CSUS, College of Engineering and Computer Science
Department of Computer Science
CSC/CPE 139 – Operating System

Programming Assignment 4 – Multi-threading

Acknowledgement: Thanks to Prof. Ghassan Shobaki for this assignment
Goal: To enable concrete understanding of multi-threading to you

Instructions:
In this assignment, you will experiment with the performance impact of multithreading using real time measurements. You will use the POSIX thread library on a UNIX system. Your job is to write a program that sorts an array of random integers first sequentially and then using multi-threading. Your program should take the following three command-line arguments:

1. The array size (a positive integer between 1 and 100 000 000)
2. The number of threads (a positive integer between 1 and 16)
3. The sorting algorithm (I for InsertionSort, Q for QuickSort)

Your program will perform the following steps:

1. Fill the array with random numbers
2. Based on the number of threads (T), compute the indices for dividing the array into T equal parts. For example, if the array size (N) is 1000 and T is 2, the indices will be 0, 499, 500, 999.
3. Sort sequentially by applying the sorting algorithm to each part of the array, and then combining the sorted parts into one sorted array using an O(n) algorithm.
4. Apply an O(n) algorithm to check if the array has been sorted correctly, and print a message indicating correct/incorrect sorting.
5. Refill the array with random numbers.
6. Sort using multi-threading. This step should be done the same way you did sequential sorting with the only difference that the sorting of each part is done in a separate thread. Combining the sorted parts into one sorted array should be done in the main (parent) thread after all child threads have completed. So, the parent thread must wait for all child threads.
7. Apply an O(n) algorithm to check if the array has been sorted correctly, and print a message indicating correct/incorrect sorting.

To measure the time, use the code provided below. All what you need to do is calling SetTime() to start the timing and GetTime() to get the time elapsed since the last call to SetTime(). So, you will first call SetTime right before Step 3 and GetTime right after its completion and print that time as the sequential sorting time. Then, do the same before and after Step 6 and print that time as the multithreaded sorting time.

Use the following command to compile your code:

```
g++ -O3 tsort.c -lpthread -o tsort
```

Note that we are using the –O3 option to enable a high-level of compiler optimizations. For your own benefit, you may want to try compiling it without the –O3 option and measuring the difference in speed when you run it.

When your program is working correctly, run the following tests on server athena.ecs.csus.edu:

1. Run InsertionSort using two threads with array sizes 10K, 100K and 300K.
2. Run InsertionSort using four threads with an array size of 100K.
3. Run QuickSort using two threads with array sizes 1M, 10M and 100M.
4. Run QuickSort using four threads with an array size of 10M.

Deliverable:
Submit the following through SacCT, in a tar-ball:

1. Your source code in c language, and corresponding executables: .o files;
2. A document having the following:
   a. necessary screenshots;
   b. four **tables** summarizing the results of the tests that you ran (one Table for each test)
   c. A brief report discussing the results and the conclusions that you have made

Requirement: The report will all be evaluated based on the following grading criteria.

Functionality	60%
Robustness	40%

Code for Timing and Sorting (**no robustness guaranteed**, you need to validate and correct if any)

```
#include <sys/timeb.h>
#include <stdio.h>
#include <stdlib.h>

long gRefTime;

long GetMilliSecondTime(struct timeb timeBuf)
{
    long mliScndTime;

    mliScndTime = timeBuf.time;
    mliScndTime *= 1000;
    mliScndTime += timeBuf.millitm;
    return mliScndTime;
}

long GetCurrentTime(void)
{
    long crntTime=0;
```

```
	struct timeb timeBuf;
	ftime(&timeBuf);
	crntTime = GetMilliSecondTime(timeBuf);

	return crntTime;
}

void SetTime(void)
{
	gRefTime = GetCurrentTime();
}

long GetTime(void)
{
	long crntTime = GetCurrentTime();

	return (crntTime - gRefTime);
}

void InsertionSort(int data[], int size)
{
	int i, j, temp;

	for(i=1; i<size; i++)
	{
		temp = data[i];
		for(j=i-1; j>=0 && data[j]>temp; j--)
			data[j+1] = data[j];
		data[j+1] = temp;
	}
}

void QuickSort(int data[], int p, int r)
{
	int q;
	if(p >= r) return;
	q = Partition(data, p, r);
	QuickSort(data, p, q-1);
	QuickSort(data, q+1, r);
}

int Partition(int data[], int p, int r)
{
	int i, j, x, pi;
```

```
	pi = Rand(p, r);
	Swap(data[r], data[pi]);

	x = data[r];
	i = p-1;
	for(j=p; j<r; j++)
	{
		if(data[j] < x)
		{
			i++;
			Swap(data[i], data[j]);
		}
	}
	Swap(data[i+1], data[r]);
	return i+1;
}

void Swap(int& x, int& y)
{
	int temp = x;
	x = y;
	y = temp;
}

int Rand(int x, int y)
{
	int range = y-x+1;
	int r = rand() % range;
	r += x;
	return r;
}
```