**Department of Computer Science and Engineering**

|  |  |
| --- | --- |
| **Course Code: CSE 321** | **Credits: 1.5** |
| **Course Name: Operating Systems** | **Semester: Fall 18** |

**Lab 09  
Process Synchronization**

1. **Overview:**

Process Synchronization means sharing system resources by processes in such a way that, Concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. Maintaining data consistency demands mechanisms to ensure synchronized execution of cooperating processes. In this lab students will examine several classical process-synchronization problems and explore several ways that are used to solve this problem.

1. **Lesson Fit:**

Programming knowledge is required for this lab.

1. **Learning Outcome:**

Process Synchronization has introduced to handle problems that arose while multiple process executions. After this lab, students will learn different approaches of process synchronization.

1. **Anticipated Challenges and Possible Solutions**
   1. When interaction occurs among processes, student may find difficulties while manipulating the variables.
   2. **Solutions:** Students should have the knowledge about previous algorithm labs.
2. **Acceptance and Evaluation**

Students will show their progress as they complete each task. They will be marked according to their lab performance.

**Activity Detail**

* 1. **Hour: 1  
     Discussion:**

1. Discussion on Critical Section Problem

Critical Section is a code segment that accesses shared variables and has to be executed as an atomic action. It means that in a group of cooperating processes, at a given point of time, only one process must be executing its critical section. If any other process also wants to execute its critical section, it must wait until the first one finishes.

1. Discuss the possible solution to Critical Section Problem
   1. Mutual Exclusion
   2. Progress
   3. Bounded waiting
   4. **Hour: 2**
2. Discussion on **Semaphore**

Managing concurrent processes by using the value of a simple integer variable to synchronize the progress of interacting processes. This integer variable is called **semaphore**. So it is basically a synchronizing tool and is accessed only through two atomic operations, **wait** and **signal** designated by P(S) and V(S) respectively.

1. Make an Example to implement the concept of Semaphore.

**P(S): if S ≥ 1 then S: = S - 1**

**else <block and enqueue the process>;**

**V(S): if <some process is blocked on the queue>**

**then <unblock a process>**

**else S: = S + 1;**

* 1. **Hour: 3**

1. Discussion on Producer Consumer Problem

Producer-consumer problem (also known as the bounded-buffer problem) is a multi-process synchronization problem.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

1. Explain the possible solution of this problem
   1. Producer either go to sleep or discard data if the buffer is full.
   2. The next time the consumer removes an item from the buffer and it notifies the producer who starts to fill the buffer again.
   3. Consumer can go to sleep if it finds the buffer to be empty.
   4. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
2. **Home tasks**

Implement Peterson Solution to solve critical section problem.

In Peterson’s solution, we have two shared variables:

boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section

int turn : The process whose turn is to enter the critical section.

**Lab Activity List**

**Task 1 – Solve Producer Consumer Problem**

**Solution using**[**inter-thread communication**](https://www.geeksforgeeks.org/inter-thread-communication-java/) **(wait(), notify(), sleep()).**

 // **Function called by producer thread**

        public void produce() throws InterruptedException

        {

            int value = 0;

            while (true)

            {

                synchronized (this)

                {

                    // producer thread waits while list

                    // is full

                    while (list.size()==capacity)

                        wait();

                    System.out.println("Producer produced-"

                                                  + value);

                    // to insert the jobs in the list

                    list.add(value++);

                    // notifies the consumer thread that

                    // now it can start consuming

                    notify();

                    // makes the working of program easier

                    // to  understand

                    Thread.sleep(1000);

                }

            }

        }

        // **Function called by consumer thread**

        public void consume() throws InterruptedException

        {

            while (true)

            {

                synchronized (this)

                {

                    // consumer thread waits while list

                    // is empty

                    while (list.size()==0)

                        wait();

                    //to retrive the ifrst job in the list

                    int val = list.removeFirst();

                    System.out.println("Consumer consumed-"

                                                    + val);

                    // Wake up producer thread

                    notify();

                    // and sleep

                    Thread.sleep(1000);

                }

            }

        }

**Task 2-** Try to solve the **Producer Consumer Problem** using **semaphore.**

BufferSize = 3;

Semaphore mutex= 1; // Controls access to critical section

Semaphore empty= Buffer Size; // counts number of empty buffer slots

Semaphore full= 0; // counts number of full buffer slots

Producer ()

{

int item;

while (TRUE)

{

make\_new(item); // create a new item to put in the buffer

down(&empty); // decrement the empty semaphore

down(&mutex);// enter critical section

put\_item(item); // put item in buffer

up(&mutex); // leave critical section

up(&full); // increment the full semaphore

}

Consumer (){

int item;

while(TRUE) {

down(&full); // decrement the full semaphore

down(&mutex); // enter critical section

remove\_item(item); // take a item from the buffer

up(&mutex); // leave critical section

up(&empty); // increment the empty semaphore

consume\_item(item); // consume the item

}

}