CSE 321 Operating Systems
Lab Assignment 4
**Total Marks: 20**

# Semaphores and Mutexes

The POSIX thread library contains functions for working with semaphores and mutexes. There is much more to say than what is mentioned here. A good place to find more information is linux.die.net.
The functions should all be compiled and linked with -pthread.

## What is a semaphore in LINUX?

**(Library: #include <semaphore.h>)**
A semaphore is fundamentally an integer whose value is never allowed to fall below 0. There are two operations on a semaphore: wait and post. The post operation increments the semaphore by 1, and the wait operation does the following: If the semaphore has a value > 0, the semaphore is decremented by 1. If the semaphore has value 0, the caller will be blocked (busy-waiting or more likely on a queue) until the semaphore has a value larger than 0, and then it is decremented by 1.We declare a semaphore as:
**sem_t sem;**
where sem_t is a typedef defined in a header file as (apparently) a kind of unsigned char.
An example of this might be that we have a set of N interchangeable resources. We start with semaphore S = N. We use a resource, so there are now N-1 available (wait), and we return it when we are done (post). If the semaphore has value 0, there are no resources available, and we have to wait (until someone does a post).
Semaphores are thus used to coordinate concurrent processes.
This is what some people call a "counted semaphore". There is a similar notion called a "binary semaphore" which is limited to the values 0 and 1.
A semaphore may be named or unnamed. These notes assume we are using named semaphores.

# Semaphore Functions in C

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

*Purpose:*

This initializes the semaphore *sem. The initial value of the semaphore will be value. If pshared is 0, the semaphore is shared among all threads of a process (and hence need to be visible to all of them such as a global variable). If pshared is not zero, the semaphore is shared but should be in shared memory.

Notes:
- On success, the return value is 0, and on failure, the return value is -1.
- An attempt to initialize a semaphore that has already been initialized results in undefined behavior.

2. **int sem_wait(sem_t * sem);**

Purpose: This implements the wait function described above on the semaphore *sem.

Notes:

- Here sem_t is a typdef defined in the header file as (apparently) some variety of integer.
- On success, the return value is 0, and on failure, the return value is -1 (and the value of the semaphore is unchanged).
- There are related functions sem_trywait() and sem_timedwait().

3. **int sem_post(sem_t * sem);**

Purpose: This implements the post function described above on the semaphore *sem.

Note: On success, the return value is 0, and on failure, the return value is -1 (and the value of the semaphore is unchanged).

4. **int sem_destroy(sem_t * sem);**

Prototype: int sem_destroy(sem_t * sem);

Purpose: This destroys the semaphore *sem, so *sem becomes uninitialized.

Notes:

- On success, the return value is o, and on failure, the return value is -1.
- Destroying a semaphore on which other processes or threads are waiting (using sem_wait()) or destroying an uninitialized semaphore will produce undefined results.

---

## What is a mutex in LINUX?

**(Library: #include <pthread.h>)**

A mutex (named for "mutual exclusion") is a binary semaphore with an ownership restriction: it can be unlocked (the post operation) only by whoever locked it (the wait operation). Thus a mutex offers a somewhat stronger protection than an ordinary semaphore.
We declare a mutex as:
**pthread_mutex_t mutex;**

## mutex Functions in C

1. **int pthread_mutex_init(pthread_mutex_t * restrict mutex, const pthread_mutexattr_t * restrict attr);**

Purpose: This initializes *mutex with the attributes specified by attr. If attr is NULL, a default set of attributes is used. The initial state of *mutex will be "initialized and unlocked".
Notes:

- If we attempt to initialize a mutex already initialized, the result is undefined.
- On success, the return value is 0, and on failure, the return value is a nonzero value indicating the type of error.
- In the prototype, the keyword restrict (part of the C99 standard) means that this pointer will be the only pointer to the object.

2. **int pthread_mutex_destroy(pthread_mutex_t * restrict mutex);**

Purpose: This destroys the mutex object *mutex, so *mutex becomes uninitialized.
Notes:

- It is safe to destroy an unlocked mutex but not a locked mutex.
- The object *mutex could be reused, i.e., reinitialized.
- On success, the return value is 0, and on failure, the return value is a nonzero value indicating the type of error.

3. **int pthread_mutex_lock(pthread_mutex_t * mutex);**

Purpose: This locks *mutex. If necessary, the caller is blocked until *mutex is unlocked (by someone else) and then &mutex is locked. When the function call ends, *mutex will be in a locked state.

Notes:

- Suppose we try to relock a locked mutex. Depending on the attributes of the mutex, we may have an error, or a count may be kept of how many times the caller has locked the same mutex (and thus will have to unlock it the same number of times).
- On success, the return value is 0, and on failure, the return value is a nonzero value indicating the type of error.

4. **int pthread_mutex_unlock(pthread_mutex_t * mutex);**

Purpose: This unlocks *mutex.

Notes:

- Suppose we try to unlock an unlocked mutex. Depending on the attributes of the mutex, we may have an error.
- On success, the return value is 0, and on failure, the return value is a nonzero value indicating the type of error.

# Lab Tasks

## Task1:

Solve the producer and consumer problem with inter thread communication (join(), wait(), sleep() etc.) modifying the given C code.

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#define MAX 10 //producers and consumers can produce and consume upto MAX
#define BUFLEN 6
#define NUMTHREAD 2      /* number of threads */
void * consumer(int *id);
void * producer(int *id);

char buffer[BUFLEN];
char source[BUFLEN]; //from this array producer will store it's production into
buffer
int pCount = 0;
int cCount = 0;
int buflen;
```

```
//initializing pthread mutex and condition variables
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nonEmpty  = PTHREAD_COND_INITIALIZER;
pthread_cond_t full  = PTHREAD_COND_INITIALIZER;
int thread_id[NUMTHREAD]  = {0,1};
int i = 0;
int j = 0;

main()
{
    int i;
    /* define the type to be pthread */
    pthread_t thread[NUMTHREAD];

    strcpy(source,"abcdef");
    buflen = strlen(source);
    /* create 2 threads*/
    /* create one consumer and one producer */
    /* define the properties of multi threads for both threads  */
    //Write Code Here







}


void * producer(int *id)
{
       /*
       1. Producer stores the values in the buffer (Here copies values from source[]
to buffer[]).
       2. Use mutex and thread communication (wait(), sleep() etc.) for the critical
section.
       3. Print which producer is storing which values using which thread inside the
       critical section.
       4. Producer can produce up to MAX.
       */
       //Write code here
}

void * consumer(int *id)
{
       /*
       1. Consumer takes out the value from the buffer and makes it empty.
```

```
	2. Use mutex and thread communication (wait(), sleep() etc.) for critical
section
	3. Print which consumer is taking which values using which thread inside the
	critical section.
	4. Consumer can consume up to MAX
	*/
	//Write code here
}
```

**Example Output:**

```
0 produced  a  by Thread  0
1 produced  b  by Thread  0
0 consumed  a  by Thread  1
2 produced  c  by Thread  0
1 consumed  b  by Thread  1
3 produced  d  by Thread  0
4 produced  e  by Thread  0
2 consumed  c  by Thread  1
5 produced  f  by Thread  0
6 produced  a  by Thread  0
3 consumed  d  by Thread  1
7 produced  b  by Thread  0
8 produced  c  by Thread  0
9 produced  d  by Thread  0
4 consumed  e  by Thread  1
```

## Task 2:

The task is similar to the producer–consumer problem discussed in the class. The farmer and Shopowner share a fixed-size buffer named ***warehouse*** used as a queue. The farmer's job is to harvest crops(Rice=R, Wheat=W, Potato=P, Sugarcane=S, Maize=M) and put this in the warehouse.
Imagine that warehouses have different rooms for different crops.
The Shopowner's job is to take the crops from this wearhouse and make that crops room empty(=N).You have 5 Farmers and 5 Shop Owners

**You need to modify the following C code**:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
/*
This program provides a possible solution using mutex and semaphore.
use 5 Farmers and 5 ShopOwners to demonstrate the solution.
*/
#define MaxCrops 5 // Maximum crops a Farmer can produce or a Shpoowner can
take
#define warehouseSize 5 // Size of the warehouse
sem_t empty;
sem_t full;
int in = 0;
int out = 0;
char crops[warehouseSize]={'R','W','P','S','M'}; //indicating room for
different crops
char warehouse[warehouseSize]={'N','N','N','N','N'}; //initially all the room
is empty
pthread_mutex_t mutex;

void *Farmer(void *far)
{
   /*
   1.Farmer harvest crops and put them in particular room. For example, room
0 for Rice(R).
   2.use mutex and semaphore for critical section.
   3.print which farmer is keeping which crops in which room inside the
critical section.
   4.print the whole warehouse buffer outside of the critical section
   */


}
void *ShopOwner(void *sho)
{
/*
   1.Shop owner takes crops and makes that particular room empty.
   2.use mutex and semaphore for critical section.
   3.print which shop owner is taking which crops from which room inside the
critical section.
   4.print the whole warehouse buffer outside of the critical section
   */
}
int main()
```

```
{
    /*initializing thread,mutex,semaphore
    */
    pthread_t Far[5],Sho[5];
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty,0,warehouseSize);//when the warehouse is full thread will
wait
    sem_init(&full,0,0);//when the warehouse is empty thread will wait

    int a[5] = {1,2,3,4,5}; //Just used for numbering the Farmer and
ShopOwner

    /*create 5 thread for Farmer 5 thread for ShopOwner
    -------------------------------------------------
    -------------------------------------------------
    */


    //  Closing or destroying mutex and semaphore
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);


    return 0;

}
```

**Example Output :**

Farmer 3: Insert crops R at 0
Farmer 4: Insert crops W at 1
Farmer 5: Insert crops P at 2
Shop owner 1: Remove crops R from 0
Shop owner 1: Remove crops W from 1
Shop owner 1: Remove crops P from 2
Farmer 4: Insert crops S at 3
Farmer 4: Insert crops M at 4
Farmer 4: Insert crops R at 0
Shop owner 1: Remove crops S from 3
Shop owner 1: Remove crops M from 4
ShopOwner1: RNNNN

……………………………………
……………………………………
……………………………………
……………………………………
Shop owner 5: Remove crops W from 1
Farmer 5: Insert crops P at 2
Farmer 5: Insert crops S at 3
Farmer 5: Insert crops M at 4
Farmer5: NNPSM
Shop owner 5: Remove crops P from 2
Shop owner 5: Remove crops S from 3
Shop owner 5: Remove crops M from 4
ShopOwner5: NNNNN