Assignment 5

Thread basics ,Semaphore and Classical synchronization problems

1. Thread basics:

Thread is a lightweight process. That provides a way to improve application performance through parallelism. This is done by dividing the process into multiple tasks executed by these threads.

Basic functions:

1) pthread_create: used for thread creation. Returns 0 if success.

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

2) pthread_self: used to get the thread id of the current thread

```
pthread_t pthread_self(void);
```

3) pthread_exit: terminate calling thread `pthread_exit(NULL);`

4) sleep: places into an inactive state for a period of time

```
sleep();
```

5) pthread_join: wait for thread termination

```
P


thread_join(thread, NULL);
```

Basic program for Single Thread

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

void * PrintHello(void * data)
{
    int my_data = (int)data;

    printf("\n Hello from new thread - got %d !\n",
    my_data); pthread_exit(NULL);
}

int main()
{
    int rc;
    pthread_t thread_id;

    int t = 11;

    rc = pthread_create(&thread_id, NULL, PrintHello,
    (void*)t); if(rc)
    {
      printf("\n ERROR: return code from pthread_create is
%d \n", rc);
```



```
      exit(1);

    }
    printf("\n Created new thread (%u)... \n", thread_id);
```

```
    pthread_exit(NULL);
}
```

Basic program for Multithread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter;

void* trythis(void* arg)
{
     unsigned long i = 0;
     counter += 1;
     printf("\n Job %d has started\n", counter);

     for (i = 0; i < (0xFFFFFFFF); i++)
          ;
     printf("\n Job %d has finished\n", counter);

     return NULL;
}

int main(void)
{
     int i = 0;
```

```
     int error;


     while (i < 2) {
          error = pthread_create(&(tid[i]), NULL,
&trythis, NULL);
          if (error != 0)
               printf("\nThread can't be created : [%s]",
strerror(error));
          i++;
     }

     pthread_join(tid[0], NULL);
     pthread_join(tid[1], NULL);

     return 0;
}
```

How to compile the above program?

To compile a multithreaded program using gcc, we need to link it with the pthreads library. Following is the command used to compile the program.

```
gcc filename.c -lpthread
```

Exercise :

1. Implement the multi threading For adding the elements in an user defined array. Addition must be done using two threads only.

2. Once you are done with the above problem you have to code for adding two user defined arrays (can be of same size or

different size). Using Two threads only.

2. Semaphore

Semaphore is an integer variable used by various processes in a mutually exclusive manner to achieve synchronization.

- Semaphores are catagorized into two types :
    1. Counting semaphore
    2. Binary Semaphore

- Two different operations are performed on semaphore:
    - Down() / Wait() / p()
    - Up() / signal() / V() / release()

    1. Counting semaphore:

```
down (semaphore s)
{
        s.value=s.value-1;
        if(s.value<0){
                Block the process and place its PCB in the
                suspended list
        }
}

up (semaphore s){
        s.value=s.value+1;
        if(s.value<= 0){
                Select a process from a suspended list and wakeup
        }

}
```

2. **Binary semaphore:**

```
down (semaphore s)
{
        if(s.value==1){
                s.value=0
        }
        else{
                Block the process and place it's PCB in suspended
        list
        }

}

up (semaphore s){

        if(suspended list is empty){
                s.value=1
        }
        else{
                Select a process from suspended list and wakeup();
        }

}
```

Mutex:

When the same region of the code executes by multiple thread and race condition is accrued to change the value of the shared variable then it causes an issue in finding the final value.

Mutexes are used to protect shared resources by implementing lock mechanisms on the shared region of code.

1) pthread_mutex_init:

Creates a mutex, referenced by mutex, with attributes specified by attr.


```
int pthread_mutex_init(pthread_mutex_t *restrict mp, const
pthread_mutexattr_t *restrict mattr);
```

2) pthread_mutex_lock:

Locks a mutex object, which identifies a mutex. If the mutex is already locked by another thread, the thread waits for the mutex to become available. If the same thread locks the same variable multiple times then the counter will increase and The owning thread must call pthread_mutex_unlock() the same number of times to decrement the count to zero.

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

3) pthread_mutex_unlock:

Release the mutex object.

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

4) pthread_mutex_destroy:

Deletes a mutex object, which identifies a mutex.

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

All the above thread mutex functions return 0 on successful execution of function else it returns -1.

Example Mutex Lock Examples for Linux Thread Synchronization

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

```
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;
```



```
void* trythis(void* arg)
{
     pthread_mutex_lock(&lock);

     unsigned long i = 0;
     counter += 1;
     printf("\n Job %d has started\n", counter);

     for (i = 0; i < (0xFFFFFFFF); i++);

     printf("\n Job %d has finished\n", counter);

     pthread_mutex_unlock(&lock);

     return NULL;
}

int main(void)
{
     int i = 0;
     int error;

     if (pthread_mutex_init(&lock, NULL) != 0) {
          printf("\n mutex init has failed\n");
```

```
            return 1;
        }

        while (i < 2) {
            error = pthread_create(&(tid[i]),NULL,&trythis,
            NULL); if (error != 0)
                printf("\nThread can't be created :[%s]",
                    strerror(error));
            i++;
        }
```



```
        pthread_join(tid[0], NULL);
        pthread_join(tid[1], NULL);
        pthread_mutex_destroy(&lock);

        return 0;
}
```

Exercise:

1) Implement classical producer-consumer problems to get insight on semaphores and mutex.

2) In your bank account, your current balance is 500 Rs. You have 2 functions:

1. Credit -> (reads amount, credits 50 Rs., prints final amount)
2. Debit -> (reads amount, debits 50 Rs., prints final amount)

Implement a C program to complete both transactions if both transactions occur concurrently safely.

3) Consider a system where we have counting semaphore S. Various semaphore operations like 20p, 12v are performed then what is the largest initial value of semaphore so that one process will remain blocked?

3. Classical synchronization problems

a) *Reader writer's problem*

***mutex*:** Binary semaphore used by readers in a mutual exclusive manner.

***db*:** Binary semaphore used by readers and writers in a mutual exclusive manner.

Example Code snippet:

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

/*
This program provides a possible solution for first reader writer problems using mutex and semaphore.
I have used 10 readers and 5 producers to demonstrate the solution. You can always play with these values.
*/

sem_t db;
pthread_mutex_t mutex;
int cnt  = 1; // content of db
int rc = 0;

void *writer(void *wno)
```

```
{
    sem_wait(&db);
    cnt = cnt*2;
    printf("Writer %d modified cnt to %d\n",(*((int *)wno)),cnt);
    sem_post(&db);

}
void *reader(void *rno)
{
    // Reader acquire the lock before modifying rc
    pthread_mutex_lock(&mutex);     // ……(1)
    rc++;  //……(2)
    if(rc == 1) {
        sem_wait(&db); // If this id the first reader, then it will block
the writer
    }
    pthread_mutex_unlock(&mutex);
    // Reading Section
    printf("Reader %d: read cnt as %d\n",*((int *)rno),cnt);

    // Reader acquire the lock before modifying rc
    pthread_mutex_lock(&mutex); // …….(3)
    rc--; // ……(4)
    if(rc == 0) {
        sem_post(&db); // If this is the last reader, it will wake up
the writer.
    }
    pthread_mutex_unlock(&mutex);
}

int main()
{

    pthread_t read[10],write[5];
    pthread_mutex_init(&mutex, NULL);
```

```
    sem_init(&db,0,1);

    int a[10] = {1,2,3,4,5,6,7,8,9,10}; //Just used for numbering
the producer and consumer

    for(int i = 0; i < 10; i++) {
        pthread_create(&read[i], NULL, (void *)reader, (void
*)&a[i]);
    }
    for(int i = 0; i < 5; i++) {
        pthread_create(&write[i], NULL, (void *)writer, (void
*)&a[i]);
    }

    for(int i = 0; i < 10; i++) {
        pthread_join(read[i], NULL);
    }
    for(int i = 0; i < 5; i++) {
        pthread_join(write[i], NULL);
    }

    pthread_mutex_destroy(&mutex);
    sem_destroy(&db);

    return 0;

}
```

Exercise:

1. What happens if we interchange line (1) and line (2) in the reader's function? Run the code after modifications and write the correct answer from the below options with proper explanation.

2. What happens if we interchange line (3) and line (4) in the reader's function? Run the code after modifications and write the correct answer from the below options with proper explanation.

(You need to check on paper to answer the following questions manually)

Options for both the questions are:

a) No problem, the solution still works fine
b) Multiple readers are not allowed into the database
c) Both reader and writer will enter into the database at the same time
d) Deadlock possible

3. Implement the Reader writer problem's variant where
    a) Priority is given to the reader.
    b) Priority is given to the writer.