# Practical Assignment 2. Synchronization: thread pool implementation

AUTHOR
Vadym Katsel

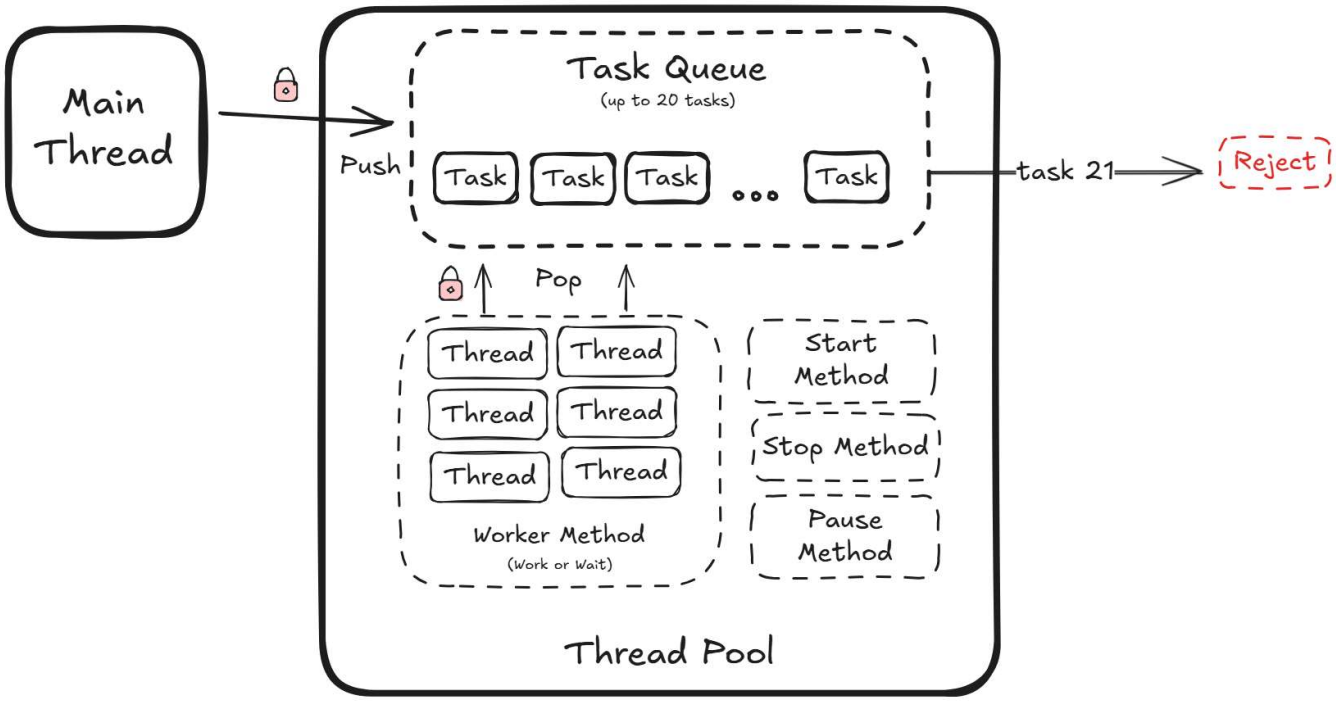
PUBLISHED
2026-02-01

| | |
|---|---|
| **Group** | Group 1 |
| **Variant** | 2 |
| **Repository** | GitHub Link |

## Task

For this practical assignment I had to build a thread pool that manages 6 workers and has one execution queue, which is limited to size 20. If the there's a task to add to the full queue - it's rejected. The implemented thread pool also supports such operations as `Start()` `Stop()` `Pause()` and `Resume()`.

## System Model




The Diagram view of an implemented Thread Pool

## Solution Description

For the purpose of achieving the lab's goal I implemented a class `ThreadPool.cs` using C# programming language.

To ensure the synchronization I used a `private readonly object _lock`. You can see full list of logic implemented in the class below:

- **Adding Tasks:**
  - `AddTask` method locks the queue, and if queue size is less than max size, it adds the task and signals a worker thread, using `Monitor.Pulse(_lock)`. If the queue is full, it rejects the task.
- **Worker Threads:**
  - Each worker thread runs `WorkerMethod` with an infinite loop inside it. It checks different conditions, in order to manage pause, resume, stop and waiting scenarios.
- **Stop, Pause and Resume:**
  - When `Pause` method is called, it sets `_isPaused` flag to true. Respectively, workers check this flag and operate accordingly.
  - When `Resume` is called, it does the opposite thing and calls `Monitor.PulseAll(_lock)` to wake up all paused workers.
  - `Stop` in its turn, sets another `_isWorking` flag to false and pulses all the workers in order to finish added tasks and stop working.

# Testing

## Testing process

In order to ensure the correctness of the execution, I conduct 5 testing scenarios, all located in `Program.cs`. Below, you can see the whole flow and consoles output

- **First Step**: The initialization of the `ThreadPool.cs` with 6 threads starts the whole process
- **Overloading Test**: In order to check the correcteness of refusing and accepting tasks, main thread tries to add 30 tasks to the queue. As expected, 4 of them are not added. In math terms, we have 20 free spots in queue and 6 workers, which take 6 tasks instantly. This leaves the last 4 tasks refused
- **Pause Test**: Then, to check the `Pause` method, we are firstly trying to pause the pool. The threads are expected to complete their current jobs but not to take new ones from queue. Also, another special task is added, to check the possibility of adding new tasks to queue during the pause.
- **Resume Test**: After that, the thread pool was resumed, and, as expected, the pending tasks were back in work, along with the task added during the pause.
- **Shutdown**: 3 seconds after the resume, `Stop` method was applied. It let check that all the task in the queue are completed before the stop, and so called soft-stop is performed.

## Performance evaluation

All the metrcis were recorded during the testing scenario presented above.

- **Number of Threads**: 6
- **Queue Fill Time**: As the queue is limited to size 20 and tasks are added immediatly, the filling time is almost 0ms. There was no direct time measurement due to the question of preciseness of such results
- **Rejected Tasks**: 4 (30 - 6 - 20 = 4)
- **Average Wait Time**: 4404,08 ms

## Conclusions

In this assignment I implemented a fully functional Thread Pool using C#. This thread pool supports different kind of operations and use cases, as well as successfully handles synchronization and parallelism, by using `Monitor` set of tools to keep the Queue safe.