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.
2. 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.
2. 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.