CPE 435: OPERATING SYSTEMS LABORATORY.

Lab06

<u>POSIX Threads.</u>

Submitted by: <u>Dan Otieno.</u>

Date of Experiment: 02/22/23.

Report Deadline: 02/26/23.

Demonstration Deadline: 02/29/23.

Introduction:

This lab served as an introduction to Threads in an operating system, and how they can be used to carry out tasks. For this lab, we are going to use threads for rectangular decomposition in the calculation of integrals.

Theory:

According to the lab manual, 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). Some of the theory topics explored in this lab are below:

- Processes vs. Threads:
- LINK TO SOURCE.
    - Processes are programs defined within the Process Control Block, which are scheduled for execution in the CPU. An example of a process involves forking, where child processes are created to complete different tasks. Processes can be in the following states:
        - New.
        - Ready.
        - Running.
        - Waiting.
        - Terminated.
        - Suspended.

    - Threads are part of a process. They are segments within a process, and one process can include several threads. Threads take lesser time to terminate, compared to processes, and do not isolate. A thread can have the following states:
        - Running.
        - Ready.
        - Blocked.

- Rectangular Decomposition Method:
    - This is a method of approximation for integrals where we determine the area under a curve by using Riemann's sum of rectangles. The more rectangles we set under the curve, the more accurate we can determine the area. We calculate the area of the rectangle and sum the results to determine the overall area under the curve. We set upper and lower limits, and the width of each rectangle lies on the x-axis, while the height spans the y-axis.

- Thread Local Storage:
- LINK TO SOURCE 1
- LINK TO SOURCE 2
    - This is a memory management method that uses global memory local to a thread, also denoted as TLS. TLS provides foundations to build POSIX interfaces for allocating thread-specific data. It also offers a more convenient and more efficient mechanism for direct use by applications and libraries, and It allows compilers to allocate TLS as necessary when performing loop-parallelizing optimizations.

- Mutex & Semaphore:
- LINK TO SOURCE.
    - Both Mutexes and Semaphores are kernel resources that provide synchronization services in an operating system. According to the definition in the lab lecture, A mutex is a MUTual EXclusion device, and is useful for protecting shared data structures from concurrent modifications and implementing critical sections. A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A semaphore is a generalized mutex.

Results & Observation:

Assignment 1:

Description:

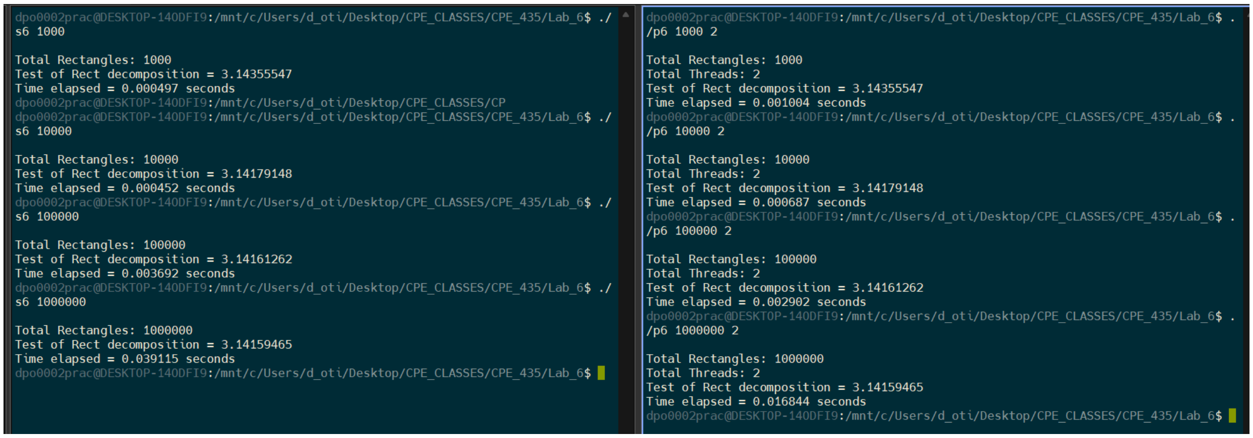
The goal for this assignment was to use C/C++ to write two programs, a parallel and serial program. The parallel program uses an arbitrary number of threads to decompose rectangles in intervals and compute the following integral:

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

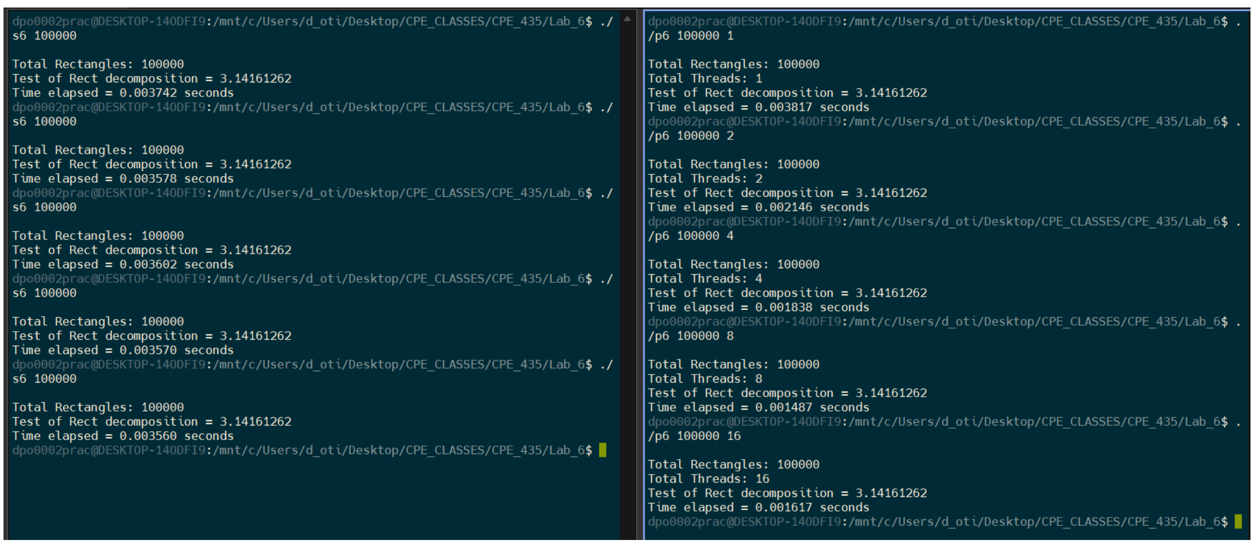
We also include a timer function to determine which program executes the decomposition quicker with the same intervals, i.e., number of rectangles. Finally, using the execution times noted in the test runs, we will include performance graphs comparing the two models.

Program Outputs:

- Side-by-side comparison of the serial and parallel (pthreads) model outputs. For this test, we use 1,000, 10,000, 100,000 and 1,000,000 rectangles (intervals), and for the pthreads model we use 2 threads for all intervals.



- For the next test run, we use 100,000 intervals, or, rectangles, and we adjust the number of threads for each run, we use 1, 2, 4, 8 and 16 threads and compare execution time against the serial model. The outputs are captured below:

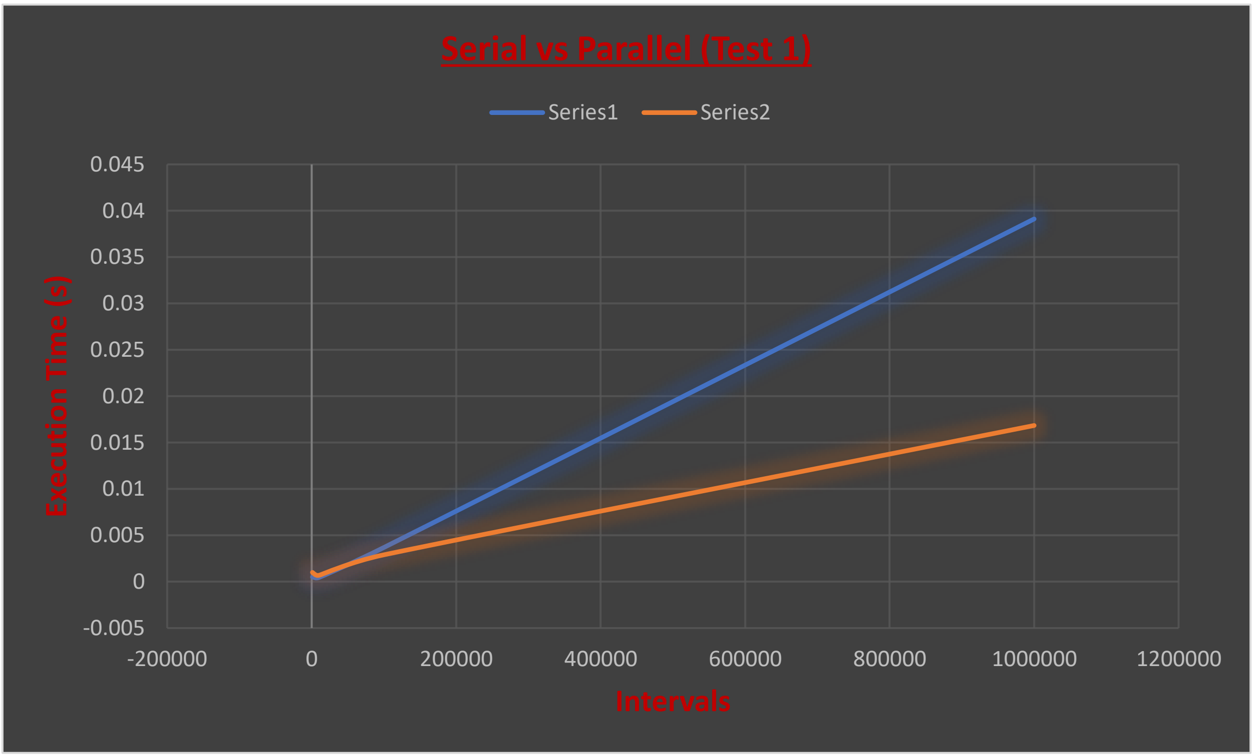


Serial vs Parallel comparison plots:

The following table shows the comparison data collected for the first test run, where we have 2 threads for the parallel method, and we test several intervals. Below the table, we can also see the comparison plots for the two methods. The x-axis has the number of intervals, and the y-axis has the execution time.

Table 1

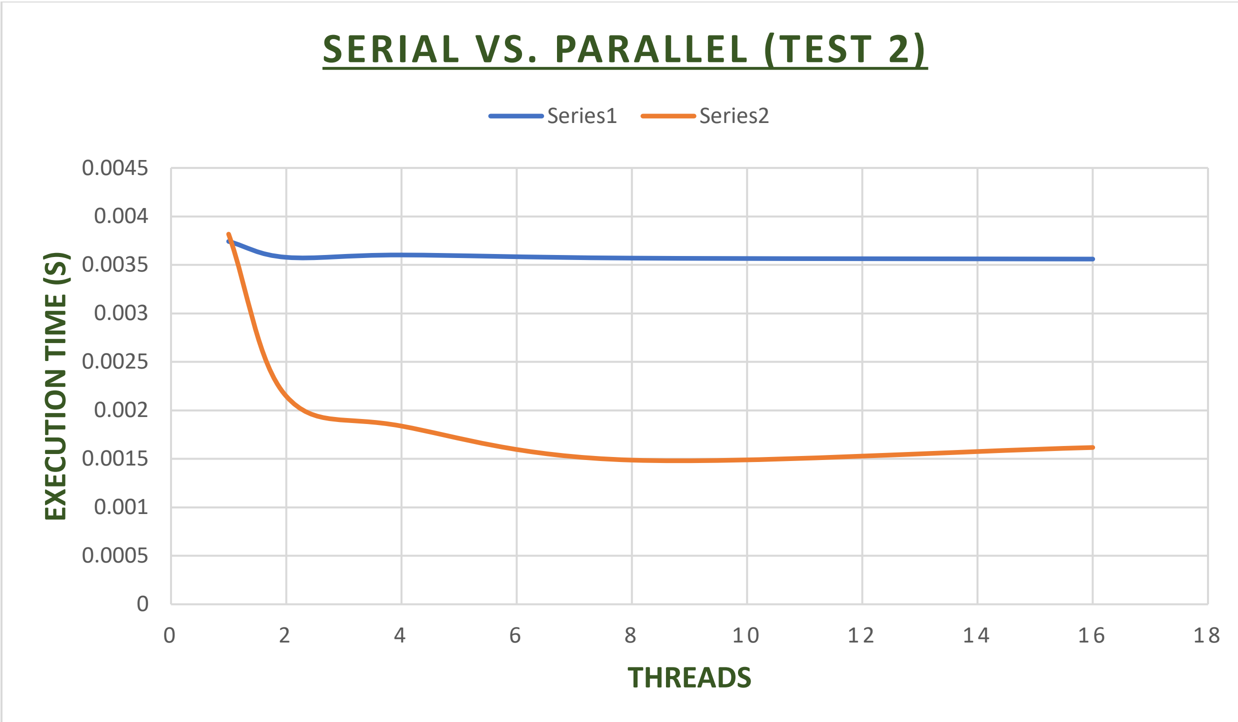
Intervals	Serial Model Time (s)	Parallel Model Time (s)
1000	0.000497	0.001004
10000	0.000452	0.000687
100000	0.003692	0.002902
1000000	0.039115	0.016844



The following table shows the data collected for the second test run, where we had a 100,000 intervals for a serial vs. parallel run with 1, 2, 4, 8 and 16 threads.

Table 2

Threads	Serial Model Time (s)	Parallel Model Time (s)
1	0.003742	0.003817
2	0.003578	0.002146
4	0.003602	0.001838
8	0.00357	0.001487
16	0.00356	0.001617



Conclusion:

All programs worked as expected, and it was interesting to see the efficiency of threads in executing tasks within the operating system. While the serial model was great at rectangular decomposition, using threads in the parallel model was much more efficient and completed the process in a shorter period.

Appendix:

Assignment 1 code – Serial Model.

```
/*
DAN OTIENO
CPE 435-01
LAB 6
Serial Model.
*/

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <math.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;

unsigned long rects;

double func(double val)
{
	return (sqrt(1 - pow(val, 2)));
}

double Width(double upper, double lower, int num)
{
   double width;

   width = ((upper - lower) / num);

   return width;
}

double Height(double width, int i)
{
	double x, height;

	x = width * i;
   height = func(x);

   return height;
}
```

```
int main(int argc, char* argv[])
{
	double sum=0, w;
	int i;

	TIMER_CLEAR;
	TIMER_START;

	if(argc != 2)
	{
		printf("Two input arguments required, please try again.\n");
		printf("Program has exited!\n");
	}

	rects = atoi(argv[1]);

	if(rects < 0)
	{
		printf("Error! Input must be a positive number, please try again.\n");
		printf("Program has exited!\n");
		return 1;
	}

  w = Width(1, 0, rects);

  for(i = 0; i < rects; i++)
  {
    sum += w * Height(w, i);
  }

  sum *= 4;
  printf("\nTotal Rectangles: %ld\n", rects);
  printf("Test of Rect decomposition = %1.9g\n", sum);
  TIMER_STOP;
	printf("Time elapsed = %f seconds\n", TIMER_ELAPSED);
	return 0;
}
```

<u>Assignment 1 code – Parallel Model.</u>

```
/*

DAN OTIENO

CPE 435-01

LAB 6

Parallel Model.

*/

#include <pthread.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <time.h>

#include <sys/time.h>

#include <math.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;


unsigned long threads, rects;

double rng, sum;

pthread_mutex_t mutx;


double func(double val)

{

        return(sqrt(1 - pow(val, 2)));

}

void* Sum(void* arg)

{

        double x, temp = 0;
```

```
        unsigned long i;
        int aStart = *(int*) arg;

        for (i = aStart; i < rects; i += threads)
        {
                x = rng * i;
                temp += rng * func(x);
        }

        pthread_mutex_lock(&mutx);
        sum += temp;
        pthread_mutex_unlock(&mutx);
        return 0;
}
int main(int argc, char* argv[])
{
        int i, j, rStatus;

        TIMER_CLEAR;
        TIMER_START;

        if(argc != 3)
        {
                printf("Three input arguments required, please try 
again.\n");
                printf("Program has exited!\n");
                return 1;
        }

        sum = 0;
        rects = atoi(argv[1]);
        threads = atoi(argv[2]);
```

```
        printf("\nTotal Rectangles: %ld\n", rects);
        printf("Total Threads: %ld\n", threads);

        rng = 1 / (double)rects;
        pthread_t myThreads[threads];
        int start[threads];
        if(rects < 0)
        {
                printf("Error! Input must be a positive number, please try
again.\n");
                printf("Program has exited!\n");
                return 1;
        }

    for(i = 0; i < threads; i++)
        {
                start[i] = i;
                if(pthread_create(&myThreads[i], 0, Sum, &start[i]))
                {
                        printf("Error! Thread creation failed!\n");
                        printf("Program has exited!\n");
                        return 1;
                }
        }

        for(j = 0; j < threads; j++)
        {
                rStatus = pthread_join(myThreads[j], 0);

                if(rStatus)
                {
                        printf("Error encountered at thread%d.\n", j);
                        printf("Error number:%d\n", rStatus);
```

```
                }
        }

    pthread_mutex_destroy(&mutx);
    sum *= 4;
        printf("Test of Rect decomposition = %1.9g\n", sum);

    TIMER_STOP;
        printf("Time elapsed = %f seconds\n", TIMER_ELAPSED);

        return 0;
}
```