МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"

Кафедра 806 "Вычислительная математика и программирование"

Курсовой проект

По курсу «Операционные системы»

Студент: Степанов Н.Е.

Группа: М8О-208Б-23

Вариант: 36

Преподаватель: Миронов Е. С.

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2024

**Содержание**

1. Репозиторий

2. Постановка задачи

3. Общие сведения о программе

4. Общий метод и алгоритм решения

5. Исходный код

6. Сборка программы

7. Демонстрация работы программы

8. Выводы

**Репозиторий**

https://github.com/n0w3e/os\_labs

**Постановка задачи**

**Создание планировщика DAG\*’a «джобов» (jobs)\*\***

На языке C\C++ написать программу, которая:

1. По конфигурационному файлу в формате yaml, json или ini принимает спроектированный DAG джобов и проверяет на корректность: отсутствие циклов, наличие только одной компоненты связанности, наличие стартовых и завершающих джоб. Структура описания джоб и их связей произвольная.

2. При завершении джобы с ошибкой, необходимо прервать выполнение всего DAG’а и всех запущенных джоб.

3. (на оценку 4) Джобы должны запускаться максимально параллельно. Должны быть ограниченны параметром – максимальным числом одновременно выполняемых джоб.

4. (на оценку 5) Реализовать для джобов один из примитивов синхронизации мьютекс\семафор\барьер. То есть в конфиге дать возможность определять имена семафоров (с их степенями)\мьютексов\барьеров и указывать их в определение джобов в конфиге. Джобы указанные с одним мьютексом могут выполняться только последовательно (в любом порядке допустимом в DAG). Джобы указанные с одним семафором могут выполнятся параллельно с максимальным числом параллельно выполняемых джоб равным степени семафору. Джобы указанные с одним барьером имеют следующие свойство – зависимые от них джобы начнут выполнятся не раньше того момента времени, когда выполнятся все джобы с указанным барьером.

\* DAG - Directed acyclic graph. Направленный ациклический

граф.

\*\* Джоб(Job) – процесс, который зависит от результата выполнения других процессов (если он не стартовый), которые исполняются до него в DAG, и который порождает данные от которых может быть зависят другие процессы, которые исполняются после него в DAG (если он не завершающий).

**Исходный код**

**Barrier.h:**

#ifndef BARRIER\_H

#define BARRIER\_H

#include <mutex>

#include <condition\_variable>

class Barrier {

public:

explicit Barrier(size\_t count);

void wait();

private:

std::mutex mtx;

std::condition\_variable cv;

size\_t threshold;

size\_t count;

size\_t generation;

};

#endif

**DAG.h:**  
#ifndef DAG\_H

#define DAG\_H

#include "Job.h"

#include <map>

#include <queue>

#include <nlohmann/json.hpp>

class DAG {

public:

std::map<std::string, Job> jobs;

DAG(const nlohmann::json& dag\_json);

bool is\_valid() const;

void execute(int max\_concurrent\_jobs);

private:

bool has\_cycle(const std::string& job\_id, std::map<std::string, bool>& visited, std::map<std::string, bool>& rec\_stack) const;

bool is\_connected() const;

bool has\_start\_and\_end\_jobs() const;

};

#endif

**Job.h:**  
#ifndef JOB\_H

#define JOB\_H

#include <string>

#include <vector>

#include <mutex>

class Job {

public:

std::string id;

std::vector<std::string> dependencies;

bool completed = false;

bool failed = false;

Job() = default;

Job(const std::string& id, const std::vector<std::string>& dependencies);

void execute();

private:

static std::mutex output\_mutex;

};

#endif

**ThreadPool.h:**  
#ifndef THREADPOOL\_H

#define THREADPOOL\_H

#include <vector>

#include <queue>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <functional>

class ThreadPool {

public:

ThreadPool(size\_t threads);

~ThreadPool();

void enqueue(std::function<void()> task);

private:

std::vector<std::thread> workers;

std::queue<std::function<void()>> tasks;

std::mutex queue\_mutex;

std::condition\_variable condition;

bool stop;

};

#endif

**Barrier.cpp:**  
#include "../include/Barrier.h"

Barrier::Barrier(size\_t count)

: threshold(count), count(count), generation(0) {}

void Barrier::wait() {

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

auto gen = generation;

if (--count == 0) {

generation++;

count = threshold;

cv.notify\_all();

} else {

cv.wait(lock, [this, gen] { return gen != generation; });

}

}

**DAG.cpp:**  
#include "../include/DAG.h"

#include "../include/ThreadPool.h"

#include "../include/Barrier.h"

#include <iostream>

#include <queue>

DAG::DAG(const nlohmann::json& dag\_json) {

for (const auto& job\_json : dag\_json["jobs"]) {

std::string id = job\_json["id"];

std::vector<std::string> dependencies = job\_json["dependencies"].get<std::vector<std::string>>();

jobs[id] = Job(id, dependencies);

}

}

bool DAG::is\_valid() const {

for (const auto& [id, job] : jobs) {

std::map<std::string, bool> visited;

std::map<std::string, bool> rec\_stack;

if (has\_cycle(id, visited, rec\_stack)) {

return false;

}

}

if (!is\_connected()) {

return false;

}

if (!has\_start\_and\_end\_jobs()) {

return false;

}

return true;

}

bool DAG::has\_cycle(const std::string& job\_id, std::map<std::string, bool>& visited, std::map<std::string, bool>& rec\_stack) const {

if (!visited[job\_id]) {

visited[job\_id] = true;

rec\_stack[job\_id] = true;

for (const auto& dep : jobs.at(job\_id).dependencies) {

if (!visited[dep] && has\_cycle(dep, visited, rec\_stack)) {

return true;

} else if (rec\_stack[dep]) {

return true;

}

}

}

rec\_stack[job\_id] = false;

return false;

}

bool DAG::is\_connected() const {

if (jobs.empty()) {

return true;

}

std::map<std::string, bool> visited;

for (const auto& [id, job] : jobs) {

if (job.dependencies.empty()) {

std::queue<std::string> queue;

queue.push(id);

visited[id] = true;

while (!queue.empty()) {

std::string current\_job\_id = queue.front();

queue.pop();

for (const auto& [other\_id, other\_job] : jobs) {

if (std::find(other\_job.dependencies.begin(), other\_job.dependencies.end(), current\_job\_id) != other\_job.dependencies.end()) {

if (!visited[other\_id]) {

queue.push(other\_id);

visited[other\_id] = true;

}

}

}

}

}

}

for (const auto& [id, job] : jobs) {

if (!visited[id]) {

return false;

}

}

return true;

}

bool DAG::has\_start\_and\_end\_jobs() const {

bool has\_start = false;

bool has\_end = false;

for (const auto& [id, job] : jobs) {

if (job.dependencies.empty()) {

has\_start = true;

}

bool is\_end = true;

for (const auto& [\_, other\_job] : jobs) {

if (std::find(other\_job.dependencies.begin(), other\_job.dependencies.end(), id) != other\_job.dependencies.end()) {

is\_end = false;

break;

}

}

if (is\_end) {

has\_end = true;

}

}

return has\_start && has\_end;

}

void DAG::execute(int max\_concurrent\_jobs) {

ThreadPool pool(max\_concurrent\_jobs);

Barrier barrier(jobs.size());

for (auto& [id, job] : jobs) {

pool.enqueue([&job, &barrier, this]() {

for (const auto& dep : job.dependencies) {

while (!jobs[dep].completed && !jobs[dep].failed) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(10));

}

if (jobs[dep].failed) {

job.failed = true;

return;

}

}

job.execute();

barrier.wait();

});

}

}

**Job.cpp:**  
#include "../include/Job.h"

#include <iostream>

#include <thread>

std::mutex Job::output\_mutex;

Job::Job(const std::string& id, const std::vector<std::string>& dependencies)

: id(id), dependencies(dependencies) {}

void Job::execute() {

std::cout << "Executing job: " << id << std::endl;

std::this\_thread::sleep\_for(std::chrono::seconds(1));

completed = true;

}

**ThreadPool.cpp:**  
#include "../include/ThreadPool.h"

ThreadPool::ThreadPool(size\_t threads) : stop(false) {

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

workers.emplace\_back([this] {

while (true) {

std::function<void()> task;

{

std::unique\_lock<std::mutex> lock(this->queue\_mutex);

this->condition.wait(lock, [this] { return this->stop || !this->tasks.empty();});

if (this->stop && this->tasks.empty()) return;

task = std::move(this->tasks.front());

this->tasks.pop();

}

task();

}

});

}

}

ThreadPool::~ThreadPool() {

{

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

stop = true;

}

condition.notify\_all();

for (std::thread& worker : workers) {

worker.join();

}

}

void ThreadPool::enqueue(std::function<void()> task) {

{

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

if (stop) throw std::runtime\_error("enqueue on stopped ThreadPool");

tasks.emplace(task);

}

condition.notify\_one();

}

**main.cpp:**  
#include <fstream>

#include <iostream>

#include <nlohmann/json.hpp>

#include "../include/DAG.h"

using json = nlohmann::json;

int main() {

std::ifstream file("dag.json");

json dag\_json;

file >> dag\_json;

DAG dag(dag\_json);

if (!dag.is\_valid()) {

std::cout << "DAG is invalid!" << std::endl;

return 1;

}

dag.execute(4);

return 0;

}

**Демонстрация работы программы**

n0wee@DESKTOP-8QSPN1P:~/Coding/os\_labs/build/KP$ ./lab8

Executing job: job1

Executing job: job2

Executing job: job3

Executing job: job4

Исходный файл dag.json:

{

"jobs": [

{

"id": "job1",

"dependencies": []

},

{

"id": "job2",

"dependencies": ["job1"]

},

{

"id": "job3",

"dependencies": ["job1"]

},

{

"id": "job4",

"dependencies": ["job2", "job3"]

}

]

}

**Вывод**

В данном курсовом проекте я изучил и реализовал механизм выполнения задач с использованием Directed Acyclic Graph (DAG) для управления зависимостями между задачами. Я освоил работу с пулом потоков для параллельного выполнения задач и применил барьер для синхронизации их завершения. Также я научился проверять валидность графа, включая отсутствие циклов, связность и наличие начальных и конечных задач. Это позволило мне создать систему, которая эффективно управляет сложными workflows с учетом зависимостей и параллелизма.