# LABWORK – 2

## OBJECTIVE:

**1.** Study how to use POSIX threads.

**2.** Implement multithreading in using C.

**3**. Implement POSIX thread functions for thread create, join, exit.

## THEORY:

### POSIX thread (pthread) libraries

The POSIX thread libraries are a standards based thread API for C/C++. It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. Threads require less overhead than "forking" or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process. While most effective on a multiprocessor system, gains are also found on uniprocessor systems which exploit latency in I/O and other system functions which may halt process execution. (One thread may execute while another is waiting for I/O or some other system latency.) Parallel programming technologies such as MPI and PVM are used in a distributed computing environment while threads are limited to a single computer system. All threads within a process share the same address space. A thread is spawned by defining a function and it's arguments which will be processed in the thread. The purpose of using the POSIX thread library in your software is to execute software faster.

### Thread Basics:

Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.

### pthread_create

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

Arguments:

*thread* - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)

*attr* - Set to NULL if default thread attributes are used.

*void * (*start_routine)* - pointer to the function to be threaded. Function has a single argument: pointer to void.

**arg* - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

### pthread_exit

*void pthread_exit(void *retval);*

Arguments:

*retval* - Return value of thread.

This routine kills the thread. The pthread_exit function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join.

## Thread Synchronization:

The threads library provides three synchronization mechanisms:

- *mutexes* - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
- *joins* - Make a thread wait till others are complete (terminated).
- *condition variables* - data type pthread_cond_t

## Mutexes:

Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed. Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. One can apply a mutex to protect a segment of memory ("critical region") from other threads. Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

## Joins:

A join is performed when one wants to wait for a thread to finish. A thread calling routine may launch multiple threads then wait for them to finish to get the results. One wait for the completion of the threads with a join.

int pthread_join(pthread_t thread, void **retval);

The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

## Mutex functions:

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

The pthread_mutex_init() function initialises the mutex referenced by mutex with attributes specified by attr. If attr is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

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

The mutex object referenced by mutex is locked by calling pthread_mutex_lock(). If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

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

The pthread_mutex_unlock() function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by mutex when pthread_mutex_unlock() is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread shall acquire the mutex.

## Example 1: Threads

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

void* say_hello(void* data)
{
   char *str;
   str = (char*)data;
   while(1)
   {
      printf("%s\n",str);
      sleep(1);
   }
}

void main()
{
   pthread_t t1,t2;

   pthread_create(&t1,NULL,say_hello,"hello from 1");
   pthread_create(&t2,NULL,say_hello,"hello from 2");
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
}
```

## Example 2: Threads without mutex

```
#include<stdio.h>
#include<pthread.h>
int num = 0;

void square()
{

   while(1)
   {
      printf("The Number is %d\n",num);
         sleep(2);
      printf("And its square is %d\n",num*num);
         sleep(1);
   }
}

void inc()
{
   while(1)
   {
      num++;
      sleep(1);
   }
}
```

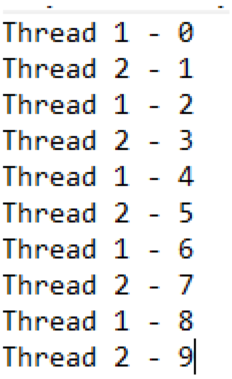
```
void main()
{
   pthread_t t1,t2;
   pthread_create(&t1,NULL,square,NULL);
   pthread_create(&t2,NULL,inc,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
}
```

### Example 3: Threads with mutex

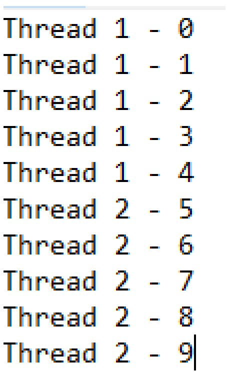
```
#include<stdio.h>
#include<pthread.h>

int num = 0;
pthread_mutex_t lock;

void square()
{
   while(1)
   {
      pthread_mutex_lock(&lock);
      printf("The Number is %d\n",num);
      sleep(2);
      printf("And its square is %d\n",num*num);
      pthread_mutex_unlock(&lock);
      sleep(1);
   }
}

void inc()
{
   while(1)
   {
      pthread_mutex_lock(&lock);
      num++;
      pthread_mutex_unlock(&lock);
      sleep(1);

   }
}

void main()
{
   pthread_t t1,t2;
   pthread_mutex_init(&lock, NULL);
   pthread_create(&t1,NULL,square,NULL);
   pthread_create(&t2,NULL,inc,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
}
```

**YOUR TASK :** Implement two program using multithreading. Program should have pthread_create, pthread_join, pthread_exit and pthread_mutex. Your program also should have two threads. Programs should write following outputs to .txt file. You should submit your .c files (Ex. program1.c ,program2.c).

***PROGRAM 1 OUTPUT:   PROGRAM 2 OUTPUT:***



```
Thread 1 - 0
Thread 2 - 1
Thread 1 - 2
Thread 2 - 3
Thread 1 - 4
Thread 2 - 5
Thread 1 - 6
Thread 2 - 7
Thread 1 - 8
Thread 2 - 9
```



```
Thread 1 - 0
Thread 1 - 1
Thread 1 - 2
Thread 1 - 3
Thread 1 - 4
Thread 2 - 5
Thread 2 - 6
Thread 2 - 7
Thread 2 - 8
Thread 2 - 9
```