LAB NO: 7 **Date:**

CLASSICAL PROBLEMS OF SYNCHRONIZATION

Objectives:

2. To synchronize various processes with the use of semaphore.
2. To understand how communication between two processes can take place with the help of named pipe.

For a multithreaded application spanning a single process or multiple processes to do useful work, it is necessary for some kind of common state to be shared between the threads. The degree of sharing that is necessary depends on the task. At one extreme, the only sharing necessary may be a single number that indicates the task to be performed. For example, a thread in a web server might be told only the port number to respond to. At the other extreme, a pool of threads might be passing information constantly among themselves to indicate what tasks are complete and what work is still to be completed.

Data Races

Data race occurs when multiple threads spanning single process or multiple processes use the same data item and one or more of those threads are updating it.

Suppose there is a function update, which takes an integer pointer and updates the value of the content pointer by 4. If multiple threads call the function, then there is a possibility of data race. If the current value of *a is 10, then when two threads simultaneously call update function, then the final value of *a might be 14, instead of 18. To visualize this, we need to write the corresponding assembly language code for this function.

```
void update(int * a)
{
*a = *a + 4;
}
```

Another situation might be when one thread is running, but the other thread has been context switched off of the processor. Imagine that the first thread has loaded the value of the variable **a** and then gets context switched off the processor. When it eventually runs again, the value of the variable a will have changed, and the final store of the restored thread will cause the value of the variable a to regress to an old value. The following code has data race.

```
//race.c
#include <pthread.h>
int counter = 0;
void * func(void * params)
{
        counter++;
}
void main()
{
        pthread_t thread1, thread2;
        pthread_create(&thread1, 0, func, 0);
        pthread_create(&thread2, 0, func, 0);
        pthread_join(thread1, 0 );
        pthread_join(thread2, 0 );
}
```

Using tools to detect data races

We can compile the above code using gcc, and then use Helgrind tool which is part of Valgrind suite to identify the data race.

```
$ gcc –g race.c -lpthread

$ valgrind –tool=helgrind ./a.out
```

Avoiding Data Races

Although it is hard to identify data races, avoiding them can be very simple. The easiest way to do this is to place a synchronization lock around all accesses to that variable and ensure that before referencing the variable, the thread must acquire the lock.

Synchronization Primitives:

Mutex Locks:

A mutex lock is a mechanism that can be acquired by only one thread at a time. Other threads that attempt to acquire the same mutex must wait until it is released by the thread that currently has it.

Mutex locks need to be initialized to the appropriate state by a call to pthread_mutex_init() or for statically defined mutexes by assignment with the PTHREAD_MUTEX_INITIALIZER. The call to pthread_mutex_init() takes an optional parameter that points to attributes describing the type of mutex required. Initialization through static assignment uses default parameters, as does passing in a null pointer in the call to pthread_mutex_init().

Once a mutex is no longer needed, the resources it consumes can be freed with a call to pthread_mutex_destroy().

```
#include <pthread.h>

...

pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2;
pthread_mutex_init( &m2, 0 );
...
pthread_mutex_destroy( &m1 );
pthread_mutex_destroy( &m2 );
```

A thread can lock a mutex by calling pthread_mutex_lock(). Once it has finished with the mutex, the thread calls pthread_mutex_unlock(). If a thread calls pthread_mutex_lock() while another thread holds the mutex, the calling thread will wait, or *block*, until the other thread releases the mutex, allowing the calling thread to attempt to acquire the released mutex.

```
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex;
volatile int counter = 0;
void * count( void * param)
{
        for ( int i=0; i<100; i++)
        {
                pthread_mutex_lock(&mutex);
```

```
                counter++;
                printf("Count = %i\n", counter);
                pthread_mutex_unlock(&mutex);
        }
}
int main()
{
        pthread_t thread1, thread2;
        pthread_mutex_init( &mutex, 0 );
        pthread_create( &thread1, 0, count, 0 );
        pthread_create( &thread2, 0, count, 0 );
        pthread_join( thread1, 0 );
        pthread_join( thread2, 0 );
        pthread_mutex_destroy( &mutex );
        return 0;
}
```

Semaphores:

A semaphore is a counting and signaling mechanism. One use for it is to allow threads access to a specified number of items. If there is a single item, then a semaphore is essentially the same as a mutex, but it is more commonly useful in a situation where there are multiple items to be managed.

A semaphore is initialized with a call to sem_init(). This function takes three parameters. The first parameter is a pointer to the semaphore. The next is an integer to indicate whether the semaphore is shared between multiple processes or private to a single process. The final parameter is the value with which to initialize the semaphore. A semaphore created by a call to sem_init() is destroyed with a call to sem_destroy().

The code below initializes a semaphore with a count of 10. The middle parameter of the call to sem_init() is zero, and this makes the semaphore private to the process; passing the value one rather than zero would enable the semaphore to be shared between multiple processes.

```
#include <semaphore.h>
int main()
{
        sem_t semaphore;
        sem_init( &semaphore, 0, 10 );
        ...
        sem_destroy( &semaphore );
}
```

The semaphore is used through a combination of two methods. The function sem_wait() will attempt to decrement the semaphore. If the semaphore is already zero, the calling thread will wait until the semaphore becomes nonzero and then return, having decremented the semaphore. The call to sem_post() will increment the semaphore. One more call, sem_getvalue(), will write the current value of the semaphore into an integer variable.

In the following program a order is maintained in displaying Thread 1 and Thread 2. Try removing the semaphore and observe the output.

```
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>

sem_t semaphore;

void *func1( void * param )
{
        printf( "Thread 1\n" );
        sem_post( &semaphore );
}
void *func2( void * param )
{
        sem_wait( &semaphore );
        printf( "Thread 2\n" );
}
int main()
{
        pthread_t threads[2];
        sem_init( &semaphore, 0, 1 );
        pthread_create( &threads[0], 0, func1, 0 );
        pthread_create( &threads[1], 0, func2, 0 );
        pthread_join( threads[0], 0 );
        pthread_join( threads[1], 0 );
        sem_destroy( &semaphore );
}
```

1. Classical Problems :

1.1 The Bounded – Buffer Problem

Here the pool consists of *n* buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0. The classical example is the production line.

1.2 The Producer Consumer Problem:

A producer process produces information that is consumed by a consumer process. For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader. The producer-consumer problem also provides a useful metaphor for the client-server paradigm. We generally think of a server as a producer and a client as a consumer. For example, a Web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client Web browser requesting the resource.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

1.3 The Readers Writers Problem:

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database. We distinguish between these two types of processes by referring to the former as readers and to the latter as writers. Obviously, if two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the *readers-writers problem.*

Solution to Producer-Consumer problem

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
int buf[5],f,r;
sem_t mutex,full,empty;
void *produce(void *arg)
{
   int i;
   for(i=0;i<10;i++)
   {
      sem_wait(&empty);
      sem_wait(&mutex);
      printf("produced item is %d\n",i);
      buf[(++r)%5]=i;
      sleep(1);
      sem_post(&mutex);
      sem_post(&full);
      printf("full %u\n",full);
   }
}
void *consume(void *arg)
{
      int item,i;
      for(i=0;i<10;i++)
      {
            sem_wait(&full);
            printf("full %u\n",full);
            sem_wait(&mutex);
            item=buf[(++f)%5];
            printf("consumed item is %d\n",item);
            sleep(1);
            sem_post(&mutex);
            sem_post(&empty);
            int val;
            sem_getvalue(&wt,&val)
if( val <10)
sem_post(wrt);
      }
}
main()
```

```
{
   pthread_t tid1,tid2;
   sem_init(&mutex,0,1);
   sem_init(&full,0,0);
   sem_init(&empty,0,5);
   pthread_create(&tid1,NULL,produce,NULL);
   pthread_create(&tid2,NULL,consume,NULL);
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
}
```

Solution to First Readers-Writers Problem using semaphores:

- The reader processes share the following data structures:

  semaphore mutex , wrt;

  int readcount;

- The binary semaphores mutex and wrt are initialized to 1; readcount is initialized to 0;
- Semaphore wrt is common to both reader and writer process
  - wrt functions as a mutual exclusion for the writers
  - It is also used by the first or last reader that enters or exits the critical section
  - It is not used by readers who enter or exit while other readers are in their critical section
- The readcount variable keeps track of how many processes are currently reading the object
- The mutex semaphore is used to ensure mutual exclusion when readcount is updated

The structure of a writer process

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

The structure of a reader process

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
     wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
     signal(wrt);
   signal(mutex);
} while (TRUE);
```

The Dining Philosophers Problem:

- Five philosophers sit at a round table - thinking and eating
- Each philosopher has one chopstick
  - five chopsticks total
- A philosopher needs two chopsticks to eat
  - philosophers must share chopsticks to eat
- No interaction occurs while thinking

The situation of the dining philosophers is shown in Figure 7.1

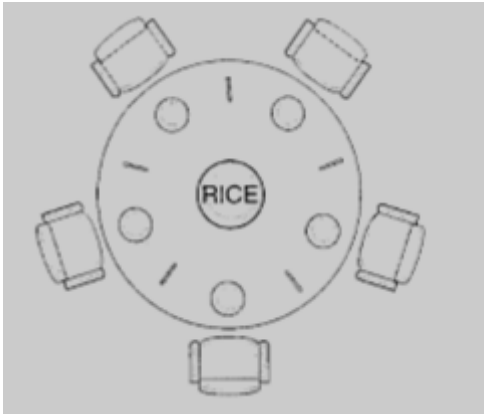

Figure 7.1

Lab Exercise

1. Modify the above Producer-Consumer program so that, a producer can produce at the

most 10 items more than what the consumer has consumed.

2. Write a C/C++ program for first readers-writers problem using semaphores.

Additional Exercise

1. Write a C/C++ program for Dining-Philosophers problem using semaphores.