Computação em Larga Escala

Short Intro. to Thread Pools

Eurico Pedrosa

Universidade de Aveiro - DETI

2025-03-09

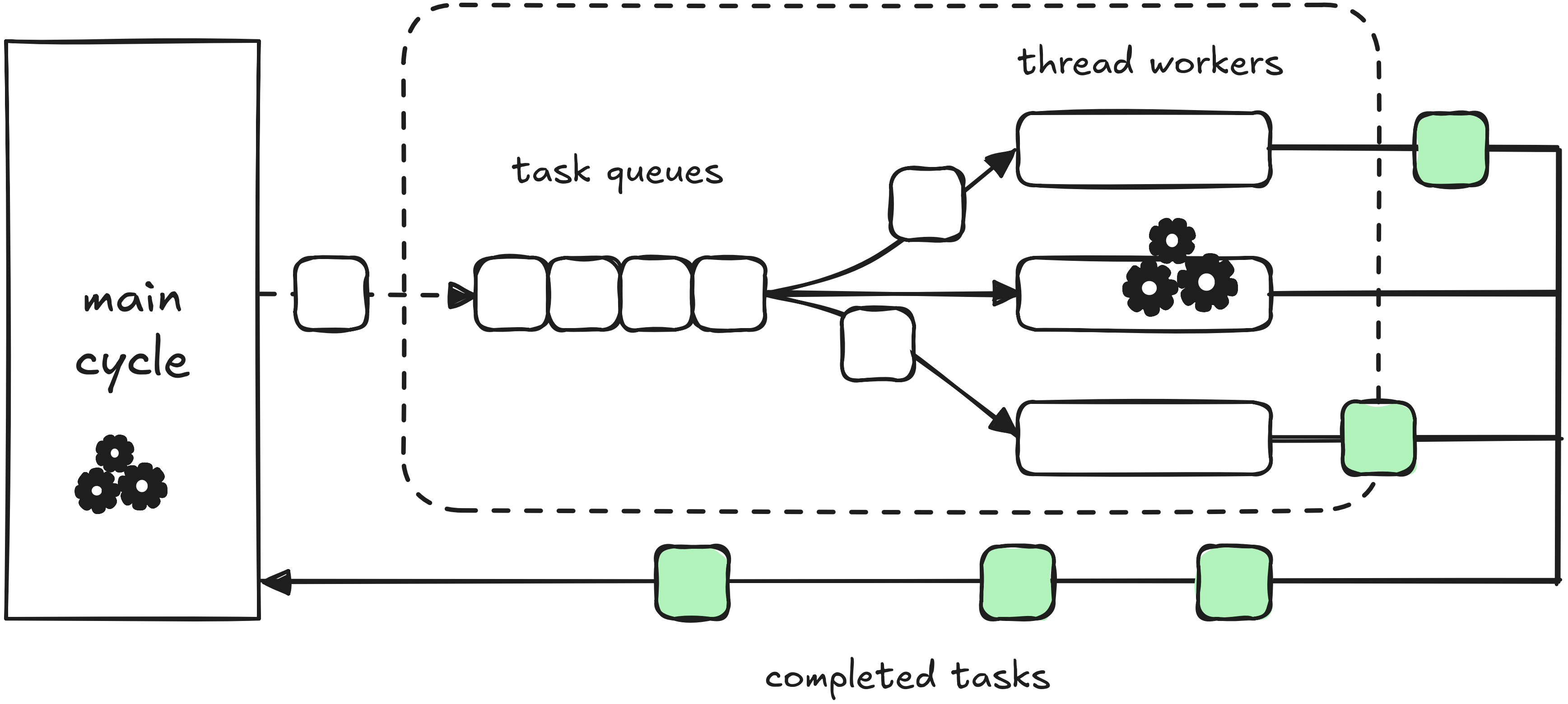
Intro. to Thread Pools

What is a Thread Poll?

- A **thread pool** is a collection of worker threads that efficiently manage task execution.
- Instead of creating and destroying threads frequently, a fixed set of threads is reused.

Why Use a Thread Pool?

- Reduces overhead of thread creation and destruction.
- Efficient task scheduling and execution.
- Prevents system overloading by limiting the number of active threads.
- Useful in high-performance applications like web servers and parallel computing.


thread workers
task queues
main
cycle
completed tasks

How Thread Pools Work

Key Components:

1. **Worker Threads**: Persistent threads waiting for tasks.
2. **Task Queue**: A queue where tasks are stored before execution.
3. **Thread Manager**: A mechanism to assign tasks to threads.
4. **Synchronization Primitives**: Mutexes and condition variables to manage concurrency.

Workflow:

- A task is submitted to the queue.
- An idle thread picks up the task and executes it.
- Once completed, the thread returns to the pool, waiting for the next task.

Implementing and Using a Thread Pool in C++

C++ Standard Libraries Used

- `<thread>`: To manage worker threads.
- `<queue>`: To store pending tasks.
- `<functional>`: To store callable tasks.
- `<atomic>`: For atomic increments.
- `<condition_variable>` & `<mutex>`: For thread synchronization.

Implementing and Using a Thread Pool in C++

```cpp
struct ThreadPool {

    std::vector<std::thread> workers;
    std::queue<std::fuction<void>> queue;

    std::atomic<bool> stop {false};
    std::atomic<int> tasks_to_complete{0};

    std::mutex              queue_mutex;
    std::condition_variable queue_condition;

    std::mutex              wait_mutex;
    std::condition_variable wait_condition;

    ThreadPool() = default;
    virtual ~ThreadPool();

    void init(size_t size = 0);

    void enqueue(std::function<void()>&& function);
    bool dequeue_task();

    void wait();
};
```

Implementing and Using a Thread Pool in C++

```cpp
#include <iostream>
#include <chrono>
#include "thread_pool.h"

using namespace std::chrono_literals;

int main(int argc, char* argv[]){
    ThreadPool thread_pool;
    thread_pool.init(2); // initialize thread pool with 2 workers

    for (int i = 0; i < 10; ++i){
        // enqueue a lambda function
        thread_pool.enqueue([ i ]{
            std::this_thread::sleep_for(1000ms);
            std::cout << "End of thread " << i << std::endl;
        });

    }// end for

    thread_pool.wait()
    return 0;
}
```

Summary and Benefits

Key Takeaways

- Thread pools optimize thread management, reducing overhead.
- Efficient synchronization using mutexes and condition variables.
- Reusable worker threads avoid unnecessary creation and destruction.
- Flexible task scheduling allows efficient execution of concurrent tasks.

Where to Use Thread Pools?

- Web servers for handling multiple client requests.
- Parallel computations in scientific computing.
- Big Data Processing (MapReduce).
- Game engines for managing background tasks.