ASSIGNMENT NO. :04

AIM:

Thread Synchronization

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.

PREREQUISITE:

1. C Programming
2. Fundamentals of Data Structure

OBJECTIVE:

To study

- Semaphores
- Mutex
- Producer-Consumer Problem
- Reader- Writer problem

THEORY:

Semaphores:

Semaphore is 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. It is known as a counting semaphore or a general semaphore. Semaphores are the OS tools for synchronization.

Two types:

1. Binary Semaphore.
2. Counting Semaphore

Binary semaphore

Semaphores, which are restricted to the values 0 and 1 (or locked/unlocked, unavailable / available), are called binary semaphores and are used to implement locks.

It is a means of suspending active processes, which are later to be reactivated at such time conditions are right for it to continue. A binary semaphore is a pointer which when held by a process grants them exclusive use to their critical section. It is a (sort of) integer variable which can take the values 0 or 1 and be operated upon only by two commands termed in English wait and signal.

Counting semaphore

Semaphores which allow an arbitrary resource count are called counting 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.

A counting semaphore can be implemented as follows:

- Initialize – initialize to non-negative integer
- Decrement (semWait)
    - Process executes this to receive a signal via semaphore.
    - If signal is not transmitted, process is suspended.
    - Decrements semaphore value
    - If value becomes negative , process is blocked
    - Otherwise it continues execution.
- Increment (semSignal)
    - Process executes it to transmit a signal via semaphore.
    - Increments semaphore value
    - If value is less than or equal to zero, process blocked by semWait is unblocked

```
struct semaphore {
     int count;
     queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Value of semaphore

- Positive

  Indicates number of processes that can issue wait & immediately continue to execute.

- Zero

  By initialization or because number of processes equal to initial semaphore value have issued a wait Next process to issue a wait is blocked.

- Negative

  Indicates number of processes waiting to be unblocked

  Each signal unblocks one waiting process.

The Producer/Consumer Problem

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:

```
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 */;
}
```

Figure illustrates the structure of buffer b. The producer can generate items and store them in the buffer at its own pace. Each time, an index (in) into the buffer is incremented. The consumer proceeds in a similar fashion but must make sure that it does not attempt to read from an empty buffer. Hence, the

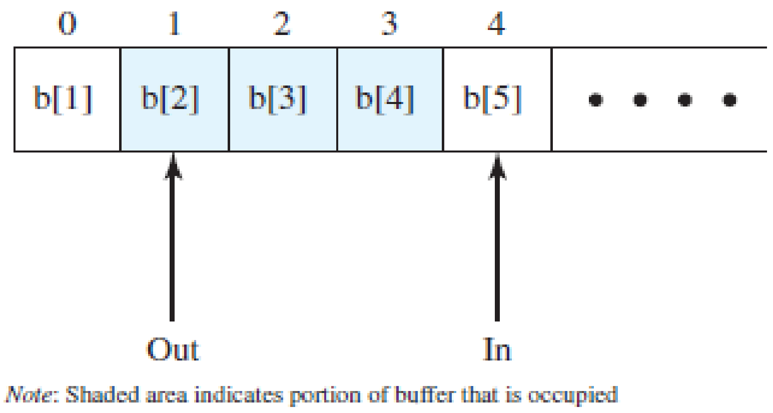

0
1
2
3
4
b[1]
b[2]
b[3]
b[4]
b[5]
• • • •
Out
In
Note: Shaded area indicates portion of buffer that is occupied

Figure: Infinite buffer for producer/consumer problem

Solution for bounded buffer using counting semaphore

```
/* program boundedbuffer */
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);
}
```

The Reader Writer Problem

In dealing with the design of synchronization and concurrency mechanisms, it is useful to be able to relate the problem at hand to known problems and to be able to test any solution in terms of its ability to solve these known problems. 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.

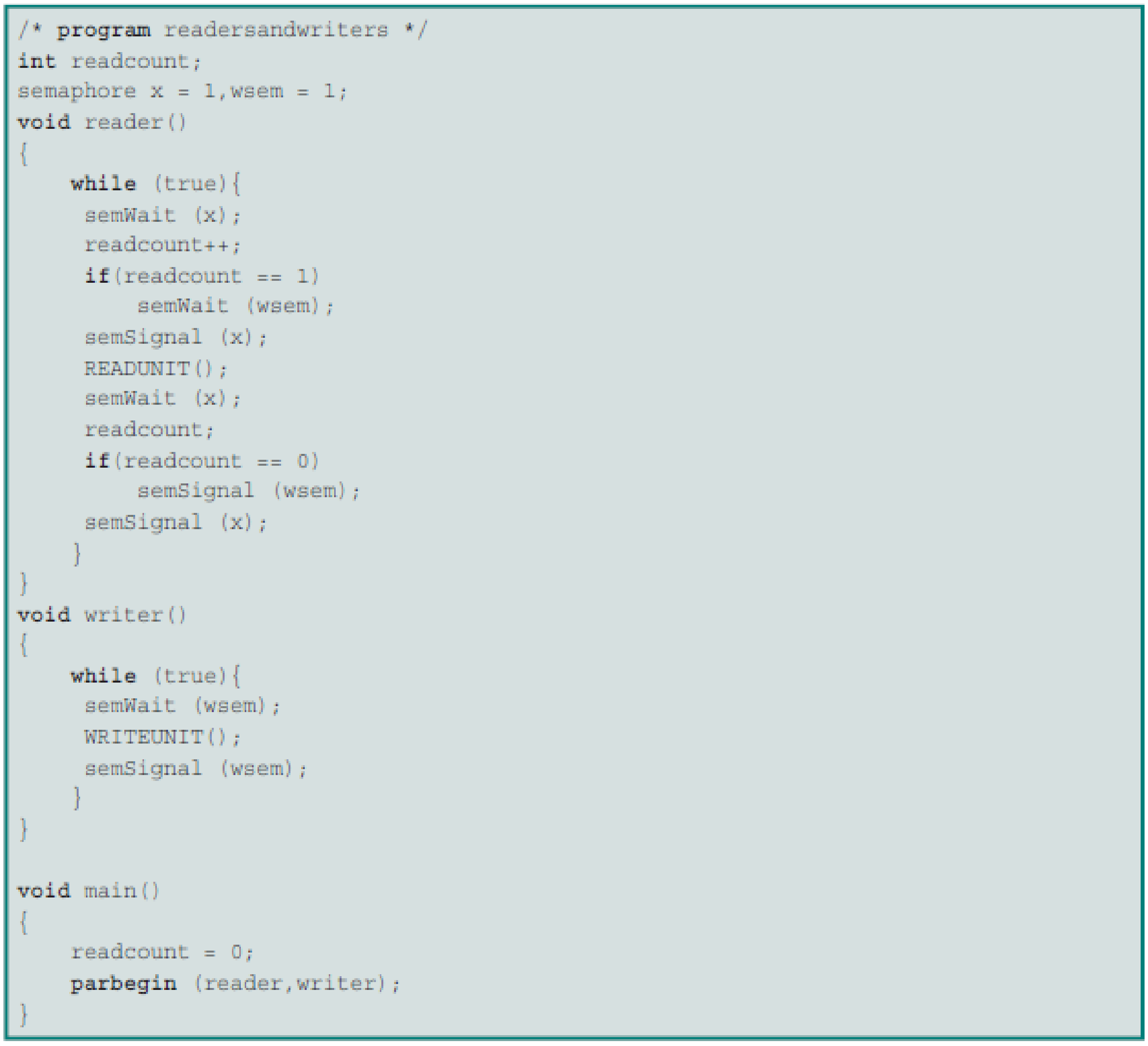
In the readers/writers problem readers do not also write to the data area, nor do writers read the data area while writing.

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.

Readers Have Priority

Figure is a solution using semaphores, showing one instance each of a reader and a writer; the solution does not change for multiple readers and writers. The writer process is simple. The semaphore wsem is used to enforce mutual exclusion. As long as one writer is accessing the shared data area, no other writers and no readers may access it. The reader process also makes use of wsem to enforce mutual exclusion. However, to allow multiple readers, we require that, when there are no readers reading, the first reader that attempts to read should wait on wsem. When there is already at least one reader reading, subsequent readers need not wait before entering. The global variable read count is used to keep track of the number of readers, and the semaphore x is used to assure that read count is updated properly.



```
/* program readersandwriters */
int readcount;
semaphore x = 1,wsem = 1;
void reader()
{
    while (true){
     semWait (x);
     readcount++;
     if(readcount == 1)
         semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount;
     if(readcount == 0)
         semSignal (wsem);
     semSignal (x);
    }
}
void writer()
{
    while (true){
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader,writer);
}
```

POSIX Semaphores

POSIX semaphores allow processes and threads to synchronize their actions. A semaphore is an integer whose value is never allowed to fall below zero. Two operations can be performed on semaphores: increment the semaphore value by one (sem_post(3)); and decrement the semaphore value by one (sem_wait(3)). If the value of a semaphore is currently zero, then a sem_wait(3) operation will block until the value becomes greater than zero.

Semaphore functions:

1. **sem_init()**

   It initializes the unnamed semaphore at the address pointed to by sem. The value argument specifies the initial value for the semaphore.

   int sem_init(sem_t *sem, int pshared, unsigned int value);

2. sem_wait()

It decrements (locks) the semaphore pointed to by sem. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until it becomes possible to perform the decrement.

```
int sem_wait(sem_t *sem);
```

3. sem_post()

It increments (unlocks) the semaphore pointed to by sem. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up and proceed to lock the semaphore.

```
int sem_post(sem_t *sem);
```

1. sem_unlink

It removes the named semaphore referred to by name. The semaphore name is removed immediately. The semaphore is destroyed once all other processes that have the semaphore open close it.

```
int sem_unlink(const char *name)
```

All the above functions returns

0 : Success

-1 : Error

Mutex

Mutexes are a method used to be sure two threads, including the parent thread, do not attempt to access shared resource at the same time. A mutex lock allows only one thread to enter the part that's locked and the lock is not shared with any other processes.

1. pthread_mutex_init()

The function shall initialize the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect shall be the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

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

2. **pthread_mutex_lock()**

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

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

3. **pthread_mutex_unlock()**

   The function shall release 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 shall determine which thread shall acquire the mutex.

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

4. **pthread_mutex_destroy()**

   The function shall destroy the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. A destroyed mutex object can be reinitialized using pthread_mutex_init(); the results of otherwise referencing the object after it has been destroyed are undefined.

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

CONCLUSION:

Thus, we have implemented producer-consumer problem and Reader Writer Problem using 'C' in Linux.

FAQ

1. Explain the concept of semaphore.
2. Explain wait and signal functions associated with semaphores.
3. What do binary and counting semaphores mean?