LAB # 08

Statement Purpose:

This lab will give you'll implement multithreading in C++ language using POSIX library, and implement synchronization Peterson's solution using threads.

Activity Outcomes:

The primary objective of this lab is to implement the Thread Management Functions:

- Creating Threads
- Terminating Thread Execution
- Passing Arguments to Threads
- Thread Identifiers
- Joining Threads
- Detaching / Undetaching Threads
- Peterson's solution using threads

Instructor Note:

https://www.geeksforgeeks.org/multithreading-in-cpp/
https://www.geeksforgeeks.org/petersons-algorithm-for-mutual-exclusion-set-2-cpu-cycles-and-memory-fence/

1) Stage J (Journey)

Introduction

Multitasking is the feature that allows your computer to run two or more programs concurrently and Multithreading is a specialized form of multitasking. In general, there are two types of multitasking: process-based and thread-based.

Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A thread is a semi-process, that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

C++ does not contain any built-in support for multithreaded applications. Instead, it relies entirely upon the operating system to provide this feature.

What are pthreads? 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.

2) Stage a1 (apply)

Lab Activities:

Creating Threads

The following routine is used to create a POSIX thread

```
#include <pthread.h>
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

Sr.No	Parameter & Description
1	**Thread** An unique identifier for the new thread returned by the subroutine.
2	**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.
3	**start_routine** The C++ routine that the thread will execute once it is created.
4	**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.

The maximum number of threads that may be created by a process is implementation dependent. Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

Terminating Threads

Following routine terminates a POSIX thread

```
#include <pthread.h>
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.

Activity 1:

Write a program in C++ to create and terminate threads.

This simple 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.

```
#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);
}
```

Compile the following program using -lpthread library as follows −

```
$gcc test.cpp -lpthread
```

Now, execute your program which gives the following output −

```
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:

Passing Arguments to Threads

This example shows how to pass multiple arguments via a structure. You can pass any data type in a thread callback because it points to void as explained in the following example

```
 #include <iostream>
#include <cstdlib>
#include <pthread.h>

using namespace std;

#define NUM_THREADS 5

struct thread_data {
  int  thread_id;
  char *message;
};

void *PrintHello(void *threadarg) {
  struct thread_data *my_data;
  my_data = (struct thread_data *) threadarg;

  cout << "Thread ID : " << my_data->thread_id ;
  cout << " Message : " << my_data->message << endl;
```

```
   pthread_exit(NULL);
}

int main () {
   pthread_t threads[NUM_THREADS];
   struct thread_data td[NUM_THREADS];
   int rc;
   int i;

   for( i = 0; i < NUM_THREADS; i++ ) {
      cout <<"main() : creating thread, " << i << endl;
      td[i].thread_id = i;
      td[i].message = "This is message";
      rc = pthread_create(&threads[i], NULL, PrintHello, (void *)&td[i]);

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

When the above code is compiled and executed, it produces the following result –

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
main() : creating thread, 4
Thread ID : 3 Message : This is message
Thread ID : 2 Message : This is message
Thread ID : 0 Message : This is message
Thread ID : 1 Message : This is message
Thread ID : 4 Message : This is message
```

Activity 3:

Joining and Detaching Threads

There are following two routines which we can use to join or detach threads –

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

The pthread_join subroutine blocks the calling thread until the specified threadid thread terminates. When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.

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

```
#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);
}
```

When the above code is compiled and executed, it produces the following result −

```
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.
```

Activity 4

Peterson solution for 2 processes critical section problem

```
#include<pthread.h>
#include<stdio.h>
void *func1(void *);
void *func2(void *);
int flag[2];
int turn=0;
int global=100;
int main()
{
   pthread_t tid1,tid2;
   pthread_create(&tid1,NULL,func1,NULL);
```

```
   pthread_create(&tid2,NULL,func2,NULL);
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
}

void *func1(void *param)
{
   int i=0;
   while(i<2)
   {
      flag[0]=1;
      turn=1;
      while(flag[1]==1 && turn==1);
      global+=100;
     // printf("FT: g: %d",global);
      // printf("value of i in func1 is %d \n",i);
      printf("value of global in func1 is %d \n",global);
      flag[0]=0;
      i++;
   }
}
void *func2(void *param)
{
   int i=0;
   while(i<2)
   {
      flag[1]=1;
      turn=0;
      while(flag[0]==1 && turn==0);
      global-=75;
     // printf("SP: g: %d",global);
      printf("value of i in func2 is %d \n",i);
         printf("value of global in func2 is %d \n",global);
      flag[1]=0;
      i++;
   }
}
```

Output

```
value of i in func2 is 0
value of global in func2 is 25
value of i in func2 is 1
value of global in func2 is -50
value of global in func1 is 50
```

value of global in func1 is 150

This **solution do not guarantee** mutual exclusion between the two process. The code in earlier Activity 4 might have worked on most systems, but is was not 100% correct. The logic was perfect, but most modern CPUs employ performance optimizations that can result in out-of-order execution. This reordering of memory operations (loads and stores) normally goes unnoticed within a single thread of execution, but can cause unpredictable behaviour in concurrent programs. When a thread was waiting for its turn, it ended in a long while loop which tested the condition millions of times per second thus doing unnecessary computation. There is a better way to wait, and it is known as ***"yield"*.**

In above activity the compiler considers the following 2 statements as independent of each other and thus tries to increase the code efficiency by re-ordering them, which can lead to problems for concurrent programs.

while (f = = 0);

// Memory fence required here

print x;

To avoid this we place a memory fence to give hint to the compiler about the possible relationship between the statements across the barrier.

So the modified code becomes,

```
#include<pthread.h>
#include<stdio.h>
void *func1(void *);
void *func2(void *);
int flag[2];
int turn=0;
int global=100;
int main()
{
   pthread_t tid1,tid2;
   pthread_create(&tid1,NULL,func1,NULL);
   pthread_create(&tid2,NULL,func2,NULL);
   pthread_join(tid1,NULL);
   pthread_join(tid2,NULL);
}
```

```
void *func1(void *param)
{
   int i=0;
   while(i<2)
   {
      flag[0]=1;
      turn=1;
      __sync_synchronize();
      while(flag[1]==1 && turn==1) sched_yield();
      global+=100;
     // printf("FT: g: %d",global);
      // printf("value of i in func1 is %d \n",i);
       printf("value of global in func1 is %d \n",global);
       flag[0]=0;
       i++;
   }
}
void *func2(void *param)
{
   int i=0;
   while(i<2)
   {
      flag[1]=1;
      turn=0;
      __sync_synchronize();
      while(flag[0]==1 && turn==0)sched_yield();
      global-=75;
     // printf("SP: g: %d",global);
      printf("value of i in func2 is %d \n",i);
        printf("value of global in func2 is %d \n",global);
      flag[1]=0;
      i++;
   }
}
```

3) Stage V (verify)

Home Activities:

1. Write a program that uses multiple threads to find which integer between 1 and 100,000 has the largest number of divisors, and how many divisors does it have?. By

using threads, your program will take less time to do the computation when it is run on a multiprocessor computer. At the end of the program, output the elapsed time, the integer that has the largest number of divisors, and the number of divisors that it has.

1) Stage V (verify)

Home Activities:

1) Practice the shell functions