Experiment Number: 13

TITLE: Reader-Writers Problem (Using Semaphore)

OBJECTIVE:

1. To understand the use of POSIX threads and semaphore in UNIX.
2. To study reader-writers problem of operating system.

Theory:

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.

Solution using semaphore:

```
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);
}
```

Threads

Multiple strands of execution in a single program are called threads. A more precise definition is that a thread is a sequence of control within a process. Like many other operating systems, Linux is quite capable of running multiple processes simultaneously. Indeed, all processes have at least one thread of execution.

POSIX Thread in Unix

Including the file pthread.h provides us with other definitions and prototypes that we will need in our code, much like stdio.h for standard input and output routines.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void*(*start_routine)(void *), void *arg);
```

This function is used to create the thread. The first argument is a pointer to pthread_t. When a thread is created, an identifier is written to the memory location to which this variable points. This identifier enables us to refer to the thread. The next argument sets the thread attributes. We do not usually need any special attributes, and we can simply pass NULL as this argument. The final two arguments tell the thread the function that it is to start executing and the arguments that are to be passed to this function.

```
void *(*start_routine)(void *)
```

We must pass the address of a function taking a pointer to void as a parameter and the function will return a pointer to void. Thus, we can pass any type of single argument and return a pointer to any type. Using fork causes execution to continue in the same location with a different return code, whereas using a new thread explicitly provides a pointer to a function where the new thread should start executing. The return value is 0 for success or an error number if anything goes wrong.

When a thread terminates, it calls the pthread_exit function, much as a process calls exit when it terminates. This function terminates the calling thread, returning a pointer to an object. Never use it to return a pointer to a local variable, because the variable will cease to exist when the thread does so, causing a serious bug. pthread_exit is declared as follows:

```
#include <pthread.h>
void pthread_exit(void *retval);
```

pthread_join is the thread equivalent of wait that processes use to collect child processes. This function is declared as follows:

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

The first parameter is the thread for which to wait, the identifier that pthread_create filled in for us. The second argument is a pointer to a pointer that itself points to the return value from the thread. This function returns zero for success and an error code on failure.

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.

References:

1. "Beginning Linux Programming" by Neil Mathew and Richard Stones, Wrox Publications.
2. "Operating System Internals and Design Implementation" by William Stallings, Pearson Education.

Tushar B Kute
(Subject Teacher)