Thread Pool Implementation

# Contents:

## Project Overview:………………………………………………………………………………………

## Scope:……………………………………………………………………………………………………….

## Functional Requirements :…………………………………………………………………………

## Non-functional Requirements :…………………………………………………………………

## System Design:………………………………………………………………………………………….

## Input/Output Specifications:…………………………………………………………………….

## Challenges:……………………………………………………………………………………………….

## Flowchart:………………………………………………………………………………………………..

## Skeleton Code Structure:………………………………………………………………………….

## Testing and Validation:……………………………………………………………………………

## Bug Tracker:……………………………………………………………………………………………

1. Milestone:………………………………………………………………………………………………………..

## References:……………………………………………………………………………………………..

# 1. Project Overview

Project Title: Thread Pool Implementation

Description: The project involves the design and implementation of a thread pool in C++. A thread pool is a collection of pre-initialized threads that can be reused to perform multiple tasks concurrently. The key purpose of this project is to enhance performance and resource utilization by minimizing the overhead associated with the creation and destruction of threads. This is achieved by maintaining a pool of worker threads that are always available to handle tasks, thus reducing the latency and system load associated with thread management.

Objective:

* **Task Management Efficiency**: Implement a thread pool that efficiently manages multiple tasks by reusing a fixed number of threads.
* **Performance Optimization**: Reduce the overhead of thread creation and destruction, improving overall system performance.
* **Scalable Design**: Ensure the thread pool can handle varying workloads effectively with

dynamic task submissions.

* **Thread Safety**: Maintain thread safety through proper synchronization mechanisms to prevent race conditions.
* **Graceful Shutdown**: Implement a method to gracefully shut down the thread pool, ensuring

all tasks are completed before termination.

* **Error Handling**: Provide robust error handling to manage task submission and thread pool operations.

# 2. Scope

Target Environment: Unix/Linux-based operating systems.

Primary Audience: System programmers, software developers, and students interested in concurrent programming and multithreading.

### Technologies:

* **Programming Language:** C++
* **Libraries:** POSIX Threads (pthreads)

# 3. Functional Requirements

## Thread Pool Management

* Initialization: Initialize the thread pool with a fixed number of threads.
* Task Submission: Submit tasks to the thread pool using a task queue.
* Task Execution: Threads should fetch tasks from the queue and execute them.
* Shutdown: Provide a mechanism to gracefully shut down the thread pool, ensuring all tasks are completed before termination.

## Concurrency Control

* Implement synchronization mechanisms to manage access to shared resources, including the task queue.
* Use condition variables or semaphores to manage thread waiting and signaling.

## Error Handling

* Implement error handling for thread creation, task submission, and resource allocation failures.

# 4. Non-functional Requirements

* Performance: Minimize latency and maximize throughput by efficiently managing thread resources.
* Scalability: Implement the thread pool to support a large number of tasks.
* Thread Safety: Ensure the implementation is thread-safe with proper synchronization mechanisms in place.

# 5. System Design

## Thread Pool Structure

* Thread Pool: Manages a pool of worker threads.
* Task Queue: A queue to store tasks submitted to the thread pool.
* Worker Threads: Threads that fetch tasks from the task queue and execute them.

## Data Structures

* Task Structure: Defines the task, including a function pointer and arguments.
* Queue Structure: A thread-safe queue for managing tasks.

## Synchronization Mechanisms

* Mutex: Protects shared data, such as the task queue.
* Condition Variable: Signals threads when new tasks are available.

## 6. Input/Output Specifications

### Input:

* Tasks submitted by the user, defined as functions with arguments.

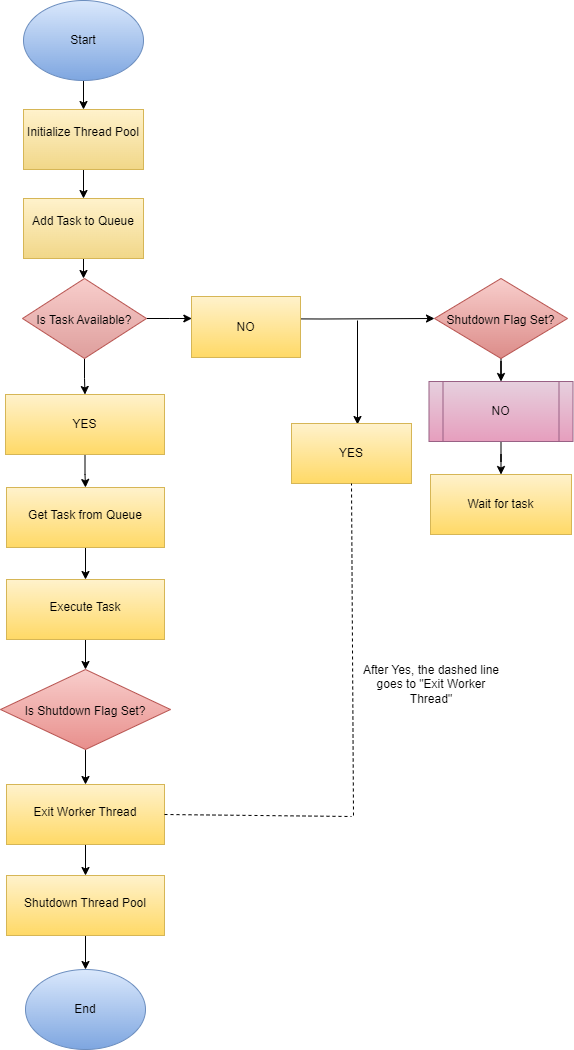
### Output:

* Results of task execution, typically returned via callback functions or stored in shared data structures.

# 7. Challenges

* Deadlock Prevention: Avoid deadlocks by managing locks and condition variables carefully.
* Efficient Task Management: Optimize the task queue and minimize context switching overhead.
* Graceful Shutdown: Ensure all tasks are completed before shutting down the thread pool.

# 8. Flow-Chart Diagram

****

# 9. Skeleton Code Structure

## **Makefile:**

# Compiler and flags

CXX = g++

CXXFLAGS = -std=c++11 -pthread -Wall -Wextra

# Target executable

TARGET = thread\_pool

# Source files

SRCS = main.cpp

# Object files

OBJS = $(SRCS:.cpp=.o)

# Default rule to build the target executable

all: $(TARGET)

# Rule to link object files and create the executable

$(TARGET): $(OBJS)

$(CXX) $(CXXFLAGS) -o $@ $^

# Rule to compile source files into object files

%.o: %.cpp

$(CXX) $(CXXFLAGS) -c $< -o $@

# Rule to clean up the generated files

clean:

rm -f $(OBJS) $(TARGET)

# Rule to run the program

run: $(TARGET)

./$(TARGET)

# Phony targets

.PHONY: all clean run

# Code:

#include <iostream>

#include <vector>

#include <queue>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <functional>

#include <stdexcept>

#include <chrono>

#include <exception>

// Task structure

struct Task {

std::function<void()> function;

};

// Task queue class

class TaskQueue {

public:

explicit TaskQueue(size\_t size) : capacity(size) {}

void push(Task task) {

std::unique\_lock<std::mutex> lock(mutex);

// Wait until there is space in the queue

condition\_full.wait(lock, [this]() { return queue.size() < capacity || shutdown; });

if (shutdown) {

throw std::runtime\_error("Cannot add tasks to a shutdown queue");

}

queue.push(task);

condition\_empty.notify\_one();

}

Task pop() {

std::unique\_lock<std::mutex> lock(mutex);

// Wait until there is at least one task in the queue

condition\_empty.wait(lock, [this]() { return !queue.empty() || shutdown; });

if (shutdown && queue.empty()) {

throw std::runtime\_error("Cannot pop tasks from a shutdown queue");

}

Task task = queue.front();

queue.pop();

condition\_full.notify\_one();

return task;

}

bool is\_empty() const {

std::unique\_lock<std::mutex> lock(mutex);

return queue.empty();

}

void set\_shutdown() {

std::unique\_lock<std::mutex> lock(mutex);

shutdown = true;

condition\_empty.notify\_all();

condition\_full.notify\_all();

}

private:

std::queue<Task> queue;

mutable std::mutex mutex;

std::condition\_variable condition\_full;

std::condition\_variable condition\_empty;

size\_t capacity;

bool shutdown = false;

};

// Thread pool class

class ThreadPool {

public:

ThreadPool(size\_t thread\_count, size\_t queue\_size)

: task\_queue(queue\_size), shutdown(false) {

for (size\_t i = 0; i < thread\_count; ++i) {

threads.emplace\_back(&ThreadPool::worker, this);

}

}

~ThreadPool() {

{

std::unique\_lock<std::mutex> lock(mutex);

shutdown = true;

}

task\_queue.set\_shutdown();

condition.notify\_all();

for (std::thread &thread : threads) {

if (thread.joinable()) {

thread.join();

}

}

}

void add\_task(std::function<void()> function) {

{

std::unique\_lock<std::mutex> lock(mutex);

if (shutdown) {

throw std::runtime\_error("Cannot add task to a shutdown thread pool");

}

}

task\_queue.push({function});

condition.notify\_one();

}

private:

void worker() {

while (true) {

Task task;

try {

{

std::unique\_lock<std::mutex> lock(mutex);

condition.wait(lock, [this]() { return shutdown || !task\_queue.is\_empty(); });

if (shutdown && task\_queue.is\_empty()) {

return;

}

task = task\_queue.pop();

}

task.function();

} catch (const std::exception& e) {

std::cerr << "Task failed: " << e.what() << std::endl;

}

}

}

TaskQueue task\_queue;

std::vector<std::thread> threads;

std::mutex mutex;

std::condition\_variable condition;

bool shutdown;

};

// Example task function

void example\_task(int num) {

std::cout << "Task " << num << " is being processed.\n";

std::this\_thread::sleep\_for(std::chrono::seconds(1)); // Simulate task processing

}

int main() {

try {

ThreadPool pool(4, 10); // Create a thread pool with 4 threads and a queue size of 10

for (int i = 0; i < 10; ++i) {

pool.add\_task([i]() { example\_task(i + 1); });

}

std::this\_thread::sleep\_for(std::chrono::seconds(5)); // Wait for tasks to complete

} catch (const std::exception &e) {

std::cerr << "Error: " << e.what() << std::endl;

}

return 0;

}

# 10. Testing and Validation

### Unit Testing

* Test individual functions such as task addition, thread creation, and task execution.

### Load Testing

* Submit a large number of tasks to the thread pool to ensure it handles high loads efficiently.

### Concurrency Testing

* Test the thread pool under concurrent task submissions to ensure thread safety and correct execution.

# 11. Bug Tracker

# Description

The thread pool implementation in C++ has several issues, including improper handling of task queue overflow, potential race conditions, and inadequate error handling. These issues could lead to task mismanagement, program crashes, or unpredictable behaviour under concurrent load.

## Steps to Reproduce

1. **Compile the Code:**
   * Save the provided code as main.cpp.
   * Compile the code using: g++ -o thread\_pool main.cpp -pthread.
2. **Run the Executable:**
   * Execute the program with: ./thread\_pool.
3. **Test Queue Overflow Handling:**
   * Adjust the queue size and add a number of tasks that exceed the queue capacity to observe behavior.
4. **Stress Test Concurrent Task Addition:**
   * Modify the code to add tasks from multiple threads simultaneously to test thread safety and synchronization.

# Expected Behaviour

1. **Queue Overflow Handling:**
   * The system should handle task overflow gracefully, either by blocking or by rejecting new tasks when the queue is full.
2. **Thread Safety:**
   * Access to the shared task queue should be properly synchronized to prevent race conditions.
3. **Error Handling:**
   * Tasks should not be added if the thread pool is in a shutdown state, and appropriate exceptions should be thrown.

# Actual Behaviour

1. **Queue Overflow Handling:**
   * The queue might not handle overflow scenarios correctly, potentially causing crashes or undefined behaviour when the queue is full.
2. **Thread Safety Issues:**
   * There could be race conditions due to improper synchronization, leading to unpredictable behaviour when tasks are added or removed concurrently.
3. **Error Handling:**
   * The code may not correctly handle errors related to adding tasks after the thread pool has been shut down, leading to possible exceptions or program crashes.

# Code Affected

1. **TaskQueue Class:**
   * Methods: push and pop
   * Issue: Improper handling of queue capacity and synchronization.
2. **ThreadPool Class:**
   * Methods: add\_task and worker
   * Issue: Potential race conditions and improper handling of the shutdown state.

# Suggested Fixes

1. **Update TaskQueue::push:**
   * Ensure that the push method blocks or handles cases where the queue is full. Properly synchronize access to the queue to prevent race conditions.

void push(Task task) {

std::unique\_lock<std::mutex> lock(mutex);

// Wait until there is space in the queue

condition\_full.wait(lock, [this]() { return queue.size() < capacity; });

queue.push(task);

condition\_empty.notify\_one();

}

1. **Update ThreadPool::add\_task:**
   * Modify the add\_task method to handle errors when attempting to add tasks after shutdown has been initiated.

void add\_task(std::function<void()> function) {

{

std::unique\_lock<std::mutex> lock(mutex);

if (shutdown) {

throw std::runtime\_error("Cannot add task to a shutdown thread pool");

}

}

task\_queue.push({function});

condition.notify\_one();

}

1. **Update ThreadPool::worker:**
   * Improve the worker method to ensure correct handling of shutdown signals and proper processing of tasks.

void worker() {

while (true) {

Task task;

{

std::unique\_lock<std::mutex> lock(mutex);

condition.wait(lock, [this]() { return shutdown || !task\_queue.is\_empty(); });

if (shutdown && task\_queue.is\_empty()) {

return;

}

task = task\_queue.pop();

}

task.function();

}

}

# Testing After Fixes

1. **Recompile and Run:**
   * Ensure the code compiles without errors.
   * Run the executable and observe task handling, especially during high load and concurrent task addition.
2. **Test Overflow Handling:**
   * Validate that the queue correctly handles overflow situations by adding tasks beyond the queue’s capacity.
3. **Verify Thread Safety:**
   * Stress test with multiple concurrent task additions to ensure proper synchronization and prevent race conditions.
4. **Check Error Handling:**
   * Verify that attempting to add tasks after shutdown triggers appropriate exceptions and does not lead to undefined behaviour.

# 12. Milestones

**Milestone 1: Initial Setup**

* **Description**: Set up the project environment and basic structure.
* **Tasks**:
  + Create the initial project repository.
  + Set up basic project files and directories.
  + Create a basic Makefile for compilation.
* **Expected Completion**: [29-08-2024]

**Milestone 2: Basic Thread Pool Implementation**

* **Description**: Implement the core functionality of the thread pool.
* **Tasks**:
  + Develop the Task structure.
  + Implement the TaskQueue class with basic functionality.
  + Implement the ThreadPool class with basic worker threads.
  + Test basic functionality with example tasks.
* **Expected Completion**: [30-08-2024]

**Milestone 3: Enhance Task Queue Handling**

* **Description**: Improve task queue handling, including overflow management.
* **Tasks**:
  + Modify TaskQueue to handle overflow scenarios.
  + Ensure proper synchronization and thread safety.
  + Test queue behavior under various loads.
* **Expected Completion**: [30-08-2024]

**Milestone 4: Improve Error Handling and Shutdown Process**

* **Description**: Enhance error handling and shutdown mechanisms.
* **Tasks**:
  + Improve error handling in ThreadPool::add\_task.
  + Refine shutdown process to prevent task addition after shutdown.
  + Implement exception handling in task execution.
* **Expected Completion**: [31-08-2024]

**Milestone 5: Testing and Validation**

* **Description**: Perform thorough testing and validation of the thread pool implementation.
* **Tasks**:
  + Stress test the thread pool with concurrent task additions.
  + Validate queue overflow handling.
  + Verify proper functioning of error handling and shutdown.
* **Expected Completion**: [01-09-2024]

**Milestone 6: Documentation and Final Review**

* **Description**: Complete the documentation and review the project.
* **Tasks**:
  + Finalize project documentation, including usage instructions and code comments.
  + Review and clean up the codebase.
  + Prepare the project for submission or deployment.
* **Expected Completion**: [02-09-2024]

# 12. Main References

**Special thanks to Kiran Sir.** This project has greatly benefited from his expert guidance and insightful feedback. I am truly grateful for his support, mentorship, and the valuable knowledge he has shared throughout this journey.

* **POSIX Threads Programming Guide: Provides documentation and examples for using pthreads.**
* **C++ Standard Library Documentation: Reference for standard libraries used in implementation.**