San José State University
Engineering Extended Studies

CMPE 202
Software Systems Engineering

Section 47

Spring 2026
Instructor: Ron Mak

Assignment #2

Assigned: Wednesday, February 4
Due: Wednesday, February 11 at 5:30 pm
Team assignment, 200 points max

Multithreaded Programming

The purpose of this assignment is to practice using the POSIX multithreading library in C++. You will compare the timings of various versions of the Quicksort algorithm as it sorts one million random values:

- the classic recursive version as presented in class
- a non-recursive version that uses an explicit stack of subarrays to sort
- a non-recursive multithreaded version that uses separate threads to sort the subarrays, without yield calls
- a non-recursive multithreaded version that uses separate threads to sort the subarrays, with yield calls

Problem 1: Baseline timing of classic recursive Quicksort

Dynamically allocate a one-million element integer array and fill it with random values:

```
#include <time.h>
...
using namespace std;
using namespace std::chrono;

const int DATA_SIZE = 1000000;
int *data = new int[DATA_SIZE];
...
srand(time(0));
for (int i = 0; i < DATA_SIZE; i++) data[i] = rand();
```

If you don't have enough memory, you can lower the number of elements for this and the subsequent problems to 100,000.

As a baseline, first calculate and print the timing of the classic recursive Quicksort algorithm sorting the random values.

The following C++ code snippet shows one way to time the execution of some code in milliseconds:

```
#include <chrono>
...
using namespace std
using namespace std::chrono;
...
auto start_time = steady_clock::now();
//
// execute code to be timed here
//
auto end_time = steady_clock::now();

long ms = duration_cast<milliseconds>(end_time - start_time).count();
printf("%20ld ms\n", ms);
```

Problem 2: Baseline timing of non-recursive Quicksort

The recursive version of Quicksort from Problem 1 uses an underlying stack automatically managed by the C++ runtime system.

Thought question: If instead of using recursion, your version of Quicksort explicitly managed a runtime stack, how would the performance of your non-recursive stack-based version of Quicksort compare to the classic recursive version? Can you account for any significant differences in performance?

Write a non-recursive version of Quicksort by explicitly using a stack from the C++ Standard Template Library. Use the stack to keep track of the subarrays to sort. After each partitioning of a subarray, push the bounds of the resulting two smaller subarrays, each bounds as a pair of the smaller subarray's leftmost and rightmost index values. In other words, instead of passing the bounds of a subarray to a recursive sorting function, push the subarray bounds onto the stack.

An example C++ stack declaration:

```
#include <stack>

using namespace std;
...
stack<pair<int, int>> subarray_stack;
```

This **`stack`** object contains **`pair`** objects. Class **`pair`** is also from the C++ Standard Template Library. Use each **`pair`** object to hold the leftmost and rightmost index values of a subarray.

Initialize **`subarray_stack`** by pushing the bounds of the entire unsorted array onto the stack. Then, in a loop:

- Pop the bounds of a subarray off the top of the stack.
- Partition that subarray into two smaller subarrays.
- Push the bounds of the two smaller subarrays onto the stack.

Repeat this loop until the entire array is sorted. Get and print a baseline timing of how long this non-recursive stack-based version of Quicksort takes to sort one million random values. How does this timing compare with the timing from Problem 1?

Problem 3: Time a multithreaded Quicksort with no yields

Quicksort's subarrays are processed independently, since they don't overlap. Therefore, you can spawn a number of threads, where each thread separately performs the loop from Problem 2. Theoretically, Quicksort should go faster and faster as you increase the number of threads that are concurrently sorting different parts of the original array. Is that true in actual practice?

Write a multithreaded program that implements non-recursive stack-based version of Quicksort from Problem 2. Each thread executes the above loop to sort subarrays. Time how long it takes to sort one million random values using just one thread. Then time how long it takes to sort the same random values using two threads. Continue this with 3 through 24 threads, timing the entire sort for each number of threads. Make copies of the original array of random values so that each sort is with the same data.

Use the timing from **one spawned child thread** as a baseline, show how the timings with the additional spawned child threads are different from the one-thread timing.

For this problem, use mutexes and condition variables as necessary, but do not include calls to **`yield()`**.

In your program's comments, clearly explain

- What is/are the shared resource(s)?
- What is/are the critical region(s) of your code?
- How does your program protect its critical region(s)?
- Is it necessary to synchronize the threads' operations? If so, how does your program do the synchronization?

Print a table of the number of threads, sort time for each thread count, and the percentage changes in timings from the one-thread timing.

Run a system tool to monitor CPU use, such as Activity Monitor on MacOS or Task Manager on Windows, and watch how the amount of use changes as more and more threads are spawned.

Thought question: Can you explain the trend of the timings as you increase the number of threads?

Problem 4: Time a multithreaded Quicksort with yields

Repeat Problem 3, but this time include calls to `this_thread::yield()` that you strategically place in the code. Again, time the total sorts for 1 through 24 threads. Print a similar table.

Thought question: Can you explain the difference in the timing trends between having no calls to yield and having yields?

Bottom line: How does adding more threads affect Quicksort performance?

What to turn in

This is team assignment. Create four zip files, one containing the code files for each problem. Include a copy of your program's **runtime output text** in each zip file. Name the zip files after your team, for example, `SuperCoders1.zip`, `SuperCoders2.zip`, etc. Create a combined zip file of the problem zip files and name it after your team, e.g., `SuperCoders.zip`. Submit the combined zip file to Canvas.

Rubric

Your submissions will be graded according to these criteria:

Criteria	Max points
Problem 1	**10**
• Timing of recursive Quicksort	• 10
Problem 2	**50**
• Working non-recursive stack-based Quicksort	• 40
• Timing	• 10
Problem 3	**100**
• Working multithreaded program with no calls to `yield()`	• 40
• Shared resource(s) identified in comments	• 10
• Critical region(s) identified in comments	• 10
• How critical region(s) is/are protected, explained in comments	• 10
• How threads are synchronized, if necessary, explained in comments	• 10
• Timing table	• 20
Problem 4	**40**
• Include strategically placed calls to `yield()`	• 20
• Timing table	• 20