Introduction to processes

Early computer system allowed any one program to be executed at a once, current days computer system allow multiple program to be loaded in memory and to be executed at the same time or step by step. This executing multiple programs is being resulted as process. A process is the unit of work in a modern time-sharing system.

What is a Process?

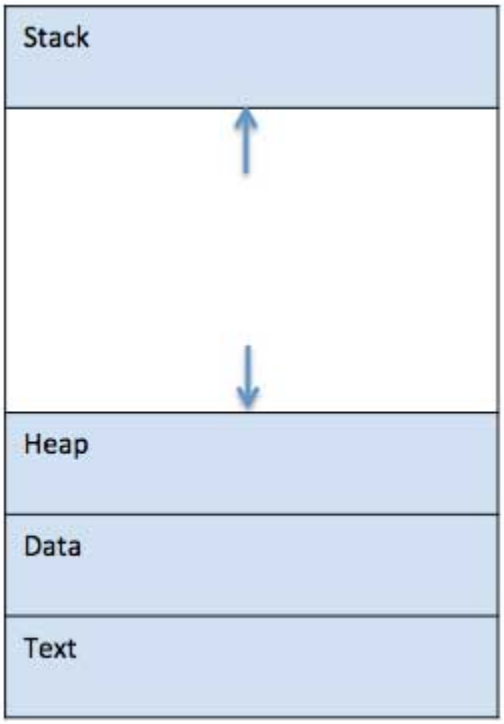
A process is a program in execution. Process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ **stack, heap, text and data**. The following image shows a simplified layout of a process inside main memory.

Process memory is divided into four sections for efficient working:

- The **Text section** is made up of **the compiled program code, read in from non-volatile storage** when the program is launched.
- The **Data section** is made up of the **global and static variables, allocated and initialized prior to executing the main.**

- The **Heap** is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The **Stack** is used for local variables. Space on the stack is reserved for **local variables when they are declared**.



States of Process:

A process is in one of the following states:

1. New: Newly Created Process (or) being-created process. A program which is going to be picked up by the OS into the main memory is called a new process.

2. **Ready:** After creation process moves to Ready state, i.e. the process is ready for execution. Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned. The OS picks the new processes from the secondary memory and put all of them in the main memory.

The **processes which are ready for the execution and reside in the main memory are called ready state processes**. There can be many processes present in the ready state.

3. Run: Currently running process in CPU (only one process at a time can be under execution in a single processor). One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm. Hence, if we have only one CPU in our system, the number of running processes for a particular time will always be one. If we have n processors in the system then we can have n processes running simultaneously.

4. Wait (or Block): When a process requests I/O access. When a process waits for a certain resource to be assigned or for the input from

the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.

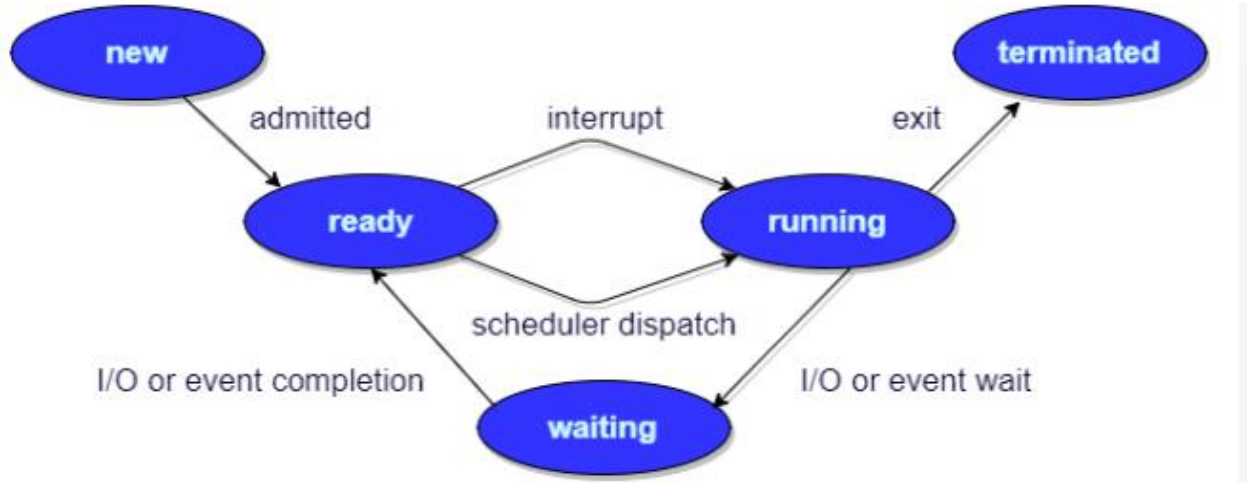
5. Complete (or Terminated): The process completed its execution. When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.

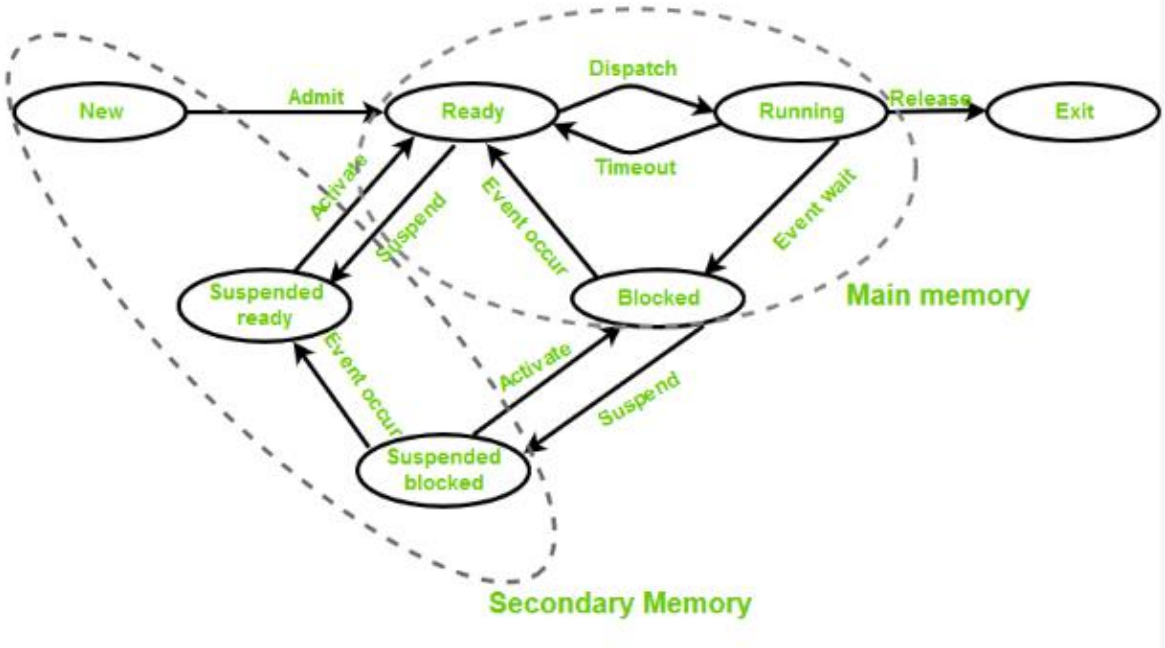
6. Suspended Ready: When the ready queue becomes full, some processes are moved to suspended ready state. A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.

If the main memory is full and a higher priority process comes for the execution then the OS have to make the room for the process in the main memory by throwing the lower priority process out into the secondary memory. The suspend ready processes remain in the secondary memory until the main memory gets available.

7. Suspended Block: When waiting queue becomes full. Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main

memory. Since it is already waiting for some resource to get available hence it is better if it waits in the secondary memory and make room for the higher priority process. These processes complete their execution once the main memory gets available and their wait is finished.





Process Control Block

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process. There is a Process Control Block for each process, enclosing all the information about the process. It is a data structure, which contains the following:

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.
3	**Process ID** Unique identification for each of the process in the operating system.
4	**Pointer**

	A pointer to parent process.
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, etc.

10	**IO status information** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −

The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

Concurrent process

Two processes are said to be concurrent if their execution can overlap in the time i.e. the execution of second process starts before the first completes. Typically, the processor will be executing instructions for one program while another (or several other) program(s) are waiting for I/O from external devices or from an end-user.

Parallel processing

Simultaneous use of more than one CPU or processors to execute a program or multiple computation threads. **Parallel Processing Systems** are designed to speed up the execution of programs by dividing the program into multiple fragments and processing these fragments simultaneously. Such systems are multiprocessor systems also known as tightly coupled systems. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors.

Inter Process Communication (IPC)

A process can be of two type:

- Independent process.

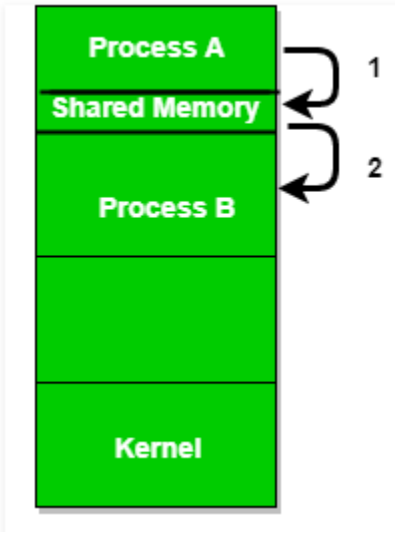
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

1. Shared Memory
2. Message passing

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some or memory. 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 other process, process1 generate information about certain computations or resources being used and

keeps it as a record in shared memory. When process2 need 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 other process as well as for delivering any specific information to other process.



Messaging Passing Method

In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

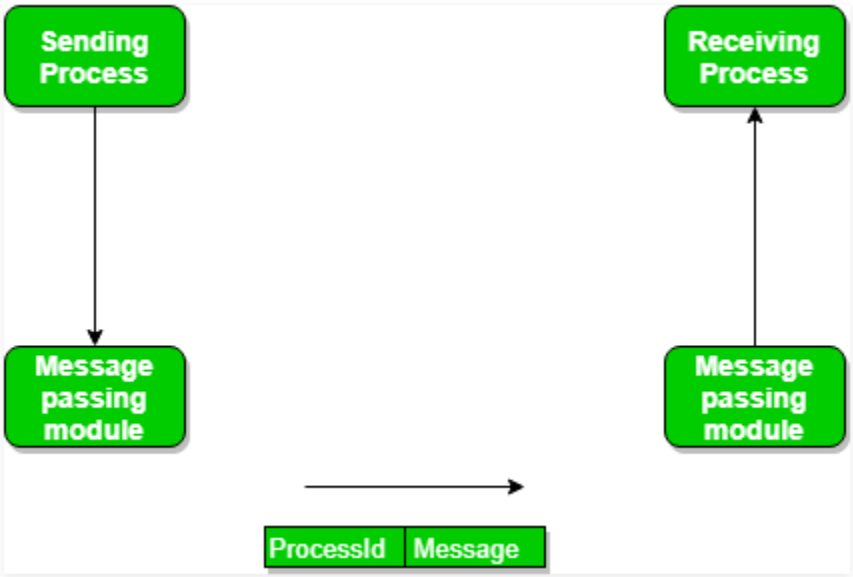
- Establish a communication link (if a link already exists, no need to establish it again.)

- Start exchanging messages using basic primitives.

  We need at least two primitives:

  – **send** (message, destination) or **send**(message)

  – **receive** (message, host) or **receive**(message)

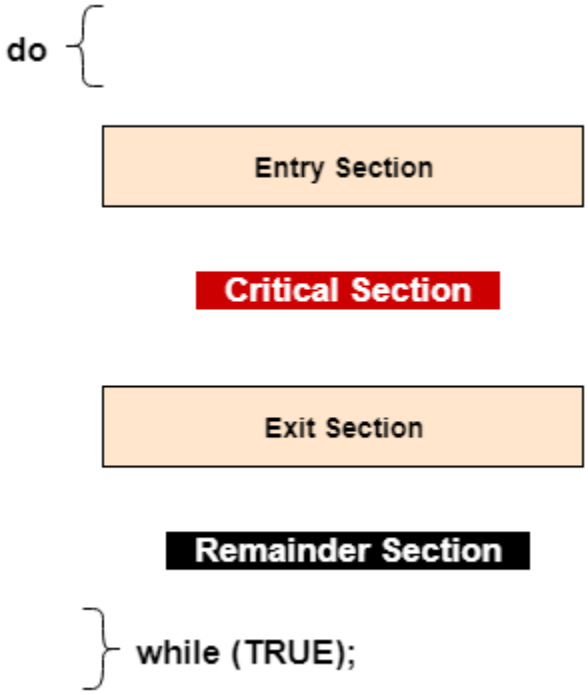


Critical section

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:

1. **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.

2. **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.

3. **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 a situation that may occur inside a critical section. This happens when the result of multiple thread execution in 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.

Mutual exclusion

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.

Mutual Exclusion Conditions

If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We

need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process should outside its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

Dekker's Algorithm for critical section solution

- Dekker's algorithm was the first probably correct solution to the critical section problem.
- It makes use of variable to control which thread can execute.
- The algorithm has two processes which access a critical section inside a loop. Both processes set their flags to true when the process is in its critical section and reset it to false when leaving the critical section.
- To enter a critical section, a process checks the other process's flag. If the other process's flag is false then it writes true on its own flag and enters the critical section.
- When leaving the critical section, it writes false on its own flag. Dekker's algorithm assumes that there are two processes: P_{first} and P_{second} where

  int first =0;

  int second=1-0;

  The two processes share two variables:

  boolean flag[2];

  int turn;

here initially, flag[first]=flag[second]=false

int first=0, second=1-0, int turn;

Boolean flag[2];

P_{first}

//Entry section begins

flag[first]=true;

while(flag[second])

{

If(turn==second) {

flag[first]=false;

while(turn==second) {

//do nothing, wait as P_{second} is executing in its critical section.

;

}

flag[first]=true;

}

}

//entry section ends

{critical section}

//exit section begins

turn= second;

flag[first]=false;

//exit section ends

{remainder section}

P_{second}

//Entry section begins

flag[second]=true;

while(flag[first])

{

If(turn==first) {

flag[second]=false;

while(turn==first) {

//do nothing, wait as P_{first} is executing in its critical section.

;

}

flag[second]=true;

}

}

//entry section ends

{critical section}

//exit section begins

turn= first;

flag[second]=false;

//exit section ends

{remainder section}

Disadvantages of Dekker's algorithm

- It is limited to two processes.
- Makes use of busy waiting instead of process suspension

Peterson's solution to critical section problem

- Peterson's solution is a classic software-based solution to the critical-section problem.
- It provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- Peterson's solution requires two data items to be shared between the two processes:

int turn;

boolean flag[2]

where initially flag[i]=flag[j]=false

- The variable turn indicates whose turn it is to enter its critical section. That is, if **turn == i, then process P_i is allowed to execute in its critical section**.

- The flag array is used to indicate if a process is ready to enter its critical section. For example, if **flag[i] is true**, this value indicates that **Pi is ready to enter its critical section**.

P_i	P_j
do {	**do {**
flag[i]==true;	**flag[j]==true;**
turn=j;	**turn=i;**
while (flag[j] && turn==j);	**while (flag[i] && turn==i);**
/* critical section */	**/* critical section */**
flag[i]=false;	**flag[j]=false;**
/* remainder section */	**/* remainder section */**
}	**}**
while (true);	**while (true);**

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.

Test and set lock

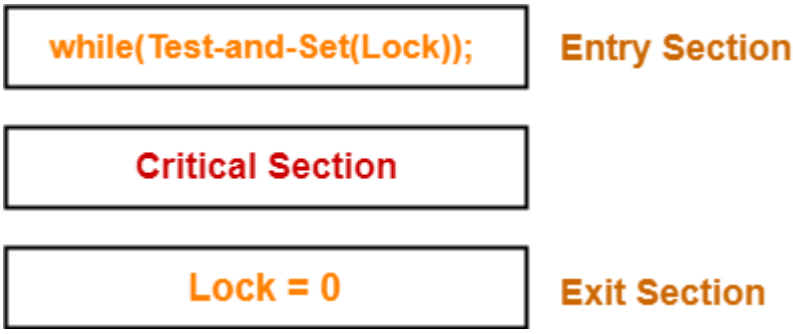
- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.

Test-and-Set Instruction

It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.

If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

It is a hardware solution.



The assembly code of the solution will look like following.

Enter region: // before entering its critical section process calls enter region

1. TSL R0, Lock // copy lock variable to register and set lock to 1
2. CMP R0, #0 // is lock variable 0
3. JNZ step 1 // if it not zero lock is set so loop
4. RET // return to caller; critical section entered

Leave region: // calls leave region when a process wants to leave a critical section

Move Lock, #0 // store 0 in lock variable

RET // return to caller

Initially, lock value is set to 0.

- Lock value = 0 means the critical section is currently vacant and no process is present inside it.

- Lock value = 1 means the critical section is currently occupied and a process is present inside it.

Scene-01:

- Process P_0 arrives.
- It executes the test-and-set (Lock) instruction.
- Since lock value is set to 0, so it returns value 0 to the while loop and sets the lock value to 1.
- The returned value 0 breaks the while loop condition.
- Process P0 enters the critical section and executes.
- Now, even if process P0 gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process P0 completes and sets the lock value to 0.

Scene-02:

- Another process P1 arrives.
- It executes the test-and-set (Lock) instruction.

- Since lock value is now 1, so it returns value 1 to the while loop and sets the lock value to 1.
- The returned value 1 does not break the while loop condition.
- The process P1 is trapped inside an infinite while loop.
- The while loop keeps the process P1 busy until the lock value becomes 0 and its condition breaks.

Scene-03:

Process P0 comes out of the critical section and sets the lock value to 0.

- The while loop condition breaks.
- Now, process P1 waiting for the critical section enters the critical section.
- Now, even if process P1 gets preempted in the middle, no other process can enter the critical section.
- Any other process can enter only after process P1 completes and sets the lock value to 0.

Characteristics-

The characteristics of this synchronization mechanism are-

- It ensures mutual exclusion.
- It is deadlock free.
- It does not guarantee bounded waiting and may cause starvation.
- It suffers from spin lock.
- It is not architectural neutral since it requires the operating system to support test-and-set instruction.
- It is a busy waiting solution which keeps the CPU busy when the process is actually waiting.

Let's examine TSL on the basis of the four conditions.

- **Mutual Exclusion**

  Mutual Exclusion is guaranteed in TSL mechanism since a process can never be preempted just before setting the lock variable. Only one process can see the lock variable as 0 at a particular time and that's why, the mutual exclusion is guaranteed.

- **Progress**

  According to the definition of the progress, a process which doesn't want to enter in the critical section should not stop other processes to get into it. In TSL mechanism, a process will execute the TSL instruction only when it wants to get into the critical section. The value of the lock will always be 0 if no process doesn't want to enter into the critical section hence the progress is always guaranteed in TSL.

- **Bounded Waiting**

  Bounded Waiting is not guaranteed in TSL. Some process might not get a chance for so long. We cannot predict for a process that it will definitely get a chance to enter in critical section after a certain time.

- **Architectural Neutrality**

  TSL doesn't provide Architectural Neutrality. It depends on the hardware platform. The TSL instruction is provided by the operating system. Some platforms might not provide that. Hence it is not Architectural natural.

Lock variable

This is the simplest synchronization mechanism. This is a Software Mechanism implemented in User mode. This is a busy waiting solution which can be used for more than two processes.

Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

The pseudo code of the mechanism looks like following.

```
Entry Section →
While (lock! = 0);
Lock = 1;
//Critical Section
Exit Section →
Lock =0;
```

Unfortunately, this idea contains flaw that violates mutual exclusion. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

Here’s the assembly code of the lock implementation

1. *Load R0, Lock*
2. *CMP R0, #0*
3. *JNZ Step 1*
4. *Store Lock, #1*
5. Critical section
6. *Store Lock, #0*