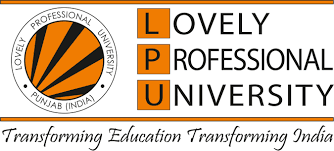
****

**SEMAPHORES AS A SYNCHRONIZATION MECHANISM.**

**Student Name: K Rohit Satya**

**Student ID: 11706990**

**Section: E1708**

**Roll no: 41**

**Email Address: rohitsatya45@gmail.com**

**GitHub Link:**

**Code : semaphores as a synchronization mechanism.**

**Submitted to :**

**Dr Aarti**

**QUESTION:**20.A new shop opened recently on Lovely Street. It is extremely popular and you have been asked to help organize how the shoppers can enter and leave the shop. You must use semaphores as your synchronization mechanism for this problem. Before a shopper can enter, they must call EnterShop(), and this function blocks until it is ok to enter the shop. To enter the shop, the shopper then calls ShopForAWhile(). This function is already written and provided to you; it returns when you are done shopping. To indicate to the rest of the world that they are done shopping, a shopper must call LeaveShop(). You will write the EnterShop() and LeaveShop()

functions.

So, a left-handed shopper would call these functions in this way:

EnterShop(LEFT);

ShopForAWhile();

LeaveShop(LEFT);

The shopkeeper is a bit peculiar and has some strange rules that you must enforce:

1. Shoppers can only enter the shop in pairs (groups of 2). Each group must have one left-handed person

and one right-hand person. You must wait outside until you have found a pair. This means that both

EnterShop() and LeaveShop() are called with a parameter of LEFT or RIGHT.

2. At most one pair of shoppers can be in the store at the same time.

3. Shoppers can leave the shop one at a time, but both shoppers must leave before the next pair can

enter.

**We can find this problem by using Semiphores as a Synchronization** Mechanism.

# Semaphores in Process Synchronization

Prerequisite: [process-synchronization](https://www.geeksforgeeks.org/process-synchronization-set-1/), [Mutex vs Semaphore](https://www.geeksforgeeks.org/mutex-vs-semaphore/amp/)

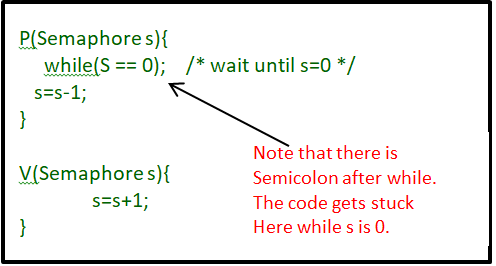
Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

Semaphores are of two types:

1. Binary Semaphore – This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
2. Counting Semaphore – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

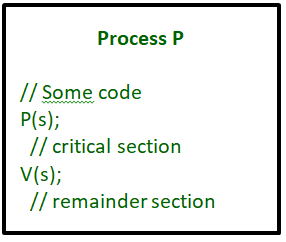
Now let us see how it do so.

First, look at two operations which can be used to access and change the value of the semaphore variable.

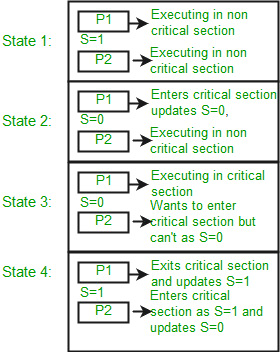


Some point regarding P and V operation

1. P operation is also called wait, sleep or down operation and V operation is also called signal, wake-up or up operation.
2. Both operations are atomic and semaphore(s) is always initialized to one.Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in between read, modify and update no other operation is performed that may change the variable.
3. A critical section is surrounded by both operations to implement process synchronization.See below image.critical section of Process P is in between P and V operation.



Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until s > 0, this can only happen when P1 finishes its critical section and calls V operation on semaphore s. This way mutual exclusion is achieved. Look at the below image for details which is Binary semaphore.

 The description above is for binary semaphore which can take only two values 0 and 1 and ensure the mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one.

Now suppose there is a resource whose number of instance is 4. Now we initialize S = 4 and rest is same as for binary semaphore. Whenever process wants that resource it calls P or wait function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 process P1, P2, P3, P4 and they all call wait operation on S(initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls signal function and value of semaphore becomes positive.

Limitations:

One of the biggest limitation of semaphore is priority inversion.

Deadlock, suppose a process is trying to wake up another process which is not in sleep state.Therefore a deadlock may block indefinitely.

The operating system has to keep track of all calls to wait and to signal the semaphore.

Problem in this implementation of semaphore

Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle

CODE: #include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

#include<errno.h>

#include<sys/ipc.h>

#include<semaphore.h>

time\_t end\_time;/\*end time\*/

sem\_t mutex,shoppers,shopkeepers;/\*Three semaphors\*/

int count=0;/\*The number of shoppers waiting \*/

void shopkeeper(void \*arg);

void shopper(void \*arg);

int main(int argc,char \*argv[])

{

pthread\_t id1,id2;

int status=0;

end\_time=time(NULL)+20;/\* Shop Hours is 20s\*/

/\*Semaphore initialization\*/

sem\_init(&mutex,0,1);

sem\_init(&shoppers,0,0);

sem\_init(&shopkeepers,0,1);

/\*shopkeeper\_thread initialization\*/

status=pthread\_create(&id1,NULL,(void \*)shopkeeper,NULL);

if(status!=0)

perror("create shopkeeper is failure!\n");

/\*shopper\_thread initialization\*/

status=pthread\_create(&id2,NULL,(void \*)shopper,NULL);

if(status!=0)

perror("create shopper is failure!\n");

/\*shopper\_thread first blocked\*/

pthread\_join(id2,NULL);

pthread\_join(id1,NULL);

exit(0);

}

void shopkeeper(void \*arg)/\*shopkeeper Process\*/

{

while(time(NULL)<end\_time || count>0)

{

sem\_wait(&shopper);/\*P(shopper)\*/

sem\_wait(&mutex);/\*P(mutex)\*/

count--;

printf("shopkeeper:shoppers inside ,count is:%d.\n",count);

sem\_post(&mutex);/\*V(mutex)\*/

sem\_post(&shopkeeper);/\*V(shopkeeper)\*/

sleep(3);

}

}

void shopper(void \*arg)/\*shopper Process\*/

{

while(time(NULL)<end\_time)

{

sem\_wait(&mutex);/\*P(mutex)\*/

if(count<N)

{

count++;

printf("Shoppers:add count,count is:%d\n",count);

sem\_post(&mutex);/\*V(mutex)\*/

sem\_post(&shoppers);/\*V(shoppers)\*/

sem\_wait(&shopkeeper);/\*P(shopkeeper)\*/

}

else

/\*V(mutex)\*/

/\*If the number is full of customers,just put the mutex lock let go\*/

sem\_post(&mutex);

sleep(1);

}

}