**CS 4345: Operating Systems**

**Assignment 2 (Spring 2023)**

***Print your name: \_\_\_\_\_\_\_\_\_\_\_John James\_\_\_\_\_\_\_\_\_\_\_***

**[Submission instructions:** write your name at the top and include answers in this document after each question. Please do not write your answers on a separate document or file. Submit this file through BlazeVIEW assignment submission box.**]**

**Answer the following:**

1. Suppose an IPC system uses *send()* and *receive()* primitives to support communication among processes. The receive primitive is designed as *receive(p)* where the calling process specifies the process from which it is expecting message (in this case, from process p). The calling process blocks if no message is available from the process specified in *receive()*. There may be messages available from other processes, but that does not affect the calling process. It lifts the block only when a message is available from the specified process. Explain whether deadlock is possible in this IPC system. [3 points]

***Answer:*** Yes, deadlock is possible. Deadlock happens when two or more processes are blocked waiting for each other to release resources that they hold. In this example, both processes would call the send() primitive and then call the receive() primitive. Now both processes are waiting for a response from the other process. This waiting can result in a circular dependency, which can lead to a deadlock. In a deadlock both processes would continue to wait for a message from the other process and they would ignore messages from other processes in the system until they are acted on by external intervention.

1. At a grocery’s Deli counter, customers pick up a number (ticket) from a ticket-dispensing machine that generates the number sequentially increasing by one. There are 3 associates working at the Deli counter and they serve the customers in order of their ticket numbers. If an associate is free when a customer takes a ticket, the associate immediately serves the customer. If no associate is free (serving existing customers) when the customer takes a ticket, then the customer waits in a queue. Assuming that a customer takes a ticket immediately after entering the Deli counter and leaves after getting service, write a semaphore-based solution for synchronizing the above scenario. [6 points]

Note: You cannot put restriction on number of customers or on number of tickets.

(Hint: basically you need to write a semaphore-based code for the *customer\_entry()* and *customer\_leave()* methods. You may use more than one semaphore – binary semaphore and counting semaphore, both. You may define additional variables (for example, ticket number).

***Answer:***

import java.util.concurrent.Semaphore;

public class DeliCounter {

private final Semaphore ticketSemaphore = new Semaphore(1);

private final Semaphore associateSemaphore = new Semaphore(3);

private int nextTicketNumber = 1;

public void customerEntry() throws InterruptedException {

// Acquire the ticket semaphore to get the next ticket number

ticketSemaphore.acquire();

int ticketNumber = nextTicketNumber++;

ticketSemaphore.release();

// Acquire the associate semaphore to wait for a free associate or get served immediately

associateSemaphore.acquire();

serveCustomer(ticketNumber);

}

public void customerLeave() {

// Release the associate semaphore to signal that a customer has left

associateSemaphore.release();

}

private void serveCustomer(int ticketNumber) throws InterruptedException {

// Serve the customer and release the associate semaphore

System.out.println("Serving customer with ticket number " + ticketNumber);

Thread.sleep(1000); // simulate serving time

customerLeave();

}

}

1. Consider the following synchronization primitives:

|  |  |
| --- | --- |
| Wait (S, T) {  while ( S < T)  block();  } | Go (S) {  S++;  } |

We have one producer process and one consumer process working on a bounded buffer of size N. Both processes use the above synchronization primitives to solve the bounded buffer problem. The two variables, p and c, indicating the produce and consume, are initialized to 0. The pseudocode for the **producer** process is as follows:

while(true){

.. some method to produce/create a data item;

Wait(N-1, p-c);

.. some method to place the data into the buffer;

Go(p);

}

Suggest the corresponding consumer process pseudocode to complete the solution. [5 points]

(Hint: similar to the producer code, you can use *.. some method to remove data from buffer;* and *.. some method to consume/process data item;* in the consumer code. Basically, your job is to include the Wait() and Go() with appropriate arguments with these two lines. )

***Answer:***

while(true){

Wait(p-c, 0);

.. some method to remove data from buffer;

.. some method to consume/process data item;

Go(c);

}

1. Suppose there are two processes – i and j, sharing atomic variables turn and flag[2]. Consider the following solution to critical section problem for the processes. This code is for process i and process j has similar code.

**do *{***

**flag[i] = true;**

**while (flag[j]) *{***

**if (turn == j) *{***

**flag[i] = false;**

**while (turn == j); /\* do nothing \*/**

**flag[i] = true;**

***}***

***}***

**critical\_task(); /\* critical region \*/**

**turn = j;**

**flag[i] = false;**

**other\_tasks(); /\* remainder section \*/**

***}* while (true);**

Examine whether the above solution satisfies all three conditions of critical section problem for two processes i and j. [6 points]

**(***Hint: check whether mutual exclusion, progress, and bounded waiting are satisfied. Just writing ‘yes’, or ‘no’ is not sufficient. You need to provide justification.*)

***Answer:***

Yes all three are satisfied.

Mutual exclusion: This is satisfied because only one process can execute the critical section at a given time. When one process enters the critical region, it sets its flag to true and then checks if the other process’s flag is set to true, if it is it waits until that processes sets its flag to false and then it can execute.

Progress: This is satisfied because of the turn variable. The turn variable allows for both processes a chance to execute the critical task. When one process completes its critical region, it sets the turn variable to the other process.

Bounded waiting: This is satisfied because of the while loop in the code. The while loop allows that a process will eventually reach the critical region and won’t wait indefinitely. The program constantly checks the flag of the other process and the turn variable, and will only allow the process to enter the critical region until both conditions are met.