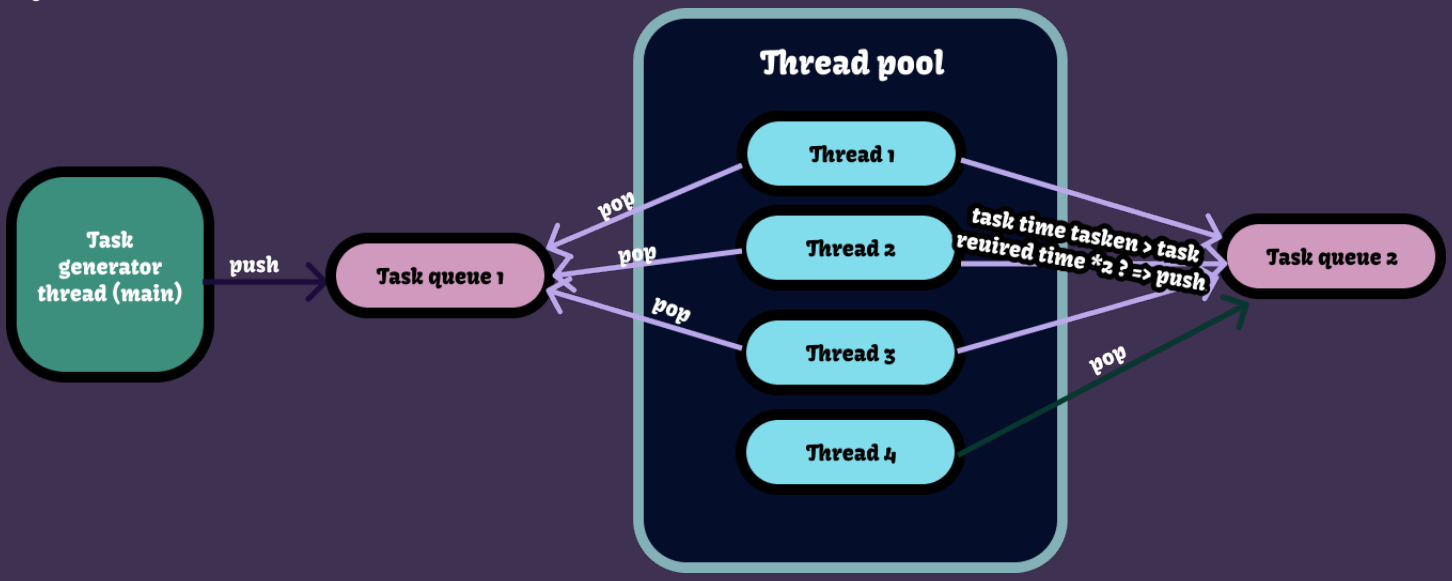
Thread Pool Implementation

Student Name: Filipovych Marharyta
Group: ПКСП-1
Chosen Variant: 10
Github Link: *https://github.com/VY-Assignments/parallel-assignment-2-MarharytaFilipovych*

Task

Implement your own thread pool. It should contain two queues, the first is served by 3 threads, and the second is served by 1. Tasks are added in the first order of execution through one interface (the user does not have explicit access to the execution queues) in the order of "difficulty" (shorter tasks have a higher priority). If the task from the first queue is not completed within twice the time required for its completion, it is transferred to the second queue. The task is taken up from both queues immediately if there is a corresponding free workflow. The task takes a random time of 5 to 10 seconds. Additionally, this thread pool should be able to complete its work correctly (immediately, with the abandonment of all active tasks, and with the completion of active tasks), have the possibility of temporarily stopping its work, and work with the use of conditional variables. The operations of initializing and destroying the pool and adding tasks from the queue must be thread safe. Check and prove the program's correct operation using the console input/output system. Perform time-limited testing and calculate the number of threads created and the average time a thread is in the waiting state. Determine the average length of each queue and the average task execution time.

System Model



Solution Description

Provide a description of the implemented software solution, explaining how the system was designed and developed:
I designed and developed a thread pool system to manage task execution in a multithreaded environment.

To simulate and manage tasks, I created a Task struct that includes key details such as the task's required time, the actual time taken, and the time it entered the queue. This time is assigned within the addTask function after the task is added to the first queue. I also implemented an overloaded operator() to invoke task objects as if they were functions.

The system handles tasks using two priority queues (q1 and q2). Initially, tasks are added to q1, with their priority determined by the required execution time (if the required time is shorter, it is executed first), which is randomly assigned between 5 and 10 seconds. If a task's execution time exceeds twice its required time, it is moved to the second queue q2.

The thread pool consists of four worker threads: three process tasks from q1, and the fourth handles tasks from q2. To synchronize access to these queues, I used two separate mutexes and condition variables (mutex1, mutex2, cv1, and cv2). This prevents race conditions and deadlocks by ensuring each queue has its own locking mechanism.

The two main functions responsible for task execution are executeTaskQ1(int id) and executeTaskQ2(). The thread enters a while (true) loop in each of these functions. First, the thread waits until the system is unpaused or stopped, using the relevant condition variable and pauseMutex. If it needs to terminate, it exits the loop immediately. If the system is running, it waits for a task to be available in the corresponding queue. Once a task is available, the thread pops it from the queue and processes it.

In executeTaskQ1, after calculating the task execution time, if the task's time taken exceeds twice the required time, it is moved to q2. To move the task to the q2, I lock mutex2. Otherwise, the task is executed directly. executeTaskQ2 simply executes the task without further checks.

The system also supports pausing and resuming task execution. During a pause, tasks in progress continue, but new tasks wait until the system is restarted. The pause/resume functionality is controlled using a separate mutex (pauseMutex) and a boolean flag paused (initially set to false). When the system is paused, the flag is set to true, and both condition variables (cv1 and cv2) notify all threads.

When the thread pool goes out of scope, its destructor is invoked. In the destructor, I set the stop flag to true, which is done without the need for mutexes since this flag is not toggled (I checked its work, and it works). Once stop is set to true, the cv1 and cv2 condition variables are notified, signaling the threads to stop. Afterward, the threads are joined, and a message indicating the termination of the pool is printed. Finally, the statistics are displayed.

I implemented a terminate() function to terminate the pool manually. The TERMINATE flag is set to true, and the stop flag is also set to true. The threads are then notified, and the queues are cleared. Initially, I cleared the queues using a well-known loop that pops elements until the queue is empty. However, this approach was too slow, so I substituted it for a faster method: I reassign the queues to new empty ones.

During the first waiting stage in the while loops of executeTaskQ1 and executeTaskQ2, the TERMINATE flag is checked alongside the paused flag. If TERMINATE is true, the function returns immediately.

To calculate all necessary statistics, I introduced a struct called ForStatistics, which contains fields such as timeTaken, numberOfTasks, and waitTime. These fields are used to track data for each thread. I created an unordered_map where the keys are thread IDs, and the values are instances of ForStatistics. This map is updated whenever necessary, using the statsMutex to ensure thread-safety. Additionally, I introduced three atomic int variables: q1Size, q2Size, and finishedTasks to track the max size of the queues and the number of finished tasks.

To output information in the console in a thread-safe manner, I added an outputMutex. This mutex ensures that only one thread outputs data at a time, avoiding conflicts. To maintain a positive mood, I made the output colorful.

To handle mutexes, I used unique_lock where condition variables (cv1 and cv2) were needed. For scenarios where mutexes were only needed for short, simple locking (without waiting on condition variables), I utilized lock_guard.
For time measurement, I used high_resolution_clock from the chrono library. However, during the while loop, I switched to system_clock. I encountered issues with using high_resolution_clock in this loop (the loop in main was infinite for some reasons), so I switched tosystem_clock.

Testing

- **Describe how the testing was conducted:** I conducted thorough testing using console output. I simulated various scenarios in the main function to verify the following features:
    - **Task Addition:** I verified that tasks were successfully added by observing the console output, which displayed the following information about each task upon its completion: the ID of the thread responsible for executing the task, the required time for the task's execution, and the actual time it took to complete the task.
    - **Pause and Resume:** To test the pause and resume functionalities, I added a specific number of tasks (e.g., 40) in the main program. Afterward, I put the main thread to sleep for some time (e.g., 10 seconds) before invoking pool.pause(). Following this, I called pool.resume(20), where the argument in the resume function specifies the duration (in seconds) for which the thread pool remains paused. The console output successfully and efficiently demonstrates the correctness of those operations. **P.S.** If the user initiates a pause while a task is already assigned for execution, the assigned task will complete its work. However, all other tasks will remain on hold until the thread pool is resumed.
    - **Termination:** To test the termination functionality, I added several tasks (e.g., 100) and put the main thread to sleep for some time (e.g., 15 seconds). Afterwards, I invoked pool.terminate(). Upon termination, the console output displays "**TERMINATED!**" to indicate that the thread pool has stopped.
- **Performance Evaluation:** The number of threads is determined within the task variant: 3 threads for the first queue and 1 thread for the second. The size of the queues is unlimited. Here are screenshots with all necessary statistics computed during the program execution within time-limited testing. The tasks were added in the main function to the first queue in the while loop. The loop was excited when a certain earlier-determined amount of time finished: 1-5-10-15-30 minutes.
    - **1 minute:**

```
Executed 46 tasks in 1 minutes.
Number of rejected tasks: 85844853.
Thread #1 executed 12 tasks with a total time of 78 seconds. Hence, average time execution of one task is 6.5 seconds. Total wait time: 0 seconds. Average w
ait time: 0 seconds.
Thread #3 executed 12 tasks with a total time of 73 seconds. Hence, average time execution of one task is 6.08333 seconds. Total wait time: 0 seconds. Avera
ge wait time: 0 seconds.
Thread #2 executed 12 tasks with a total time of 73 seconds. Hence, average time execution of one task is 6.08333 seconds. Total wait time: 0 seconds. Avera
ge wait time: 0 seconds.
Thread #4 executed 10 tasks with a total time of 332 seconds. Hence, average time execution of one task is 33.2 seconds. Total wait time: 10 seconds. Averag
e wait time: 1 seconds.
Max size of the first queue was 85844899.
Max size of the second queue was 2188.
```

- **5 minutes:**

```
Executed 194 tasks in 5 minutes.
Number of rejected tasks: 255707790.
Thread #2 executed 47 tasks with a total time of 268 seconds. Hence, average time execution of one task is 5.70213 seconds. Total wait time: 50 seconds. Average wait time: 1.06383 seconds.
Thread #1 executed 47 tasks with a total time of 260 seconds. Hence, average time execution of one task is 5.53191 seconds. Total wait time: 50 seconds. Average wait time: 1.06383 seconds.
Thread #3 executed 47 tasks with a total time of 275 seconds. Hence, average time execution of one task is 5.85106 seconds. Total wait time: 50 seconds. Average wait time: 1.06383 seconds.
Thread #4 executed 53 tasks with a total time of 5590 seconds. Hence, average time execution of one task is 105.472 seconds. Total wait time: 10 seconds. Average wait time: 0.188679 seconds.
Max size of the first queue was 255707984.
Max size of the second queue was 23297.
```

- **10 minutes:**

```
Executed 397 tasks in 10 minutes.
Number of rejected tasks: 427421426.
Thread #1 executed 96 tasks with a total time of 561 seconds. Hence, average time execution of one task is 5.84375 seconds. Total wait time: 94 seconds. Average wait time: 0.979167 seconds.
Thread #3 executed 96 tasks with a total time of 564 seconds. Hence, average time execution of one task is 5.875 seconds. Total wait time: 94 seconds. Average wait time: 0.979167 seconds.
Thread #2 executed 96 tasks with a total time of 542 seconds. Hence, average time execution of one task is 5.64583 seconds. Total wait time: 94 seconds. Average wait time: 0.979167 seconds.
Thread #4 executed 109 tasks with a total time of 15864 seconds. Hence, average time execution of one task is 145.541 seconds. Total wait time: 10 seconds. Average wait time: 0.0917431 seconds.
Max size of the first queue was 427421823.
Max size of the second queue was 50783.
```

- **15 minutes:**

```
Executed 605 tasks in 15 minutes.
Number of rejected tasks: 454505575.
Thread #1 executed 147 tasks with a total time of 865 seconds. Hence, average time execution of one task is 5.88435 seconds. Total wait time: 128 seconds. Average wait time: 0.870748 seconds.
Thread #3 executed 147 tasks with a total time of 851 seconds. Hence, average time execution of one task is 5.78912 seconds. Total wait time: 128 seconds. Average wait time: 0.870748 seconds.
Thread #2 executed 147 tasks with a total time of 809 seconds. Hence, average time execution of one task is 5.5034 seconds. Total wait time: 128 seconds. Average wait time: 0.870748 seconds.
Thread #4 executed 164 tasks with a total time of 34013 seconds. Hence, average time execution of one task is 207.396 seconds. Total wait time: 10 seconds. Average wait time: 0.0609756 seconds.
Max size of the first queue was 454506180.
Max size of the second queue was 68201.
```

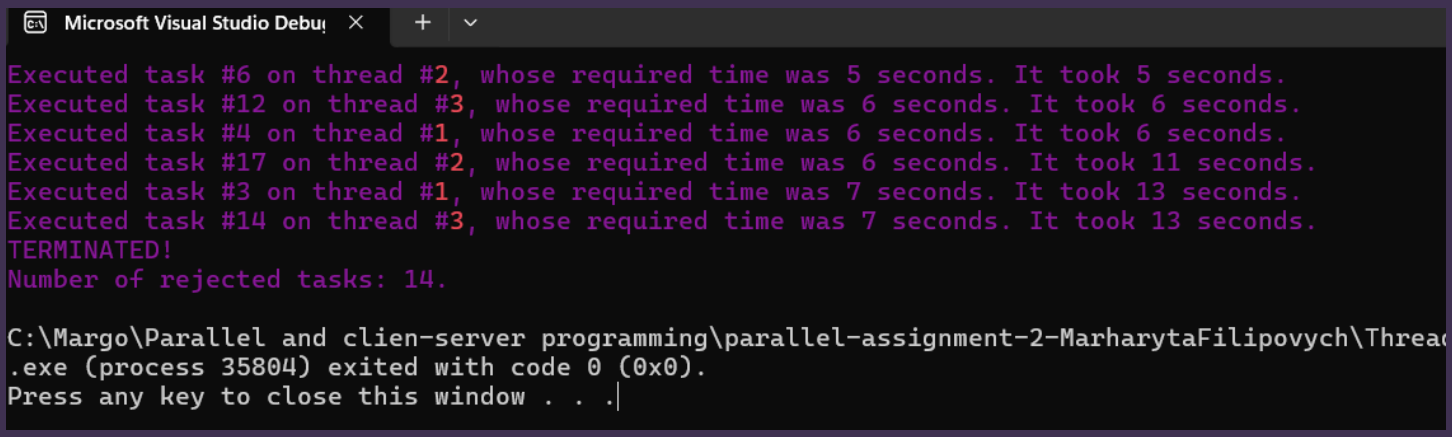
- **30 minutes:**

```
Executed 846 tasks in 30 minutes.
Number of rejected tasks: 931878699.
Thread #3 executed 198 tasks with a total time of 1116 seconds. Hence, average time execution of one task is 5.63636 seconds. Total wait time: 415 seconds. Average wait time: 2.09596 seconds.
Thread #2 executed 198 tasks with a total time of 1102 seconds. Hence, average time execution of one task is 5.56566 seconds. Total wait time: 415 seconds. Average wait time: 2.09596 seconds.
Thread #1 executed 198 tasks with a total time of 1150 seconds. Hence, average time execution of one task is 5.80808 seconds. Total wait time: 415 seconds. Average wait time: 2.09596 seconds.
Thread #4 executed 252 tasks with a total time of 85232 seconds. Hence, average time execution of one task is 338.222 seconds. Total wait time: 10 seconds. Average wait time: 0.0396825 seconds.
Max size of the first queue was 931879545.
Max size of the second queue was 108008.
```

- **Insert screenshots of the program during testing:**
  - *Time-limited testing main function:*

```
int main() {
    srand(time(0));
    ThreadPool pool;
    for (int i = 0; i < 10; i++)
    {
        pool.addTask(Task(i));
    }
    this_thread::sleep_for(seconds(5));
    pool.pause();
    pool.resume(30);
    return 0;
}
```

- *Pause and resume testing:*




```cpp
int main() {
    srand(time(0));
    ThreadPool pool;
    for (int i = 0; i < 20; i++)
    {
        pool.addTask(Task(i));
    }
    this_thread::sleep_for(seconds(10));
    pool.terminate();
    return 0;
}
```

- *Termination testing:*

```
Executed task #8 on thread #2, whose required time was 5 seconds. It took 5 seconds.
Paused
Executed task #1 on thread #3, whose required time was 6 seconds. It took 6 seconds.
Executed task #4 on thread #1, whose required time was 6 seconds. It took 6 seconds.
Resumed after pause for 30 seconds.
Executed task #5 on thread #4, whose required time was 9 seconds. It took 44 seconds.
Executed task #3 on thread #4, whose required time was 9 seconds. It took 53 seconds.
Executed task #6 on thread #4, whose required time was 9 seconds. It took 62 seconds.
Executed task #9 on thread #4, whose required time was 9 seconds. It took 71 seconds.
Executed task #7 on thread #4, whose required time was 10 seconds. It took 81 seconds.
Executed task #0 on thread #4, whose required time was 10 seconds. It took 91 seconds.
Executed task #2 on thread #4, whose required time was 10 seconds. It took 101 seconds.
Finished.
```

```cpp
int main() {
    srand(time(0));
    ThreadPool pool;
    int i = 0;
    auto finish = system_clock::now() + minutes(TIME);
    while (system_clock::now() < finish) {
        pool.addTask(Task(i));
        i++;
    }
    pool.terminate();
    return 0;
}
```

Conclusions

- **What was implemented:** I designed the best thread pool according to the assigned variant with the following functionalities:

    - supports task addition and prioritization using two priority queues.
    - pause, resume, and termination functionalities
    - **graceful shutdown**: automatically stops when all tasks are completed and the pool goes out of scope.

  To handle synchronization challenges, I employed **mutexes** and **condition variables**. Additionally, I utilized **atomic variables (since I decided it would make my life easier)** to track the number of finished tasks and the maximum sizes of both priority queues, simplifying statistical tracking.

- **Additional Notes:** In my humble opinion, this assignment's most challenging part was implementing pause and termination mechanisms. I spent lots of time figuring out whether I needed additional mutexes for that or not and adjusting the conditions. I didn't like statistics measurement and writing this document.

Appendices

- **https://www.geeksforgeeks.org/thread-pool-in-cpp/**
- **gpt**
- https://www.geeksforgeeks.org/priority-queue-in-cpp-stl/