Lab No. 08
Creating Multithreaded Applications

Objective
This lab is designed to familiarize the students with the implementation of multithreading.

Activity Outcomes:
On completion of this lab students will be able to

- Implement simple multithreaded applications

Instructor Notes
As pre-lab activity, read the content from the following (or some other) internet source:
https://www.geeksforgeeks.org/multithreading-in-cpp/

1) Useful Concepts

Multithreading

A thread is a path of execution within a process. A process can contain multiple threads. A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. Parallelism is the feature that allows your computer to run two or more programs from same application simultaneously. As a result, our applications are executed in less time and become more interactive. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

We can implement parallel applications in two ways: process-based and thread-based. Process-based parallelism is achieved through the simultaneous execution of more than one processes from the same application while thread-based parallelism deals with the simultaneous execution of pieces of the same processes. Generally, thread-based parallelism is considered efficient than process-based parallelisms.

Pthread Library

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's. Pthreads are defined as a set of C language programming types and procedure calls. Vendors usually provide a Pthreads implementation in the form of a header/include file and a library, which you link with your program.

In this lab we are going to write multi-threaded C++ program using POSIX. POSIX Threads, or Pthreads provides API which are available on many Unix-like POSIX systems such as FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris.

Creating Threads

We can use the following routine to create a POSIX thread

```
pthread_create (thread, attr, start_routine, arg)
```

Here, pthread_create creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code. Here is the description of the parameters is as follows:

Parameter	Description
Thread	An unique identifier for the new thread returned by the subroutine.
Attr	An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
start_routine	The C++ routine that the thread will execute once it is created.
Arg	A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

It is to be noted that global variables can be accessed from all threads while variables defined within a thread are not accessible to other threads.

Displaying Thread ID

We can use the pthread_self( ) routine to display the id of the current thread.

Terminating Threads

Following routine terminates a POSIX thread

```
pthread_exit (status)
```

Here **pthread_exit** is used to explicitly exit a thread. Typically, the pthread_exit routine is called after a thread has completed its work and is no longer required to exist. If main finishes before the threads it has created, and exits with pthread_exit, the other threads will continue to execute. Otherwise, they will be automatically terminated when main finishes.

Joining and Detaching Threads

Following routines are used for joining threads

```
pthread_join (threadid, status)
```

The pthread_join subroutine blocks the calling thread until the specified threadid thread terminates.
Note: To compile a program that uses pthread library, we need to use the –pthread option. For example, to compile a program test.cpp we can write the following command

```
g++  test.cpp –o  test  -pthread
```

2) Solved Lab Activities

Sr.No	*Allocated Time*	*Level of Complexity*	*CLO Mapping*
1	20	Medium	CLO-7
2	25	Medium	CLO-7

Activity 1:

The following example code creates 5 threads with the pthread_create routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit.

Solution:

Code

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
using namespace std;
#define NUM_THREADS 5
void *PrintHello (void *threadid)
{
  long tid;
  tid = (long)threadid;
  cout << "Hello World! Thread ID, " << tid << endl;
  pthread_exit(NULL);
}
int main () {
  pthread_t threads[NUM_THREADS];
  int rc;
  int i;
for( i = 0; i < NUM_THREADS; i++ )
{
    cout << "main() : creating thread, " << i << endl;
    rc = pthread_create(&threads[i], NULL, PrintHello, (void *)i);

    if (rc)
   {
      cout << "Error:unable to create thread," << rc << endl;
      exit(-1);
    }
  }
  pthread_exit(NULL);
```

Out-put

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

Activity 2:

This example demonstrates how to wait for thread completions by using the Pthread join routine.

Solution:

Code

```
#include <iostream>
#include <cstdlib>
#include <pthread.h>
#include <unistd.h>
using namespace std;
#define NUM_THREADS 5
void *wait(void *t)
{
  int i;
  long tid;
  tid = (long)t;
  sleep(1);
  cout << "Sleeping in thread " << endl;
  cout << "Thread with id : " << tid << "  ...exiting " << endl;
  pthread_exit(NULL);
}
int main ()
{
  int rc;
  int i;
  pthread_t threads[NUM_THREADS];
  pthread_attr_t attr;
  void *status;
  // Initialize and set thread joinable
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  for( i = 0; i < NUM_THREADS; i++ ) {
    cout << "main() : creating thread, " << i << endl;
    rc = pthread_create(&threads[i], &attr, wait, (void *)i );
   if (rc) {
      cout << "Error:unable to create thread," << rc << endl;
      exit(-1);
    }
  }

  // free attribute and wait for the other threads
  pthread_attr_destroy(&attr);
  for( i = 0; i < NUM_THREADS; i++ )
{
    rc = pthread_join(threads[i], &status);
    if (rc)
      {
      cout << "Error:unable to join," << rc << endl;
      exit(-1);
      }
    cout << "Main: completed thread id :" << i ;
```

```
            cout << "  exiting with status :" << status << endl;
        }
        cout << "Main: program exiting." << endl;
        pthread_exit(NULL);
    }
```

Out-put

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Sleeping in thread
Thread with id : 0 .... exiting
Sleeping in thread
Thread with id : 1 .... exiting
Sleeping in thread
Thread with id : 2 .... exiting
Sleeping in thread
Thread with id : 3 .... exiting
Sleeping in thread
Thread with id : 4 .... exiting
Main: completed thread id :0  exiting with status :0
Main: completed thread id :1  exiting with status :0
Main: completed thread id :2  exiting with status :0
Main: completed thread id :3  exiting with status :0
Main: completed thread id :4  exiting with status :0
Main: program exiting.
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Task 1:

Write C++ program that

- Declares an array of size 1000 and populates it with random numbers between 1 and 100. Then it finds factorial of these numbers and checks how many of these prime numbers are. Also find the time taken by the system to perform these tasks.
- Now, create two thread such that the first thread process finds the factorial of these numbers (given in the above array) while the other finds how many of these are prime. Now, calculate the time taken by these children processes to perform these tasks.
- Now, compare which approach performs better