Assignment No. 4(A)

Date:

TITLE: Thread synchronization using counting semaphores and mutual exclusion using mutex. Application to demonstrate: producer-consumer problem with counting semaphores and mutex.

OBJECTIVE:

- To study use of counting semaphore in Linux.
- To study Producer-Consumer problem of operating system.

SOFTWARE REQUIREMENTS:

1. Ubuntu 16.04
2. GNU C Compiler

THEORY:

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

Semaphore

It is a special type of variable containing 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 of two types −

1. Binary semaphore
2. Counting semaphore

Binary semaphore can take the value 0 & 1 only. Counting semaphore can take nonnegative integer values.

Generalized use of semaphore for forcing critical section

```
semaphore sv = 1;
loop forever
{
    Wait(sv);
    critical code section;
    signal(sv);
    noncritical code section;
}
```

Linux Semaphore Facilities (Binary Semaphore)

A semaphore is created with the sem_init function, which is declared as follows:

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

This function initializes a semaphore object pointed to by *sem*, sets its sharing option and gives it an initial integer value. The *pshared* parameter controls the type of semaphore. If the value of *pshared* is 0, the semaphore is local to the current process. Otherwise, the semaphore may be shared between processes. Here we are interested only in semaphores that are not shared between processes. At the time of writing, Linux doesn't support this sharing, and passing a nonzero value for *pshared* will cause the call to fail.

The next pair of functions controls the value of the semaphore and is declared as follows:

```
#include <semaphore.h>
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);
```

These both take a pointer to the semaphore object initialized by a call to sem_init. The sem_post function atomically increases the value of the semaphore by 1. Atomically here means that if two threads simultaneously try to increase the value of a single semaphore by 1, they do not interfere with each other, as might happen if two programs read, increment, and write a value to a file at the same time. If both programs try to increase the value by 1, the semaphore will always be correctly increased in value by 2.

The sem_wait function atomically decreases the value of the semaphore by one, but always waits until the semaphore has a nonzero count first. Thus, if you call sem_wait on a semaphore with a value of 2, the thread will continue executing but the semaphore will be decreased to 1. If sem_wait is called on a semaphore with a value of 0, the function will wait until some other thread has incremented the value so that it is no longer 0. If two threads are both waiting in sem_wait for the same semaphore to become nonzero and it is incremented once by a third process, only one of the two waiting processes will get to decrement the semaphore and continue; the other will remain waiting. This atomic “test and set” ability in a single function is what makes semaphores so valuable.

The last semaphore function is sem_destroy. This function tidies up the semaphore when we have finished with it. It is declared as follows:

```
#include <semaphore.h>
int sem_destroy(sem_t * sem);
```

Again, this function takes a pointer to a semaphore and tidies up any resources that it may have. If we attempt to destroy a semaphore for which some thread is waiting, we will get an error. Like most Linux functions, these functions all return 0 on success.

Mutex functions:

```
int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

The pthread_mutex_init() function initialises the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

The mutex object referenced by mutex is locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

The pthread_mutex_unlock() function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread_mutex_unlock() is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex.

Conclusion: Thus we have studied and demonstrated producer-consumer problem with counting semaphores and mutex