Name: Gia Dao

Student ID: 10017047062

Date: March 27th, 2021

Programming Assignment 2 Report

The purpose of this project is to practice threaded programming by solving various problems. The objectives of this project is to learn:

1. Get familiar with threading.
2. How to use mutexes, semaphores, and conditional variables in a threading library.
3. How to design efficient solutions for mutual exclusion problems.

Part 1:

Problem explanation:

Given two strings s1 and s2. Write a threaded program to find out the number of substrings, in string s1, that is exactly the same as s2.

Evaluation metrics that will be considered in this case will be:

- NUM_THREADS: the number of threads running in parallel will be defined as a constant and can be modified due to user's preference, assume that n1 (the size of s1) mod NUM_THREADS = 0 and n2 (the size of s2) < n1 / NUM_THREADS.

- MAX: the number of maximum characters in the string given by the user.

- mtime: the calculated execution time of the program.

Choice of threading libraries:

Use POSIX thread or pthread for part 1 since the implementation of pthread is available with gcc compiler and the library is already installed in C, which is the <pthread.h>. Pthread is also common thread and easy to use.

Design of the experiment and develop programs for evaluation:

General solution for part 1 is to partitioned string s1 into equal size of substrings, and the size of each substrings is equal to the length of s1 divided by NUM_THREADS. Then, NUM_THREADS threads will then search each substring concurrently to find the matching string with s2. An array, which has the size of NUM_THREADS, will be created to save the total of local matching strings in each substring. Finally, sum up all the values in the array to a global variable, which is the number of s2 in s1.

First, calculate the size of local strings. In the program, "part_size = n1 / NUM_THREADS", where n1 is the length of string s1. Then, run a for loop from 0 to NUM_THREADS – 1. For each iteration, the program will save the starting index of each local string, and save them in an array. By passing the address of the starting index of the local string, each thread will then search for the match of s2 concurrently starting at that index. The function "num_substring()" will be responsible for calculating the total number of matching strings in each local string. To guard the critical region from being accessed at the same time by all the threads, we created a mutex at the top of the program "pthread_mutex_t mutex". As mentioned previously, "sum[NUM_THREADS]" array will be created as global and saved the total of s2 found per local strings. After all the threads are joined, goes through sum array and add all the values to "total" variable. "total" is the result that needs to be printed out.

For part 1, the execution time will be measured. If we were to created more NUM_THREADS, specifically, increase from one thread to two, then to four threads, the execution time is expected to be reduced since parallelism will reduce the workload. Create variables mtime of type float, secs and usecs of type int, the execution time will be measured by the following:

secs = end.tv_sec – start.tv_sec;

usecs = end.tv_usec – start.tv_usecs;

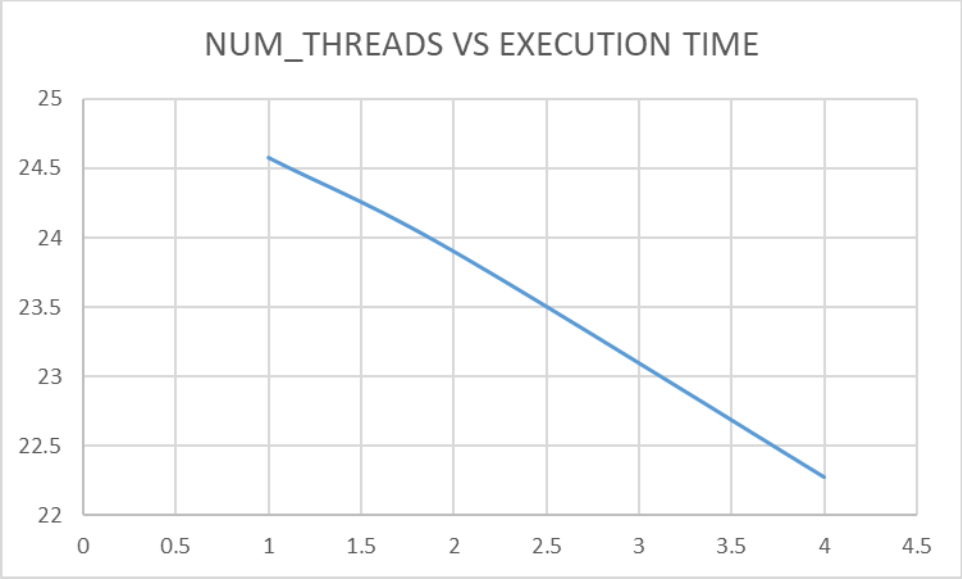
mtime = ((secs) * 1000 + usecs / 1000.0) + 0.5;

Measurement time is the variable mtime and will be printed out.

Detail collected experimental results:

The original program with no threads created has execution time: 24.27199 milliseconds

NUM_THREADS	EXECUTION TIME
1	24.579000 milliseconds
2	23.903000 milliseconds
4	22.274000 milliseconds



As we can see from the graph, it's decreasing linearly as the threads are increased, therefore, the interpretation is true that threads will reduce the execution time of the program since it has divided the work and ran them concurrently.

Part 2:

Problem explanation:

Part 2 is the implementation of the producer – consumer algorithm. There are two threads: one producer and one consumer. The producer will read characters one by one from a string stored in a file name "message.txt", then writes sequentially these characters into a circular queue. Meanwhile, the consumer reads sequentially from the queue and prints them in the same order.

Choice of threading libraries:

Using the same thread as part 1.

Design of the experiment and develop programs for evaluation:

The program will use a buffer of size 5, which acts as a circular queue for producer thread to push the character in it, or enqueue(). The consumer thread will act as dequeue() function in this case. We'll create two variables of type sem_t from semaphore library to wait and signal the producer thread and consumer thread simultaneously. In this case, semVacant, which represents the number of available space in the buffer, will be set to 5. sem_wait(&semVacant) will be

implemented inside the producer function to notify that, if there is no available space, the producer thread stop reading the characters, otherwise, continue. In return, semFull, represents the already taken position inside the buffer, and sem_wait(&semFull) will be implemented in the consumer function. semFull will be initialized to 0, since at the beginning there are no characters inside the buffer, and the consumer thread will stop consuming if the buffer is empty.