Assignment 4A

Name: Kumar Deshmukh

Roll: 33143

Title:

Semaphores and Mutex

Problem Statement:

Implement C program to demonstrate producer-consumer problem with counting semaphores and mutex

Theory:

Semaphores:

An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore.

Semaphores are the OS tools for synchronization. Two types:

1. Binary Semaphore.
2. Counting Semaphore.

Counting semaphore

The counting semaphores are free of the limitations of the binary semaphores. A counting semaphore comprises:

An integer variable, initialized to a value K (K>=0).

During operation it can assume any value <= K, a pointer to a process queue. The queue will hold the PCBs of all those processes, waiting to enter their critical sections. The queue is implemented as a FCFS, so that the waiting processes are served in a FCFS order.

Operation of a counting semaphore:

1. Let the initial value of the semaphore count be 1.
2. When semaphore count = 1, it implies that no process is executing in its critical section and no process is waiting in the semaphore queue.
3. When semaphore count = 0, it implies that one process is executing in its critical section but no process is waiting in the semaphore queue.
4. When semaphore count = N, it implies that one process is executing in its critical section and N processes are waiting in the semaphore queue.
5. When a process is waiting in semaphore queue, it is not performing any busy waiting. It is rather in a “waiting” or “blocked” state.
6. When a waiting process is selected for entry into its critical section, it is transferred from “Blocked” state to “ready” state. The Producer/Consumer Problem We now examine one of the most common problems faced in concurrent processing: the producer/consumer problem. The general statement is this: there are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a single consumer that is taking items out of the buffer one at a time. The system is to be constrained to prevent the overlap of buffer operations. That is, only one agent (producer or consumer) may access the buffer at any one time. The problem is to make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer. We will look at a number of solutions to this problem to illustrate both the power and the pitfalls of semaphores. To begin, let us assume that the buffer is infinite and consists of a linear array of elements.

Conclusion:

We implemented producer-consumer problem with counting semaphores and mutex

CODE:

```
#include<iostream>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <semaphore.h>

using namespace std;

#define total_thread 15

sem_t semEmpty;
sem_t semFull;
pthread_mutex_t mutexBuffer;
int N;

int count = 0;

void* producer(void* args)
{
   sem_wait(&semEmpty);

   pthread_mutex_lock(&mutexBuffer);

   count++;
   cout<<"Item Produced. Count = "<<count<<endl;

   pthread_mutex_unlock(&mutexBuffer);

   sem_post(&semFull);

   return NULL;
}

void* consumer(void* args)
{
   sem_wait(&semFull);

   pthread_mutex_lock(&mutexBuffer);

   count--;
   cout<<"Item Consumed. Count = "<<count<<endl;
   pthread_mutex_unlock(&mutexBuffer);
   sem_post(&semEmpty);

   return NULL;

}

int main(){

   cout<<"Enter buffer size: ";
   cin>>N;

   pthread_t th[total_thread];

   pthread_mutex_init(&mutexBuffer, NULL);
   sem_init(&semEmpty, 0, N);
   sem_init(&semFull, 0, 0);

   int choice, count1=0, i;
```

```
    do{
       cout<<"\n 1.Producer\n 2.Consumer\n 3.Exit\nChoice: ";
       cin>>choice;
       cout<<endl;
       switch(choice){
          case 1:
                pthread_create(&th[count1],NULL,&producer,NULL);
                count1++;
                sleep(1);
                break;

          case 2:
                pthread_create(&th[count1],NULL,&consumer,NULL);
                count1++;
                sleep(1);
                break;

          case 3:
             break;
       }

    }while(choice!=3 && count1!=total_thread);


    for (i = 0; i < total_thread; i++) {
       if (pthread_join(th[i], NULL) != 0)
          perror("Failed to join thread");

     }

     sem_destroy(&semEmpty);
     sem_destroy(&semFull);
     pthread_mutex_destroy(&mutexBuffer);

     return 0;

}
```

OUTPUT:

```
(base) kumar@pop-os:~/Desktop/OS/Assignments$ g++ -pthread Assig4a.cpp
(base) kumar@pop-os:~/Desktop/OS/Assignments$ ./a.out
Enter buffer size: 5

 1.Producer
 2.Consumer
 3.Exit
Choice: 1

Item Produced. Count = 1

 1.Producer
 2.Consumer
 3.Exit
Choice: 1

Item Produced. Count = 2

 1.Producer
 2.Consumer
 3.Exit
Choice: 2

Item Consumed. Count = 1

 1.Producer
 2.Consumer
 3.Exit
Choice: 2

Item Consumed. Count = 0

 1.Producer
 2.Consumer
 3.Exit
Choice: 2


 1.Producer
 2.Consumer
 3.Exit
Choice: 2


 1.Producer
 2.Consumer
 3.Exit
Choice: 1

Item Produced. Count = 1
Item Consumed. Count = 0

 1.Producer
 2.Consumer
 3.Exit
Choice: 3
```