The Department of Electrical and Computer Engineering

The University of Alabama in Huntsville

CPE 435 Lab-06

POSIX Threads

Introduction

A thread is a semi-process, which has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (whereas 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).

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. You can link the pthread library by adding **-lpthread** to your compile command.

Demo Codes

Demo 1

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void* printHello(void *threadId)
{
        printf("\n%d:Hello World!\n",threadId);
        pthread_exit(NULL);
}

int main()
{
        pthread_t threads[NUM_THREADS];
        int rc,t;
        for(t=0;t<NUM_THREADS;t++)
        {
                printf("Creating thread %d\n",t);
                rc = pthread_create(&threads[t],NULL,printHello,(void*)t);
                if(rc)
                {
                                        printf("ERROR:Return Code from pthread_create() is %d\n",rc);
                }
        }
        pthread_exit(NULL);
}
```

You can also define a global variable such that it is thread-specific. The variable is global, but the value it holds will be thread specific. In the snippet below, the modifier **__thread** defines the variable *myData* as **Thread Local Storage**.

```
#include statements
__thread int myData;
void *threadFunc()
{
//access and modification of data “myData” will be local
}
int main()
{
//create and launch threads
//join threads
}
```

There is another way to achieve the same thing. Here you can use **pthread_key_t** as the modifier.

```
#include statements
pthread_key_t myKey;
void* threadFunc()
{
        int* data = malloc(sizeof(int));
        //this modification is thread specific
        pthread_setspecific(myKey,data);
        //access the data using following
        // the real use for this is when you need to access it from another function, but want it to
be specific to a thread
        int* myLoc = pthread_getspecific(myKey);
}
int main()
{
        pthread_key_create(&myKey,NULL);
        //create and launch threads
        //join threads
}
```

Demo 2

```
/*
	Written By: Prawar Poudel
	13 Feb 2018
	This is written to demonstrate simple creation and waiting for pthread to terminate
*/
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#define NUM_THREADS 5

//the argument that will be sent will be the (int) id
void *simpleThreadFunc(void* argument)
{
	int myId = (int)argument;
	printf("My Id is %d\n",myId);
	int a = 0;
}
int main()
{
	//you can create these dynamically also
	pthread_t myThreads[NUM_THREADS];
	int status = 0;
	int i;
	for( i=0;i<NUM_THREADS;i++)
	{
		printf("Creating thread no. %d, and sending ID %d\n",i,i);
		status = pthread_create(&myThreads[i],NULL,simpleThreadFunc,(void*)i);
		if(status)
		{
			printf("Error in creating the threads: %d\n",i);
			return -1;
		}else
		{
			printf("Successful creation of thread..\n");
		}
	}

	//this is the area that threads will run

	//we will wait for the threads here
	for( i=0;i<NUM_THREADS;i++)
	{
		int retStatus = pthread_join(myThreads[i],NULL);
		if(!retStatus)
		{
			printf("Successful termination of thread id %d\n",i);
		}else
					printf("Well..... some problem at thread id %d, error no:
%d\n",i,retStatus);
	}
	return 0;
}
```

Demo 3

```
/*
	Written By: Prawar Poudel
	13 Feb 2018
	This is written to demonstrate simple creation and waiting for pthread to terminate
*/
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

#define NUM_THREADS 100

int mutexProtectedGlobalVariable;
int unprotectedProtectedGlobalVariable;
pthread_mutex_t myMutex;

//this function will update the value without any protection void
*unprotectedThreadFunc(void* argument)
{
	int i;
	for( i=0;i<10000;i++) unprotectedProtectedGlobalVariable++; }

//this function will update the value without any protection void
*protectedThreadFunc(void* argument)
{
	int i;
	pthread_mutex_lock (&myMutex);
	for( i=0;i<10000;i++) mutexProtectedGlobalVariable++;
	pthread_mutex_unlock (&myMutex);
}

int main()
{
	mutexProtectedGlobalVariable = 0;
	unprotectedProtectedGlobalVariable = 0;
	int i;

	//you can create these dynamically also
	pthread_t myThreads[NUM_THREADS];
	int status = 0;


	printf("Calling unprotected set of threads\n");
	//first set of five threads will call a function that will update the variable unprotected
	for( i=0;i<NUM_THREADS;i++)
	{
		status = pthread_create(&myThreads[i],NULL,unprotectedThreadFunc, (void*)i);
		if(status)
		{
			printf("Error in creating the threads: %d\n",i);
```

```
                return -1;
            }
        }

        //this is the area that threads will run

        //we will wait for the threads here
        for( i=0;i<NUM_THREADS;i++)
        {
            int retStatus = pthread_join(myThreads[i],NULL);
            if(retStatus)
            {
                printf("Well..... some problem at thread id %d, error no: %d\n",i,retStatus);
            }
        }
        printf("Unprotected sum is %d\n", unprotectedProtectedGlobalVariable); printf("\t\t...end of
        unprotected set of threads\n");

        printf("Calling protected set of threads\n");
        pthread_mutex_init(&myMutex, NULL);
        //next set of five threads will call a function that will update the variable protected
        for( i=0;i<NUM_THREADS;i++)
        {
            status = pthread_create(&myThreads[i],NULL,protectedThreadFunc, (void*)i);
            if(status)
            {
                printf("Error in creating the threads: %d\n",i);
                return -1;
            }
        }

        //this is the area that threads will run

        //we will wait for the threads here
        for( i=0;i<NUM_THREADS;i++)
        {
            int retStatus = pthread_join(myThreads[i],NULL);
            if(retStatus)
            {
                                    printf("Well..... some problem at thread id %d, error no:
%d\n",i,retStatus);
            }
        }
        pthread_mutex_destroy(&myMutex);
        printf("Protected sum is %d \n", mutexProtectedGlobalVariable); printf("\t\t...end of
        unprotected set of threads\n");

        return 0;
}
```

Demo 4

```
/*
	Written By: Prawar Poudel
	13 Feb 2018
	This is written to demonstrate the usage of Thread Local Storage Here using identifier __thread , we have made myVal and myArr[] thread local */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

#define NUM_THREADS 5
#define ARRSIZE 5

//this value is a global variable,
// but we will store it as thread local, meaning while it is still global the value will can be modified such that the modified value is thread specific
__thread int myVal;
__thread int myArr[ARRSIZE];

void printMyVal(int id)
{
	int i;
	printf("My value myVal from thread %d is %d\n", id,myVal);
	printf("My arr value are\n");
	for( i=0;i<ARRSIZE;i++)
	{
		printf("%dthe element is %d\n",i,myArr[i]);
	}
}
//the argument that will be sent will be the (int) id
void *simpleThreadFunc(void* argument)
{
	int i;
	int myId = (int)argument;
	printf("My Id is %d\n",myId);

	//just setting some thread specific value to the variable
	myVal = myId*100;
	for( i=0;i<5;i++)
		myArr[i] = (myId*100+i);

	printMyVal(myId);
}

int main()
{
	//you can create these dynamically also
	pthread_t myThreads[NUM_THREADS];
	int status = 0;
	int i;

	for( i=0;i<NUM_THREADS;i++)
	{
```

```
		printf("Creating thread no. %d, and sending ID %d\n",i,i);
		status = pthread_create(&myThreads[i],NULL,simpleThreadFunc,(void*)i); if(status)
		{
			printf("Error in creating the threads: %d\n",i);
			return -1;
		}else
		{
			printf("Successful creation of thread..\n");
		}
	}

	//this is the area that threads will run

	//we will wait for the threads here
	for( i=0;i<NUM_THREADS;i++)
	{
		int retStatus = pthread_join(myThreads[i],NULL);
		if(!retStatus)
		{
			printf("Successful termination of thread id %d\n",i);
		}else
					printf("Well..... some problem at thread id %d, error no:
%d\n",i,retStatus);
	}

	return 0;
}
```

Demo 5

```
/*
	Written By: Prawar Poudel
	13 Feb 2018
	This is written to demonstrate the usage of Thread Local Storage using key */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

#define NUM_THREADS 5
#define ARRSIZE 5

//in this program we will use a key defined by pthread_key_t to define a key

pthread_key_t myKey;

void printMyVal(int myId)
{
	printf("Getting specific value for thread %d using key\n",myId);
	 int *myVal = pthread_getspecific(myKey);
	printf("\t..The thread local value in thread id %d is %d\n",myId,*myVal); }
```

```
//the argument that will be sent will be the (int) id
void *simpleThreadFunc(void* argument)
{
	int myId = (int)argument;
	//this variable will be thread specific value that we will print from other function
	int myVal = myId*100;
	printf("Creating the variable ::%d that will be referred to by Key from threadid
%d\n",myVal,myId);
	if(!pthread_setspecific(myKey,(void*)&myVal))
	{}else
	{
		printf("Erorr in setting specific key in thread id %d\n",myId); }

	printMyVal(myId);
}

int main()
{
	//you can create these dynamically also
	pthread_t myThreads[NUM_THREADS];
	int status = 0;
	int i;
	pthread_key_create(&myKey,NULL);

	for( i=0;i<NUM_THREADS;i++)
	{
		printf("Creating thread no. %d, and sending ID %d\n",i,i);
		status = pthread_create(&myThreads[i],NULL,simpleThreadFunc,(void*)i); if(status)
		{
			printf("Error in creating the threads: %d\n",i);
			return -1;
		}else
		{
			printf("Successful creation of thread %d..\n",i);
		}
	}
	//this is the area that threads will run

	//we will wait for the threads here
	for( i=0;i<NUM_THREADS;i++)
	{
		int retStatus = pthread_join(myThreads[i],NULL);
		if(!retStatus)
		{
			printf("Successful termination of thread id %d\n",i);
		}else
					printf("Well..... some problem at thread id %d, error no:
%d\n",i,retStatus);
	}

	return 0;
}
```

Assignment

Write a parallel program to compute the definite integral using Rectangular decomposition:

$$4 * \int_0^1 \sqrt{1 - x^2}\, dx$$

Write a program in a general pthreads manner, so that they can utilize an arbitrary number of threads so the computation is divided among these computational entities as evenly as possible. Design program so that the **number of intervals and the number of operating threads** can be entered as a **run time parameter (i.e. command line arguments)** by the user under the constraint that it must be greater than or equal to the number of threads that are used.

You are expected to do the following tasks:

- Write a serial code version to compute integral, using a timing function to evaluate its execution time.
- Write a parallel version of code using pthreads model, compare the result and execution time.
- **Compare and graph** the performance of the two models (serial and pthreads) in terms of time and number of intervals used.. For this comparison use the following set of intervals: 1,000, 10,000, 100,000, and 1,000,000). For the pthread model use only 2 threads.
- **Compare and graph** the performance of the serial model and parallel based on the execution time and number of threads for 100,000 intervals. The graph should include serial vs parallel with 1, 2, 4, 8 and 16 threads.

Sample Output

Your programs are expected to output something similar to the following:

Serial Method	Parallel Method

`-bash-4.2$ ./serial_decomp 10000000`
`Test of rect_decomp, 3.141593`
`Time = 0.031398 seconds`
	`-bash-4.2$ ./par_decomp 10000000 2`
	`Test of rect_decomp, 3.141593`
	`Time = 0.017692 seconds`

Hint:

You need to study the integral using rectangular decomposition, or the trapezoidal method. The integral should evaluate to approximately 3.14. You may need to include the math.h library for the sqrt() method and link it with **-lm** for your programs. Regarding timing, an example of how to do timing measurements is given below.

Timing Example

You may use the following program as a starting point for timing execution:

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

#define TIMER_CLEAR  (tv1.tv_sec = tv1.tv_usec = tv2.tv_sec = tv2.tv_usec = 0)
#define TIMER_START  gettimeofday(&tv1, (struct timezone*)0)
#define TIMER_ELAPSED   (double) (tv2.tv_usec-tv1.tv_usec)/1000000.0+(tv2.tv_sec-tv1.tv_sec)
#define TIMER_STOP    gettimeofday(&tv2, (struct timezone*)0)
struct timeval tv1,tv2;

int main(int argc, char* argv[])
{
    int i;
    TIMER_CLEAR;
    TIMER_START;
    // Call to function that you want to time
    // example
    sleep(5); //Timer will give us ~5 seconds
    TIMER_STOP;
    printf("Time elapsed = %f seconds\n", TIMER_ELAPSED);
    return 0;
}
```

Topics for Theory

- Difference between Threads and Processes
- Rectangular Decomposition Method
- Thread Local Storage
- Mutex and Semaphore

Deliverables

Lab Report

The following material in each section is expected:

1. Cover page with your name, lab number, course name, and dates
2. Theory/Background (Material or methods relevant to the lab, a few sentences on each in your own words)
    a. Difference between Threads and Processes
    b. Rectangular Decomposition Method
    c. Thread Local Storage
    d. Mutex and Semaphore
3. Observations
    a. Comparison and graph of performance of the two models (serial and pthreads).This should be between the serial method vs the parallel method with 2 pthreads for intervals 1,000, 10,000, 100,000, and 1,000,000. Program output should be included here, each program should print out the integral and the time it took to compute it.
    b. Comparison and graph of performance between the serial model and parallel model based on execution time. Should include the serial method vs the parallel method with 1, 2, 4, 8, and 16 threads for 100,000 intervals. Program output should be included here, each program should print out the integral and the time it took to compute it.
4. Conclusion (Did your program work as expected, what can you take away from the lab?)
5. Appendix (for source code, submit the text in a table)

The report should be submitted as a single pdf document with the source code for your program within it.

Recorded Demonstration

The following material in each section is expected:

1. Introduce yourself and give the name of the lab
2. Compile the program
3. Show that the programs run similar to what was reported under observations

The demonstration may be in person or recorded and submitted an mp4 file alongside the report.