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

Репозиторий

https://github.com/n0w3e/os_labs

Постановка задачи

Создание планировщика DAG*'а «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.