CSE 3201: Operating System

Assignment 01: Investor Producer Synchronization

Jubayer Rahman (FH-06)

Farhan Fuad Haque (AE-35)

Nafisa Naznin (RH-41)

Table of Contents

Investor Producer Synchronization

Problem Statement

The given problem is an investor-producer synchronization problem. We have to deal with the critical regions so that the concurrent processes can be executed properly. Customers will order products from the producer. After accepting the requested order, the producer takes a loan from the bank and then supplies the items. Customers have to pay before receiving their orders. At the end, the bank takes the money back from the producer with additional charges. Now, there are a lot of concurrent processes going on. We have to make sure the threads execute without any race condition or deadlock.

Definitions

Critical Region

The segment which has the shared code or variables is called the critical region. Every process has to run an *entry section* to get access to the region and an *exit section* when done.

Semaphore

Semaphores are integer variables. They can be manipulated in many ways to make sure that the critical region is safe. A semaphore mostly uses two functions, P() and V(). P() is used to decrease the semaphore value and V() increases the semaphore value.

Components of the System

Customer Thread

order item():

Customers order different items. For the system we designed, the structure item has several attributes. Among them, the *i_price*(item price) and *item_quantity*(item quantity) attributes are

given by the customer. This function takes an *item* as input. After taking the *item* we add the new item to our linked list and increase the value of the total item by 1. We used semaphore *sem_queue* to make sure that no other process is working with the item list at that moment.

consume_item():

This function takes *customernum* as an input. We check the *req_serv_item* list first. After that, we try to see if any items are requested by this customer. If the order has already been *SERVICED* we just add the money for the correspondent customer in our *customer_spending_amount[customernum]*. There are two shared entities used in this function. One is the *customer_spending_amount[]* array and another one is *req_serv_item*. We used semaphore *sem_consume[]* and *sem_queue* for them respectively.

end shopping():

This function is executed when a customer ends shopping. We keep track of the total customers by using the *customers* variable. So, if we find the value of the variable equal to 0, we can just stop producing. The *customer's* variable is shared. We used *sem_customer* semaphore to lock it. As we are stopping all producers, we used *sem_producer_take_order* semaphore for synchronization.

Producer Thread

take order():

This function waits for customers to order. If there isn't any customer(the *customer* variable is 0) it stops. Then the function looks in the *req_serv_item* list. If there is an item which has been requested this function returns the item. Otherwise, it keeps searching. If the *customer* variable is 0 we are stopping the producer. So, this process is shared with the end shopping() process. As before, we used *sem_producer_take_order* to synchronize them. We also stopped the producer from producing the same item. To do this, we used *sem_producer_req_item* semaphore.

produce item():

This function produces an item. Mostly the producer sets a price in *i_price* variable according to adjusting the profit and bank loan.

serve_order():

This function is pretty self explanatory. Producer serves the ordered item to the customer. After serving, the *order_type* attribute is set to *SERVICED*. We update the *total_served[]* array, which stores how many orders have already been served to each customer. We also make sure that

all the previous orders requested by the customer were already served. For this, we used *sem_consume[]*, an array of semaphores.

calculate loan amount():

This function helps the producer to calculate the bank loan. After deciding the prices of the items in the producer_item() function, we calculate and return the total loan here.

loan_request():

This function takes two parameters. The total loan*(amount*) that is being requested and the producer(*producernumber*) who is requesting it. Then, it selects a random bank to request the loan from. We also implemented a critical region here, which works with the bank account attributes. In this region we update *acu_loan_amount*(accumulated loan amount), *prod_loan[producernumber]*(producer's current loan amount for this bank), *remaining_cash*(the actual cash ready for investment*)* attributes. We use the semaphore *sem_bank_account* to make sure this region executes properly.

loan_reimburse():

The producer has to pay his debt to the bank at the end. This function calculates the total amount of money the producer has to give back(with additional charge). We update *acu_loan_amount*(accumulated loan amount), *prod_loan[producernumber]*(producer's current loan amount for this bank), *remaining_cash*(the actual cash ready for investment*)* attributes here. This is a critical region and we use the semaphore *sem_bank_account* to deal with this.

initialize():

We initialize the important variables here. We also create the semaphores with appropriate values.

finish():

We destroy the semaphores and free the used memory after everything is done.

Semaphores

sem_queue:

This semaphore is used to lock the *req_serv_item* linked list to ensure that only a single thread works on the item list at a given time.

sem_producer_take_order:

The producer thread should wait while a customer is placing an order. The *sem_producer_take_order* semaphore ensures that the producer thread is waiting and resuming properly. It is also used to enable synchronization when a customer ends shopping and the producer thread needs to stop.

sem_customer:

Since the *customer* variable is shared between multiple threads, the *sem_customer* semaphore is necessary to avoid potential synchronization issues like race condition.

sem_bank_account:

Similar to the previous semaphore, *sem_bank_account* is used to ensure that no two threads are operating on the bank account array simultaneously.

sem_producer_req_item:

To avoid multiple producers producing the same order item, *sem_producer_req_item* semaphore is used to restrict the variable *req_serv_item* for other producers before a producer accesses it so that other producers cannot retrieve the same item in the midway.

sem_consume[NCUSTOMER]:

sem_cosume is an array of semaphores dedicated to each customer. It keeps track of the previous order of each customer. This semaphore ensures that no customer can place a new order before his/her previous order is served.

sem_total_served:

This semaphore is used to lock the *total_served[]* array to ensure synchronization.