Delhi Technological University

CO – 204

OPERATING SYSTEMS DESIGN

Submitted To: **Dr. Daya Gupta**

Submitted By:

Name:	**Roll No:**
SHACHI INTODIA	2K19/CO/350
SHIKHAR CHAUHAN	2K19/CO/359

Topic: ***Deadlock and Concurrency Algorithms Visualiser* – A Website**

ACKNOWLEDGMENT

We (Shikhar Chaunhan and Shachi Intodia) would like to express our sincere gratitude to our supervisor Dr. Daya Gupta for providing her valuable guidance, comments and suggestions throughout the course of the project. We would like to thank her for providing us the opportunity to work on the topic - "***Deadlock and Concurrency Algorithms Visualiser* – A Website".**

We would also like thank my fellow classmates who have been very supportive and have motivated us throughout the course of the project.

Lastly, we would like to mention our parents who have helped and guided us to focus on our project. They taught us the basic ethical principles that one must follow in life which has ultimately led to the successful completion of the project.

We hope you have a great time while reading the project!

INTRODUCTION

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Concurrency is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel.

Our website consists of various deadlock and concurrency algorithms:

- Strict Alteration
- Counting Semaphore
- Producer Consumer Algorithm
- Lock variable
- Binary Semaphore
- Peterson’s Algorithm
- Test and Lock Set
- Banker’s Algorithm

These are all implemented using **html, css and javascript** on visual code studio. The code files and simulation/visualization is included in the zip file.

CONTENTS

This zip file of the project contains:

/Plagirism Check for .js – this includes plagiarism check for one of the .jss codes

/Plagirism Check for .css – this includes plagiarism check for one of the .css codes

/PPT_project – ppt file used for presentation

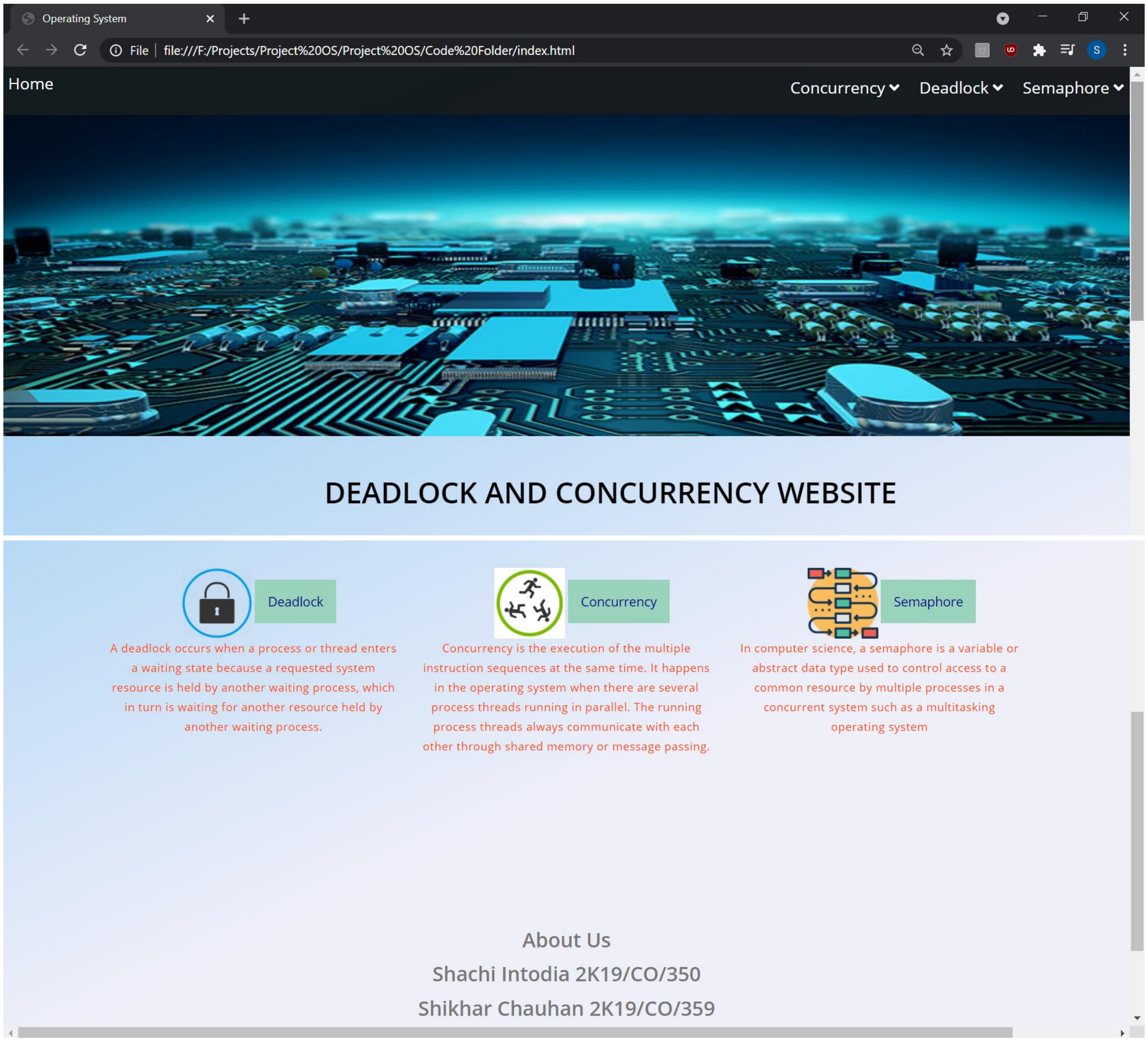
/Report.pdf - This file

/Code Folder - Contains all the .html, .css and .js files along with used images and fonts file. It also includes snapshots of execution.

❖ HOW TO ACCESS THEM?

- **Unzip the code folder**, you will find the **'index.html'** file, right click on that file and open with VS Code.
- To view other algorithms, open the other files by right clicking on them.
- **To directly open in browser**, double click on .html files.
- To open .css files or .js files, right click on them and open with VS Code.

HOME PAGE OF WEBSITE



Operating System
File | file:///F:/Projects/Project%20OS/Project%20OS/Code%20Folder/index.html
Home
Concurrency
Deadlock
Semaphore

DEADLOCK AND CONCURRENCY WEBSITE

Deadlock

A deadlock occurs when a process or thread enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process.

Concurrency

Concurrency is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel. The running process threads always communicate with each other through shared memory or message passing.

Semaphore

In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system

About Us

Shachi Intodia 2K19/CO/350

Shikhar Chauhan 2K19/CO/359

Few features of the website:

- If you hover on any icon/button you'll see the data it consists and its brief information.

- The website is user-friendly and easy to operate, since all the algorithms theory and simulation both are available along with the steps on how to simulate/visualize.
- We can also save a pdf version of output if we want after simulation of any algorithm.

STRICT ALTERATION

Strict Alternation Approach is the software mechanism implemented at user mode. It is a busy waiting solution which can be implemented only for two processes. In this approach, A turn variable is used which is actually a lock. This approach can only be used for only two processes. In general, let the two processes be P0 and P1. They share a variable called turn variable.

❖ Analysis of Strict Alternation approach

Let's analyze Strict Alternation approach on the basis of four requirements.

- **Mutual Exclusion**

The strict alternation approach provides mutual exclusion in every case. This procedure works only for two processes. The pseudo code is different for both of the processes. The process will only enter when it sees that the turn variable is equal to its Process ID otherwise not Hence No process can enter in the critical section regardless of its turn.

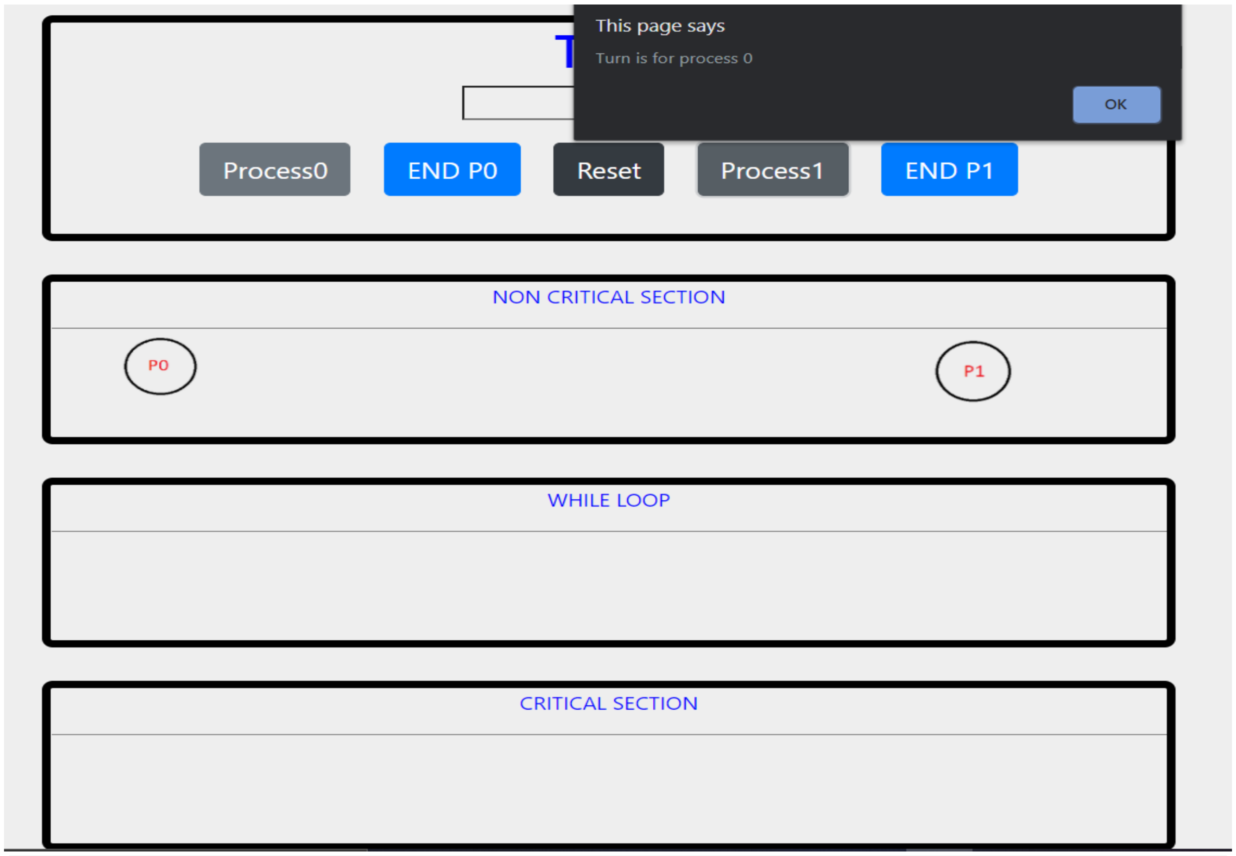
- **Progress**

Progress is not guaranteed in this mechanism. If Pi doesn't want to get enter into the critical section on its turn then Pj got blocked for infinite time. Pj has to wait for so long for its turn since the turn variable will remain 0 until Pi assigns it to j.

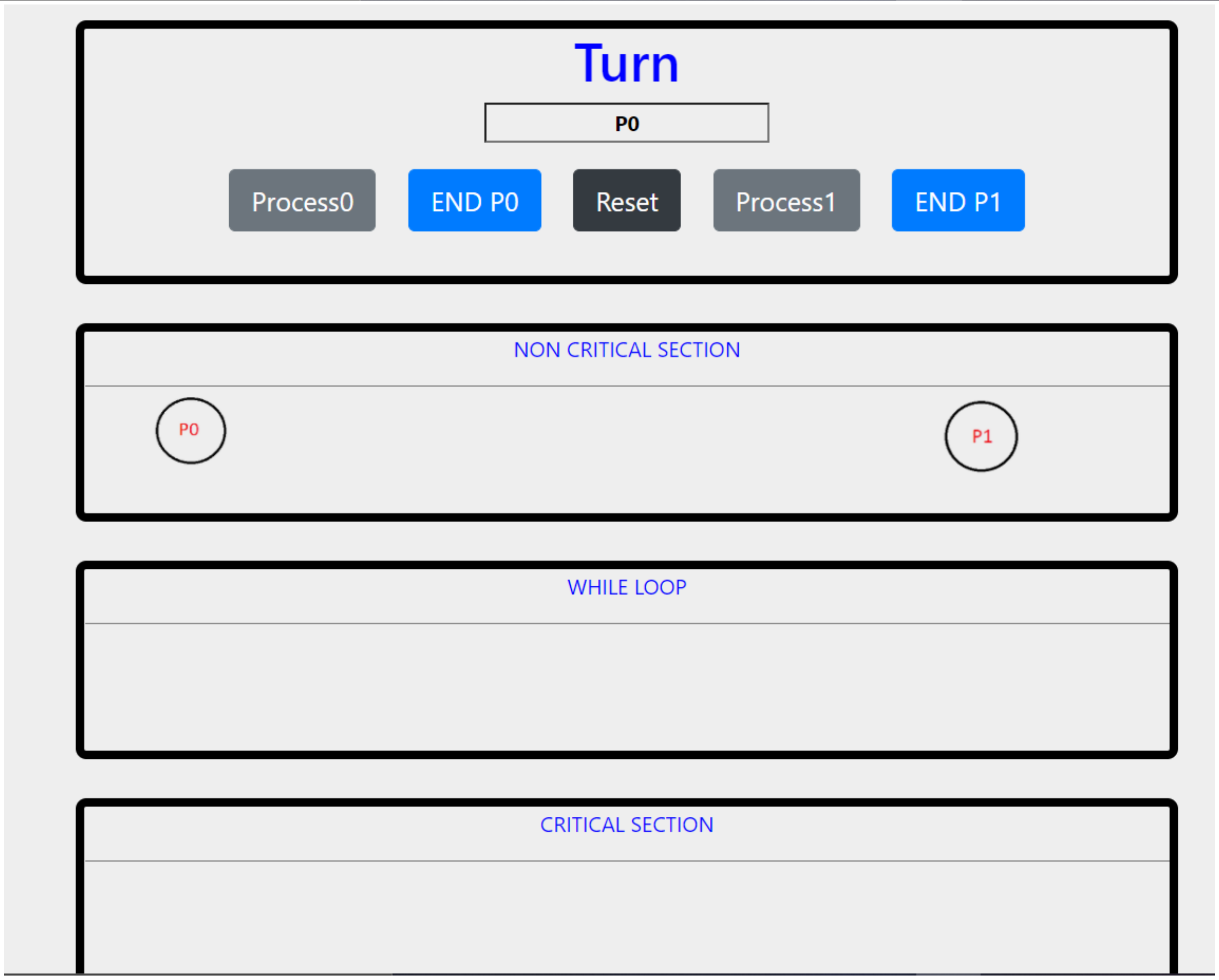
- **Portability**

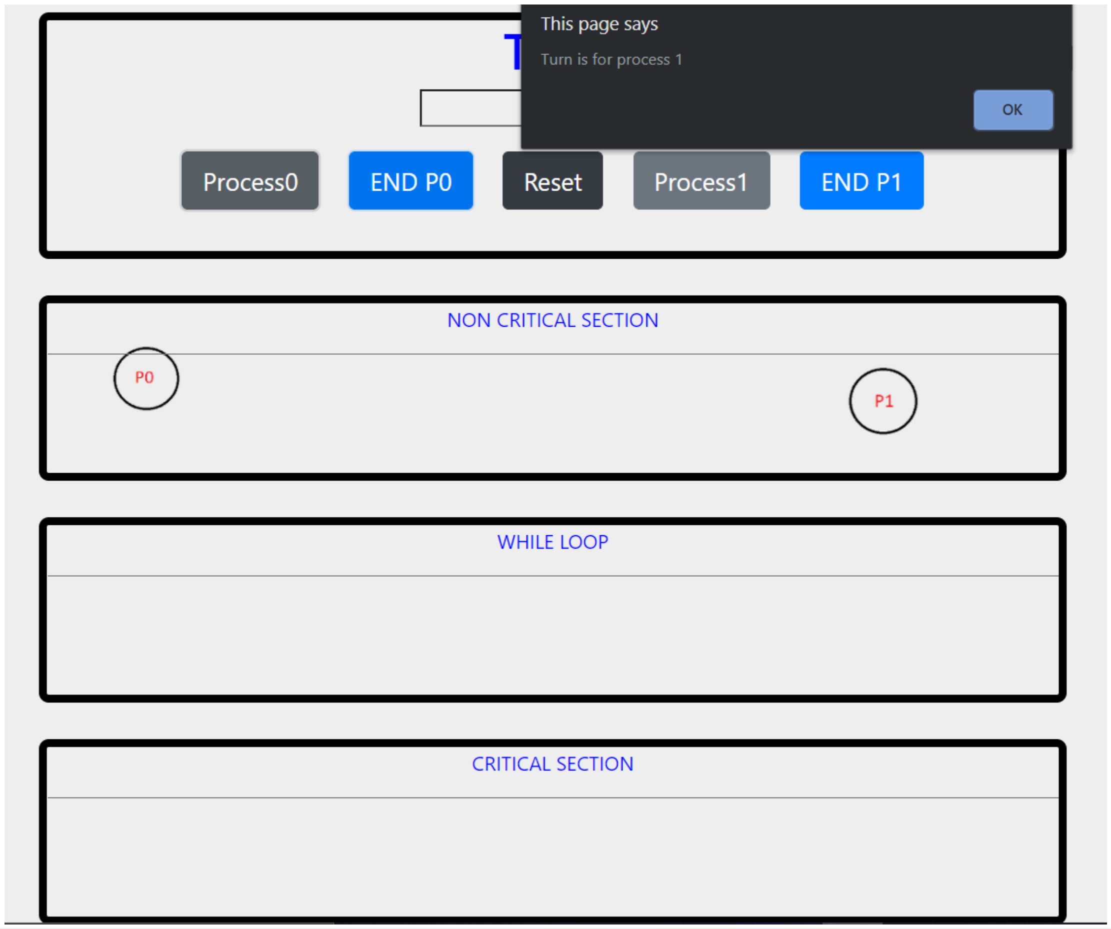
The solution provides portability. It is a pure software mechanism implemented at user mode and doesn't need any special instruction from the Operating System.

Mutual Exclusion	✓
Progress	✗
Bounded Waiting	✓
Portability	✓

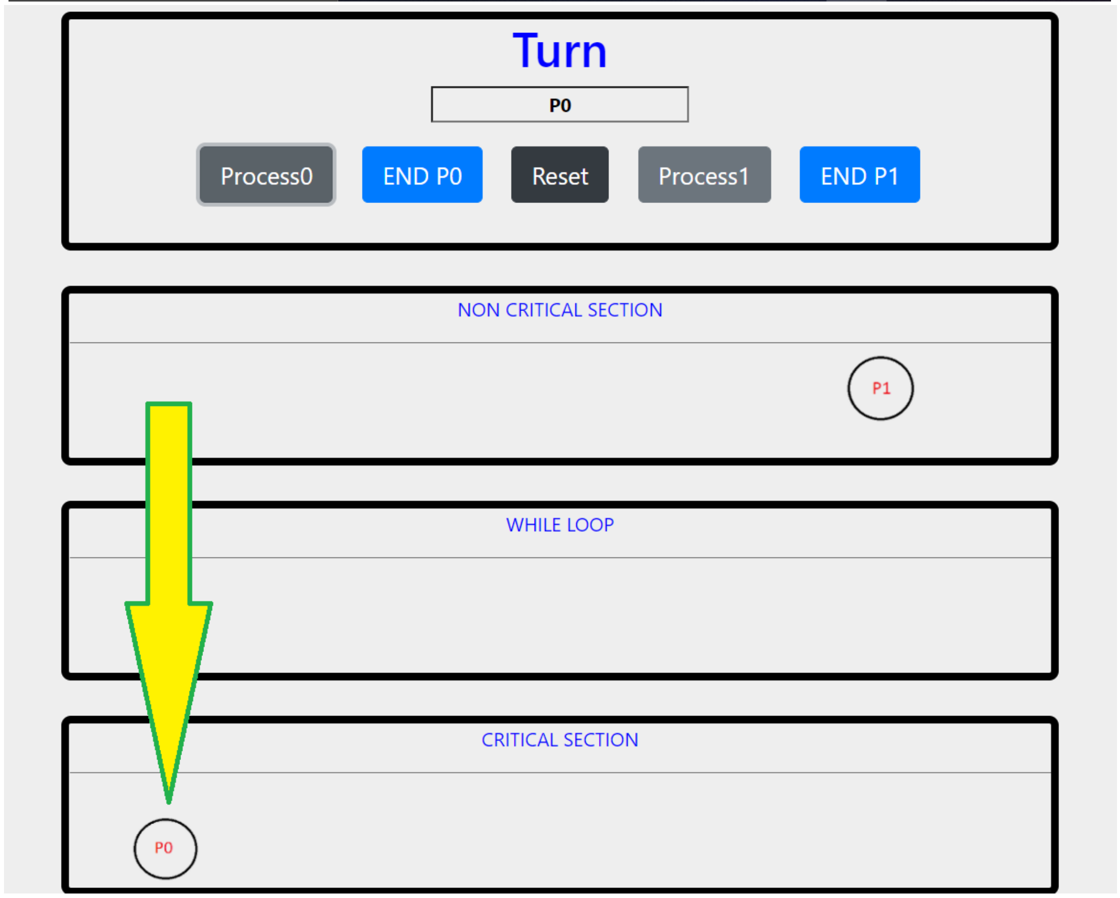

This page says

Turn is for process 0




This page says

Turn is for process 1

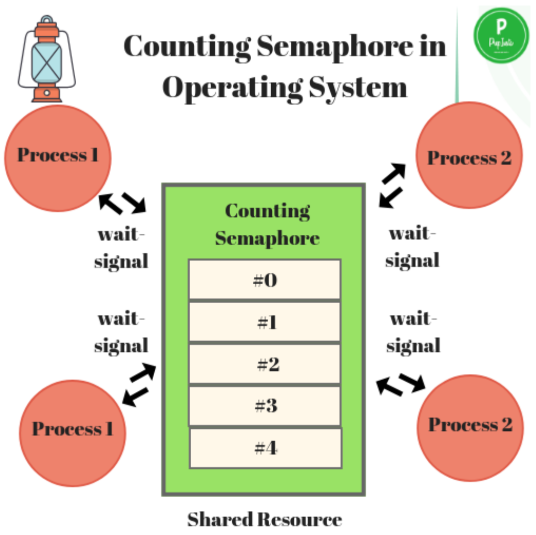


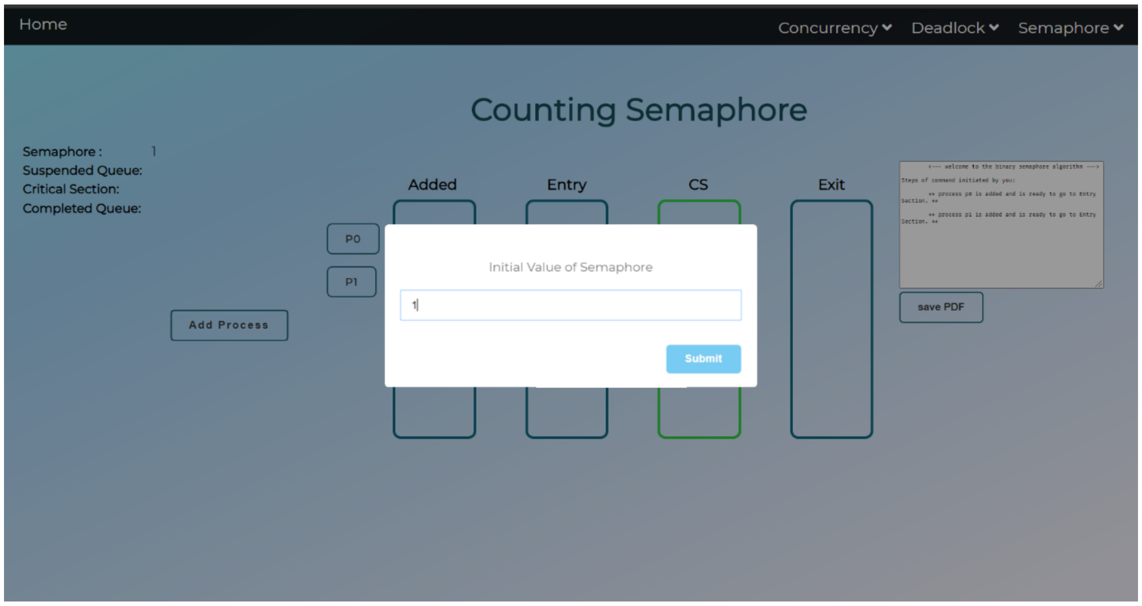
COUNTING SEMAPHORES

Semaphore is an integer variable S, that is initialized with the number of resources present in the system and is used for process synchronization. It uses two functions to change the value of S i.e. wait() and signal(). Both these functions are used to modify the value of semaphore but the functions allow only one process to change the value at a particular time i.e. no two processes can change the value of semaphore simultaneously. There are two categories of semaphores i.e. Counting semaphores and Binary semaphores.

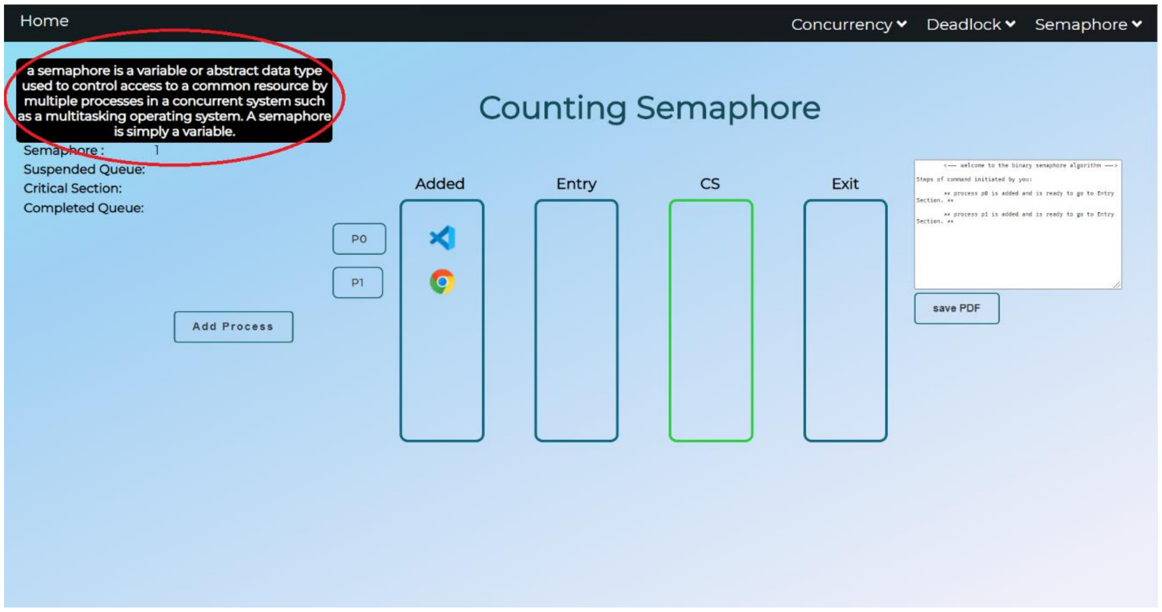
In **Counting semaphores**, firstly, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the wait() function is called and the value of the semaphore variable is decreased by one. The process then uses the resource and after using the resource, the signal() function is called and the value of the semaphore variable is increased by one. So, when the value of the semaphore variable goes to 0 i.e. all the resources are taken by the process and there is no resource left to be used, then if some other process wants to use resources, then that process has to wait for its turn. In this way, we achieve the process synchronization.

In the **implementation**, whenever the process waits it is added to a waiting queue of processes associated with that semaphore. This is done through system call block() on that process. When a process is completed it calls the signal function and one process in the queue is resumed. It uses wakeup() system call.

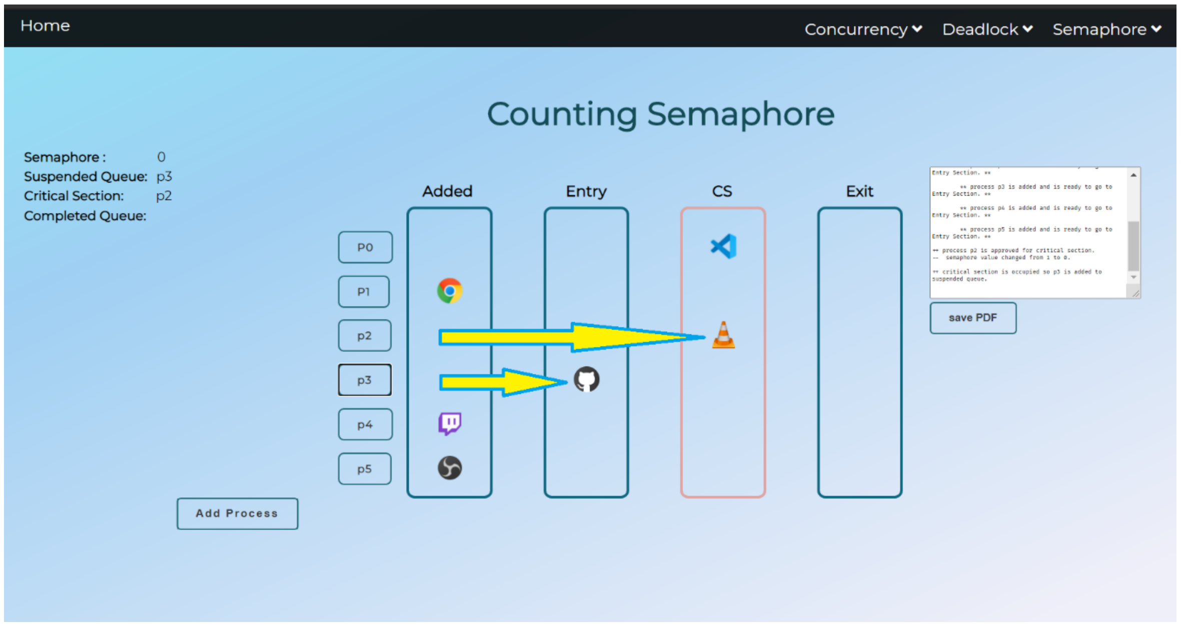




```
        <--- welcome to the binary semaphore algorithm --->

Steps of command initiated by you:

        ** process p0 is added and is ready to go to Entry
Section. **

        ** process p1 is added and is ready to go to Entry
Section. **
```


Home
Concurrency
Deadlock
Semaphore
a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system. A semaphore is simply a variable.
Counting Semaphore
Semaphore : 1
Suspended Queue:
Critical Section:
Completed Queue:
Added
Entry
CS
Exit
P0
P1
Add Process
save PDF

```
        <--- welcome to the binary semaphore algorithm --->

Steps of command initiated by you:

        ** process p0 is added and is ready to go to Entry
Section. **

        ** process p1 is added and is ready to go to Entry
Section. **
```



```
Entry Section. **

        ** process p3 is added and is ready to go to
Entry Section. **

        ** process p4 is added and is ready to go to
Entry Section. **

        ** process p5 is added and is ready to go to
Entry Section. **

++ process p2 is approved for critical section.
--  semaphore value changed from 1 to 0.

++ critical section is occupied so p3 is added to
suspended queue.
```

PRODUCER AND CONSUMER

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes.

In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.

What's the problem here?

The following are the problems that might occur in the Producer-Consumer:

- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

What's the solution?

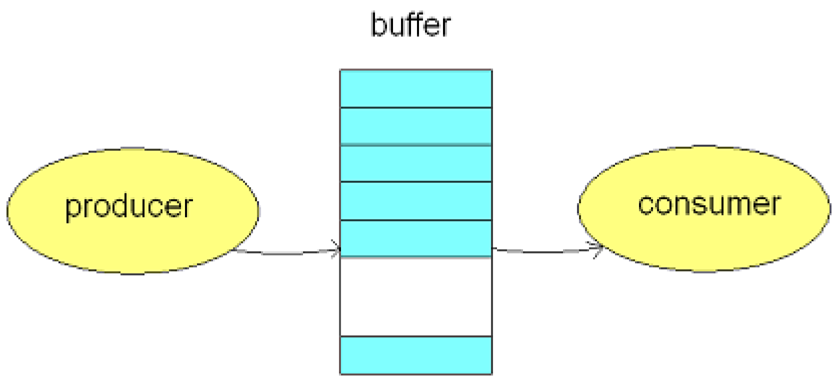
The above three problems can be solved with the help of semaphores.

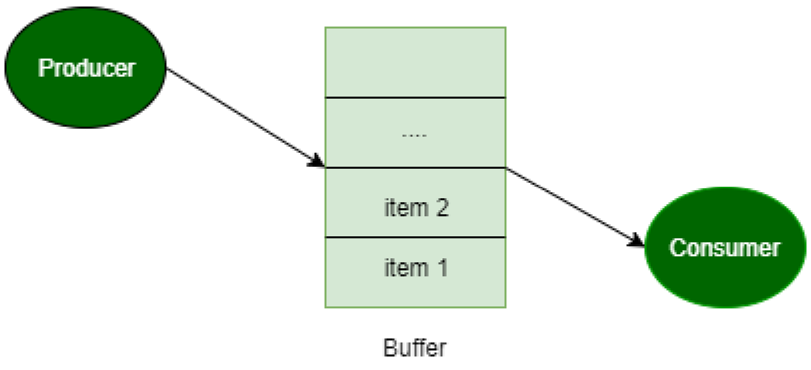
In the producer-consumer problem, we use three semaphore variables:

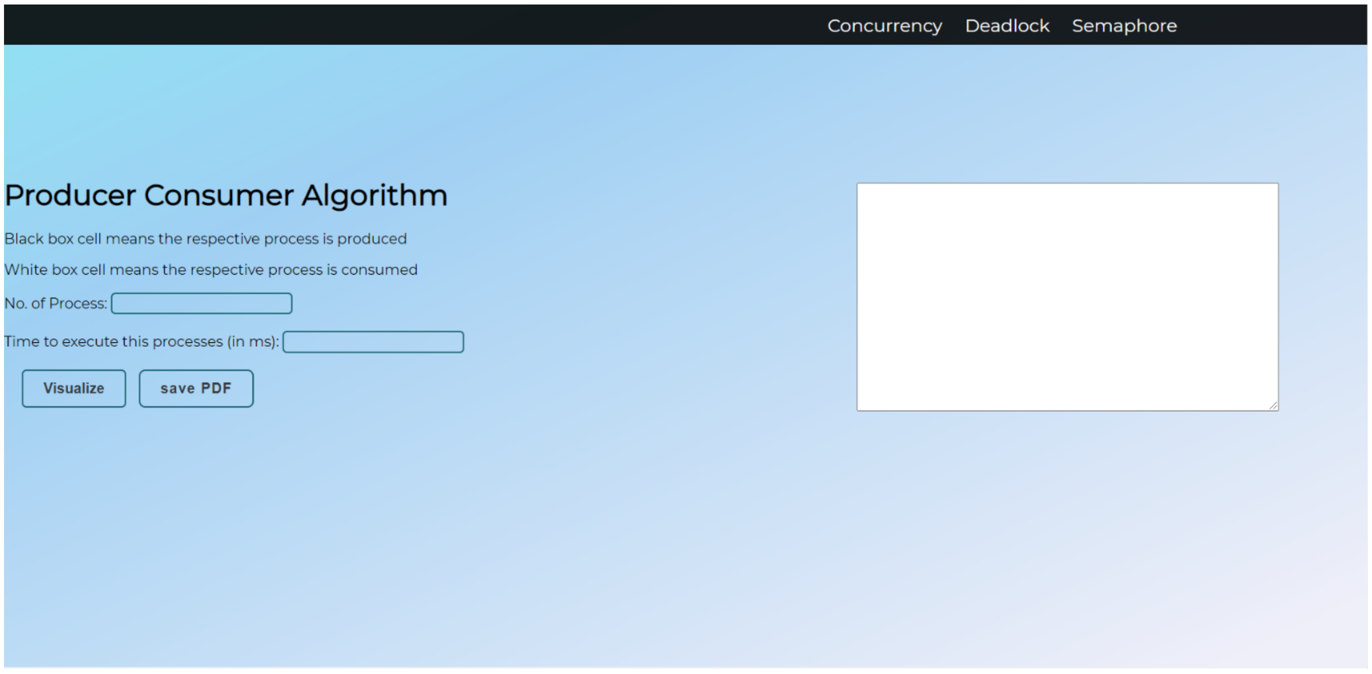
- Semaphore S: This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.

- Semaphore E: This semaphore variable is used to define the empty space in the buffer. Initially, it is set to the whole space of the buffer i.e. "n" because the buffer is initially empty.
- Semaphore F: This semaphore variable is used to define the space that is filled by the producer. Initially, it is set to "0" because there is no space filled by the producer initially.

By using the above three semaphore variables and by using the wait() and signal() function, we can solve our problem(the wait() function decreases the semaphore variable by 1 and the signal() function increases the semaphore variable by 1).






Concurrency Deadlock Semaphore

Producer Consumer Algorithm

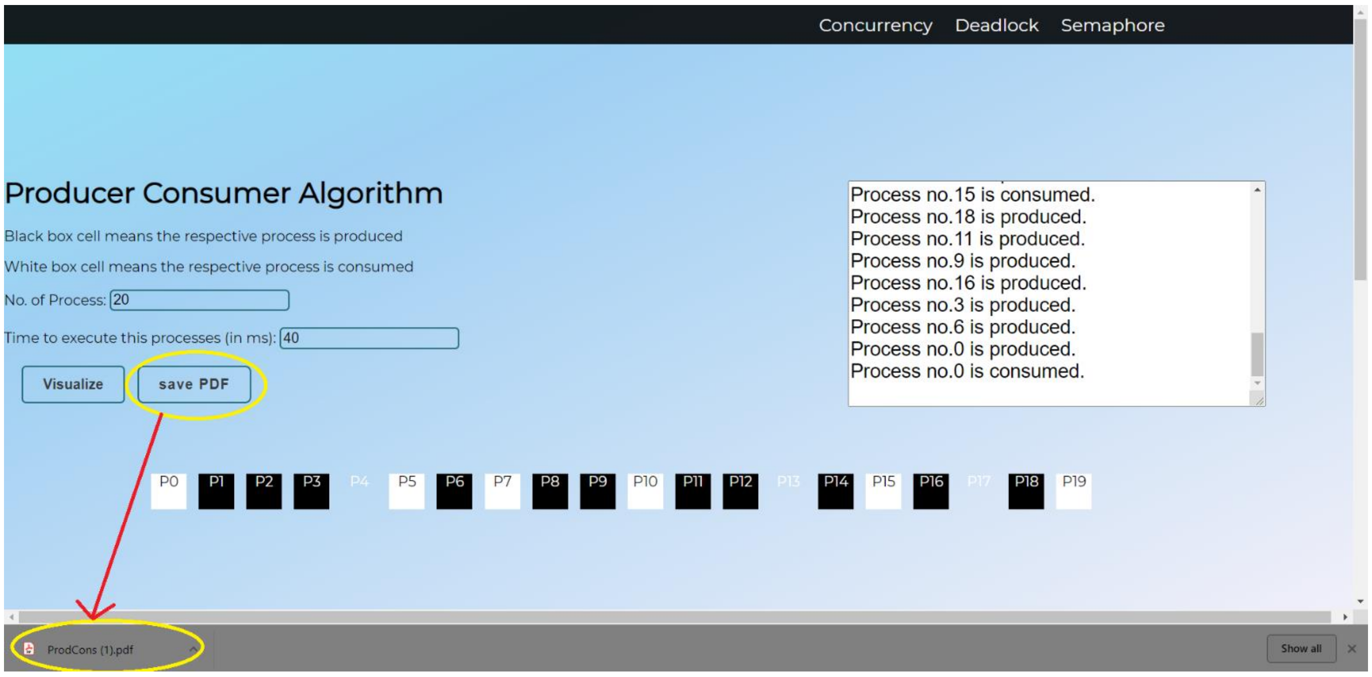
Black box cell means the respective process is produced

White box cell means the respective process is consumed

No. of Process:

Time to execute this processes (in ms):

Visualize save PDF


Concurrency Deadlock Semaphore

Producer Consumer Algorithm

Black box cell means the respective process is produced

White box cell means the respective process is consumed

No. of Process: 20

Time to execute this processes (in ms): 40

Visualize save PDF

```
Process no.15 is consumed.
Process no.18 is produced.
Process no.11 is produced.
Process no.9 is produced.
Process no.16 is produced.
Process no.3 is produced.
Process no.6 is produced.
Process no.0 is produced.
Process no.0 is consumed.
```

LOCK VARIABLE

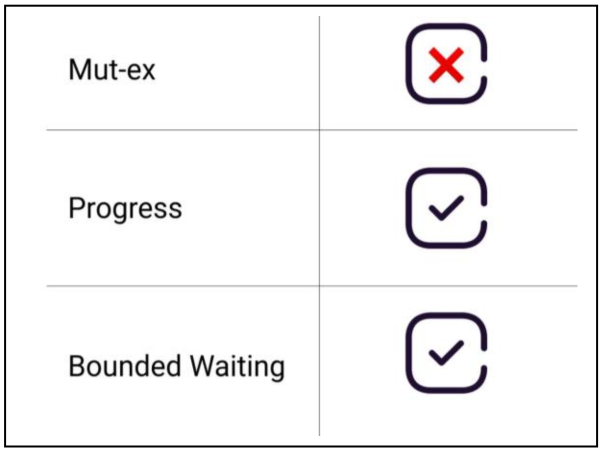
This is the simplest synchronization mechanism. This is a Software Mechanism implemented in User mode. This is a busy waiting solution which can be used for more than two processes.

In this mechanism, a Lock variable lockis used. Two values of lock can be possible, either 0 or 1. Lock value 0 means that the critical section is vacant while the lock value 1 means that it is occupied.

A process which wants to get into the critical section first checks the value of the lock variable. If it is 0 then it sets the value of lock as 1 and enters into the critical section, otherwise it waits.

The pseudo code of the mechanism looks like following.

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



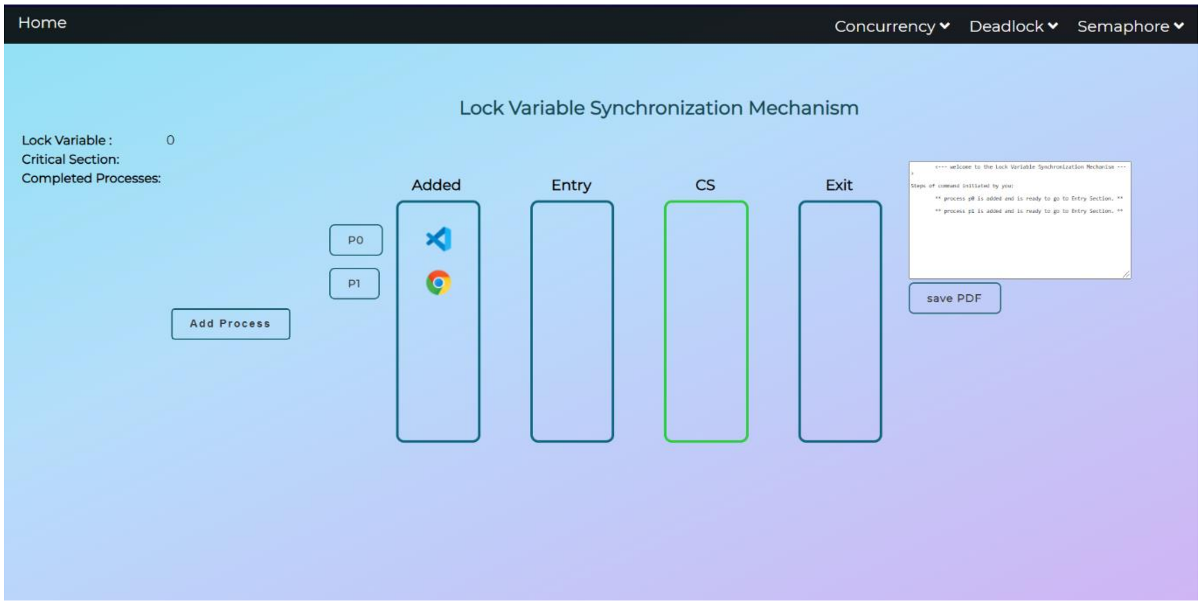
Mut-ex	✗
Progress	✓
Bounded Waiting	✓

If we look at the Pseudo Code, we find that there are three sections in the code. Entry Section, Critical Section and the exit section.

Initially the value of **lock variable** is **0**. The process which needs to get into the **critical section**, enters into the entry section and checks the condition provided in the while loop.

The process will wait infinitely until the value of **lock** is **1** (that is implied by while loop). Since, at the very first time critical section is vacant hence the process will enter the critical section by setting the lock variable as **1**.

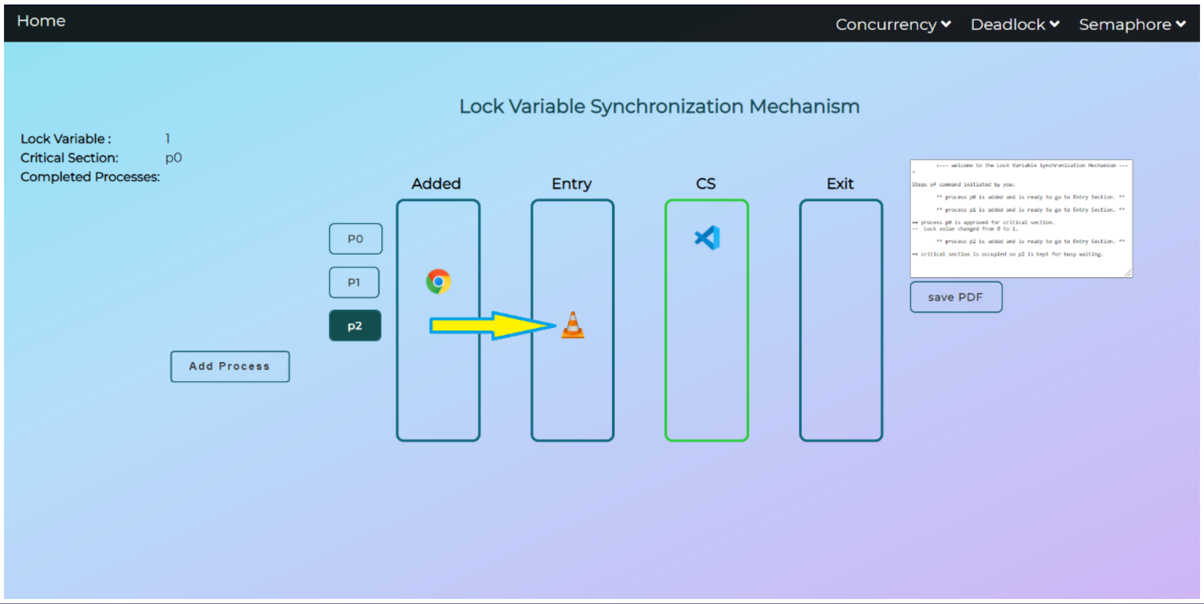
When the process exits from the critical section, then in the exit section, it reassigns the value of **lock** as **0**.



Lock Variable Synchronization Mechanism

Lock Variable : 0
Critical Section:
Completed Processes:

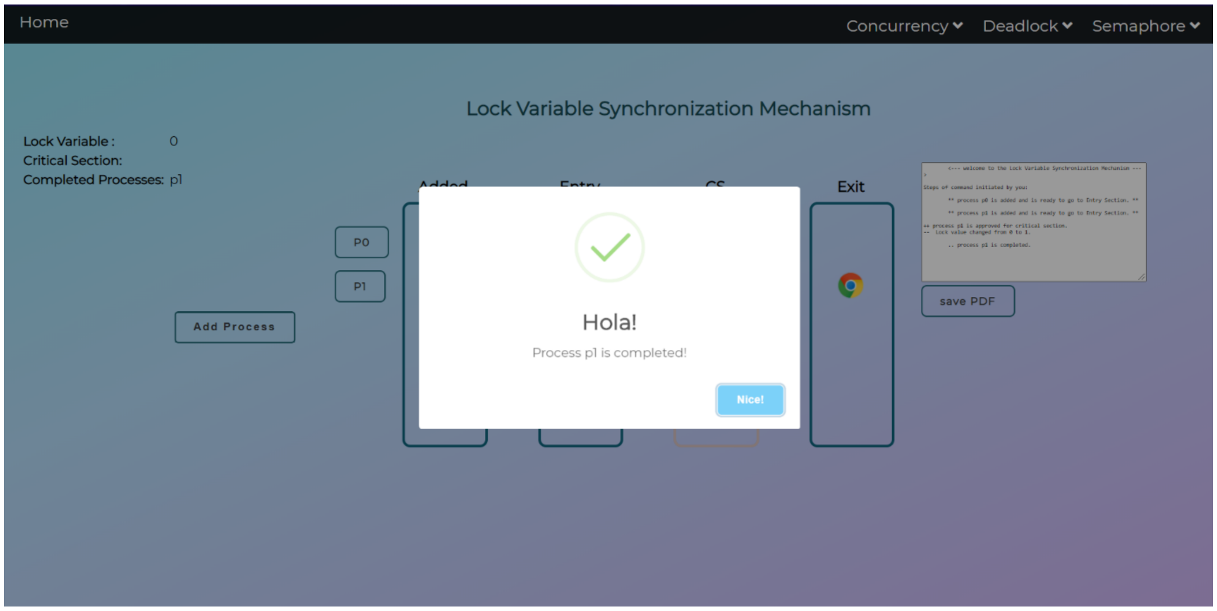
```
        <--- welcome to the Lock Variable Synchronization Mechanism ---
>

Steps of command initiated by you:

        ** process p0 is added and is ready to go to Entry Section. **

        ** process p1 is added and is ready to go to Entry Section. **
```



Lock Variable Synchronization Mechanism

Lock Variable : 1
Critical Section: p0
Completed Processes:

```
        <--- welcome to the Lock Variable Synchronization Mechanism ---
>

Steps of command initiated by you:

        ** process p0 is added and is ready to go to Entry Section. **

        ** process p1 is added and is ready to go to Entry Section. **

++ process p0 is approved for critical section.
--  lock value changed from 0 to 1.

        ** process p2 is added and is ready to go to Entry Section. **

++ critical section is occupied so p2 is kept for busy waiting.
```



Lock Variable Synchronization Mechanism

Lock Variable : 0
Critical Section:
Completed Processes: p1

Hola!

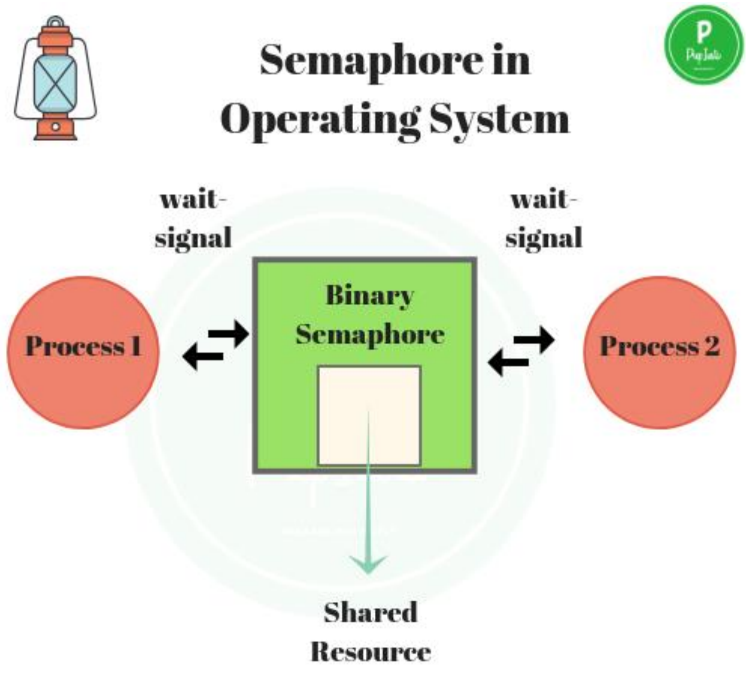
Process p1 is completed!

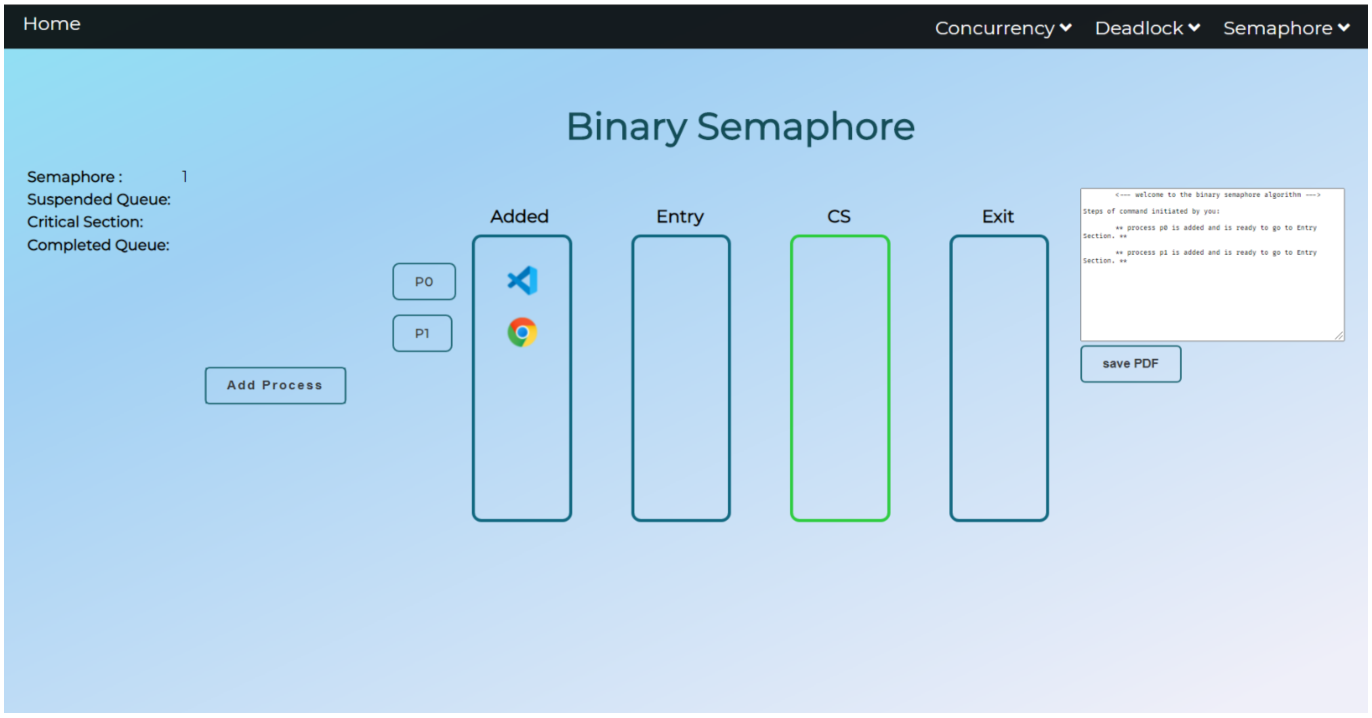
```
        <--- welcome to the Lock Variable Synchronization Mechanism ---
>

Steps of command initiated by you:

        ** process p0 is added and is ready to go to Entry Section. **

        ** process p1 is added and is ready to go to Entry Section. **

++ process p1 is approved for critical section.
--  lock value changed from 0 to 1.

        .. process p1 is completed.
```

BINARY SEMAPHORE

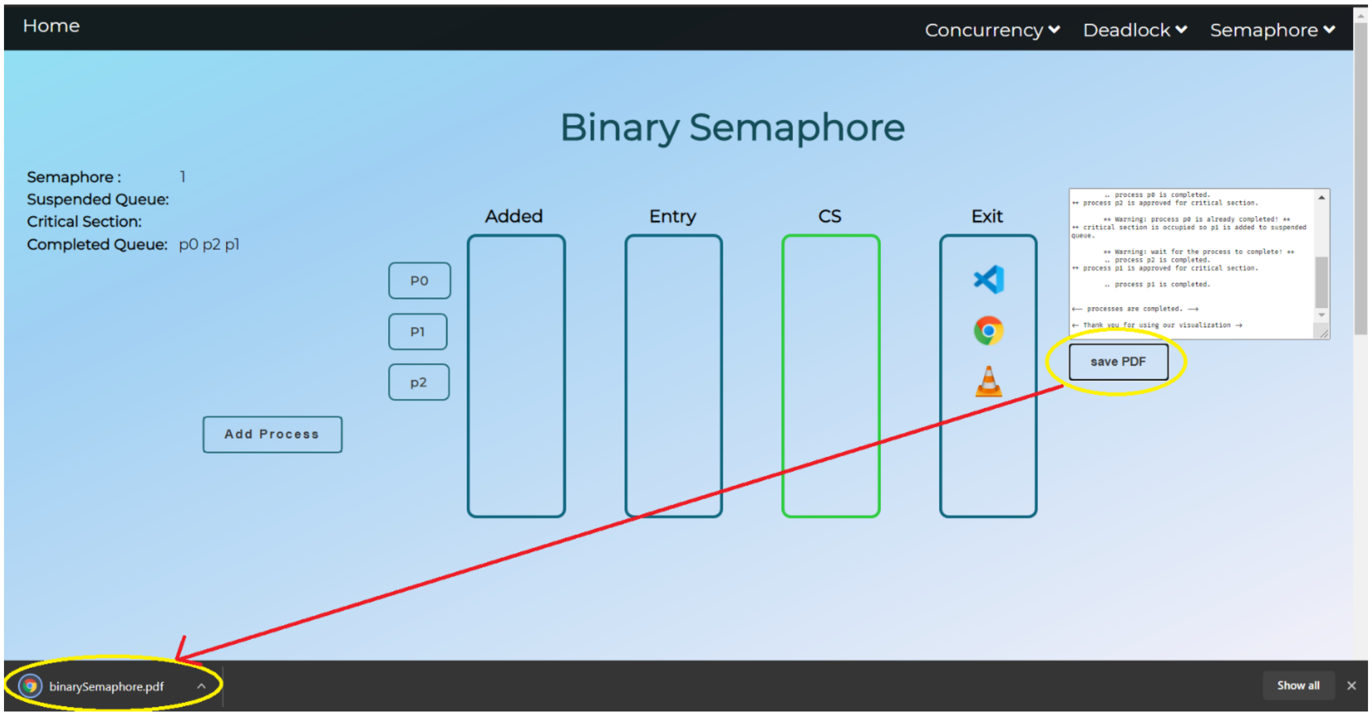
Semaphore is an integer variable S, that is initialized with the number of resources present in the system and is used for process synchronization. It uses two functions to change the value of S i.e. wait() and signal(). Both these functions are used to modify the value of semaphore but the functions allow only one process to change the value at a particular time i.e. no two processes can change the value of semaphore simultaneously. There are two categories of semaphores i.e. Counting semaphores and Binary semaphores.

In **Binary semaphores**, the value of the semaphore variable will be 0 or 1. Initially, the value of semaphore variable is set to 1 and if some process wants to use some resource then the wait() function is called and the value of the semaphore is changed to 0 from 1. The process then uses the resource and when it releases the resource then the signal() function is called and the value of the semaphore variable is increased to 1. If at a particular instant of time, the value of the semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of the resource by the previous process. In this way, process synchronization can be achieved. It is similar to mutex but here locking is not performed.





```
        <--- welcome to the binary semaphore algorithm --->

Steps of command initiated by you:

        ** process p0 is added and is ready to go to Entry
Section. **

        ** process p1 is added and is ready to go to Entry
Section. **
```



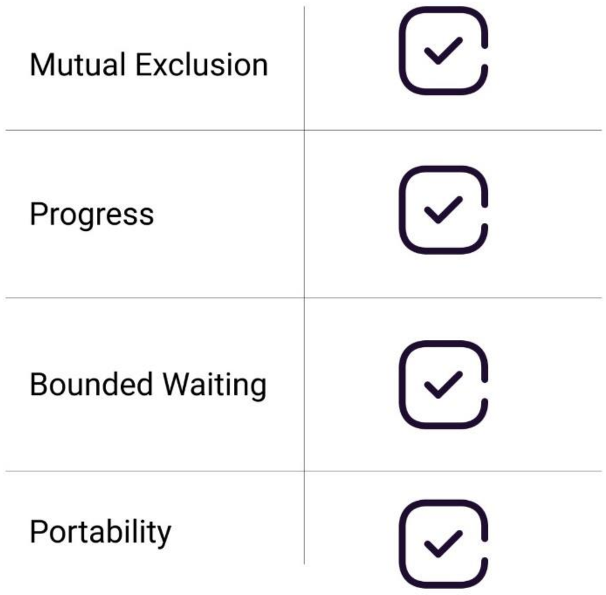
```
        .. process p0 is completed.
++ process p2 is approved for critical section.

        ** Warning: process p0 is already completed! **
++ critical section is occupied so p1 is added to suspended
queue.

        ** Warning: wait for the process to complete! **
        .. process p2 is completed.
++ process p1 is approved for critical section.

        .. process p1 is completed.


<-- processes are completed. -->

<- Thank you for using our visualization ->
```

PETERSON’S ALGORITHM

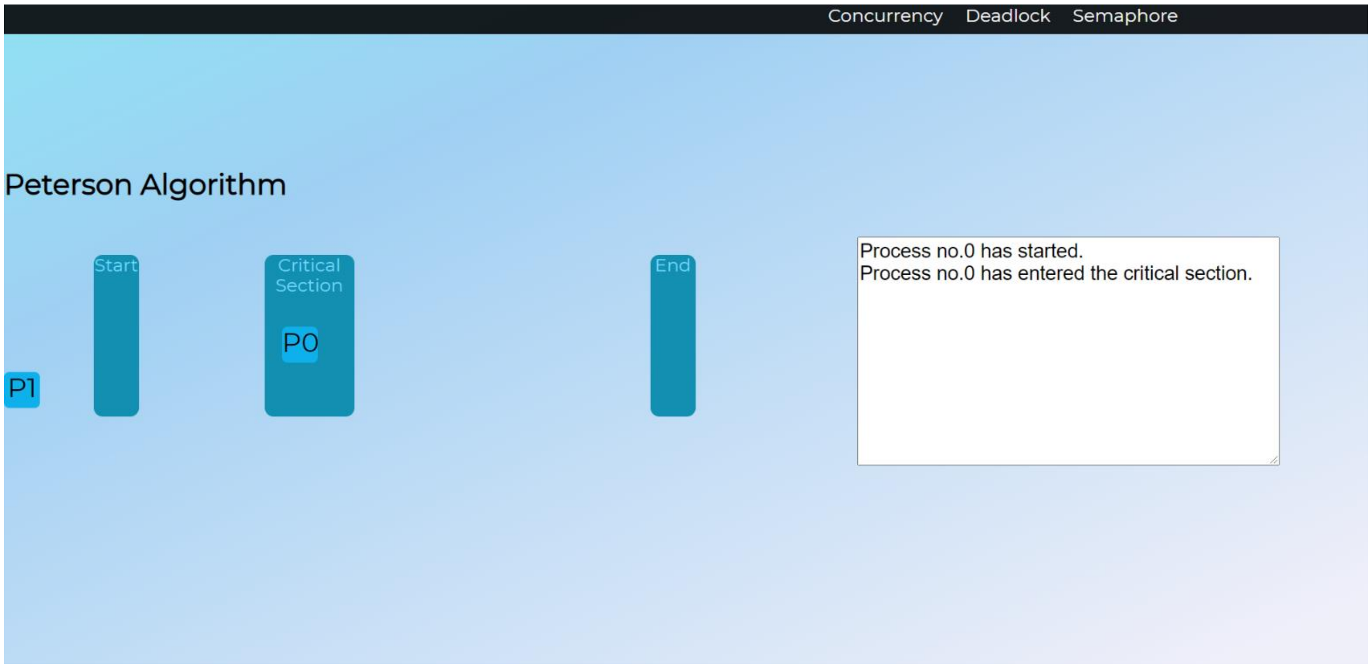
Peterson’s Algorithm is used to synchronize two processes. It uses two variables, a **bool array flag of size 2** and an int variable turn to accomplish it.

In the solution i represents the Consumer and j represents the Producer. Initially the flags are false. When a process wants to execute it’s critical section, it sets it’s flag to true and turn as the index of the other process. This means that the process wants to execute but it will allow the other process to run first. The process performs busy waiting until the other process has finished it’s own critical section.

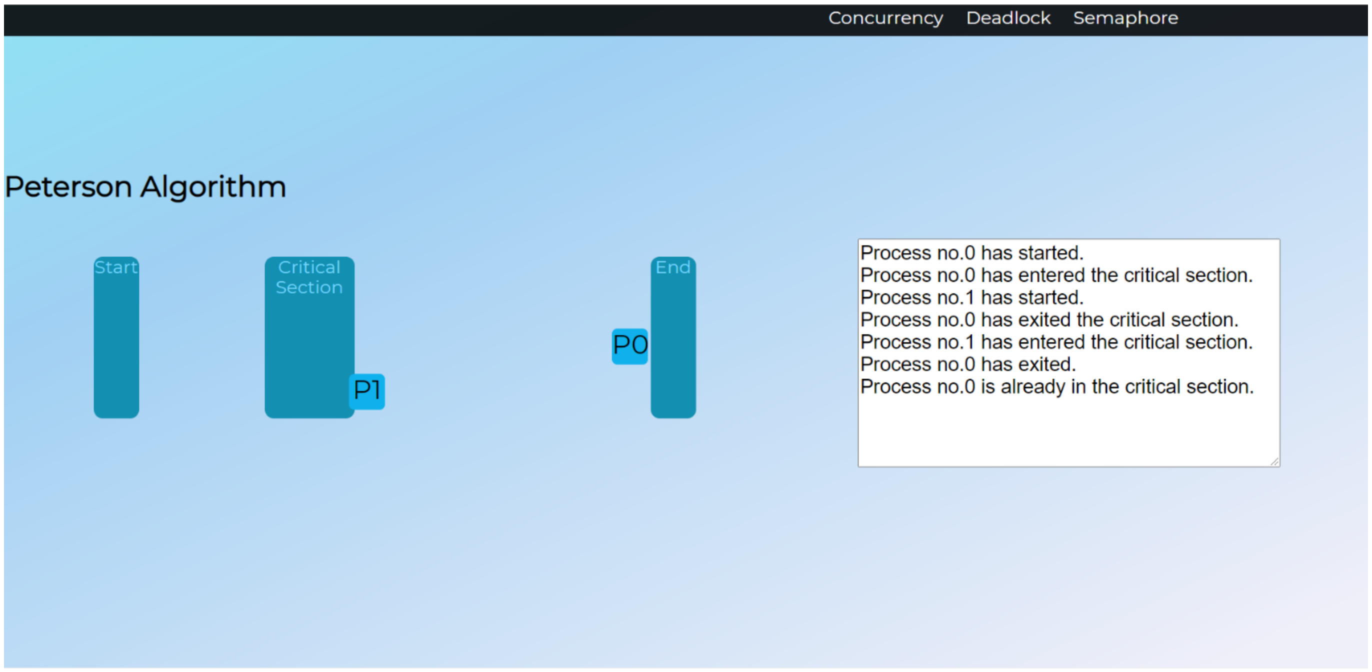
After this the current process enters it’s critical section and adds or removes a random number from the shared buffer. After completing the critical section, it sets it’s own flag to false, indication it does not wish to execute anymore.



Mutual Exclusion	✓
Progress	✓
Bounded Waiting	✓
Portability	✓



```
Process no.0 has started.
Process no.0 has entered the critical section.
```



```
Process no.0 has started.
Process no.0 has entered the critical section.
Process no.1 has started.
Process no.0 has exited the critical section.
Process no.1 has entered the critical section.
Process no.0 has exited.
Process no.0 is already in the critical section.
```

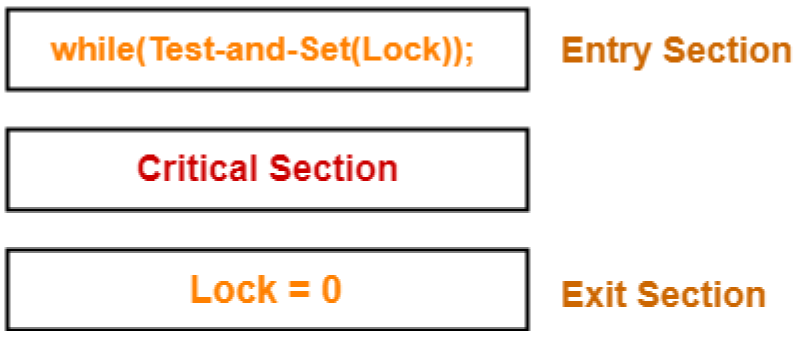
TEST AND SET

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

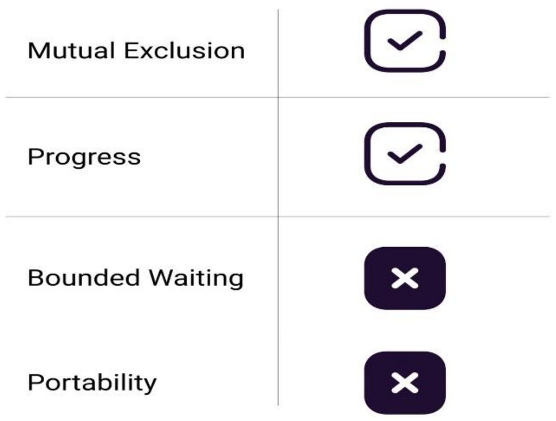
- It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.
- If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

It is implemented as-

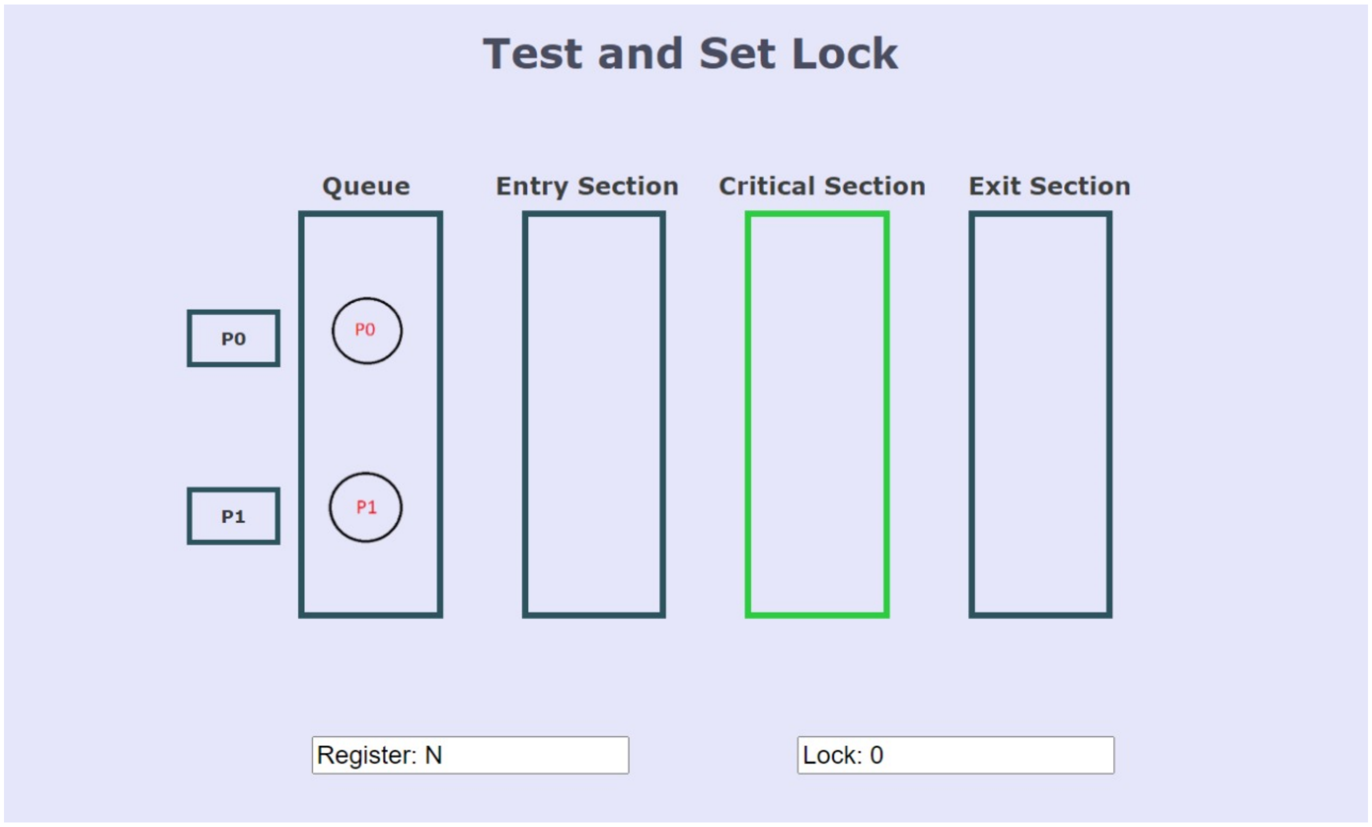


Initially, lock value is set to 0.

- 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.

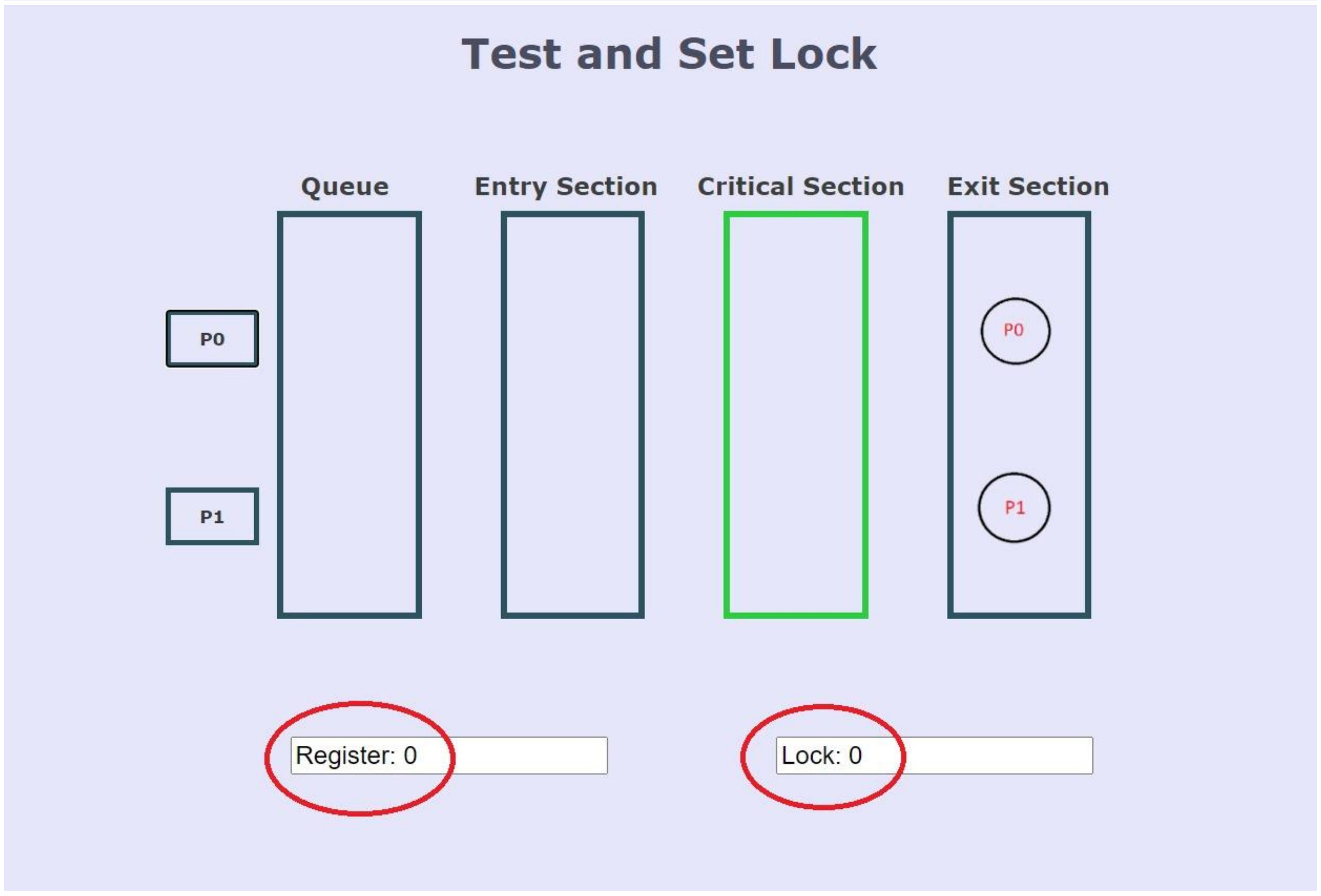


Mutual Exclusion	✓
Progress	✓
Bounded Waiting	✗
Portability	✗



Register: N

Lock: 0



Register: 0

Lock: 0

BANKERS ALGORITHM

The banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

❖ Why Banker’s algorithm is named so?

Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

❖ Characteristics of Banker's Algorithm

1. Keep many resources that satisfy the requirement of at least one client

2. Whenever a process gets all its resources, it needs to return them in a restricted period.

3. When a process requests a resource, it needs to wait

4. The system has a limited number of resources

5. Advance feature for max resource allocation

❖ Disadvantage of Banker's algorithm

1. Does not allow the process to change its Maximum need while processing

2. It allows all requests to be granted in restricted time, but one year is a fixed period for that.

3. All processes must know and state their maximum resource needs in advance.

❖ Summary:

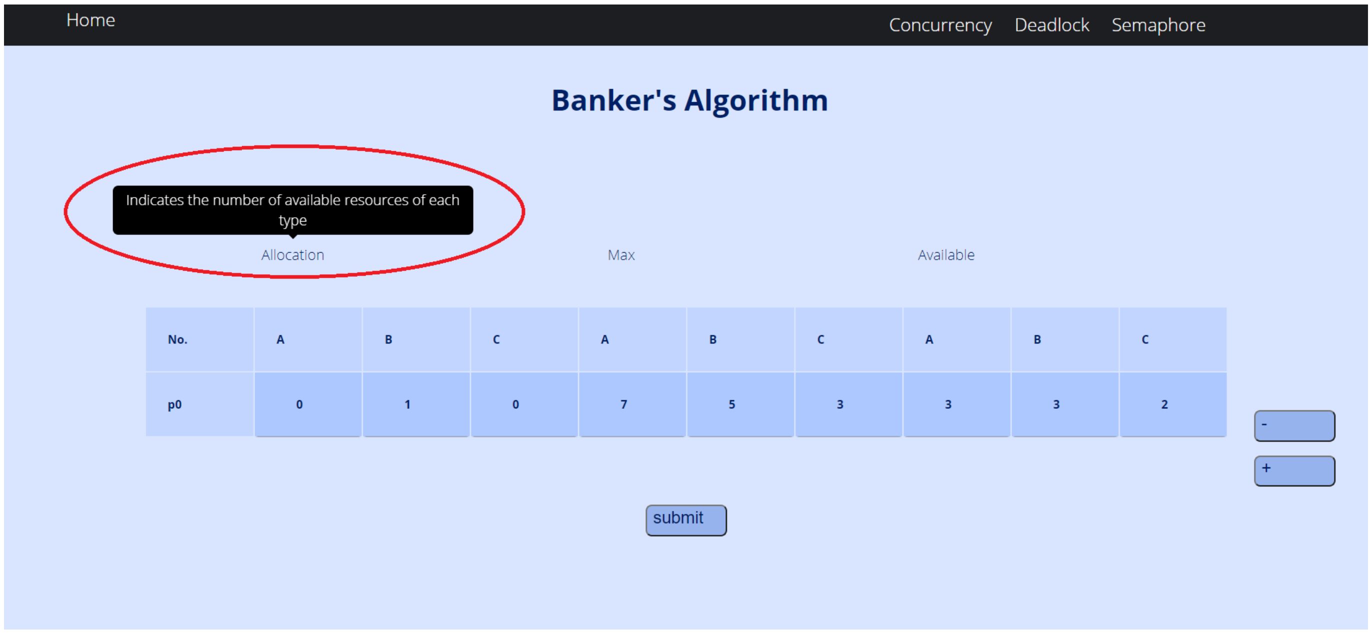
1. Banker's algorithm is used majorly in the banking system to avoid deadlock. It helps you to identify whether a loan will be given or not.

2. Notations used in banker's algorithms are 1) Available 2) Max 3) Allocation 4) Need

3. Resource request algorithm enables you to represent the system behavior when a specific process makes a resource request.

4. Banker's algorithm keeps many resources that satisfy the requirement of at least one client

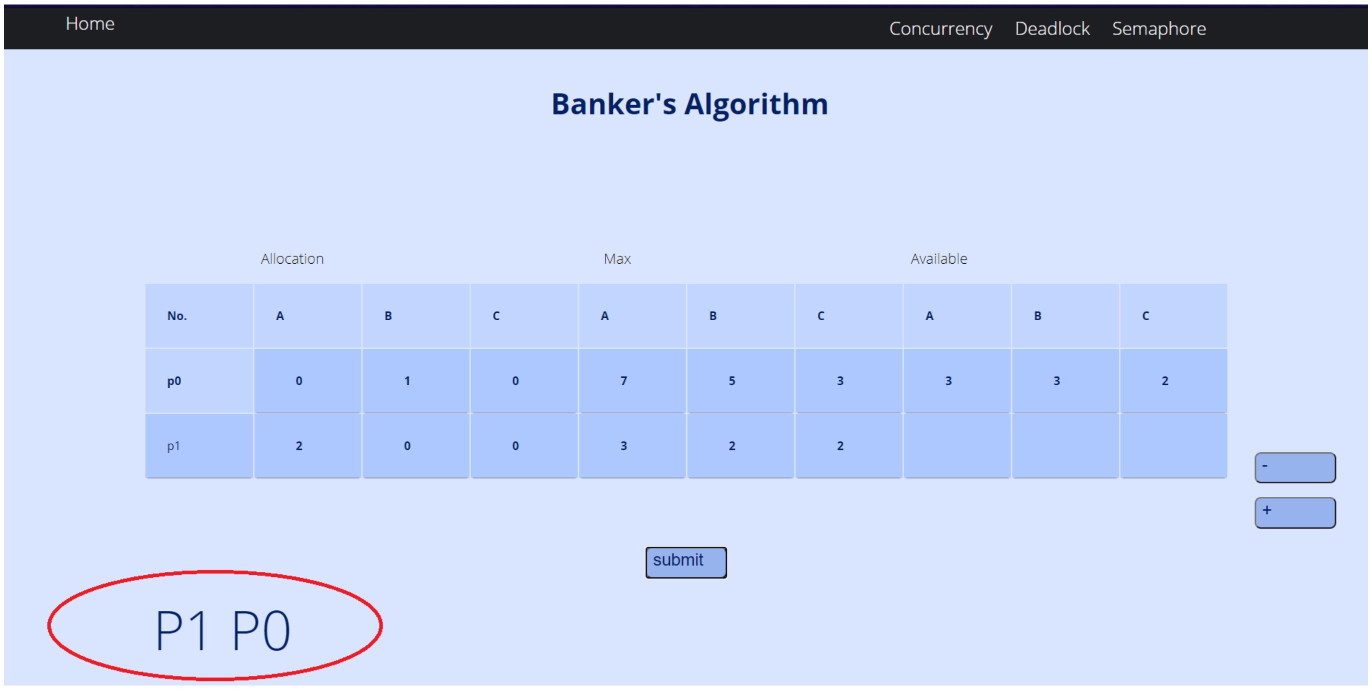
5. The biggest drawback of banker's algorithm this that it does not allow the process to change its Maximum need while processing.



Banker's Algorithm

Indicates the number of available resources of each type

	Allocation			Max			Available		
No.	A	B	C	A	B	C	A	B	C
p0	0	1	0	7	5	3	3	3	2



Banker's Algorithm

	Allocation			Max			Available		
No.	A	B	C	A	B	C	A	B	C
p0	0	1	0	7	5	3	3	3	2
p1	2	0	0	3	2	2			

P1 P0

BIBLIOGRAPHY

- https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/
- https://www.geeksforgeeks.org/concurrency-in-operating-system/
- https://www.tutorialspoint.com/mutex-vs-semaphore
- https://www.geeksforgeeks.org/producer-consumer-problem-using-semaphores-set-1/
- https://www.geeksforgeeks.org/petersons-algorithm-in-process-synchronization/
- https://www.geeksforgeeks.org/hardware-synchronization-algorithms-unlock-and-lock-test-and-set-swap/
- https://www.geeksforgeeks.org/bankers-algorithm-in-operating-system-2/
- http://lad.dsc.ufcg.edu.br/so/silber/