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»**

Группа: М8О-215Б-23

Студент: Кобзев К. А.

Преподаватель: Миронов Е.С.

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

Дата: 17.07.25

Москва, 2025

**Постановка задачи**

**Вариант 0.**

На языке C++ необходимо разработать программу-планировщик, которая управляет выполнением задач (джобов), организованных в виде направленного ациклического графа (DAG). Выполнить из приложения пункты 1 и 2.

Основные требования к программе:

1. Конфигурация и валидация DAG:
   * Программа должна считывать конфигурацию DAG из файла в формате YAML.
   * Необходимо реализовать проверку корректности графа по следующим критериям:

* Отсутствие циклов: в графе не должно быть циклических зависимостей.
* Одна компонента связности: все джобы должны быть частью единого графа.
* Наличие стартовых и завершающих джобов: должен быть хотя бы один джоб без зависимостей (стартовый) и хотя бы один джоб, от которого не зависят другие (завершающий).

Структура описания джобов и их связей в конфигурационном файле является произвольной.

1. Обработка ошибок:

В случае, если какой-либо джоб завершается с ошибкой, выполнение всего DAG и всех активных на данный момент джобов должно быть немедленно прервано.

**Общий метод и алгоритм решения**

Центральным компонентом архитектуры является класс DAGScheduler, который инкапсулирует всю логику по загрузке, валидации и выполнению графа задач.

Алгоритм решения

1. Загрузка и парсинг конфигурации:

* Начальным этапом является чтение и парсинг .yaml файла. Для этой цели используется сторонняя библиотека yaml-cpp, которая позволяет удобно работать с YAML-структурами в C++.
* Данные о каждом джобе (ID, имя, зависимости и флаг для симуляции ошибки) считываются и сохраняются в std::map<int, Job>, где ключ — это уникальный идентификатор джоба.

1. Валидация DAG:

После загрузки данных производится их валидация. Этот процесс состоит из нескольких ключевых проверок:

* Проверка на неизвестные зависимости: осуществляется итерация по всем джобам, и для каждой зависимости проверяется, существует ли джоб с соответствующим ID.
* Проверка на цикличность: для обнаружения циклов применяется алгоритм поиска в глубину (DFS). В процессе обхода графа отслеживаются посещенные узлы и узлы, находящиеся в текущем стеке рекурсии. Если при обходе встречается узел, уже находящийся в стеке, это свидетельствует о наличии цикла.
* Проверка на связность: лля проверки того, что граф состоит из одной компоненты связности, также используется DFS, запущенный от произвольного узла. После завершения обхода размер множества посещенных узлов сравнивается с общим количеством джобов.
* Проверка на наличие стартовых и конечных узлов: проверяется наличие хотя бы одного узла без входящих ребер (стартовый) и хотя бы одного узла без исходящих ребер (завершающий).

3. Выполнение джобов:

* Выполнение джобов организовано в многопоточном режиме для обеспечения параллелизма.
* Основной цикл планировщика run() периодически проверяет, какие джобы готовы к выполнению (т.е. все их зависимости выполнены).
* Для каждого готового к запуску джоба создается отдельный поток (std::thread), в котором выполняется функция execute\_job().
* Внутри execute\_job() имитируется работа с помощью std::this\_thread::sleep\_for().

4. Обработка ошибок и остановка выполнения:

* Для симуляции ошибки в структуре Job предусмотрен флаг should\_fail. Если этот флаг установлен, джоб помечается как FAILED.
* Для координации остановки используется атомарный флаг std::atomic<bool> m\_should\_stop. Когда джоб "падает", он устанавливает этот флаг в true.
* Все остальные работающие джобы перед завершением своей "работы" проверяют состояние этого флага. Если он установлен, они немедленно меняют свой статус на CANCELLED и завершают выполнение, предотвращая дальнейшую работу над DAG.
* Основной цикл планировщика также прекращает запуск новых джобов при установке флага m\_should\_stop.

**Код программы**

**DagScheduler.cpp**

#include "DagScheduler.h"

using namespace std;

bool DAGScheduler::load\_from\_yaml(const string &filename)

{

try

{

YAML::Node config = YAML::LoadFile(filename);

if (!config["jobs"])

{

cerr << "Ошибка: ключ 'jobs' не найден в файле " << filename << endl;

return false;

}

for (const auto &node : config["jobs"])

{

Job job;

job.id = node["id"].as<int>();

job.name = node["name"].as<string>();

if (node["dependencies"])

{

job.dependencies = node["dependencies"].as<vector<int>>();

}

if (node["fail"])

{

job.should\_fail = node["fail"].as<bool>();

}

m\_jobs[job.id] = job;

}

return validate\_dag();

}

catch (const YAML::Exception &e)

{

cerr << "Ошибка парсинга YAML: " << e.what() << endl;

return false;

}

catch (const exception &e)

{

cerr << "Ошибка: " << e.what() << endl;

return false;

}

}

void DAGScheduler::run()

{

m\_should\_stop = false;

vector<thread> worker\_threads;

// Основной цикл: продолжается, пока есть незавершенные задачи

while (!all\_jobs\_done())

{

// Если задача не выполнилась, нам нужно прекратить планирование новых задач

if (m\_should\_stop)

{

cout << "\n--- ОБНАРУЖЕНА ОШИБКА! ЗАПРОС НА ОСТАНОВКУ ---" << endl;

break;

}

vector<int> ready\_jobs;

{

// Находим все задачи, которые готовы к запуску (в ожидании и зависимости удовлетворены)

// Этот блок защищен мьютексом для предотвращения гонки за m\_jobs

lock\_guard<mutex> lock(m\_mutex);

for (auto &[id, job] : m\_jobs)

{

if (job.status == JobStatus::PENDING && are\_dependencies\_met(job))

{

ready\_jobs.push\_back(id);

}

}

}

// Запускаем новый поток для каждой готовой задачи

for (int id : ready\_jobs)

{

m\_jobs[id].status = JobStatus::RUNNING;

worker\_threads.emplace\_back(&DAGScheduler::execute\_job, this, id);

}

// Небольшая пауза, чтобы предотвратить активное ожидание и высокую загрузку процессора

this\_thread::sleep\_for(chrono::milliseconds(100));

}

// Ждем, пока все рабочие потоки завершат свое выполнение

for (auto &th : worker\_threads)

{

if (th.joinable())

{

th.join();

}

}

cout << "\n--- ВЫПОЛНЕНИЕ DAG ЗАВЕРШЕНО ---" << endl;

print\_summary();

}

bool DAGScheduler::validate\_dag()

{

if (m\_jobs.empty())

{

cerr << "Ошибка валидации: нет джобов для выполнения." << endl;

return false;

}

if (!check\_for\_unknown\_dependencies())

return false;

if (!check\_for\_cycles())

return false;

if (!check\_for\_single\_component())

return false;

if (!check\_for\_start\_and\_end\_jobs())

return false;

cout << "Валидация DAG прошла успешно." << endl;

return true;

}

bool DAGScheduler::check\_for\_unknown\_dependencies() const

{

for (const auto &[id, job] : m\_jobs)

{

for (int dep\_id : job.dependencies)

{

if (m\_jobs.find(dep\_id) == m\_jobs.end())

{

cerr << "Ошибка валидации: джоб " << id << " имеет неизвестную зависимость " << dep\_id << endl;

return false;

}

}

}

return true;

}

bool DAGScheduler::check\_for\_cycles()

{

set<int> visited; // Узлы, которые были полностью исследованы

set<int> recursion\_stack; // Узлы, находящиеся в данный момент в стеке рекурсии для текущего пути DFS

for (const auto &[id, job] : m\_jobs)

{

if (visited.find(id) == visited.end())

{

// Если, начиная с этого узла, обнаруживается цикл, весь граф недействителен

if (is\_cyclic\_util(id, visited, recursion\_stack))

{

cerr << "Ошибка валидации: обнаружен цикл в графе!" << endl;

return false;

}

}

}

return true;

}

// Рекурсивный помощник для обнаружения циклов на основе DFS

bool DAGScheduler::is\_cyclic\_util(int id, set<int> &visited, set<int> &recursion\_stack)

{

visited.insert(id);

recursion\_stack.insert(id);

// Находим всех соседей (задачи, которые зависят от текущей задачи)

for (const auto &[job\_id, job\_node] : m\_jobs)

{

for (int dep\_id : job\_node.dependencies)

{

if (dep\_id == id)

{ // job\_node зависит от id

// Если сосед уже находится в стеке рекурсии, значит, у нас есть цикл

if (recursion\_stack.count(job\_id))

{

return true; // Цикл найден

}

// Если сосед еще не посещен, рекурсивно вызываем для него

if (!visited.count(job\_id))

{

if (is\_cyclic\_util(job\_id, visited, recursion\_stack))

{

return true;

}

}

}

}

}

recursion\_stack.erase(id);

return false;

}

bool DAGScheduler::check\_for\_single\_component()

{

if (m\_jobs.empty())

return true;

set<int> visited;

dfs\_connectivity(m\_jobs.begin()->first, visited);

if (visited.size() != m\_jobs.size())

{

cerr << "Ошибка валидации: граф имеет несколько компонент связности." << endl;

return false;

}

return true;

}

void DAGScheduler::dfs\_connectivity(int start\_node, set<int> &visited)

{

visited.insert(start\_node);

for (int dep\_id : m\_jobs.at(start\_node).dependencies)

{

if (visited.find(dep\_id) == visited.end())

{

dfs\_connectivity(dep\_id, visited);

}

}

for (const auto &[id, job] : m\_jobs)

{

for (int dep\_id : job.dependencies)

{

if (dep\_id == start\_node && visited.find(id) == visited.end())

{

dfs\_connectivity(id, visited);

}

}

}

}

bool DAGScheduler::check\_for\_start\_and\_end\_jobs()

{

set<int> has\_outgoing\_edges;

set<int> has\_incoming\_edges;

for (const auto &[id, job] : m\_jobs)

{

if (!job.dependencies.empty())

{

has\_incoming\_edges.insert(id);

}

for (int dep\_id : job.dependencies)

{

has\_outgoing\_edges.insert(dep\_id);

}

}

bool has\_start\_job = false;

for (const auto &[id, job] : m\_jobs)

{

if (has\_incoming\_edges.find(id) == has\_incoming\_edges.end())

{

has\_start\_job = true;

break;

}

}

bool has\_end\_job = false;

for (const auto &[id, job] : m\_jobs)

{

if (has\_outgoing\_edges.find(id) == has\_outgoing\_edges.end())

{

has\_end\_job = true;

break;

}

}

if (!has\_start\_job)

{

cerr << "Ошибка валидации: нет стартовых джобов (без зависимостей)." << endl;

return false;

}

if (!has\_end\_job)

{

cerr << "Ошибка валидации: нет завершающих джобов (от которых никто не зависит)." << endl;

return false;

}

return true;

}

void DAGScheduler::execute\_job(int id)

{

Job &current\_job = m\_jobs.at(id);

{

lock\_guard<mutex> lock(m\_mutex);

cout << "[ЗАПУСК] Джоб " << id << ": " << current\_job.name << endl;

}

// Имитация работы со случайной задержкой

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<> distrib(1000, 3000);

this\_thread::sleep\_for(chrono::milliseconds(distrib(gen)));

// Проверяем, не был ли получен сигнал остановки (например, из-за сбоя другой задачи)

if (m\_should\_stop)

{

current\_job.status = JobStatus::CANCELLED;

lock\_guard<mutex> lock(m\_mutex);

cout << "[ОТМЕНЕН] Джоб " << id << ": " << current\_job.name << endl;

return;

}

// Имитация сбоя задачи

if (current\_job.should\_fail)

{

current\_job.status = JobStatus::FAILED;

// Посылаем сигнал всем остальным потокам на остановку

m\_should\_stop = true;

lock\_guard<mutex> lock(m\_mutex);

cerr << "[ОШИБКА] Джоб " << id << ": " << current\_job.name << " завершился сбоем!" << endl;

}

else

{

current\_job.status = JobStatus::COMPLETED;

lock\_guard<mutex> lock(m\_mutex);

cout << "[УСПЕХ] Джоб " << id << ": " << current\_job.name << endl;

}

}

bool DAGScheduler::are\_dependencies\_met(const Job &job) const

{

for (int dep\_id : job.dependencies)

{

if (m\_jobs.at(dep\_id).status != JobStatus::COMPLETED)

{

return false;

}

}

return true;

}

bool DAGScheduler::all\_jobs\_done() const

{

for (const auto &[id, job] : m\_jobs)

{

if (job.status == JobStatus::PENDING || job.status == JobStatus::RUNNING)

{

return false;

}

}

return true;

}

void DAGScheduler::print\_summary() const

{

cout << "\n--- Итоги выполнения ---" << endl;

for (const auto &[id, job] : m\_jobs)

{

cout << "Джоб " << id << " (" << job.name << "): ";

switch (job.status)

{

case JobStatus::COMPLETED:

cout << "УСПЕШНО";

break;

case JobStatus::FAILED:

cout << "ОШИБКА";

break;

case JobStatus::CANCELLED:

cout << "ОТМЕНЕН";

break;

case JobStatus::PENDING:

cout << "НЕ ЗАПУЩЕН";

break;

case JobStatus::RUNNING:

cout << "ВЫПОЛНЯЕТСЯ (ошибка?)";

break;

}

cout << endl;

}

}

**DagScheduler.h**

#pragma once

#include <iostream>

#include <vector>

#include <string>

#include <map>

#include <set>

#include <thread>

#include <chrono>

#include <mutex>

#include <atomic>

#include <fstream>

#include <stdexcept>

#include <algorithm>

#include <yaml-cpp/yaml.h>

#include <random>

using namespace std;

enum class JobStatus

{

PENDING,

RUNNING,

COMPLETED,

FAILED,

CANCELLED

};

struct Job

{

int id;

string name;

vector<int> dependencies; // Список идентификаторов задач, от которых зависит эта задача

JobStatus status = JobStatus::PENDING;

bool should\_fail = false; // Флаг для имитации сбоя задачи для тестирования

};

// Управляет выполнением DAG

class DAGScheduler

{

public:

bool load\_from\_yaml(const string &filename);

void run();

private:

// Выполняет все проверки валидации для DAG

bool validate\_dag();

// Проверяет, что все зависимости задач ссылаются на существующие задачи

bool check\_for\_unknown\_dependencies() const;

// Проверяет наличие циклов в DAG с помощью поиска в глубину

bool check\_for\_cycles();

// Вспомогательная функция для обнаружения циклов

bool is\_cyclic\_util(int id, set<int> &visited, set<int> &recursion\_stack);

// Проверяет, является ли DAG единой компонентой связности

bool check\_for\_single\_component();

// Вспомогательная функция для проверки связности

void dfs\_connectivity(int start\_node, set<int> &visited);

// Проверяет наличие хотя бы одной начальной и одной конечной задачи

bool check\_for\_start\_and\_end\_jobs();

// Функция, которая выполняется каждым рабочим потоком для одной задачи

void execute\_job(int id);

// Проверяет, все ли зависимости для данной задачи были выполнены

bool are\_dependencies\_met(const Job &job) const;

// Проверяет, все ли задачи в DAG завершены (успешно, с ошибкой или отменены)

bool all\_jobs\_done() const;

// Выводит сводку по конечному статусу всех задач

void print\_summary() const;

private:

// Хранит все задачи, сопоставленные по их ID

map<int, Job> m\_jobs;

// Мьютекс для защиты общего доступа к карте m\_jobs

mutex m\_mutex;

// Флаг для сигнализации всем запущенным задачам об остановке в случае сбоя

atomic<bool> m\_should\_stop{false};

};

**main.cpp**

#include <iostream>

#include "DagScheduler.h"

using namespace std;

int main()

{

DAGScheduler scheduler;

cout << "Загрузка DAG из файла config.yaml..." << endl;

if (!scheduler.load\_from\_yaml("config.yaml"))

{

cerr << "Не удалось загрузить или провалидировать DAG. Выход." << endl;

return 1;

}

cout << "\n--- ЗАПУСК ВЫПОЛНЕНИЯ DAG ---\n"

<< endl;

scheduler.run();

return 0;

}

**config.yaml**

jobs:

- id: 1

name: "Извлечение данных из источника A"

dependencies: []

- id: 2

name: "Извлечение данных из источника B"

dependencies: []

- id: 3

name: "Обработка данных A"

dependencies: [1]

- id: 4

name: "Обработка данных B"

dependencies: [2]

# Этот джоб намеренно вызовет ошибку для демонстрации

- id: 5

name: "Агрегация данных A и B (с ошибкой)"

dependencies: [3, 4]

fail: true # Пользовательский флаг для симуляции ошибки

- id: 6

name: "Формирование отчета"

dependencies: [5]

- id: 7

name: "Отправка отчета"

dependencies: [6]

**Протокол работы программы**

**Тестирование:**

Для тестирования программы был использован конфигурационный файл config.yaml, описанный выше. Он определяет граф из семи джобов. Джоб с id: 5 имеет специальный флаг fail: true, который имитирует его аварийное завершение.

**Процесс выполнения:**

1. Программа успешно загружает и валидирует DAG.
2. Начинается выполнение. Джобы 1 и 2 запускаются параллельно, так как не имеют зависимостей.
3. По мере их завершения, запускаются джобы 3 и 4.
4. После успешного завершения джобов 3 и 4 запускается джоб 5.
5. Джоб 5 завершается с ошибкой, как и было запланировано.
6. Планировщик фиксирует ошибку, устанавливает глобальный флаг остановки и прекращает планирование новых джобов.
7. Все джобы, которые могли бы выполняться в этот момент (если бы они были), получили бы сигнал к отмене.
8. Джобы 6 и 7 не будут запущены, так как их зависимость (джоб 5) не была успешно выполнена.

**Вывод программы:**

Загрузка DAG из файла config.yaml...

Валидация DAG прошла успешно.

--- ЗАПУСК ВЫПОЛНЕНИЯ DAG ---

[ЗАПУСК] Джоб 1: Извлечение данных из источника A

[ЗАПУСК] Джоб 2: Извлечение данных из источника B

[УСПЕХ] Джоб 1: Извлечение данных из источника A

[ЗАПУСК] Джоб 3: Обработка данных A

[УСПЕХ] Джоб 2: Извлечение данных из источника B

[ЗАПУСК] Джоб 4: Обработка данных B

[УСПЕХ] Джоб 3: Обработка данных A

[УСПЕХ] Джоб 4: Обработка данных B

[ЗАПУСК] Джоб 5: Агрегация данных A и B (с ошибкой)

[ОШИБКА] Джоб 5: Агрегация данных A и B (с ошибкой) завершился сбоем!

--- ОБНАРУЖЕНА ОШИБКА! ЗАПРОС НА ОСТАНОВКУ ---

--- ВЫПОЛНЕНИЕ DAG ЗАВЕРШЕНО ---

--- Итоги выполнения ---

Джоб 1 (Извлечение данных из источника A): УСПЕШНО

Джоб 2 (Извлечение данных из источника B): УСПЕШНО

Джоб 3 (Обработка данных A): УСПЕШНО

Джоб 4 (Обработка данных B): УСПЕШНО

Джоб 5 (Агрегация данных A и B (с ошибкой)): ОШИБКА

Джоб 6 (Формирование отчета): НЕ ЗАПУЩЕН

Джоб 7 (Отправка отчета): НЕ ЗАПУЩЕН

**Вывод**

В ходе выполнения данной лабораторной работы была успешно разработана и протестирована программа-планировщик для выполнения задач, представленных в виде направленного ациклического графа.

* Реализованы все ключевые требования, поставленные в задаче:
* Чтение и парсинг конфигурации графа из YAML-файла.
* Комплексная валидация графа на отсутствие циклов, связность и наличие необходимых узлов.
* Корректная обработка ошибок во время выполнения: при сбое одного из джобов происходит немедленная остановка всего процесса, что предотвращает дальнейшее выполнение зависимых задач и экономит ресурсы.