DISTRIBUTED OPERATING SYSTEM

ITA 5006 - Project Report

WINTER SEMESTER 2021-22

Implementation Of Deadlock Avoidance And Synchronization Algorithms

by

21MCA0010 - HURAIS ADNAN C
21MCA0012 – PANTULA SARALA
21MCA0278 – MD NOUMAN S
21MCA0071 – GAYATHRI PASUPULETI

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

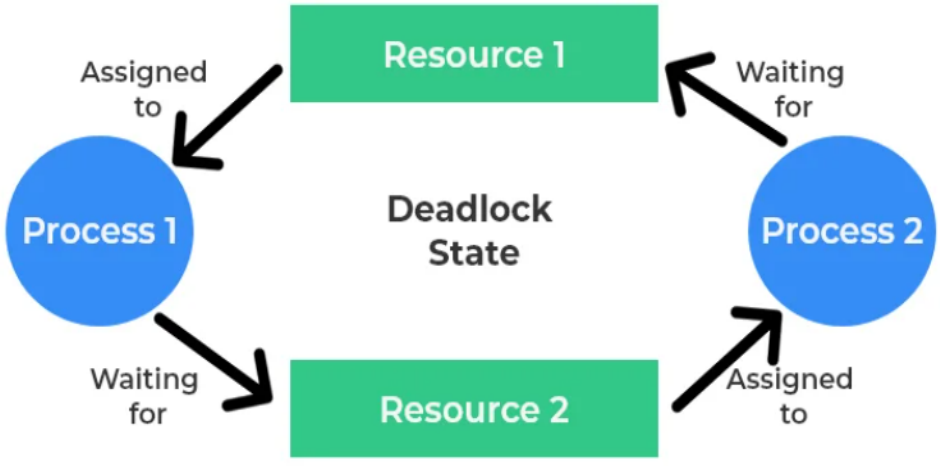
SITE

June, 2022

Abstract:
When there is a group of processes waiting for a resource that is held by other processes in the same set a deadlock takes place. Deadlocks are the set of obstructed processes every one holding a resource and waiting to obtain a resource that is held by another process. The processes in a deadlock may wait for infinite time for the resources they need and never end their executions and the resources which are held by them are not accessible to any other process which is depending on that resource. The existence of deadlocks should be organized efficiently by their recognition and resolution, but may occasionally lead the system to a serious failure. After suggesting the detection algorithm the deadlock is prevented by a deadlock prevention algorithm whose basic step is to allocate the resource only if the system will not go into a deadlock state. This step prevents deadlock easily. In our project, we are going to analyse some deadlock prevention algorithms and will implement them to analyse which one is more suitable for which type of situation.

Keywords:
Deadlock, Concurrency, Critical Section, Mutual Exclusion, Bounded Wait

Introduction:
A deadlock is a situation in which two computer programs sharing the same resource are effectively preventing each other from accessing the resource, resulting in both programs ceasing to function.



Deadlock can arise if the following four conditions hold simultaneously (Necessary Conditions)
Mutual Exclusion: Two or more resources are non-shareable (Only one process can use at a time)
Hold and Wait: A process is holding at least one resource and waiting for resources.
No Pre-emption: A resource cannot be taken from a process unless the process releases the resource.
Circular Wait: A set of processes are waiting for each other in circular form.

If a system does not employ either a deadlock prevention or deadlock avoidance algorithm then a deadlock situation may occur. In this case-

- We have to apply an algorithm to examine the state of the system to determine whether deadlock has occurred or not.
- Apply an algorithm to recover from the deadlock.

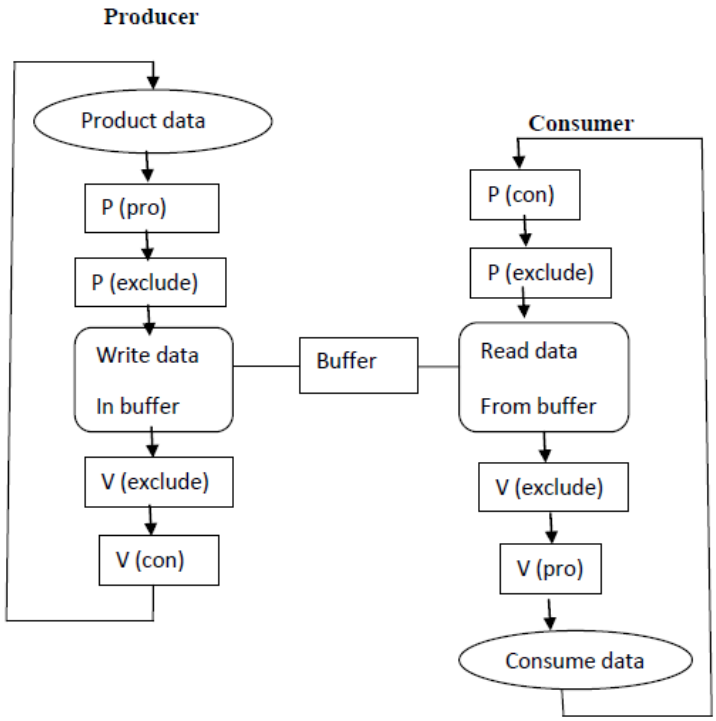
Requirement specification:

Hardware Requirements

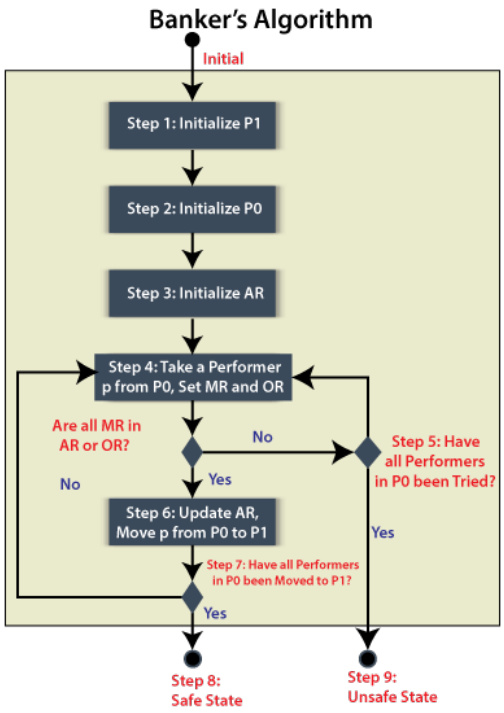
Processor	1.6 GHz or Faster Processor
RAM	1.5 GB
Disk Space	4GB of Available Hard Disk
Graphic	DirectX 9-Capable Video Card
Display	1024 X 768 or Higher Resolution

Design Methodologies:

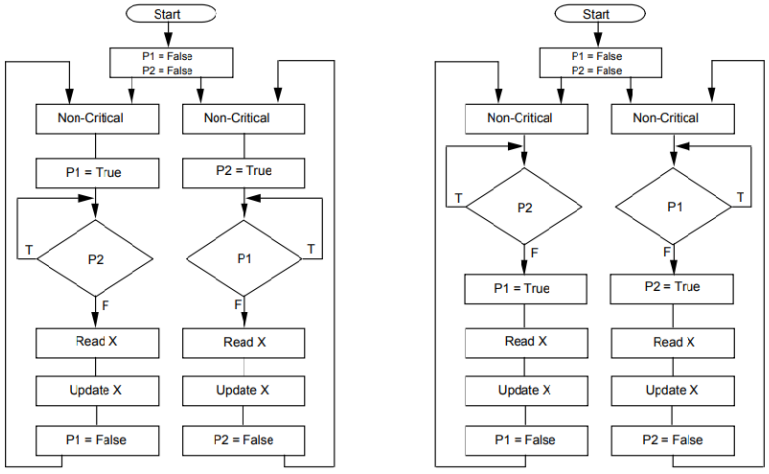
Producer-Consumer :



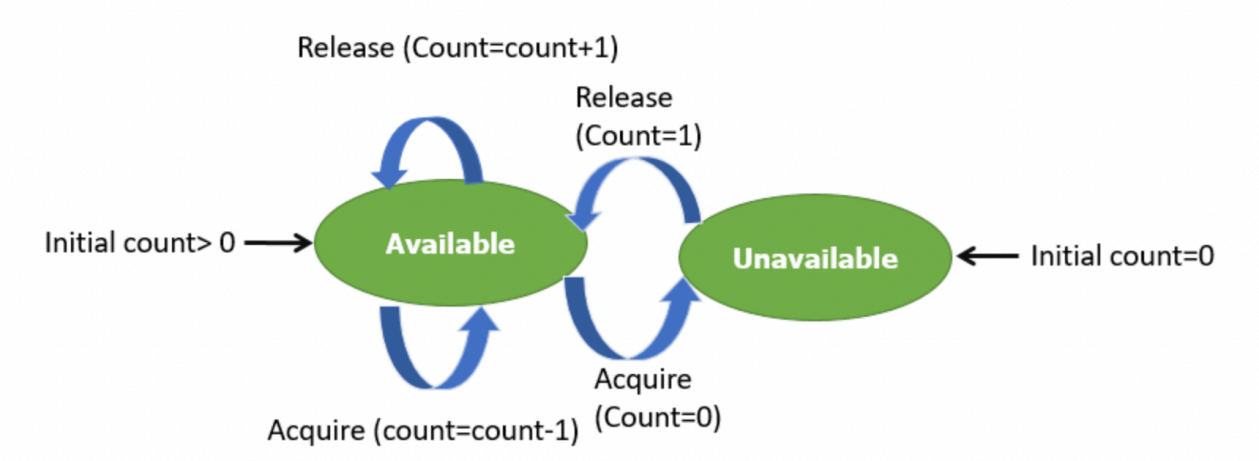
Banker's Algorithm:



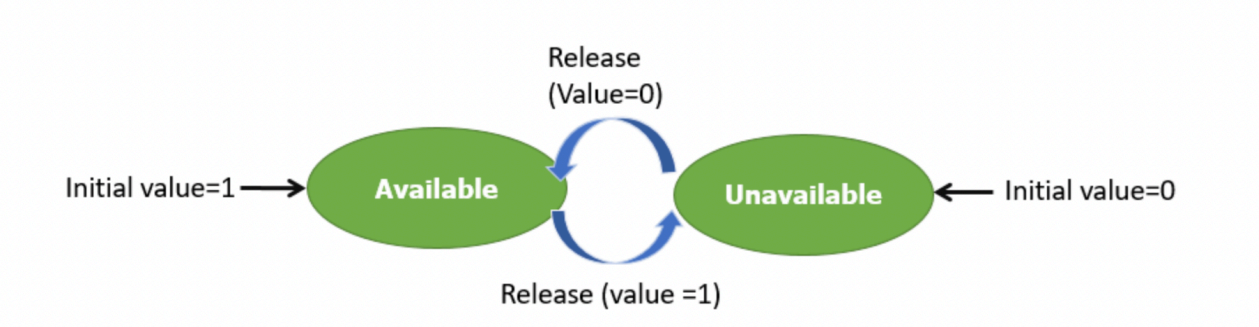
MUTEX :



Counting semaphores:



Binary Semaphores :



Lock variables :

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

Tools Used:

Operating System	Windows 10
Front End	HTML, CSS, JS
Frameworks/Library	Bootstrap, Font Awesome
Text Editor	Visual Studio
Web Browser	Google Chrome

Implementation:

The project gives a clearcut idea about how deadlock avoidance and synchronization algorithms prevent the occurrence of a deadlock.

We have visualized several deadlock algorithms using animations with the help of HTML, CSS & JavaScript.

Steps :

- Making a UI with the help of HTML
- Designing the HTML elements using CSS
- Using Javascript to implement the Algorithms.
- Animations are made responsive with the help of Javascript by manipulating the HTML DOM elements.

Source Code : (Only 1 module**)**

Mutex :

```
document.addEventListener('DOMContentLoaded', ()=>
{
   let stage = document.querySelector('#stage')
   for(let i = 0; i < 5;i++)
   {
      let process = document.createElement('div')
      process.className = "process"
      process.id = "P" + i
      process.innerHTML = `<span style="color:white;">P${i}</span>`
      process.style.animationPlayState = "paused"
      stage.appendChild(process)
   }

   let atCriticalSection = false
   let mutex = 1
```

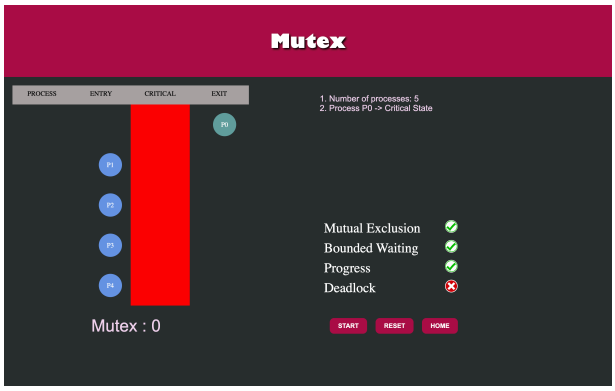
```
let slist = document.querySelector('#slist')
let lst = document.querySelectorAll('.process')
let prop = {}
document.querySelector('#val').innerHTML = mutex

let li = document.createElement('li')
li.innerHTML = `Number of processes: 5`
slist.appendChild(li)

document.querySelector('#strbtn').addEventListener('click',()=>
{
   document.querySelectorAll('.process').forEach((e)=>
   {
      e.style.animationPlayState = "running"
      e.addEventListener('webkitAnimationEnd', ()=>
      {
         document.querySelector('#val').innerHTML = 1
         let li = document.createElement('li')
         li.innerHTML = `Process ${e.id} -> Completed`
         slist.appendChild(li)
         setTimeout(()=>
         {
            mutex = 1
         }, 1000)
         document.querySelector('#critical').style.backgroundColor = "rgb(39, 46, 46)"
      })

      setTimeout(()=>
      {
         e.style.animationPlayState = "paused"
         atCriticalSection = true
      }, 3000)

      let a = setInterval(()=>
      {
         if(atCriticalSection && mutex == 1)
         {
            e.style.animationPlayState = "running"
            mutex = 0
            document.querySelector('#val').innerHTML = mutex
            document.querySelector('#critical').style.backgroundColor = "red"
            let li = document.createElement('li')
            li.innerHTML = `Process ${e.id} -> Critical State`
            slist.appendChild(li)
            clearInterval(a)
         }
      }, 100)
   })

})})
```

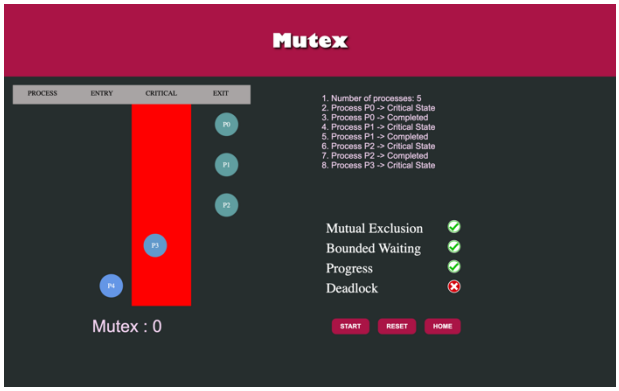
Analysis:

- The deadlock-avoidance algorithm helps you to dynamically assess the resource-allocation state so that there can never be a circular-wait situation. There is no system under-utilization as this method works dynamically to allocate the resources. Deadlock avoidance can block processes for too long. Resource allocation strategy for deadlock prevention is not conservative.
- Synchronization Algorithms prevents multiple processes accessing the critical section at the same time. Changes made in one process aren't reflected when another process accesses the same shared data. The synchronization mechanism of process well solves the numerous problems brought by process concurrency in operating system, but this is not the only method.

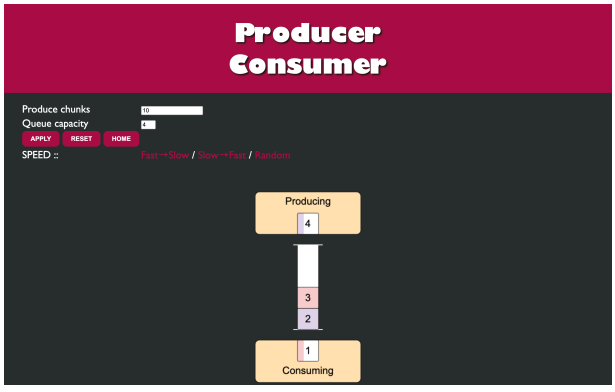
Snapshots:

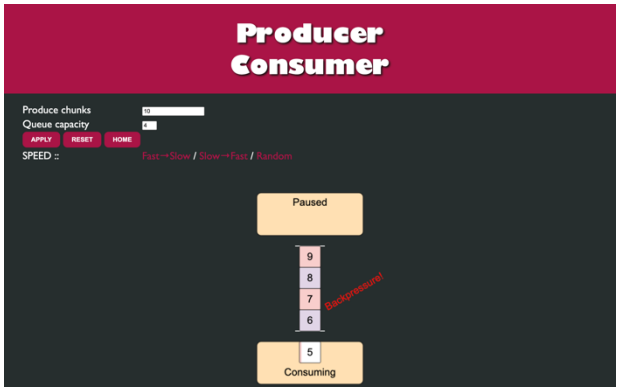
Mutex :





Producer-Consumer :





Conclusion:

Deadlock Avoidance work by letting the Operating System know the Process requirements of resources to complete their execution, and accordingly operating system checks if the requirements can be satisfied or not.

Mutex locks can be used to solve the critical section problem. But it has a drawback, busy waiting which leads to CPU cycle wastage. Hence a better tool is needed which behave like mutex locks but can also provide better methods for processes to synchronize their activities.

Semaphores are greatly used for providing solution to synchronization problems. Semaphores can be easily accessed by two atomic operations: wait and signal and can be implemented efficiently.

References:

- Implementation and Experimentation of Producer-Consumer Synchronization Problem
  [I] Stefano, A. D., Bello, L. L., Santoro, C.1997. Synchronous Producer-consumer transactions for real-time distributed process control. In Proceedings of the IEEE International Workshop on Factory Communication Systems.
  [II] Robbins, S. 2000. Experimentation with bounded buffer synchronization. In Proceedings of the 31st SIGCSE

- Synchronization using counting semaphores
  [I] Michael R. Garey and David S. Johnson. Computers and Intractibility: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, 1979.
  [ii] Zhiyuan Li. A Technique for Reducing Synchronization Overhead in Large Scale Multiprocessors, Center for Supercomputing Research & Development, U. of illinois, May 1985. CSRD Report No. 521.

- Peterson's Mutual Exclusion Algorithm as Feedback Control:
  [i] Alagarsamy K (2005) A mutual exclusion algorithm with optimally bounded bypasses. Information Processing Letters 96(1):36–40