* Assignment No : 01 *

1] Date:

2] Assignment No : 01

3] Title : Write a program to solve classical problem of synchronization using semaphore & mutex.

4] Problem statement : Implement program in C/C++/ Java/Python to solve classical problems of synchronization using mutex and semaphore.

5] software and hardware requirement.

Software Requirement:
1) IDE (Integrated Development Environment)
2) Compiler.

Hardware Requirement:
1) Windows 10 PC, keyboard, Mouse, (I5 9th gen) (8GB Ram)

6] Learning Objective :
1) To solve the given problem using programming skills.
2) Apply appropriate data structure to solve the given problem.
3) Understand the use of semaphore and mutex

7] Learning Outcome :
1) Understood the classical problem and ways to solve it using semaphore and mutex.
2) Gained problem solving skills.

3) Applied suitable data structure to solve the given problem.
4) Understood the concept of Semaphore and mutex.

8] Theory-Concept in brief:

There are various problem related to synchronization.
1) Bounded buffer problem.
2) Dining philosophers problem.
3) The readers writer problem
Lets discuss the 1) Bounded buffer problem.

In computing, the producer-consumer problem (also known as the bounded-buffer problem) is a classical example of a multi-process synchronizatio problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as queue.
Here,
- The producer's job is to generate item (data), put it into the buffer, and start again.
- At the same time, the consumer is consuming the item (i.e removing it from buffer), one piece at a time.

Problem: 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.

Solution: The producer is to either go to sleep or discard data if the buffer is full. the next

time the consumer removes an item from the buffer, it wakes up the sleeping customer.
An inadequate solution could result in deadlock where processes are waiting to be awakened

Mutex:

A mutex provide mutual exclusion, either the producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by the producer, the consumer needs to wait, and vice-versa.
At any point of time, only one thread can work with entire-benefit. The concept can be generalized using semaphore.

Semaphore:

A semaphore is a generalized mutex. In lieu of a single buffer, we can split the 4kb buffer into four 1 KB buffer (identical resources). A semaphore is ~~generalize~~ can be be associated with these four buffers. The consumer and producer can work on different buffers at the same time.

Strictly speaking, a mutex is locking mechanism used to synchronize access to a resource. Only one task (can be a thread process based on OS abstraction) can acquire the mutex. it means there is ownership associated with a mutex, and only the owner can release the lock (mutex).

Semaphore is signaling mechanism ("im done, you can carry on" kind of signal). For example if we are listening to song (one task) on your mobile and at the same time, your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

1] Algorithm.

* Algorithm for producer's Thread.

1) Start
2) While true repeat steps 5 to 8
3) Produce value 'item' to be inserted in buffer.
4) Make Semaphore wait until the shared buffer is empty (or atleast there is 1 item in the buffer)
5) Lock the mutex
6) The item earlier produced, add it in the buffer.
7) Now, free the locked mutex (unlock it)
8) Notify the threads about the item inserted in Buffer.
9) End.

* Algorithm for Consume's Thread

1) Start
2) While is True, repeat steps 5 to 7
3) Wait until there is atleast a item produced in Buffer.
4) Lock the mutex.
5) Buffer is available now for consumption, Consume item from the buffer.
6) Unlock the mutex.
7) Acknowledge all other threads about consumption of one item from buffers.
8) End.

* Additional Theory.

* Critical Section:

The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e only one process can execute in its critical section at a time. All other processes have to wait to execute in their critical sections.

Rules of critical section:

1) Mutual Exclusion
   It is a special type of binary semaphore which is used for controlling access to shared resource.

2) Process solution:
   It is used when no one is in critical section & someone wants in.

3) Bound waiting.
   After a process makes a request for getting into its critical section, there is a limit for how many other processes can get into their critical section.

14] Application of Assignment:

This Assignment can be used to understand the classical problem of Synchronization And the possible ways to solve it or find the solution using Semaphore and mutex.

Applications: file management, memory management, process management.

15] Conclusion

Understood the concept of Synchronization, Semaphore and mutex and the optimal way to witness the classical problem of consumer and producer.

16] References.

1] Geeks for Geeks.
2] www.javamadesoeasy.com.

Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <iostream>
using namespace std;

#define MAX_SIZE 10
sem_t semEmpty;
sem_t semFull;

pthread_mutex_t mutexBuffer;

class Queue
{
    long long qFront;
    long long qRear;
    long long queue_array[MAX_SIZE];

public:
    Queue() : qFront(-1), qRear(-1) {}
    bool isEmpty()
    {
        return qFront == qRear && qFront == -1;
    }
    bool IsFull()
    {
        return (qRear + 1) % MAX_SIZE == qFront ? true : false;
    }
    void enque(long long x)
    {
        if (IsFull())
        {
            cout << "Queue Full!!\n";
            return;
        }
        else if (isEmpty())
        {
            qFront = qRear = 0;
        }
        else
        {
            qRear = (qRear + 1) % MAX_SIZE;
        }
        queue_array[qRear] = x;
    }
    void deque_element()
    {
        if (isEmpty())
        {
            cout << "Error!! Cannot deque empty queue!!\n";
        }
        else if (qFront == qRear)
        {
            queue_array[qFront] = 0;
            qFront = qRear = -1;
```

```
        }
        else
        {
            queue_array[qFront] = 0;
            qFront = (qFront + 1) % MAX_SIZE;
        }
    }
    long long get_front()
    {
        if (isEmpty())
        {
            cout << "Error!! Cannot return front from empty queue!\n";
            return -1;
        }
        else
        {
            return queue_array[qFront];
        }
    }
    void printStatus()
    {
        if (isEmpty())
        {
            cout << "Buffer Holds Nothing\n\n";
        }
        else
        {
            cout << "\nCurrent State of the buffer :  ";
            for (int i = 0; i < MAX_SIZE; i++)
            {
                cout << queue_array[i] << " ";
            }
            cout << "\n\n";
        }
    }
};
Queue buffer;
int item = 0;
void *producer(void *args)
{
    long tid  = reinterpret_cast<std::uintptr_t>(args);
    while (1)
    {
        if (buffer.IsFull())
        {
            cout <<"\nProducer "<<tid<<" is Waiting for a consumer to
consume\n";
        }
        // Add to the buffer
        sem_wait(&semEmpty);
        cout << "\n----> Producer "<<tid<<" is waiting for the mutex\n";
        pthread_mutex_lock(&mutexBuffer);
        cout << "\n----> Producer "<<tid<<" received the mutex\n";
        buffer.enque(++item);
        cout << "\nproducer "<<tid<<" produced : " <<item<< endl;
        buffer.printStatus();
        pthread_mutex_unlock(&mutexBuffer);
```

```
        sem_post(&semFull);
    }
}

void *consumer(void *args)
{

    long tid  = reinterpret_cast<std::uintptr_t>(args);
    while (1)
    {
        int y;
        if (buffer.isEmpty())
        {
            cout <<"\nConsumer " << tid << " is Waiting for a producer to
produce\n";
        }
        // Consuming (Removing) from the buffer
        sem_wait(&semFull);
        cout << "\n---->Consumer " << tid << " is waiting for the mutex \n";
        pthread_mutex_lock(&mutexBuffer);
        cout << "\n---->Consumer " << tid << " received the mutex \n";
        y = buffer.get_front();
        buffer.deque_element();
        cout << "\nConsumer " << tid << " Consumed " << y << endl;
        buffer.printStatus();
        pthread_mutex_unlock(&mutexBuffer);
        sem_post(&semEmpty);
        // Consumer
    }
}

int main()
{
    int producerCount, consumerCount;
    /*cout << "Enter The Number of Producers : ";
    cin >> producerCount;
    cout << "Enter The Number of Consumers : ";
    cin >> consumerCount;*/

    producerCount = consumerCount =5;
    pthread_t th[producerCount + consumerCount];
    pthread_mutex_init(&mutexBuffer, NULL);
    sem_init(&semEmpty, 0, 10);
    sem_init(&semFull, 0, 0);
    long i;
    for (i = 0; i < producerCount; i++)
    {
        if (pthread_create(&th[i], NULL, &producer, (void *)(i + 1)) != 0)
        {
            cerr << "Failed to create thread";
        }
    }
    for (int j = 0; j < consumerCount; j++)
    {

        if (pthread_create(&th[j + i], NULL, &consumer, (void *)(i + 1)) !=
0)
```

```
        {
            cerr << "Failed to create thread";
        }
    }
    for (i = 0; i < producerCount + consumerCount; i++)
    {
        if (pthread_join(th[i], NULL) != 0)
        {
            cerr << "Failed to join thread";
        }
    }
    sem_destroy(&semEmpty);
    sem_destroy(&semFull);
    pthread_mutex_destroy(&mutexBuffer);
    return 0;
}
```

Output :

```
----> Producer
----> Producer 4 is waiting for the mutex

----> Producer 5 is waiting for the mutex
1 is waiting for the mutex

----> Producer 3 is waiting for the mutex

----> Producer 2 is waiting for the mutex

Consumer 6 is Waiting for a producer to produce

Consumer 6 is Waiting for a producer to produce

Consumer 6 is Waiting for a producer to produce

Consumer 6 is Waiting for a producer to produce

Consumer 6 is Waiting for a producer to produce

----> Producer 4 received the mutex

producer 4 produced : 1

Current State of the buffer :  1 0 0 0 0 0 0 0 0 0
```

```
 ----> Producer 5 received the mutex

 producer 5 produced : 2

 Current State of the buffer :  1 2 0 0 0 0 0 0 0 0


 ----> Producer 5 is waiting for the mutex

 ----> Producer 1 received the mutex

 producer 1 produced : 3

 Current State of the buffer :  1 2 3 0 0 0 0 0 0 0


 ----> Producer 1 is waiting for the mutex

 ---->Consumer 6 is waiting for the mutex

 ---->Consumer 6 is waiting for the mutex

 ---->Consumer 6 is waiting for the mutex

 ----> Producer 4 is waiting for the mutex

 ----> Producer 3 received the mutex
```

```
producer 3 produced : 4

Current State of the buffer :  1 2 3 4 0 0 0 0 0 0


----> Producer 3 is waiting for the mutex

----> Producer 2 received the mutex

producer 2 produced : 5

Current State of the buffer :  1
---->Consumer 6 is waiting for the mutex
 2 3 4 5 0 0 0 0 0


----> Producer 5 received the mutex

producer 5 produced : 6

Current State of the buffer :  1 2 3 4 5 6 0 0 0 0


---->Consumer 6 is waiting for the mutex

----> Producer 2 is waiting for the mutex
```

```
----> Producer 1 received the mutex

producer 1 produced : 7

Current State of the buffer :  1 2 3 4 5 6 7 0 0 0


---->Consumer 6 received the mutex

Consumer 6 Consumed 1

Current State of the buffer :  0 2 3 4 5 6 7 0 0 0


---->Consumer 6 received the mutex

Consumer 6 Consumed 2

Current State of the buffer :  0 0 3 4 5 6 7 0 0 0


----> Producer 5 is waiting for the mutex

---->Consumer 6 received the mutex

Consumer 6 Consumed 3
```

```
Current State of the buffer :  0 0 0 4 5 6 7 0 0 0


---->Consumer 6 is waiting for the mutex

----> Producer 4 received the mutex

producer 4 produced : 8

Current State of the buffer :  0 0 0 4 5 6 7 8 0 0


---->Consumer 6 is waiting for the mutex

----> Producer 3 received the mutex

producer 3 produced : 9

Current State of the buffer :  0 0 0 4 5 6 7 8 9 0


----> Producer 4 is waiting for the mutex

---->Consumer 6 received the mutex

Consumer 6 Consumed 4
```