Andrew Ma

Lab 6

CPE 435

2/22/21

Theory

Difference Between Threads and Processes

The main difference between processes and threads is that threads run in a shared memory space, while processes have separate memory space. Processes are created with fork(), and creating a new process is expensive since memory is copied from the parent to the child. Inter process communication is needed for parent and child process communication, as shown in previous labs. Threads are lightweight and faster to create than processes, and all threads share the same global memory unless the variable is declared with __thread. In that case, the variable would be thread specific.

Rectangular Decomposition Method

To calculate the integral of f(x) from endpoints a to b, we can use rectangular decomposition for an approximation of the area under the plotted function. We can divide the plotted function into rectangles, and multiply the length and width to get the rectangle's areas. We can sum up the areas of all the rectangles to get the approximated area. To increase the accuracy of the approximation, we can divide the plotted function into smaller rectangles (increasing the number of rectangles).

Thread Local Storage

Thread local storage is memory specific to each thread. Normally, global memory is shared between all threads, but declaring a variable with __thread makes it specific to each thread. We can also use pthread keys to create thread specific variables.

Mutex and Semaphore

A mutex is a mutual exclusion lock. It is a binary locking mechanism that blocks access to variables by other threads. This gives exclusive access to a thread and blocks other threads. A semaphore is also a locking mechanism, but it is not binary. It is a integer that can be used to synchronize multiple threads by incrementing and decrementing the counter. Checking or modifying the value of a semaphore can be done safely without creating a race condition. A semaphore can wait, which decrements the semaphore counter. If the counter is already 0, the operation blocks until the counter is > 0. When the counter becomes positive, it is decremented by 1 and the wait operation returns and execution can continue. A post operation increments the counter. If the semaphore was 0 and threads are blocked and waiting, then incrementing the counter with post will unblock a waiting thread.

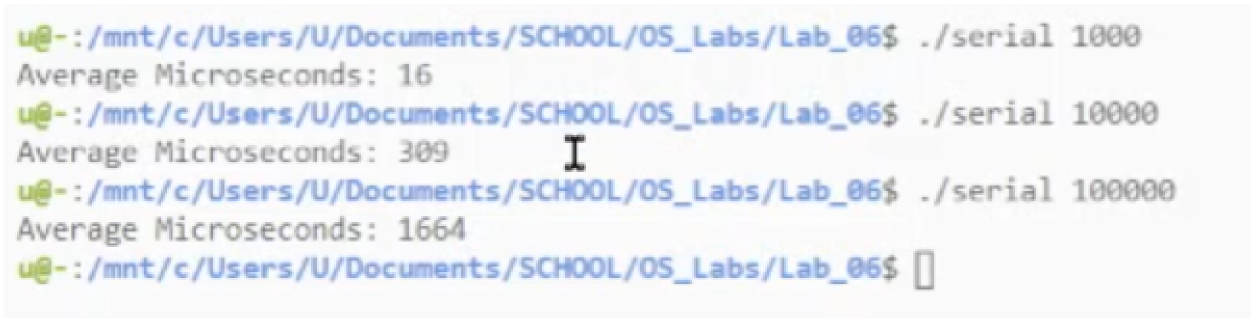
The `int pthread_create(pthread_t* thread, pthread_attr_t const* attr, void* (*start_routine) (void*), void* arg)` function starts a new thread in the calling process. The new thread invokes start_routine, and arg is passed into start_routing as the only argument. The new thread can terminate by (1) calling pthread_exit(), (2) returning from start_routine() which is same as calling pthread_exit() with return value, (3) being canceled with pthread_cancel(), or (4) if any threads in process calls exit() or if the main thread returns from main(). If the thread is successfully created, then the thread ID will be stored in the argument *thread*, and it will return 0. If there is an error, then it returns the error number.

The `int pthread_join(pthread_t thread, void** retval)` function waits for the thread specified by argument *thread* to terminate. If the thread has already terminated, then pthread_join() returns immediately. If argument *retval* is not NULL, then pthread_join() copies the exit status of the target thread into the location pointed to by *retval*. If the target thread was canceled with pthread_cancel(), then PTHREAD_CANCELED is put in location pointed to by *retval*. If multiple threads simultaneously try to join with the same thread, the results are undefined. If pthread_join() is successful, it returns 0. Otherwise it will return the error number (deadlock detected, thread is not a joinable thread, another thread is waiting to join with this thread, no thread with the ID *thread* could be found).

Observations

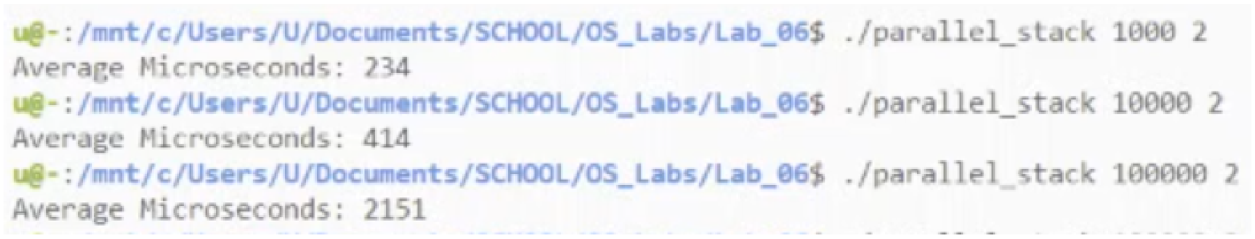
- Comparison of performance of the two models (serial and pthreads) - serial vs 2 pthreads for intervals 1000, 10000, 100000

  Serial with varying intervals

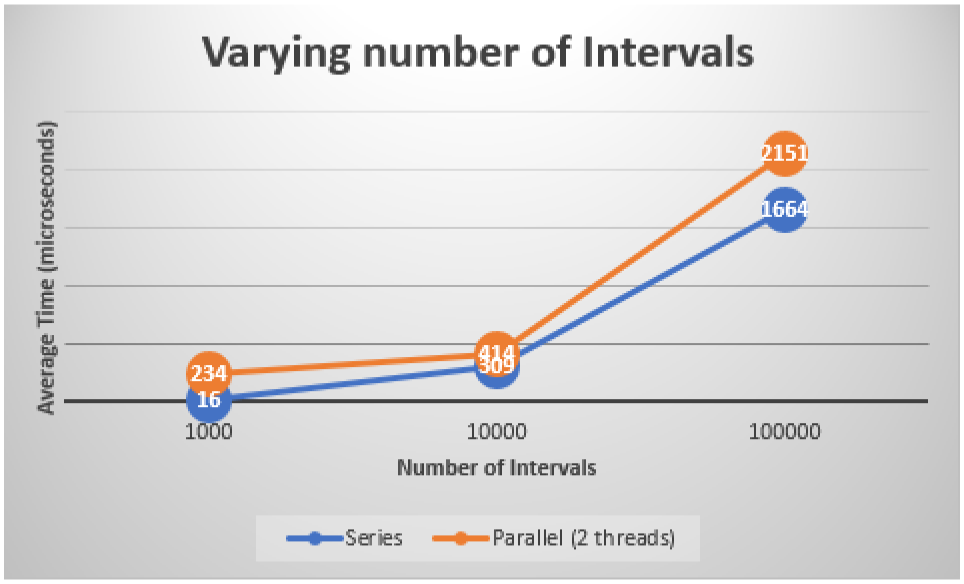


```
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./serial 1000
Average Microseconds: 16
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./serial 10000
Average Microseconds: 309
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./serial 100000
Average Microseconds: 1664
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$
```

Parallel with 2 threads and varying intervals

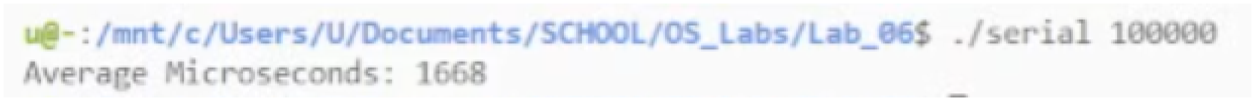


```
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./parallel_stack 1000 2
Average Microseconds: 234
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./parallel_stack 10000 2
Average Microseconds: 414
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./parallel_stack 100000 2
Average Microseconds: 2151
```



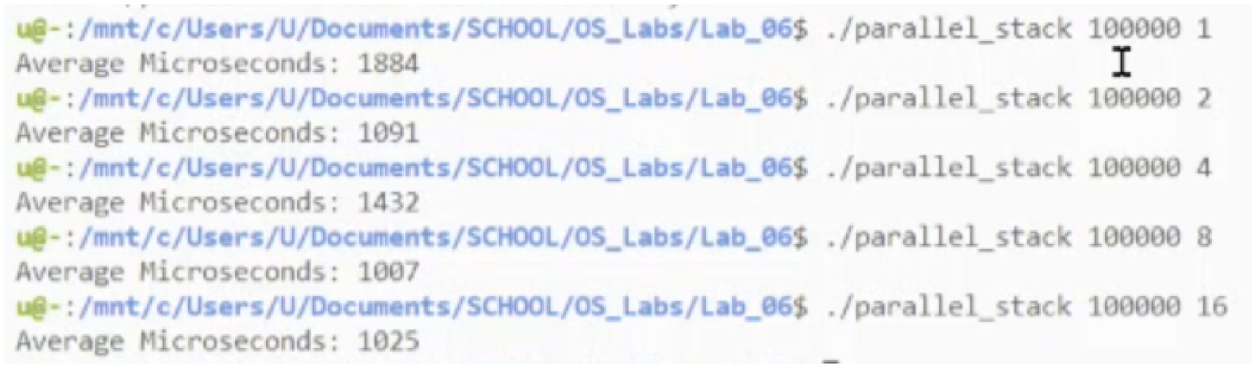
- Comparison of performance between serial and parallel model based on execution time - serial vs parallel with 1, 2, 4, 8, 16 threads for 100,000 iterations

Serial with fixed interval

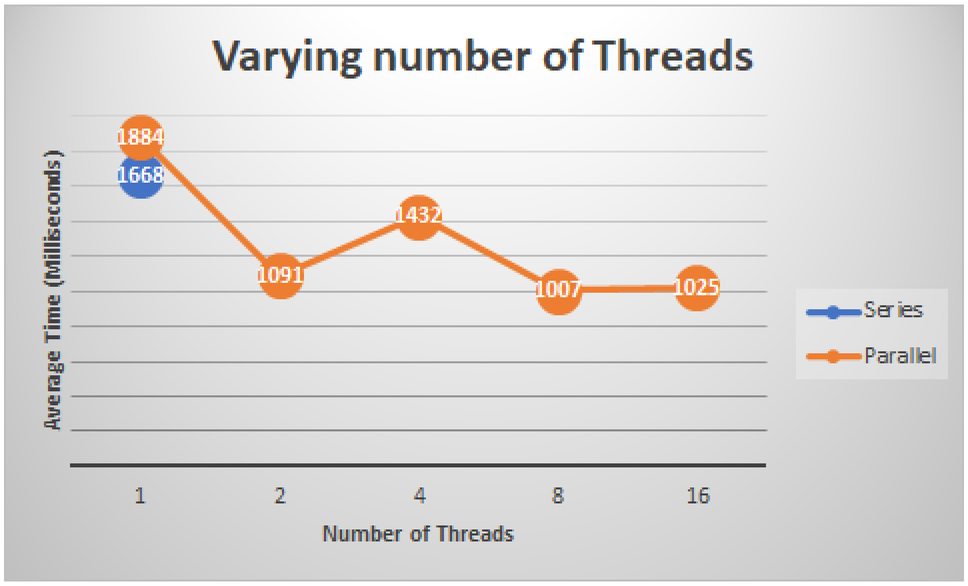


```
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./serial 100000
Average Microseconds: 1668
```

Parallel with fixed interval and varying threads



```
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./parallel_stack 100000 1
Average Microseconds: 1884
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./parallel_stack 100000 2
Average Microseconds: 1091
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./parallel_stack 100000 4
Average Microseconds: 1432
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./parallel_stack 100000 8
Average Microseconds: 1007
u@-:/mnt/c/Users/U/Documents/SCHOOL/OS_Labs/Lab_06$ ./parallel_stack 100000 16
Average Microseconds: 1025
```



Conclusion

The program worked. What was unexpected was that with a varying number of intervals from 1000 to 100000, the series solution outperformed the parallel with 2 threads solution. But, I believe this is because of the extra setup needed in the parallel solution with pthread_create(), pthread_join(), dividing the total number of intervals into subintervals, and allocating the thread argument structures. With the varying number of threads, the output was expected and the multithreaded solution outperformed the series solution with greater than 1 threads. From this lab I learned how to use pthreads to create a multithreaded solution. Even though I did not use these in the source code, I learned from the demo code how to use pthread mutexes, thread local storage, and pthread keys.

Source Code

```c
/* Serial */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <limits.h>
#include <math.h>
#include <sys/time.h>

#define NUM_TESTS 10

int main(int argc, char** argv) {
    if (argc != 2) {
        printf("Usage: program <Number Of Intervals>\n");
        return 0;
    }

    char* end;

    errno = 0;
    unsigned long const NUM_INTERVALS = strtoul(argv[1], &end, 10);

    if (end == argv[1]) {
        fprintf(stderr, "%s: not a decimal number\n", argv[1]);
        return -1;
    }
    else if ('\0' != *end) {
        fprintf(stderr, "%s: extra characters at end of input: %s\n", 
argv[1], end);
        return -1;
    }
    else if ((0 == NUM_INTERVALS || ULONG_MAX == NUM_INTERVALS) && 
ERANGE == errno) {
        fprintf(stderr, "%s out of range of type unsigned long\n", 
argv[1]);
        return -1;
    }
    else if (NUM_INTERVALS == 0) {
        fprintf(stderr, "Can't have zero intervals\n");
        return -1;
    }

    unsigned char const UPPER_BOUND = 1;
    unsigned char const LOWER_BOUND = 0;

    // Setup Timer
    struct timeval startTime, stopTime;

    long long totalTestMicroseconds = 0;

    // Loop NUM_TESTS times to and get average elapsed time
    for (int i = 0; i < NUM_TESTS; ++i) {

        gettimeofday(&startTime, NULL);

        double const EACH_RECTANGLE_WIDTH = ((double)UPPER_BOUND - 
LOWER_BOUND) / NUM_INTERVALS;

        double sum = 0.0;

        for (int i = 0; i < NUM_INTERVALS; ++i) {
            double const x = (double)i * EACH_RECTANGLE_WIDTH +
LOWER_BOUND;
            sum += 4.0 * (EACH_RECTANGLE_WIDTH * (sqrt(1 - x * x)));
        }

        // Stop Timer
        gettimeofday(&stopTime, NULL);

        // printf("Sum %f\n", sum);

        long long elapsedMicroseconds = (long long)(stopTime.tv_usec -
startTime.tv_usec) + (long long)((stopTime.tv_sec - startTime.tv_sec) *
1000000);
        // printf("Time: %lld microseconds\n", elapsedMicroseconds);

        totalTestMicroseconds += elapsedMicroseconds;
    }

    printf("Average Microseconds: %lld\n", totalTestMicroseconds /
NUM_TESTS);

    return 0;
}
```

```
/* Parallel */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <limits.h>
#include <math.h>
#include <sys/time.h>

#define NUM_TESTS 10
```

```c
unsigned char const UPPER_BOUND = 1;
unsigned char const LOWER_BOUND = 0;
double EACH_RECTANGLE_WIDTH;


struct ThreadArgument {
    unsigned long start;
    unsigned long end;
    double sum;
};

void* computeIntegral(void* argument) {
    struct ThreadArgument* arg = (struct ThreadArgument*)argument;

    arg->sum = 0.0;
    for (unsigned long i = arg->start; i < arg->end; ++i) {
        double const x = (double)i * EACH_RECTANGLE_WIDTH + LOWER_BOUND;
        arg->sum += 4.0 * (EACH_RECTANGLE_WIDTH * (sqrt(1 - x * x)));
    }
    pthread_exit(NULL);

}

int main(int argc, char** argv) {
    if (argc != 3) {
        printf("Usage: program <Number Of Intervals> <Number Of 
Threads>\n");
        return 0;
    }

    char* end;

    errno = 0;
    long const NUM_INTERVALS = strtoul(argv[1], &end, 10);
    if (end == argv[1]) {
        fprintf(stderr, "%s: not a decimal number\n", argv[1]);
        return -1;
    }
    else if ('\0' != *end) {
        fprintf(stderr, "%s: extra characters at end of input: %s\n",
argv[1], end);
        return -1;
    }
    else if ((0 == NUM_INTERVALS || ULONG_MAX == NUM_INTERVALS) &&
ERANGE == errno) {
        fprintf(stderr, "%s out of range of type unsigned long\n",
argv[1]);
        return -1;
    }
    else if (NUM_INTERVALS == 0) {
        fprintf(stderr, "Can't have zero intervals\n");
        return -1;
    }


    errno = 0;
    unsigned long const NUM_THREADS = strtoul(argv[2], &end, 10);

    if (end == argv[2]) {
        fprintf(stderr, "%s: not a decimal number\n", argv[2]);
        return -1;
    }
    else if ('\0' != *end) {
        fprintf(stderr, "%s: extra characters at end of input: %s\n",
argv[2], end);
        return -1;
    }
    else if ((0 == NUM_THREADS || ULONG_MAX == NUM_THREADS) && ERANGE ==
errno) {
        fprintf(stderr, "%s out of range of type unsigned long\n",
argv[2]);
        return -1;
    }

    // Setup Timer
    struct timeval startTime, stopTime;

    long long totalTestMicroseconds = 0;

    // loop NUM_TESTS times to and get average elapsed time
    for (int i = 0; i < NUM_TESTS; ++i) {

        gettimeofday(&startTime, NULL);


        // Create Array of pthreads
        pthread_t myThreads[NUM_THREADS];

        // the ThreadArgument structures are allocated in the main
stack, so we can address (ptr) the structures within thread functions
        struct ThreadArgument threadArguments[NUM_THREADS];


        EACH_RECTANGLE_WIDTH = ((double)UPPER_BOUND - LOWER_BOUND) /
NUM_INTERVALS;


        unsigned long const INTERVALS_PER_THREAD =
((double)NUM_INTERVALS / NUM_THREADS);
        unsigned long const LEFTOVER_INTERVALS_FOR_LAST_THREAD =
NUM_INTERVALS % NUM_THREADS;

        // printf("Intervals per thread %ld, and leftover %ld\n",
INTERVALS_PER_THREAD, LEFTOVER_INTERVALS_FOR_LAST_THREAD);

        /* Creating Threads */
        for (int i = 0; i < NUM_THREADS; ++i) {
            unsigned long start = i * INTERVALS_PER_THREAD;
            unsigned long end = start + INTERVALS_PER_THREAD - 1;

            if (i == NUM_THREADS - 1) {
                // Add leftovers to last thread
                end += LEFTOVER_INTERVALS_FOR_LAST_THREAD;
            }

            // printf("Start %lu, End %lu\n", start, end);

            threadArguments[i] = (struct ThreadArgument){ .start =
start, .end = end, .sum = 0.0 };

            int createThreadStatus = pthread_create(&myThreads[i], NULL,
computeIntegral, (void*)&threadArguments[i]);
            if (createThreadStatus != 0) {
                fprintf(stderr, "Error creating thread %d\n", i);
            }
        }


        double totalSum = 0.0;

        // /* Joining Threads */
        for (int i = 0; i < NUM_THREADS; ++i) {
            pthread_join(myThreads[i], NULL);

            totalSum += threadArguments[i].sum;

            // printf("Thread %d: %f\n", i, threadArguments[i].sum);
        }

        // Stop Timer
        gettimeofday(&stopTime, NULL);
```

```c
        // printf("Sum: %f\n", totalSum);

        long long elapsedMicroseconds = (long long)(stopTime.tv_usec -
startTime.tv_usec) + (long long)((stopTime.tv_sec - startTime.tv_sec) *
1000000);

        // printf("Time: %lld microseconds\n", elapsedMicroseconds);

        totalTestMicroseconds += elapsedMicroseconds;
    }

    printf("Average Microseconds: %lld\n", totalTestMicroseconds /
NUM_TESTS);

    return 0;
}
```