Process Synchronization in OS

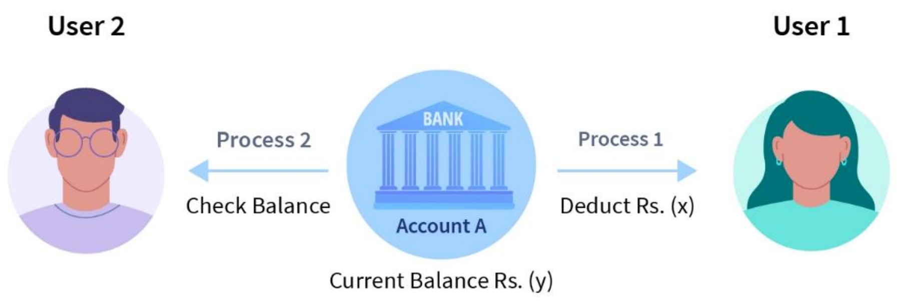
Overview

Processes Synchronization or Synchronization is the way by which processes that share the same memory space are managed in an operating system. It helps maintain the consistency of data by using variables or hardware so that only one process can make changes to the shared memory at a time. There are various solutions for the same such as semaphores, mutex locks, synchronization hardware, etc.

What is Process Synchronization in OS?

An operating system is a software that manages all applications on a device and basically helps in the smooth functioning of our computer. Because of this reason, the operating system has to perform many tasks, and sometimes simultaneously. This isn't usually a problem unless these simultaneously occurring processes use a common resource.

For example, consider a bank that stores the account balance of each customer in the same database. Now suppose you initially have x rupees in your account. Now, you take out some amount of money from your bank account, and at the same time, someone tries to look at the amount of money stored in your account. As you are taking out some money from your account, after the transaction, the total balance left will be lower than x. But, the transaction takes time, and hence the person reads x as your account balance which leads to inconsistent data. If in some way, we could make sure that only one process occurs at a time, we could ensure consistent data.

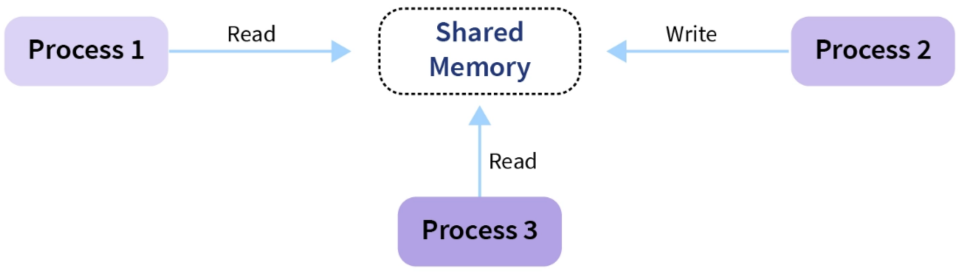


In the above image, if Process1 and Process2 happen at the same time, user 2 will get the wrong account balance as Y because of Process1 being transacted when the balance is X.

Inconsistency of data can occur when various processes share a common resource in a system which is why there is a need for process synchronization in the operating system.

How Process Synchronization in OS Works?

Let us take a look at why exactly we need Process Synchronization. For example, If a process1 is trying to read the data present in a memory location while another process2 is trying to change the data present at the same location, there is a high chance that the data read by the process1 will be incorrect.



Let us look at different elements/sections of a program:

- **Entry Section:** The entry Section decides the entry of a process.
- **Critical Section:** Critical section allows and makes sure that only one process is modifying the shared data.
- **Exit Section:** The entry of other processes in the shared data after the execution of one process is handled by the Exit section.
- **Remainder Section:** The remaining part of the code which is not categorized as above is contained in the Remainder section.

Race Condition

When more than one process is either running the same code or modifying the same memory or any shared data, there is a risk that the result or value of the shared data may be incorrect because all processes try to access and modify this shared resource. Thus, all the processes race to say that my result is correct. This condition is called the race condition. Since many processes use the same data, the results of the processes may depend on the order of their execution.

This is mostly a situation that can arise within the critical section. **<u>In the critical section, a race condition occurs when the end result of multiple thread executions varies depending on the sequence in which the threads execute.</u>**

But how to avoid this race condition? There is a simple solution:

- by treating the critical section as a section that can be accessed by only a single process at a time. This kind of section is called an atomic section.

What is the Critical Section Problem?

Why do we need to have a critical section? What problems occur if we remove it?

A part of code that can only be accessed by a single process at any moment is known as a critical section. This means that when a lot of programs want to access and change a single shared data, only one process will be allowed to change at any given moment. The other processes have to wait until the data is free to be used.

The wait() function mainly handles the entry to the critical section, while the signal() function handles the exit from the critical section. **If we remove the critical section, we cannot guarantee the consistency of the end outcome after all the processes finish executing simultaneously.**

We'll look at some solutions to Critical Section Problem but before we move on to that, let us take a look at what conditions are necessary for a solution to Critical Section Problem.

Requirements of Synchronization

The following three requirements must be met by a solution to the critical section problem:

- **Mutual exclusion:** If a process is running in the critical section, no other process should be allowed to run in that section at that time.
- **Progress:** If no process is still in the critical section and other processes are waiting outside the critical section to execute, then any one of the threads must be permitted to enter the critical section. The decision of which process will enter the critical section will be taken by only those processes that are not executing in the remaining section.
- **No starvation:** Starvation means a process keeps waiting forever to access the critical section but never gets a chance. No starvation is also known as Bounded Waiting.
  - A process should not wait forever to enter inside the critical section.
  - When a process submits a request to access its critical section, there should be a limit or bound, which is the number of other processes that are allowed to access the critical section before it.
  - After this bound is reached, this process should be allowed to access the critical section.

Solutions To the Critical Section Problem

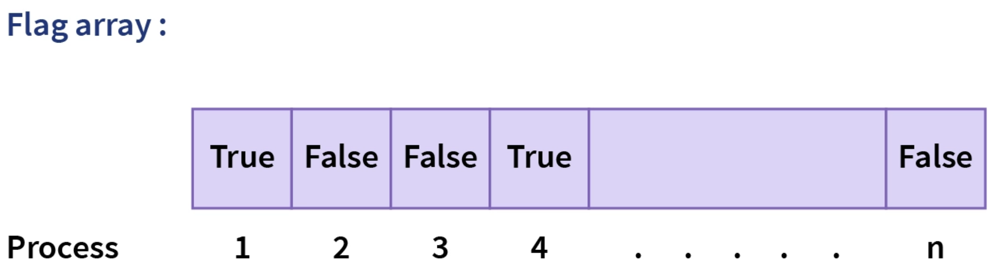
Peterson's solution

Peterson's approach to critical section problems is extensively utilized. It is a classical software-based solution.

The solution is based on the idea that when a process is executing in a critical section, then the other process executes the rest of the code and vice-versa is also possible, i.e., this solution makes sure that only one process executes the critical section at any point in time.

In Peterson's solution, we have two shared variables that are used by the processes.

- **A Boolean Flag[]**: A Boolean array Flag that is initialized to FALSE. This Flag array represents which process wants to enter into the critical solution.
- **int Turn**: A integer variable Turn indicates the process number which is ready to enter into the critical section.


Flag array :
True
False
False
True
False
Process
1
2
3
4
. . . . .
n