**Embedded Systems and RTOS**

Mini Project Assignment - Report

BTech E&TC 2017-21 Semester – VII

AY 2020-21 Odd Semester

Name: Veekshita Sai Choudhary, Ramya Bardae

PRN: 17070123001,17070123080

**TITLE**: Demonstrating use of Mutex for task synchronization using pthreads.

**AIM**: Creating Linux Pthreads and using a mutex for synchronizing their execution.

**INTRODUCTION:**

**Pthreads (POSIX)**

Pthreads is a standard API for C language programming. It is implemented with a pthread.h header file. The main purpose of this thread is to execute the program faster. Pthreads can generally be defined as a light weight process, as managing it requires only a few system resources. pthreads are preferably suitable for parallel programming in multi-core machines. They provide multiple flows of execution within a process. The threads have their own stacks but share the global data and the heap. So, the global variables are visible to multiple threads. Also, the threads need to synchronize their actions so that they jointly realize the overall objectives of the process they belong to. The core problems of concurrent programming, mutual exclusion and synchronization are relevant for threads just like these problems are relevant for multi-process systems.

**Pthread API: Mutex synchronization**

The API used for this project is mutex. Mutex which is an acronym for mutual exclusion is used by routines that deal with synchronization. Mutex helps to block access of the other threads to the variables. This kind of technique leads to restricted access to the variables by a thread. Mutex functions help in locking and unlocking mutexes.

Mutex is similar to a flag which can be set and unset; more precisely, can be locked and unlocked. Mutexes help to prevent data contradictions which occur due to race conditions. A race condition is a situation that occurs when multiple threads have to perform operations on a particular memory area and the results may eventually vary as the order of execution of different threads may differ. In such conditions, mutexes can be used for serializing the resources that are shared. Mutexes can only be applied to threads that are in a single process and not on different processes.

Deadlocks will occur when the thread which acquired the lock attempts to lock the mutex again. The best way to avoid this situation is to pay close attention towards when locking and unlocking of the mutex to ensure that program can never be in a situation to lock the same mutex that is already locked.

**PROCESS:**

Mutex helps to solve the unpredictable behaviour and the race conditions that the multiple threads create while accessing a resource that is shared. A thread tries to lock a mutex and another thread also tries to lock the same mutex that thread will be blocked till the previous thread finishes and gets unlocked. This eventually ensures only one thread to access a variable at a time and a dead lock will be created when any other thread tries to access it.

In this project we will count 5 lakh times using 12 threads with and without using mutex and observe the output.

* static pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER

Initialising a conditional variable by defining and initialising it as a variable.

* pthread\_create ()

This function starts a new thread in the calling process.

* pthread\_join ()

This function suspends the execution of the calling thread until the target thread terminates, unless the target has already terminated.

**PROGRAM:**

1. **Without the use of mutex:**

// This program helps to count up to 500000

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define COUNT 500000

#define MAX\_THREADS 12 /\* number of threads \*/

static pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

long long i = 0;

void \* start\_count (void \*arg)

{

for (; ;) // create infinite loop

{

if (i >= COUNT) // to check if maximum count is reached

{

    return NULL;

}

i++;

printf ("i = %lld \n", i);

}

}

 int main(void)

{

int i = 0;

 // create a thread group as the size of MAX\_THREADS

pthread\_t \*thread\_group = malloc(sizeof(pthread\_t) \* MAX\_THREADS);

// The threads start to work

for (i=0; i<MAX\_THREADS; i++)

{

pthread\_create(&thread\_group[i], NULL, start\_count, NULL);

}

// Wating for the threads to terminate

for (i=0; i<MAX\_THREADS; i++)

{

pthread\_join(thread\_group[i], NULL);

}

return EXIT\_SUCCESS;

}

1. **With Mutex Synchronization:**

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#define COUNT 500000

#define MAX\_THREADS 12 /\* number of threads \*/

static pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

long long i = 0;

void \*start\_count (void \*arg) {

for (; ;) {

// To start mutex lock and acquire it

pthread\_mutex\_lock(&mutex);

/\* critical section of the code \*/

// To check if maximum count is reached

if (i >= COUNT\_TO) {

pthread\_mutex\_unlock(&mutex);

return NULL;

}

++i;

/\* Critical section ends \*/

// release the mutex lock

pthread\_mutex\_unlock(&mutex);

printf ("i = %lld\n", i);

}

}

int main(void) {

int i = 0;

// create a thread group as the size of MAX\_THREADS

pthread\_t \*thread\_group = malloc(sizeof(pthread\_t) \* MAX\_THREADS);

// The threads start to work

for (i = 0; i < MAX\_THREADS; ++i) {

pthread\_create(&thread\_group[i], NULL, start\_count, NULL);

}

// Waiting for the threads to terminate

for (i = 0; i < MAX\_THREADS; ++i) {

pthread\_join(thread\_group[i], NULL);

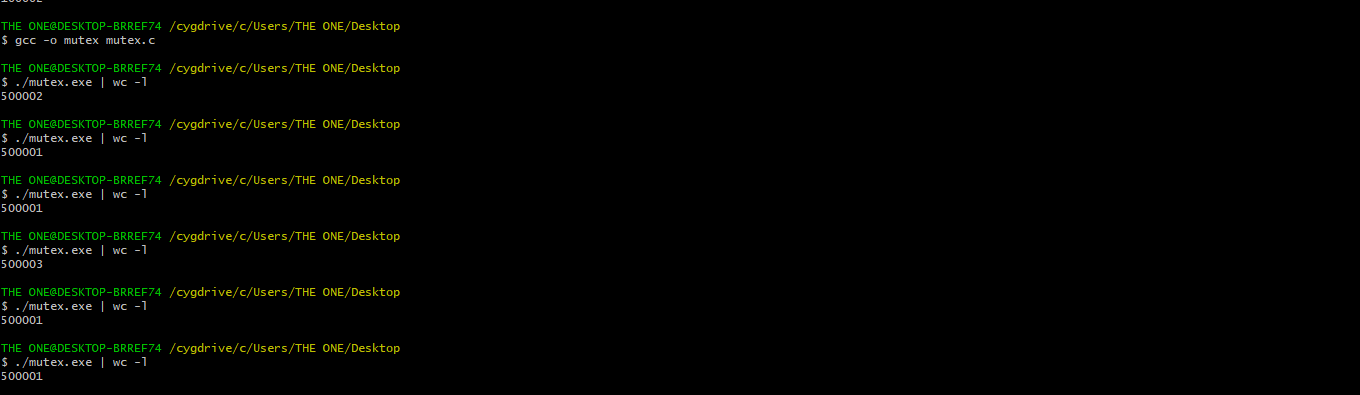
}

return EXIT\_SUCCESS;

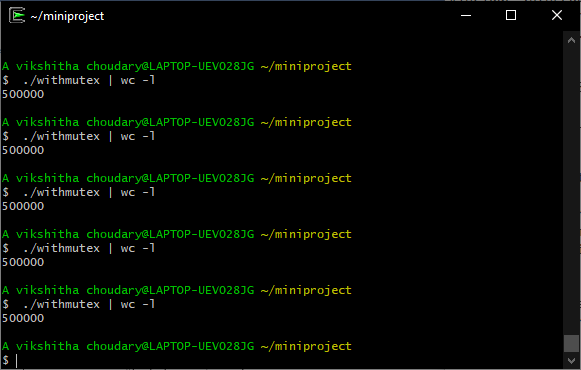
}

**OUTPUT:**

1. **Without Mutex Synchronization**



1. **With Mutex Synchronization**



**INFERENCE:**

This project resulted in recognizing the importance of mutual execution. We understood the practical difference in how random numbers are assigned when multiple threads try to access the same resource and how a constant value is provided with mutex synchronization, that is giving access to only one thread at a time.

From this experiment, we conclude that Mutex synchronization is essential for avoiding the race conditions and unpredictable behaviour of the system, and also understood about mutex ensuring that two or more concurrent processes or threads do not simultaneously execute the critical section which is meant to be executed by only one thread or process.