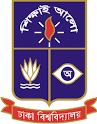
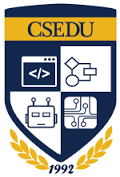
**University of Dhaka**

**Department of Computer Science and Engineering**

CSE-3211, Operating System Lab

3rd Year, 2nd Semester

**Session:** 2018-19

**Assignment No.** 1

**Title of the Assignment:** Investor-Producer Synchronization Problem.

**Lab Group**: A

**Submitted by:**

1. Meheraj Hossain (Roll - 24)

2. Rahat Rizvi Rahman (Roll - 37)

3. Md. Fahim (Roll - 26)

**Date:** 29th October, 2019

**Submitted to:**

1. Dr. Mosaddek Hossain Kamal Tushar, Professor, Dept. of CSE, DU
2. Dr. Md Mamun-or-Rashid, Professor, Dept. of CSE, DU

# Problem definition:

In this problem, a customer requests the producer to supply some items to accomplish their every day’s need; by accepting the request, the producer loan money from the bank and use to produce the items. Bank finance the business and get the money back from the producer with a service charge. The producer decides the item price combining bank service charges and profit.

Customer pays before the consumption of the requested items. Then, they finish shopping and back home and sleep. On the other hand, bank and producer evaluate the business at the end of the day.

However, investor (bank) and producer discovered that the balance sheet

is inconsistent. There were found serious discrepancies due to the mismanagement of demand, supply and financial transactions.

# Proposed Solution:

As the customer and producer threads both use a shared resource to order and serve items at the same time, there may arise a concurrency issue.

To solve this problem first we identified the critical sections of the program and implement semaphores to solve the synchronization problem and to avoid deadlock.

# Theory:

**Critical region:**

* A critical region is a region of codes where shared resources are accessed.
* Also called critical sections.
* Uncontrolled access to critical region results in race condition. As a result, it creates concurrency problem.
* To solve this type of concurrency issue, we need to control access to the critical region by implementing the synchronization techniques.

**Mutual Exclusion:** No two processes can access the critical region simultaneously.

**Semaphore:**

To solve concurrency issue, this technique was first introduced by dijkstra in 1965. This technique has two primitives which are atomic.

* 1. P() : proberen, also known as down or wait
  2. V() : verhogen, also known as up or signal

These two primitives are more powerful than simple sleep and wakeup alone.

**Semaphore definition:**

typedef struct {

int count;

struct process \*L;

}semaphore;

Semaphore operations are defined as:

P(S):

S.count-- ;

If ( S.count < 0 ){

Add this process to S.L;

Sleep ;

}

V(S):

S.count++ ;

If ( S.count <= 0 ){

Remove a process P from S.L;

wakeup(P) ;

}

# The components of the system:

The system deploys a multi-threading mechanism in the OS161 kernel mode with numerous functionalities.

## Critical region:

1. req\_serv\_item(Linked-list): We inserted customer’s ordered items and also producer’s served items in this linked-list.
2. bank\_account[] (Array): All the transactions between producer and bank are done via this array.

## Customer thread :

1. Order\_item (): In this function, we allocated memory for each item one by one and inserted orders in “req\_serv\_item” linked list.
2. Consume\_item(): In this function, a customer consumes the orders he/she has placed at a time. So we deleted those consumed items from “req\_serv\_item” (linked-list) and also updated each customer’s spending amount.
3. end\_shopping(): We kept this function unchanged.

## Producer thread :

1. take\_order(): In this function, each producer takes one item at a time and changes each orders “order\_type” as “ORDER\_TAKEN” for production.
2. calculate\_loan\_amount(): In this function, each producer calculates the loan amount needed from bank which is equivalent to (item\_quantity \* ITEM\_PRICE).
3. loan\_request(): In this function, a producer chooses a bank depending on the bank’s remaining cash and takes loan from the bank i.e updates the “bank\_account” array.
4. produce\_item(): We kept this function unchanged.
5. serve\_order(): In this function, each producer calculates the item price for customer and also updates “order\_type” as “SERVICED” so that a customer can consume the order. This function also updates producer’s income.
6. loan\_reimburse(): In this function, each producer repays the loan it took from the bank to produce the item and updates “bank\_account” array.

## Resolving the synchronization problem using semaphores:

1. There are two critical sections in our program. One is “req\_serv\_item” (linked-list) and another one is “bank\_account” (array). So to maintain mutual exclusion we used “item\_mutex” semaphore and “bank\_mutex” semaphore.
   * “item\_mutex” semaphore controls the access to the “req\_serv\_item”(linked-list) so that no two producer or two customer can access the linked list at the same time.
   * “bank\_mutex” semaphore controls the access to the “bank\_account” (array) so that no two producer can access the bank at the same time.
2. There are another two semaphore used in this program. They are “order\_take\_full” and “serv\_con\_full”. All semaphores are initialized in “initialize()” function.
3. In “order\_item()” function, it waits until it gets the access to “req\_serv\_item” (linked-list) from “item\_mutex”. After getting the access it inserts ordered items one by one in the linked-list. After the insertion of each item, it signals the “take\_order” function via “order\_take\_full” semaphore. When all the orders are queued in the linked-list, this function releases the access to the linked list by giving a signal via “item\_mutex”.
4. In “take\_order()” function, at first it waits for the signal from “order\_take\_full” semaphore. After getting the signal it waits for the signal from “item\_mutex” to access the “req\_serv\_item” (linked list). When it gets the signal, it takes an order from the linked-list and update the “order\_type” as “ORDER\_TAKEN” and returns the item.

Here, it also maintains a static global variable “count\_order\_taken” which is incremented by one while an order is taken. “count\_order\_taken” is used to check whether “count\_order\_taken” is greater or equal to (NCUSTOMER \* N\_ITEM\_TYPE \* 10) and returns NULL if the condition is fulfilled.

1. In “calculate\_loan\_amount()” function, it returns the loan amount by calculating (item\_quantity \* ITEM\_PRICE).
2. In “loan\_request()” function, it waits for the access to the bank using “bank\_mutex”. After getting access it chooses the bank depending on the maximum remaining cash. Then it updates the bank details of the corresponding producer.
3. In “serve\_order()” function, it updates the “item\_price” and calculates the producer’s income. It also changes the item’s “order\_type” as “SERVICED”.  
   It also maintains a “count\_serve\_orders” array for each individual customer and signals “consume\_item()” function via serv\_con\_full semaphore for a particular customer if total served items for that customer is equal to “N\_ITEM\_TYPE”.
4. In “consume\_item()” function, it waits for the signal from “serv\_con\_full” semaphore. After getting both “serv\_con\_full” and “item\_mutex” signal it consumes all the served items which are served for that particular customer and removes them from the “req\_serv\_item”(Linked-list).
5. In “loan\_reimburse()” function, it waits for the access to the “bank\_array” from “bank\_mutex”. It repays the loan amount taken by the particular producer to the bank.
6. In finish() function, it destroys all the declared semaphores.