**Name: Viraj Sahebrao Sonawane**

**Roll No. : 33382**

**Batch: N11**

**Subject: Operating System Lab**

**Assignment No. 4 (PART A)**

**Title:** Thread synchronization using counting semaphores. Application to demonstrate: producerconsumer problem with counting semaphores and mutex.

**Theory:**

What is thread?

A thread is a single sequential flow of execution of tasks of a process so it is also known as thread of execution or thread of control. There is a way of thread execution inside the process of any operating system. Apart from this, there can be more than one thread inside a process. Each thread of the same process makes use of a separate program counter and a stack of activation records and control blocks. Thread is often referred to as a lightweight process.

Need of Thread:

* It takes far less time to create a new thread in an existing process than to create a new process.
* Threads can share the common data, they do not need to use Inter- Process communication.
* Context switching is faster when working with threads.
* It takes less time to terminate a thread than a process.

**Types of Threads**

In the [operating system](https://www.javatpoint.com/os-tutorial), there are two types of threads.

1. Kernel level thread.
2. User-level thread.

# Producer-Consumer problem

The Producer-Consumer problem is a classical multi-process synchronization problem that is we are trying to achieve synchronization between more than one processes.

There is one Producer in the producer-consumer problem, Producer is producing some items, whereas there is one Consumer that is consuming the items produced by the Producer. The same memory buffer is shared by both producers and consumers which is of fixed-size.

The task of the Producer is to produce the item, put it into the memory buffer, and again start producing items. Whereas the task of the Consumer is to consume the item from the memory buffer.

Below are a few points that considered as the problems occur in Producer-Consumer:

* The producer should produce data only when the buffer is not full. In case it is found that the buffer is full, the producer is not allowed to store any data into the memory buffer.
* Data can only be consumed by the consumer if and only if the memory buffer is not empty. In case it is found that the buffer is empty, the consumer is not allowed to use any data from the memory buffer.
* Accessing memory buffer should not be allowed to producer and consumer at the same time.

Program and output for PART A (Producer- Consumer) :

#include <pthread.h>

#include <semaphore.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#define BufferSize 5

sem\_t empty;

sem\_t full;

int buffer[BufferSize];

pthread\_mutex\_t mutex;

int in =0,out = 0;

//display function used for producer

void display\_pro()

{

int i;

if (out == -1)

return;

//If Buffer is Full it will print complete Buffer

else if((out == in + 1) || (out == in ))

{

printf("Queue : ");

int i = out;

int counter = 1;

while(counter <= BufferSize)

{

printf("%d ", buffer[i]);

i = (i + 1) % BufferSize;

counter++;

}

}

else {

printf("Queue : ");

for (i = out; i != in; i = (i + 1) % BufferSize)

{

printf("%d ", buffer[i]);

}

}

}

//Display function for consumer

void display()

{

int i;

if (out == -1)

return;

else {

printf("Queue : ");

for (i = out; i != in; i = (i + 1) % BufferSize)

{

printf("%d ", buffer[i]);

}

}

}

void \*producer(void \*pno)

{

while(1)

{

int item;

for(int i = 0; i < BufferSize; i++)

{

item = rand()%10; // Produce an random item between 0 to 10

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex); //locking the mutex as producer is producing

//Displaying produced item with producer number with it's position in buffer

printf("\nProducer P[%d]: has Produced item %d at Position %d.\n", \*((int \*)pno),item,in);

buffer[in] = item; //inserting item at the end of buffer

in = (in + 1) % BufferSize; //updating end pointer of buffer

display\_pro(); //displaying queue

printf("\n");

sleep(2);

pthread\_mutex\_unlock(&mutex); //unlocking the mutex

sem\_post(&full);

}

}

}

//consumer

void \*consumer(void \*cno)

{

while(1)

{

for(int i = 0; i < BufferSize; i++)

{

sem\_wait(&full);

pthread\_mutex\_lock(&mutex); //Locking the mutex as consumer is going to consume

int item = buffer[out]; //Storing which element is going to remove

//Displaying consumed item with consumer number with it's position in buffer

printf("\nConsumer C[%d]: has Consumed Item %d from Position %d.\n",\*((int \*)cno),item,out);

//Updating start pointer of buffer

out = (out + 1) % BufferSize;

//displaying queue

display();

printf("\n");

sleep(3);

pthread\_mutex\_unlock(&mutex); //unlocking mutex

sem\_post(&empty);

}

}

}

int main()

{

pthread\_mutex\_init(&mutex, NULL); //intializing mutex\_lock

sem\_init(&empty,0,BufferSize); //initializing semaphores

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

int producers, consumers;

//Taking input from user about no of producers and no of consumers

printf("--------Producer Consumer Problem--------\n\n");

printf("Enter number of Producers :");

scanf("%d",&producers);

printf("Enter number of Consumers :");

scanf("%d",&consumers);

pthread\_t pro[producers],con[consumers];

int p[producers], c[consumers];

printf("\nEnter ID of Producer :: \n");

for(int i=0;i<producers;i++)

{

printf("ID of Producer [%d] : ",i+1);

scanf("%d",&p[i]);

}

printf("\n\nEnter ID of Consumer :: \n");

for(int i=0;i<consumers;i++)

{

printf("ID of Consumer [%d] : ",i+1);

scanf("%d",&c[i]);

}

printf("\n\n--------------------------------------------\n");

printf("Press Ctrl + C to stop the execution.");

printf("\n--------------------------------------------\n");

//creating producers

for(int i = 0; i < producers; i++) {

pthread\_create(&pro[i], NULL, (void \*)producer, (void \*)&p[i]);

}

//creating consumers

for(int i = 0; i < consumers; i++) {

pthread\_create(&con[i], NULL, (void \*)consumer, (void \*)&c[i]);

}

//joining producers

for(int i = 0; i < producers; i++) {

pthread\_join(pro[i], NULL);

}

//joining consumers

for(int i = 0; i < consumers; i++) {

pthread\_join(con[i], NULL);

}

//distroying thread and mutex after completion

pthread\_mutex\_destroy(&mutex);

sem\_destroy(&empty);

sem\_destroy(&full);

return 0;

}

Output:

--------Producer Consumer Problem--------

Viraj@ Viraj:~/OSL$ gcc assign4.c –o a4 -lpthread

Viraj @ Viraj:~/OSL$ ./a4

Enter number of Producers :3

Enter number of Consumers :2

Enter ID of Producer ::

ID of Producer [1] : 1

ID of Producer [2] : 2

ID of Producer [3] : 3

Enter ID of Consumer ::

ID of Consumer [1] : 1

ID of Consumer [2] : 2

--------------------------------------------

Press Ctrl + C to stop the execution.

--------------------------------------------

Producer P[3]: has Produced item 3 at Position 0.

Queue : 3

Producer P[3]: has Produced item 5 at Position 1.

Queue : 3 5

Producer P[3]: has Produced item 3 at Position 2.

Queue : 3 5 3

Consumer C[2]: has Consumed Item 3 from Position 0.

Queue : 5 3

Consumer C[2]: has Consumed Item 5 from Position 1.

Queue : 3

Consumer C[1]: has Consumed Item 3 from Position 2.

Queue :

Producer P[1]: has Produced item 7 at Position 3.

Queue : 7

Producer P[1]: has Produced item 6 at Position 4.

Queue : 7 6

Producer P[1]: has Produced item 2 at Position 0.

Queue : 7 6 2

^C

**Assignment 4 (PART B)**

**Title:** Thread synchronization and mutual exclusion using mutex. Application to demonstrate: ReaderWriter problem with reader priority.

**Theory:**

* **Readers Writers Problem in Operating System**

This is a synchronisation problem which is used to test newly proposed synchronisation scheme. The problem statement is, if a database or file is to be shared among several concurrent process, there can be broadly 2 types of users –

Readers – Reader are those processes/users which only read the data

Writers – Writers are those processes which also write, that is, they change the data .

It is allowed for 2 or more readers to access shared data, simultaneously as they are not making any change and even after the reading the file format remains the same.

**The solution of readers and writers can be implemented using binary semaphores**.

We use two binary semaphores "write" and "mutex", where binary semaphore can be defined as:

Semaphore: A semaphore is an integer variable in S, that apart from initialization is accessed by only two standard atomic operations - wait and signal, whose definitions are as follows:

1. wait( S )

{

while( S <= 0) ;

S--;

}

2. signal( S )

{

S++;

}

From the above definitions of wait, it is clear that if the value of S <= 0 then it will enter into an infinite loop (because of the semicolon; after while loop). Whereas the job of the signal is to increment the value of S.

But if one writer(Say w1) is editing or writing the file then it should locked and no other writer(Say w2) can make any changes until w1 has finished writing the file.

Writers are given to be exclusive access to shared database while writing to database. This is called Reader’s writer problem.

**CASE 1:** WRITING - WRITING → NOT ALLOWED. That is when two or more than two processes are willing to write, then it is not allowed.

**CASE 2:** READING - WRITING → NOT ALLOWED. That is when one or more than one process is reading the file, then writing by another process is not allowed. Let us see that our code is working accordingly or not?

**CASE 3**: WRITING -- READING → NOT ALLOWED. That is when if one process is writing into the file, then reading by another process is not allowed.

**CASE 4:** READING - READING → ALLOWED. That is when one process is reading the file, and other process or processes is willing to read, then they all are allowed i.e. reading - reading is not mutually exclusive. Let us see that our code is working accordingly or not?

Program and Output :

#include <pthread.h>

#include <semaphore.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

sem\_t wrt;

pthread\_mutex\_t mutex;

int k = 1;

int numreader = 0;

void \*writer(void \*wno)

{

while(1)

{

printf("\nWriter %d wants to write ",(\*((int \*)wno)));

sem\_wait(&wrt);

k = k+1;

printf("\nWriter %d is writing, value of k to %d",(\*((int \*)wno)),k);

sleep(rand()%4);

sem\_post(&wrt);

printf("\nWriter %d has finished writing\n",(\*((int \*)wno)));

sleep(1);

}

}

void \*reader(void \*rno)

{

while(1)

{

printf("\nReader %d wants to read ",\*((int \*)rno));

// Reader acquire the lock before modifying numreader

pthread\_mutex\_lock(&mutex);

numreader++;

if(numreader == 1) {

sem\_wait(&wrt); // If this id the first reader, then it will block the writer

}

pthread\_mutex\_unlock(&mutex);

// Reading Section

printf("\nReader %d: read k as %d",\*((int \*)rno),k);

sleep(rand()%4);

// Reader acquire the lock before modifying numreader

pthread\_mutex\_lock(&mutex);

numreader--;

if(numreader == 0) {

sem\_post(&wrt); // If this is the last reader, it will wake up the writer.

}

printf("\nReader %d has finished reading \n",\*((int \*)rno));

pthread\_mutex\_unlock(&mutex);

sleep(1);

}

}

int main()

{

pthread\_t read[10],write[5];

pthread\_mutex\_init(&mutex, NULL);

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

int pc,a[10];

//input from user

printf("Enter Number of Reader & Writer? ::");

scanf("%d",&pc);

printf("Enter Reader & Writer :: ");

for(int i=1;i<=pc;i++)

{

scanf("%d",&a[i]);

}

for(int i = 1; i <= pc-1; i++) {

pthread\_create(&write[i], NULL, (void \*)writer, (void \*)&a[i]);

sleep(1);

}

for(int i = 1; i <= pc; i++) {

pthread\_create(&read[i], NULL, (void \*)reader, (void \*)&a[i]);

sleep(1);

}

for(int i = 1; i <= pc; i++) {

pthread\_join(read[i], NULL);

}

for(int i = 1; i <= pc-1; i++) {

pthread\_join(write[i], NULL);

}

pthread\_mutex\_destroy(&mutex);

sem\_destroy(&wrt);

return 0;

}

Output:

Viraj @ Viraj:~/OSL$ gcc assign4B.c –o a4 - lpthread

Viraj @ Viraj:~/OSL$ ./a4

Enter Number of reader & writer? 3

Enter Reader & Writer :: 1 2 3

Writer 1 wants to write

Writer 1 is writing, value of k to 2

Writer 2 wants to write

Reader 1 wants to read

Writer 1 has finished writing

Writer 2 is writing, value of k to 3

Reader 2 wants to read

Writer 1 wants to write

Reader 3 wants to read

Writer 2 has finished writing

Reader 1: read k as 3

Reader 2: read k as 3

Reader 3: read k as 3

Reader 1 has finished reading

Writer 2 wants to write

Reader 3 has finished reading

Reader 1 wants to read

Reader 1: read k as 3

Reader 3 wants to read

Reader 3: read k as 3

Reader 2 has finished reading

Reader 2 wants to read

Reader 2: read k as 3

Reader 3 has finished reading

Reader 2 has finished reading

Reader 1 has finished reading

Writer 1 is writing, value of k to 4

Reader 3 wants to read

Reader 2 wants to read

Writer 1 has finished writing

Writer 2 is writing, value of k to 5

Reader 1 wants to read

Writer 2 has finished writing

Writer 1 wants to write

Writer 1 is writing, value of k to 6

Writer 2 wants to write

Writer 1 has finished writing

Reader 2: read k as 6

Reader 1: read k as 6

Reader 3: read k as 6

Writer 1 wants to write

Reader 1 has finished reading

Reader 2 has finished reading

Writer 2 is writing, value of k to 7

Reader 3 has finished reading

Reader 1 wants to read

Reader 2 wants to read

Reader 3 wants to read

Writer 2 has finished writing

Writer 1 is writing, value of k to 8

Writer 2 wants to write

Writer 1 has finished writing

Reader 1: read k as 8

Reader 2: read k as 8

Reader 3: read k as 8

Reader 1 has finished reading

Reader 3 has finished reading

^Z

[2]+ Stopped ./rw

**Conclusion:** Thus, Producer-Consumer and Reader-Writer problem was implemented successfully using semaphores, mutex and threads. We also learnt relevant concepts of mutual exclusion and critical section in this assignment. pthread library functions were used and understood as well.