CS69201: Computing Lab-1 **Systems Assignment 1: Basic and Non-Blocking Execution Environment for C++ Threads**

Deadline: October 14, 2024, 11:59 PM

Important Instructions

1. Using Linux

1. This assignment is based on OS programming and will be beneficial for the students to do this assignment in a Linux OS.
2. If you don't have Linux, use Virtualbox + Linux image, both free. We suggest Ubuntu but feel free to use any Linux version. If you decide to use Cygwin or WSL, it's up to you; just remember that most of your batchmates (and the instructors) will use Linux (and Ubuntu), so, naturally, we might not be able to provide support for other systems.
3. In case of any difficulties in understanding any functionalities, you may use the same functionality in a Linux Terminal to get a better understanding of how it works.

2. Programming Language

This assignment will use the C++ programming language.

3. Error Handling & Input-Output

1. Proper error handling (e.g., when the syscall or the library calls fail) is expected in this assignment.
2. The inputs should be taken from the Terminal and results displayed in the Terminal, if explicitly not mentioned.

4. Deliverables

Submit a zip file named `<your_roll_no_Assignment9.zip>` which should include:

- All .cpp files [3 files for the 3 parts] `<your_roll_no_A9_part{i}.cpp>` where $i \in \{1, 2, 3\}$
- Reports for all the 3 parts `<your_roll_no_part{i}_report.txt>` where $i \in \{1, 2, 3\}$
- Separate makefile: Resource for each part
- An additional readme file (call it `README`) containing details about how to compile the files and any additional information required for the instructor during checking that you feel is necessary.

Objective

You already know about threads in operating systems and possibly have implemented basic threads in your undergraduate curriculum. However, the basic implementation might not always suffice in terms of throughput.

For example, if a thread has a dedicated handler function to perform (which is the case in the basic version of pthreads), then you cannot use that thread for any other functions and have to create a thread for each handler function—this is expensive.

So, you might want to create a thread pool which is always waiting for tasks and can perform functions (called tasks) as and when they come, complete it and wait for the next task. Even this might be a problem due to the blocking nature of your approach. If some of these tasks have waits (e.g., to do I/O or for sleeping), then a thread does not make progress until the system call is done / sleep is over. This is called blocking behaviour.

So, for high-performance versions of threads for the real world, we need to implement non-blocking variants. In this assignment, you will implement and compare three threading approaches in C++: basic Pthreads, blocking std::threads, and a non-blocking solution using C++20 coroutines and lock-free programming.

To facilitate this comparison and provide a foundation for your implementation, you might want to check the asyncio library available at `https://github.com/netcan/asyncio`. This library offers a framework for building efficient asynchronous I/O operations in C++, which is similar for the non-blocking portion of your assignment.

Tasks

Task 1: Basic Thread Creation with Pthreads library [25 marks]

- **Mathematical Operations: [2 marks]**
  - Take five basic mathematical operations: addition, subtraction, multiplication, division, and exponentiation (power) using one function for each.

```cpp
int add(int x, int y) { return x + y; }
int subtract(int x, int y) { return x - y; }
int multiply(int x, int y) { return x * y; }
double divide(double x, double y) { return x / y; }
int power(int x, int y) { return std::pow(x, y); }
```

  - Each function will take input values, perform the operation, and return the result.
- **Wrapper Function for Thread Execution: [5 marks]**
  - Create a struct to hold the input parameters and results for each mathematical operation.
  - Implement wrapper functions that take a pointer to the struct, perform the operation, and store the result in the appropriate field of the struct.
  - These wrapper functions will serve as the entry point/handler functions for each thread.
- **Thread Creation with pthread: [10 marks]**
  - Use the pthread library to create a separate thread for each mathematical operation. Pass the appropriate struct (with input values) to each thread.
  - Ensure that the main thread waits for each thread to complete by using pthread_join().
- **Display Results: [3 marks]**

  - After joining the threads, print the input values and results for each operation to verify correctness.
  - Ensure that the results from the threaded operations are accurate and displayed in an organized manner.

- **Test function and report: [5 marks]**
  - Add a random selection of 500 mathematical operations (possibly machine-generated) to demonstrate the functioning of your approach.
  - Write a report in a text file (`<your_roll_no>_A9_part1_report.txt`) stating:
    - What is your structure?
    - What is your input format?
    - What is your output format?
    - What is the time taken for 500 operations? Please take at least 5 random samples and report average, standard deviation, median, 95th percentile, and max time.

Task 2: Blocking Threads with std::thread [35 marks]

C++ provides an abstraction over pthreads called std::thread. This part of the assignment is mastering using std::thread to do:

- **Simulate Time-Consuming Operations: [10 marks]**
  - Implement functions that perform basic tasks such as addition, subtraction, and multiplication. Introduce artificial delays in each operation to simulate long-running computations (e.g., sleeping the thread for a few seconds).
  - Introduce sleep to ensure that each operation takes a comparative large amount of time to complete – e.g., 3–7 seconds.
- **Sequential Thread Creation: [10 marks]**
  - Create threads using the C++ standard library (std::thread) for each time-consuming operation.
  - Launch each thread with the necessary arguments for the respective tasks (see section 1 above for the structure description).
  - Ensure that after launching the threads, you use appropriate mechanisms to wait for their completion, such as joining each thread before moving on to the next one.
- **Demonstrate Blocking Behavior: [5 marks]**
  - Start a timer before creating the threads to measure how long it takes for all operations to complete.
  - After launching the threads, use blocking techniques (like join()) to ensure that the main thread waits for each task to finish sequentially.
  - Once all threads have been completed, record and display the total time taken, demonstrating that tasks are executed in a blocking, sequential manner.
- **Exception Handling: [5 marks]**
  - Introduce exception handling to account for any potential issues that may arise during thread execution. Use try-catch blocks to ensure that exceptions are properly handled.

  - Ensure that even in the case of an exception, all threads are properly joined, and no resources are left unmanaged.

- **Test function and report: [5 marks]**
  - Add a random selection of 500 mathematical operations (possibly machine-generated) to demonstrate the functioning of your approach. Make sure around 10% of them will give rise to exceptions (e.g., divide by 0, taking roots of negative numbers, etc.).
  - Write a report in a text file (`<your_roll_no_A9>_part2_report.txt`) stating:
    * What is your structure?
    * What is your input format?
    * What is your output format?
    * What is the time taken for 500 operations? Please take at least 5 random samples and report average, standard deviation, median, 95th percentile, and max time.
    * How does the time taken by std::thread compare with pthreads? Why?

Task 3: Non-Blocking Threads with Thread Pools [40 marks]

Now finally, we want to explore non-blocking, asynchronous programming using C++20 coroutines and lock-free data structures. You will design and implement a system that efficiently handles concurrent tasks without relying on traditional blocking mechanisms, such as locks.

- **Asynchronous Mathematical Functions: [10 marks]**
  - Implement basic mathematical operations (e.g., addition, subtraction, multiplication, division, exponentiation) in an asynchronous, non-blocking fashion using C++20 coroutines.
  - Each function should return a result asynchronously, allowing for multiple operations to be performed concurrently.
  - You are free to choose how you structure and manage the coroutines and task system (i.e., the structure, input, and output format).

- **Thread Pool with Non-Blocking Job Management: [10 marks]**
  - Create a thread pool creation process (*allfather*).
  - Design a thread pool of 50 threads that efficiently distributes tasks across a fixed number of worker threads (how to schedule? Read on next parts). Ensure that tasks are managed and processed concurrently without traditional locking mechanisms.
  - Consider using lock-free techniques, such as atomic operations, for task management to minimize contention and avoid bottlenecks.
  - You have flexibility in choosing how the thread pool and task scheduling are structured. (Make sure to write your design choices in the report.)

- **Lock-Free Job Queue: [5 marks]**
  - Implement a non-blocking job queue that can hold tasks (above mathematical functions) and distribute them across multiple threads.
  - The queue should make use of atomic operations to handle enqueueing and dequeueing without locks. (Ensure the queue is thread-safe without using mutexes or locks.)

- **Coroutine Task System: [5 marks]**
  - Create a process (*task_system*) to manage and schedule coroutines, allowing for the execution of multiple asynchronous tasks.
  - Design a mechanism in the process for managing coroutine states, such as checking when a task is ready and retrieving its result. (Consider using system calls like poll and select.)
- **Concurrency Management: [5 marks]**
  - Implement a process (*assigner*) that efficiently assigns jobs to the thread pool and manages their execution concurrently.
  - Ensure that job submission and result processing are non-blocking and utilize atomic operations or lock-free techniques.
- **Test function and report with performance Comparison: [5 marks]**
  - Demonstrate your system by submitting a set of concurrent jobs (e.g., 100 or more) using the asynchronous mathematical functions.
  - A test script to run the performance comparison.
  - Write a report in a text file (`<your_roll_no_A9>_part3_report.txt`) stating:
    - Compare the performance (in terms of latency) of your non-blocking implementation with a traditional blocking version that executes tasks sequentially (modify the std::thread implementation for the baseline).
    - How did you structure and manage the coroutines and task system (i.e., the structure, input and output format)?
    - What lock-free techniques did you use?
    - How the thread pool and task scheduling are structured (data structure, design, etc.)?
    - What is the design of your non-blocking job queue?
    - Any challenges faced and how did you overcome them?
