlock.
3. **What would happen even if we were somehow able to lock the file?** Let's try to lock the file and see what could be the possible drawbacks. The most common locking mechanism that is available is atomic file locking. It is done using a lock file to create a

unique file on the same filesystem. We make use of `link()` to make a link to the lock file for any kind of access to the file.

- If `link()` returns 0, the lock is successful.

The most common fix available is to store the PID of the application in the lock file, which is checked against the active PID at that time. Then again a flaw with this fix is that PID may have been reused.

4. **Actual Solutions** – A better solution is to rather than creating locks on the file as a whole, lock the parts of the file to different processes. **Example** – When a process wants to write into a file, it first asks the kernel to lock that file or a part of it. As long as the process keeps the lock, no other process can ask to lock the same part of the file. Hence you could see that the issue with concurrency is getting resolved like this. In the same way, a process asks for locking before reading the content of a file, which ensures no changes will be made as long as the lock is kept. Differentiating this different kind of locks is done by the system itself. The system has the capability to distinguish between the locks required for file reading and those required for file writing. This kind of locking system is achieved by the `flock()` system call. `Flock()` call can have different values :

- `LOCK_SH` (lock for reading)
- `LOCK_EX` (for writing)
- `LOCK_UN` (release of the lock)

Using these separate calls we can tell what kind of locks are necessary. A point to note here is that many processes can be benefited from a reading lock simultaneously since no one will attempt to change the file content. However, only one process can benefit from a lock for writing at a given time which is currently using it. Thus no other lock can be allowed at the same time, even for reading. This kind of carefully crafted system works well with applications that can ask the [kernel](#) to reserve their access (their lock) before reading or writing to an important system file. Hence this way of selectively locking the file is much practical than our initial approach. So, while you are trying to implement your file system for directories you could take advantage of this secure coding technique to prevent a potential CWE-362 (Race Condition Vulnerability).

Real-Time Examples of Race Conditions

Example 1 – Consider an ATM Withdrawal

Imagine Ram and his friend Sham both have access to the same bank account. They both try to withdraw Rs,500 at the same time from different [ATMs](#). The system checks the balance and sees

there's enough money for both withdrawals. Without proper synchronization, the system might allow both transactions to go through, even if the balance is only enough for one, leaving the account overdrawn.

Example 2 – Consider a Printer Queue

Imagine two people sending print jobs at the same time. If the printer isn't managed properly, the print jobs could get mixed up, with pages from one person's document being printed in the middle of another's.

Key Terms in a Race Condition

- **Critical Section:** A code part where the shared resources are accessed. It is critical as multiple processes enter this section at same time leading to data corruption and errors.
- **Synchronization:** It is the process of controlling how and when multiple processes or threads access the shared resources ensuring that only one can enter the [critical section](#) at same time.
- **Mutual Exclusion (Mutex):** A mutex is like a lock that ensures only one process can access a resource at a time. If a process holds a lock, others must wait their turn preventing race conditions.
- **Deadlock:** A situation where two or more processes are stuck waiting for each other's resources, causing a deadlock(standstill).

What is Mutual Exclusion?

Mutual Exclusion is a property of [process synchronization](#) that states that “**no two processes can exist in the critical section at any given point of time**”. The term was first coined by **Dijkstra**. Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition.

The need for mutual exclusion comes with concurrency. There are several kinds of concurrent execution:

- Interrupt handlers
- Interleaved, preemptively scheduled processes/threads
- Multiprocessor clusters, with shared memory
- Distributed systems

[Mutual exclusion](#) methods are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections.

The requirement of mutual exclusion is that when process P1 is accessing a shared resource R1, another process should not be able to access resource R1 until process P1 has finished its operation with resource R1.

Examples of such resources include files, I/O devices such as printers, and shared data structures.

Conditions Required for Mutual Exclusion

According to the following four criteria, mutual exclusion is applicable:

- When using shared resources, it is important to ensure mutual exclusion between various processes. There cannot be two processes running simultaneously in either of their critical sections.
- It is not advisable to make assumptions about the relative speeds of the unstable processes.
- For access to the critical section, a process that is outside of it must not obstruct another process.
- Its critical section must be accessible by multiple processes in a finite amount of time; multiple processes should never be kept waiting in an infinite loop.

Approaches To Implementing Mutual Exclusion

- **Software Method:** Leave the responsibility to the processes themselves. These methods are usually highly error-prone and carry high overheads.
- **Hardware Method:** Special-purpose [machine instructions](#) are used for accessing shared resources. This method is faster but cannot provide a complete solution. Hardware solutions cannot give guarantee the absence of [deadlock](#) and [starvation](#).
- **Programming Language Method:** Provide support through the operating system or through the programming language.

Requirements of Mutual Exclusion

- At any time, only one process is allowed to enter its critical section.
- The solution is implemented purely in software on a machine.

- A process remains inside its critical section for a bounded time only.
- No assumption can be made about the relative speeds of asynchronous concurrent processes.
- A process cannot prevent any other process from entering into a critical section.
- A process must not be indefinitely postponed from entering its critical section.

Critical Section in synchronization

A **critical section** is a part of a program where shared resources like memory or files are accessed by multiple processes or threads. To avoid issues like data inconsistency or race conditions, synchronization techniques ensure that only one process or thread uses the critical section at a time.

- The critical section contains shared variables or resources that need to be synchronized to maintain the consistency of data variables.
- In simple terms, a critical section is a group of instructions/statements or regions of code that need to be executed atomically, such as accessing a resource (file, input or output port, global data, etc.) In concurrent programming, if one process tries to change the value of shared data at the same time as another thread tries to read the value (i.e., data race across threads), the result is unpredictable. The access to such shared variables (shared memory, shared files, shared port, etc.) is to be synchronized.

Few programming languages have built-in support for synchronization. It is critical to understand the importance of race conditions while writing kernel-mode programming (a device driver, kernel thread, etc.) since the programmer can directly access and modify kernel data structures

Although there are some **properties that should be followed if any code in the critical section**

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
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.

Two general approaches are used to handle critical sections:

1. **Preemptive kernels:** A preemptive kernel allows a process to be preempted while it is running in kernel mode.
2. **Non preemptive kernels:** A non-preemptive kernel does not allow a process running in kernel mode to be preempted. A kernel-mode process will run until it exists in kernel mode, blocks, or voluntarily yields control of the CPU. A non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Critical Section Problem

The use of critical sections in a program can cause a number of issues, including:

- **Deadlock:** When two or more threads or processes wait for each other to release a critical section, it can result in a deadlock situation in which none of the threads or processes can move. Deadlocks can be difficult to detect and resolve, and they can have a significant impact on a program's performance and reliability.
- **Starvation:** When a thread or process is repeatedly prevented from entering a critical section, it can result in starvation, in which the thread or process is unable to progress. This can happen if the critical section is held for an unusually long period of time, or if a high-priority thread or process is always given priority when entering the critical section.
- **Overhead:** When using critical sections, threads or processes must acquire and release locks or semaphores, which can take time and resources. This may reduce the program's overall performance.

What is Semaphores?

A semaphore is a synchronization tool used in concurrent programming to manage access to shared resources. It is a [lock-based mechanism](#) designed to achieve process synchronization, built on top of basic locking techniques.

Semaphores use a counter to control access, allowing synchronization for multiple instances of a resource. Processes can attempt to access one instance, and if it is not available, they can try other instances. Unlike basic locks, which allow only one process to access one instance of a resource. Semaphores can handle more complex synchronization scenarios, involving multiple processes or threads. It help prevent problems like [race conditions](#) by controlling when and how processes access shared data.

The process of using Semaphores provides two operations:

- **wait (P):** The wait operation decrements the value of the semaphore
- **signal (V):** The signal operation increments the value of the semaphore.

When the value of the semaphore is zero, any process that performs a wait operation will be blocked until another process performs a signal operation.

When a process performs a wait operation on a semaphore, the operation checks whether the value of the semaphore is >0 . If so, it decrements the value of the semaphore and lets the process continue its execution; otherwise, it blocks the process on the semaphore. A signal operation on a semaphore activates a process blocked on the semaphore if any, or increments the value of the semaphore by 1. Due to these semantics, semaphores are also called counting semaphores. The initial value of a semaphore determines how many processes can get past the wait operation.

Semaphores are required for [process synchronization](#) to make sure that multiple processes can safely share resources without interfering with each other. They help control when a process can access a shared resource, preventing issues like **race** conditions.

Types of Semaphores

Semaphores are of two Types:

- **Binary Semaphore:** This is also known as a mutex lock, as they are locks that provide mutual exclusion. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes and a single resource.
- **Counting Semaphore:** Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Its value can range over an unrestricted domain.

To learn more refer: [Types of Semaphores](#)

Working of Semaphore

A semaphore is a simple yet powerful synchronization tool used to manage access to shared resources in a system with multiple processes. It works by maintaining a counter that controls access to a specific resource, ensuring that no more than the allowed number of processes access the resource at the same time.

There are two primary operations that a semaphore can perform:


1. **Wait (P operation):** This operation checks the semaphore's value. If the value is greater than 0, the process is allowed to continue, and the semaphore's value is decremented by

1. If the value is 0, the process is blocked (waits) until the semaphore value becomes greater than 0.
2. **Signal (V operation):** After a process is done using the shared resource, it performs the signal operation. This increments the semaphore's value by 1, potentially unblocking other waiting processes and allowing them to access the resource.

Now let us see how it does so. First, look at two operations that can be used to access and change the value of the semaphore variable.

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

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



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

A [critical section](#) is surrounded by both operations to implement process synchronization. The below image demonstrates the basic mechanism of how semaphores are used to control access to a critical section in a multi-process environment, ensuring that only one process can access the shared resource at a time

Process P

```
// Some code
P(s);
  // critical section
V(s);
  // remainder section
```

Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until $s > 0$, this can only happen when P1 finishes its critical section and calls V operation on semaphore s .

This way mutual exclusion is achieved. Look at the below image for details which is a Binary semaphore.

