Programming Paradigms
Caucasus University – Fall 2022
Assignment 3 (10 points)

– Deadline: 23:59 on Friday 09 Dec 2022 –

Learning Goals

This assignment covers topics from the first few lectures and the first lab. You will be building your skills with:

- editing, compiling, testing, and debugging C programs under Unix
- writing code that manipulates bits and integers using the C bitwise and arithmetic operators
- C-strings (both raw manipulation and using string library functions)

Building

We use cmake to build our applications.

You can build debug version with thread sanitizers for testing your code:

```
> mkdir build-debug && cd build-debug
> cmake -DCMAKE_BUILD_TYPE=Debug..
> make
```

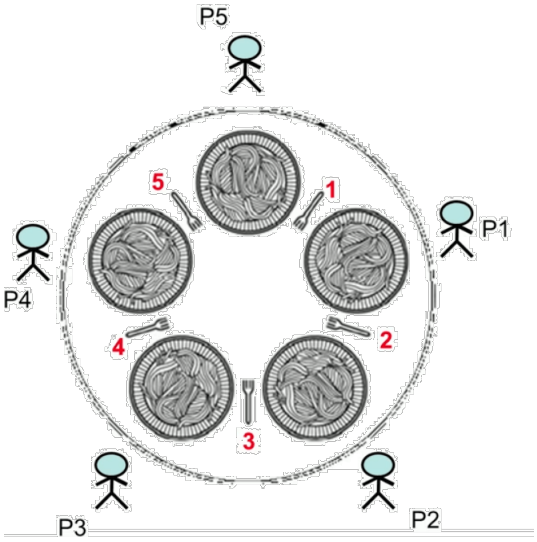
Build the release version for running benchmarks:

```
> mkdir build-release && cd build-release
> cmake -DCMAKE_BUILD_TYPE=Release ..
> make
```

Part I
Concurrency

1 Dining Philosophers (3 points)

Once upon a time, some philosophers were sitting around a table. On the table was a big bowl of spaghetti, and between each pair of adjacent philosophers was a fork.



The philosophers spent most of their time thinking. But every so often, one of them would get hungry. To eat some spaghetti, a philosopher needs to pick up both the fork to her left and the fork to her right. After eating, she puts both forks down again, so that they may be used by her neighbors.

You're provided with a program *dining-philosophers.cpp* that simulates N dining philosophers (where N is given as a command-line argument). Each philosopher behaves as follows:

1. think until the left fork is available, pick it up;
2. wait until the right fork is available, pick it up;
3. when both forks are held, eat for a fixed amount of time;
4. then, put the right fork down;
5. then, put the left fork down;
6. repeat from step 1.

Your tasks are the following:

(a) Run the program several times for different values of $N (2 \leq N \leq 10)$. What output do you observe? (You can use Ctrl+C to interrupt the program.) Explain (as far as you can) the observed output.

(b) Devise and implement a solution to the dining philosophers that does not suffer from the problem(s) you observed in part a.

For further information about this problem and some pointers, read the Wikipedia page on the dining philosophers: `http://en.wikipedia.org/wiki/Dining_philosophers_problem`. Hint: Use Thread Sanitizer `https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual` to better understand what is going on and what is the problem. (add a compiler flag -fsanitize=thread in cmakelists.txt).

Part II
Parallelism

2 Sieve of Eratosthenes. (3 points)

For this exercise, you will use Posix threads to implement a parallel version of the Sieve of Eratosthenes[1], an algorithm to find prime numbers. For some maximum positive integer Max, the Sieve of Eratosthenes works as follows:

1. Create a list of natural numbers: $1, 2, 3, ..., Max$.
2. Set k to 2, the first unmarked number in the list.
3. repeat:
   Mark all multiples of k between k^2 and Max.
   Find the smallest number greater than k that is still unmarked.
   Set k to this new value.
   until k^2 is greater than Max.
4. The unmarked numbers are all prime.

To parallelize this algorithm:

1. First sequentially compute primes up to $\sqrt{Max}$.
2. Given p cores, build p chunks of roughly equal length covering the range from $sqrtMax+1$ to Max, and allocate a thread for each chunk.
3. Each thread uses the sequentially computed "seeds" to mark the numbers in its chunk.
4. The master waits for all threads to finish and collects the unmarked numbers.

Besides your code, you need to provide a brief report that explains your solution (e.g., did you use synchronization between threads and why, how did you distribute the work to threads, how did you minimize communication, how did you achieve load balance, etc.) and reports the speedup curve you get as the number of cores is increased. You should also try to vary the size of Max to gauge the impact of program size on parallel performance. Make sure to pick large numbers for Max (e.g., in the millions to ensure there's enough work for threads to do).

[1] `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

Part III
Concurrent Data Structures

3 Sorted List (4 points)

Concurrent data structures require proper synchronization so that users of the data structure do not have to think about it themselves. The example given in this exercise is a sorted singly linked list data structure with the following interface:

- **insert** inserts a specified value into the linked list at the correct position.
- **remove** removes one copy of the specified value if it is in the list.
- **count** counts the number of elements with the specified value.

You can find a sequential implementation of this data structure in *sorted_list.hpp*. The driver program in *benchmark_example.cpp* prefills the sorted list with a number of elements and then runs three benchmarks:

- a read-only benchmark, using only counts.
- a write-only benchmark, using 50% inserts and 50% removes.
- a mixed benchmark, using 3.125% inserts, 3.125% removes, and 93.75% counts.

You can test the driver program as follows:

```
> make bench
> ./ bench 1
non-thread-safe read / threads: 1 - thousands of operations per second: 1592.289
non-thread-safe update / threads: 1 - thousands of operations per second: 76.683
non-thread-safe mixed / threads : 1 - thousands of operations per second: 182.39

> ./ bench 2
non-thread-safe read / threads: 2 - thousands of operations per second: 3125.396
^C
```

In the second run, the program stopped progressing and we had to abort it manually. As the program contains data races, it may also randomly crash, delete random files, or spawn demons from your nose.[2] Fortunately, all it does in this case is to hang. Your task is to implement (in C++) and compare thread-safe versions of the sorted list, using the following synchronization mechanisms:

1. coarse-grained locking (for example, in C++, using the **std::mutex** class from the standard library; see `http://en.cppreference.com/w/cpp/thread/mutex`);
2. fine-grained locking (again, using the **std::mutex** class from the C++ standard library);

Using the standard library's mutex locks should be straightforward. Evaluate, plot, and explain the results/behaviour of the five versions from the point of view of performance and scalability with respect to:

[2] `http://www.catb.org/jargon/html/N/nasal-demons.html`

- the number of threads;
- the performed operations.

If possible, run your experiments on a machine that allows to run at least 16 concurrent threads (e.g., one that has at least 8 physical cores, each with hyperthreading), and run your experiments using a suitable subset of worker threads in that range, making sure you include all powers of two (1, 2, ..., 2^n) in the range of numbers that your machine allows as concurrent threads. Discuss the advantages and disadvantages of each form of synchronization in terms of: ease of implementation, lock contention, performance, and scalability.

Identify situations when one version is more suitable than another, e.g. % of read vs. % of update operations. Dedicate a section of your report to explain the reasoning, challenges and solutions in implementing both versions of the data structure. Describe both the general solution and technical details. If you ended up getting 'inspiration' from code for some of these locks that you found on the internet, make sure you include a pointer to it in your source and your report.

The points of this exercise are split equally into 2 points for implementation and 2 for the report.

Submitting

Don't forget to write a report with explenations and performance graphs. Once you're finished working, zip all source code, CMakelists.txt files and pdf report together and upload them on teams.

Grading

Correctness: 80% Your code should compile without any errors or warnings. We will test your programms on different inputs and edge cases. For a full score your code should pass all tests (some exercises might already have predefined tests).

Code Quality: 20% We expect your code to be clean and readable. We will look for descriptive names, defined constants (not magic numbers!), and consistent layout. Be sure to use the most clear and direct C syntax and constructs available to you.