Assignment no -04
Title : Thread synchronization using counting semaphores and mutual exclusion using mutex

Problem statement :

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

B. Thread synchronization and mutual exclusion using mutex. Application to demonstrate: Reader-Writer problem with reader priority.

OBJECTIVE:

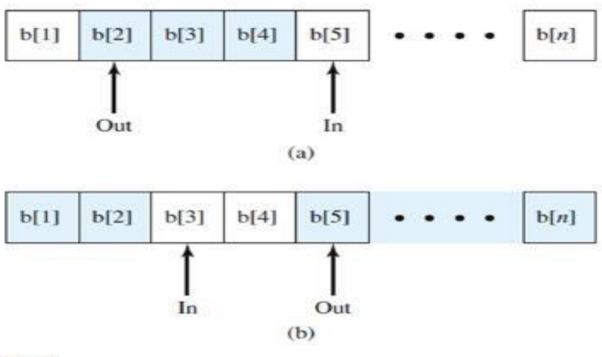
1. To study use of counting semaphore and mutex in Linux.
2. To study Producer-Consumer problem of operating system
3. To understand the use of POSIX threads and semaphore in UNIX.
4. To study reader-writers problem of operating system

Theory :

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.
To begin, let us assume that the buffer is infinite and consists of a linear array of elements. In abstract terms, we can define the producer and consumer functions as follows:

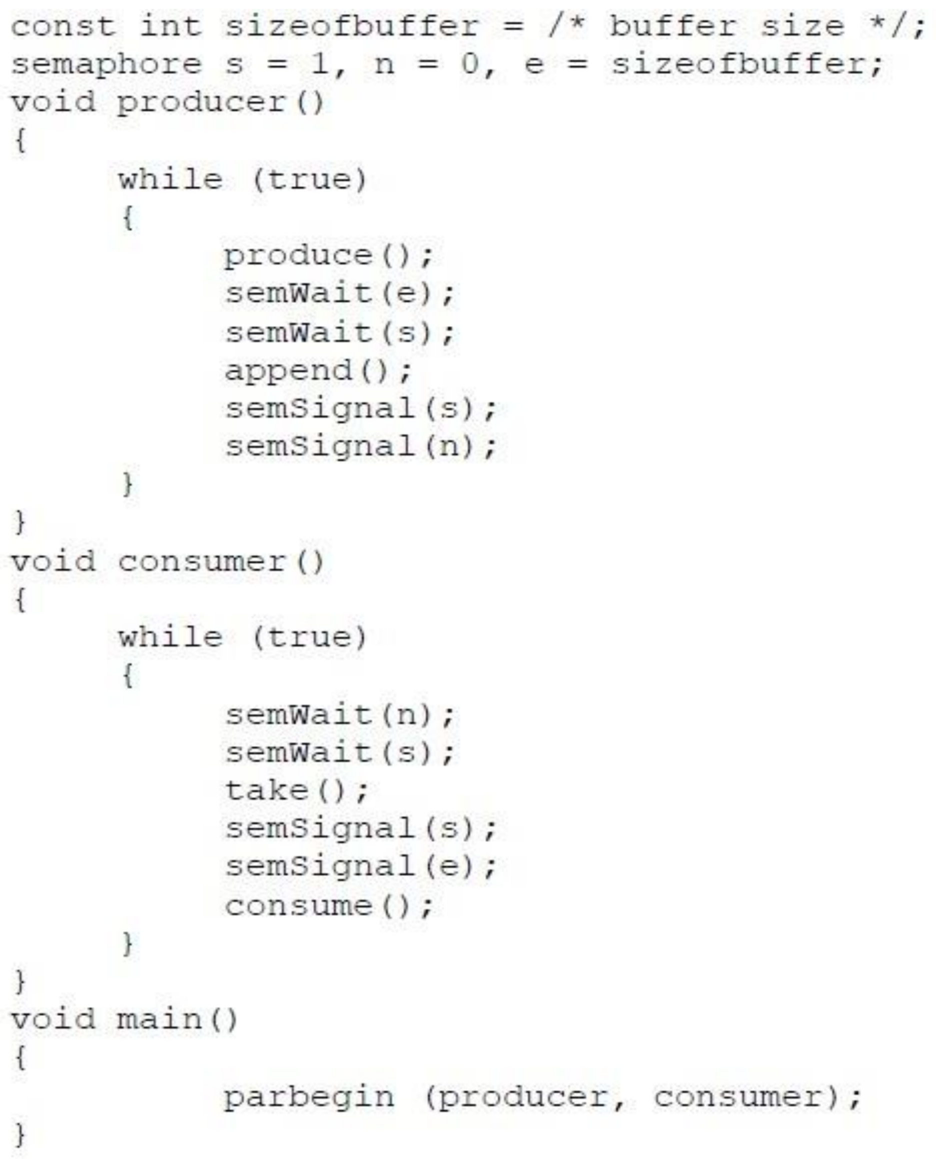
Solution to bounded/circular buffer producer-consumer problem



producer:
```
while (true) {
/* produce item v */;
b[in] = v;
in++;
}
```

consumer:
```
while (true) {
while (in <= out)
/* do nothing */;
w = b[out];
out++;
/* consume item w */;
}
```

Solutions Using Semaphore



```
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
     while (true)
     {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
          parbegin (producer, consumer);
}
```

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.
A semaphore may be initialized to a nonnegative integer value. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

int sem_wait(sem_t **sem*);

The sem_wait() function locks the semaphore referenced by sem by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, then the calling thread will not return from the call to sem_wait() until it either locks the semaphore or the call is interrupted by a signal. The sem_trywait() function locks the semaphore referenced by sem only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.
Upon successful return, the state of the semaphore is locked and remains locked until the sem_post() function is executed and returns successfully.
The sem_wait() function is interruptible by the delivery of a signal.

int sem_post(sem_t *sem);

The *sem_post()* function unlocks the semaphore referenced by *sem* by performing a semaphore unlock operation on that semaphore.
If the semaphore value resulting from this operation is positive, then no threads were blocked waiting for the semaphore to become unlocked; the semaphore value is simply incremented.
If the value of the semaphore resulting from this operation is zero, then one of the threads blocked waiting for the semaphore will be allowed to return successfully from its call to sem_wait().

The semSignal() function will perform a traditional SIGNAL operation on the specified semaphore. The semSignal() mechanism will:

- •If there are threads blocked in the semWait() waiting queue for the semaphore, the first thread will be released to execute.
- •Otherwise the semaphore value will be incremented by 1.

semctl - semaphore control operations

int semctl(int *semid***, int** *semnum***, int** *cmd***, ...);**

semctl() performs the control operation specified by *cmd* on the semaphore set identified by *semid*, or on the *semnum*-th semaphore of that set.

semget - get a semaphore set identifier

int semget(key_t *key***, int** *nsems***, int** *semflg***);**

The **semget**() system call returns the semaphore set identifier associated with the argument *key*. A new set of *nsems* semaphores is created if *key* has the value **IPC_PRIVATE** or if no existing semaphore set is associated with *key* and **IPC_CREAT** is specified in *semflg*.

Reader-Writers Problem:

The readers/writers problem is defined as follows: There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers). The conditions that must be satisfied are as follows:

1. Any number of readers may simultaneously read the file.
2. Only one writer at a time may write to the file.
3. If a writer is writing to the file, no reader may read it.

Thus, readers are processes that are not required to exclude one another and writers are processes that are required to exclude all other processes, readers and writers alike. Before proceeding, let us distinguish this problem from two others: the general mutual exclusion problem and the producer/consumer problem. In the readers/writers problem readers do not also write to the data area, nor do writers read the data area while writing. A more general case, which includes this case, is to allow any of the processes to read or write the data area. In that case, we can declare any portion of a process that accesses the data area to be a critical section and impose the general mutual exclusion solution. The reason for being concerned with the more restricted case is that more efficient solutions are possible for this case and that the less efficient solutions to the general problem are unacceptably slow.

For example, suppose that the shared area is a library catalog. Ordinary users of the library read the catalog to locate a book. One or more librarians are able to update the catalog. In the general solution, every access to the catalog would be treated as a critical section, and users would be forced to read the catalog one at a time. This would clearly impose intolerable delays. At the same time, it is important to prevent writers from interfering with each other and it is also required to prevent reading while writing is in progress to prevent the access of inconsistent information.

This is not a special case of producer-consumer. The producer is not just a writer. It must read queue pointers to determine where to write the next item, and it must determine if the buffer is full. Similarly, the consumer is not just a reader, because it must adjust the queue pointers to show that it has removed a unit from the buffer.

Unix Mutex Facilities:

The way of synchronizing access in multithreaded programs is with mutexes (short for mutual exclusions), which act by allowing the programmer to "lock" an object so that only one thread can access it. To control access to a critical section of code us lock a mutex before entering the code section and then unlock it when we have finished.
The basic functions required to use mutexes are very similar to those needed for semaphores. They are declared as follows:

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t*mutex, const
                    pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t*mutex));
int pthread_mutex_unlock(pthread_mutex_t*mutex); int
pthread_mutex_destroy(pthread_mutex_t*mutex);
```

As usual, 0 is returned for success, and on failure an error code is returned. As with semaphores, they all take a pointer to a previously declared object, in this case a pthread_mutex_t. The extra attribute parameter pthread_mutex_init allows us to provide attributes for the mutex, which control its behavior. The attribute type by default is "fast." This has the slight drawback that, if our program tries to call pthread_mutex_lock on a mutex that it has already locked, the program will block. Because the thread that holds the lock is the one that is now blocked, the mutex can never be unlocked and the program is deadlocked. It is possible to alter the attributes of the mutex so that it either checks for this and returns an error or acts recursively and allows multiple locks by the same thread if there are the same number of unlocks afterward.

Difference Between Semaphore and Mutex

Process synchronization plays an important role in maintaining the consistency of shared data. Both the software and hardware solutions are present for handling critical section problem. But hardware solutions for critical section problem are quite difficult to implement.

The basic difference between semaphore and mutex is that semaphore is a signalling mechanism i.e. processes perform wait() and signal() operation to indicate whether they are acquiring or releasing the resource, while Mutex is locking mechanism, the process has to acquire the lock on mutex object if it wants to acquire the resource.

BASIS FOR COMPARISON	SEMAPHORE	MUTEX
Basic	Semaphore is a signalling mechanism.	Mutex is a locking mechanism.
Existence	Semaphore is an integer variable.	Mutex is an object.
Function	Semaphore allow multiple program threads to access a finite instance of resources.	Mutex allow multiple program thread to access a single resource but not simultaneously.
Ownership	Semaphore value can be changed by any process acquiring or releasing the resource.	Mutex object lock is released only by the process that has acquired the lock on it.
Categorize	Semaphore can be categorized into counting semaphore and binary semaphore.	Mutex is not categorized further.
Operation	Semaphore value is modified using wait() and signal() operation.	Mutex object is locked or unlocked by the process requesting or releasing the resource.
Resources Occupied	If all resources are being used, the process requesting for resource performs wait() operation and block itself till semaphore count become greater than one.	If a mutex object is already locked, the process requesting for resources waits and queued by the system till lock is released.

Conclusion