Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Курсовая работа по курсу

«Операционные системы»

Группа: М8О-215Б-23

Студент: Тараскаев Д.М.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 12.12.24

Москва, 2024

Постановка задачи

На языке C\C++ написать программу, которая по конфигурационному файлу в формате json принимает спроектированный DAG джобов и проверяет на корректность:

- отсутствие циклов
- наличие только одной компоненты связности
- наличие стартовых и завершающих джоб

Структура описания джоб и их связей произвольная.

Общий метод и алгоритм решения

Создается DAG, который позволяет выполнять задачи с учетом их зависимостей. Отсутствие циклов проверяется с помощью поиска в глубину. Проверка на существование единственной компоненты связности происходит с помощью поиска в ширину.

Код программы

main.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <queue>
#include <nlohmann/json.hpp>

using json = nlohmann::json;


struct Job {
    std::string id;
    std::vector<std::string> dependencies; // ID джобов, от которых зависит текущий
    std::vector<std::string> next_jobs;    // ID джобов, которые зависят от текущего
    bool completed = false;
    bool failed = false;

    bool execute() {
        std::cout << "Выполняется джоб " << id << std::endl;
        if (failed) {
            return false;
        }
        return true;
    }
};
```

```
class DAGScheduler {
public:
    // Загрузка DAG из JSON-файла
    bool loadFromJson(const std::string& filename) {
        try {
            std::ifstream file(filename);
            if (!file.is_open()) {
                std::cerr << "Ошибка открытия файла: " << filename << std::endl;
                return false;
            }

            json config;
            file >> config;

            // Создаем джобы
            for (const auto& job_json : config["jobs"]) {
                std::string id = job_json["id"];
                Job* job = new Job();
                job->id = id;
                job->failed = job_json["failed"];

                if (job_json.contains("dependencies")) {
                    for (const auto& dep : job_json["dependencies"]) {
                        job->dependencies.push_back(dep);
                    }
                }

                jobs[id] = job;
            }

            // Устанавливаем связи между джобами
            for (auto& pair : jobs) {
                auto job = pair.second;
                for (const auto& dep_id : job->dependencies) {
                    if (jobs.find(dep_id) == jobs.end()) {
                        std::cerr << "Ошибка: зависимость " << dep_id << " для джоба "
<< job->id << " не существует" << std::endl;
                        return false;
                    }
                    jobs[dep_id]->next_jobs.push_back(job->id);
                }
            }

            // Определение стартовых и конечных джобов
            for (const auto& pair : jobs) {
                if (pair.second->dependencies.empty()) {
                    start_jobs.push_back(pair.first);
                }
                if (pair.second->next_jobs.empty()) {
```

```
                end_jobs.push_back(pair.first);
            }
        }

        return true;
    }
    catch (const std::exception& e) {
        std::cerr << "Ошибка при разборе JSON: " << e.what() << std::endl;
        return false;
    }
}

// Проверка на отсутствие циклов с помощью поиска в глубину
bool checkNoCycles() {
    std::unordered_map<std::string, int> visited;

    for (const auto& pair : jobs) {
        visited[pair.first] = 0;
    }

    for (const auto& pair : jobs) {
        if (visited[pair.first] == 0) {
            if (hasCycleDFS(pair.first, visited)) {
                return false;
            }
        }
    }

    return true;
}

// Проверка на наличие только одной компоненты связанности
bool checkSingleComponent() {
    if (jobs.empty()) return true;

    // Создаем неориентированный граф (игнорируем направление связей)
    std::unordered_map<std::string, std::vector<std::string>> undirected_graph;
    for (const auto& pair : jobs) {
        undirected_graph[pair.first] = {};
        for (const auto& dep : pair.second->dependencies) {
            undirected_graph[pair.first].push_back(dep);
            undirected_graph[dep].push_back(pair.first);
        }
    }

    // BFS для проверки связности
    std::unordered_set<std::string> visited;
    std::queue<std::string> q;
```

```
        // Начинаем с первого джоба
        auto first_job = jobs.begin()->first;
        q.push(first_job);
        visited.insert(first_job);

        while (!q.empty()) {
            auto current = q.front();
            q.pop();

            for (const auto& neighbor : undirected_graph[current]) {
                if (visited.find(neighbor) == visited.end()) {
                    visited.insert(neighbor);
                    q.push(neighbor);
                }
            }
        }

        return visited.size() == jobs.size();
    }

    bool checkStartAndEndJobs() {
        return !start_jobs.empty() && !end_jobs.empty();
    }

    bool validateDAG() {
        if (!checkNoCycles()) {
            std::cerr << "Ошибка: DAG содержит циклы" << std::endl;
            return false;
        }

        if (!checkSingleComponent()) {
            std::cerr << "Ошибка: DAG содержит несколько компонент связанности" <<
std::endl;
            return false;
        }

        if (!checkStartAndEndJobs()) {
            std::cerr << "Ошибка: DAG не содержит стартовых или конечных джобов" <<
std::endl;
            return false;
        }

        std::cout << "DAG корректен" << std::endl;
        return true;
    }

    bool executeDAG() {
        std::unordered_map<std::string, int> in_degree;
        std::queue<std::string> q;
```

```
        // Инициализация входящих степеней
        for (const auto& pair : jobs) {
            in_degree[pair.first] = pair.second->dependencies.size();
            if (in_degree[pair.first] == 0) {
                q.push(pair.first);
            }
        }

        while (!q.empty()) {
            std::string current_id = q.front();
            q.pop();

            Job* current_job = jobs[current_id];
            bool success = current_job->execute();
            current_job->completed = true;

            if (!success) {
                std::cerr << "Джоб " << current_id << " завершился с ошибкой,
прерываем выполнение DAG" << std::endl;
                return false;
            }

            // Обработка следующих джобов
            for (const auto& next_id : current_job->next_jobs) {
                in_degree[next_id]--;
                if (in_degree[next_id] == 0) {
                    q.push(next_id);
                }
            }
        }

        // Проверяем, что все джобы выполнены
        for (const auto& pair : jobs) {
            if (!pair.second->completed) {
                std::cerr << "Не все джобы были выполнены, возможно есть цикл" <<
std::endl;
                return false;
            }
        }

        std::cout << "DAG выполнен успешно" << std::endl;
        return true;
    }

    ~DAGScheduler() {
        // Очистка памяти
        for (auto& pair : jobs) {
            delete pair.second;
        }
    }
```

```
private:
    std::unordered_map<std::string, Job*> jobs;
    std::vector<std::string> start_jobs;   // ID стартовых джобов
    std::vector<std::string> end_jobs;     // ID конечных джобов

    // Вспомогательная функция для поиска циклов
    bool hasCycleDFS(const std::string& id, std::unordered_map<std::string, int>&
visited) {
        visited[id] = 1; // В процессе обхода

        for (const auto& next_id : jobs[id]->next_jobs) {
            if (visited[next_id] == 1) {
                return true; // Найден цикл
            }
            if (visited[next_id] == 0 && hasCycleDFS(next_id, visited)) {
                return true;
            }
        }

        visited[id] = 2; // Обработан
        return false;
    }
};

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cerr << "Использование: " << argv[0] << " <путь_к_json_файлу>" <<
std::endl;
        return 1;
    }

    std::string json_file = argv[1];
    DAGScheduler scheduler;

    if (!scheduler.loadFromJson(json_file)) {
        std::cerr << "Ошибка загрузки конфигурации DAG" << std::endl;
        return 1;
    }

    if (!scheduler.validateDAG()) {
        std::cerr << "DAG некорректен" << std::endl;
        return 1;
    }

    if (!scheduler.executeDAG()) {
        std::cerr << "Выполнение DAG прервано из-за ошибки" << std::endl;
        return 1;
    }
```

```
    return 0;
}
```

Протокол работы программы

Тестирование

```
$ ./main ../src/test1.json
DAG is valid
Executing job job1
Executing job job3
Job job3 failed, aborting DAG execution
DAG execution aborted due to an error
```

Вывод

Программа успешно демонстрирует работу планировщика с последовательным выполнением задач.