Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект по курсу

«Операционные системы»

Тема работы
“ Создание планировщика DAG’а «джобов» (jobs)”

Студент: Тутаев Владимир Владимирович
Группа: М8О-201Б-23
Вариант: 2
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

Репозиторий

https://github.com/Volan4ik/MAI_OS

Постановка задачи

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

Вариант 2: Формат конфигурационного файла *.ini

Общие сведения о программе

1. **Структура Job**
   Описывает задачу с уникальным ID, списками зависимостей и зависимых задач, а также полем для ошибок.
2. **Класс DAG**
   Управляет графом задач:
   - Добавление ребер между задачами.
   - Проверка на наличие циклов.
   - Проверка связности графа.
   - Определение стартовых и завершающих задач.
3. **Класс Parser**
   Парсит INI-файл для создания графа задач.
4. **Функция validateDAG**
   Проверяет корректность графа (нет циклов, граф связан, есть стартовые и завершающие задачи).
5. **Класс Scheduler**
   Выполняет задачи графа:
   - Определяет порядок выполнения задач с учетом зависимостей.
   - Запускает задачи в потоках.
   - Обрабатывает ошибки и завершает выполнение, если задача неуспешна.
6. **Функция main**
   Загружает конфигурацию, валидирует граф и запускает планировщик.

Код демонстрирует работу с многопоточностью, синхронизацией (std::mutex, std::atomic), структурой графа и обработкой зависимостей.

Выводы

Код реализует многопоточный планировщик задач на основе DAG, включающий парсинг конфигурационного файла, проверку корректности графа и выполнение задач с учетом зависимостей. Работа демонстрирует навыки работы с графами, многопоточностью, синхронизацией и обработкой ошибок. Итогом является эффективная и структурированная система для управления и выполнения зависимых задач.

Исходный код в Приложении 1

Приложение 1

файл main.cpp:

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <atomic>
#include <chrono>

struct Job {
    int id;
    std::vector<int> dependencies;
    std::vector<int> dependents;
    bool hasError = false;
};

// Класс, представляющий DAG джобов
class DAG {
public:
    std::unordered_map<int, Job> jobs;

    void addEdge(int from, int to) {
        jobs[from].id = from;
        jobs[to].id = to;
        jobs[from].dependents.push_back(to);
        jobs[to].dependencies.push_back(from);
    }

    // Утилита для проверки циклов (DFS)
    bool hasCycleUtil(int job_id, std::unordered_set<int>& visited, std::unordered_set<int>& recStack) {
        if (visited.find(job_id) == visited.end()) {
            visited.insert(job_id);
            recStack.insert(job_id);

            for (auto& dependent : jobs[job_id].dependents) {
                if (visited.find(dependent) == visited.end() && hasCycleUtil(dependent, visited, recStack))
                    return true;
                else if (recStack.find(dependent) != recStack.end())
                    return true;
            }
        }
        recStack.erase(job_id);
        return false;
```

```
    }

    bool hasCycle() {
        std::unordered_set<int> visited;
        std::unordered_set<int> recStack;

        for (auto& pair : jobs) {
            if (hasCycleUtil(pair.first, visited, recStack))
                return true;
        }
        return false;
    }

    bool isConnected() {
        if (jobs.empty()) return true;
        std::unordered_set<int> visited;
        std::queue<int> q;
        q.push(jobs.begin()->first);
        while (!q.empty()) {
            int current = q.front();
            q.pop();
            if (visited.find(current) == visited.end()) {
                visited.insert(current);
                for (auto& dep : jobs[current].dependencies)
                    q.push(dep);
                for (auto& dep : jobs[current].dependents)
                    q.push(dep);
            }
        }
        return visited.size() == jobs.size();
    }

    std::vector<int> getStartJobs() {
        std::vector<int> starts;
        for (auto& pair : jobs) {
            if (pair.second.dependencies.empty())
                starts.push_back(pair.first);
        }
        return starts;
    }

    std::vector<int> getEndJobs() {
        std::vector<int> ends;
        for (auto& pair : jobs) {
            if (pair.second.dependents.empty())
                ends.push_back(pair.first);
        }
        return ends;
    }
};

// Класс для парсинга INI-файла
class Parser {
```

```
public:
    static bool parseINI(const std::string& filename, DAG& dag) {
        std::ifstream infile(filename);
        if (!infile.is_open()) {
            std::cerr << "Не удалось открыть файл: " << filename << std::endl;
            return false;
        }

        std::string line;
        while (std::getline(infile, line)) {
            if (line.empty() || line[0] == ';' || line[0] == '#') continue;

            std::stringstream ss(line);
            std::string token;
            int from = -1, to = -1;

            if (!std::getline(ss, token, '=')) continue;
            size_t pos = token.find("job_id");
            if (pos != std::string::npos)
                token = token.substr(pos + 6);
            if (!(ss >> from)) continue;

            ss.ignore(std::numeric_limits<std::streamsize>::max(), '=');
            if (!(ss >> to)) continue;

            if (from != -1 && to != -1)
                dag.addEdge(from, to);
        }
        infile.close();
        return true;
    }
};

// Функция для валидации DAG
bool validateDAG(DAG& dag) {
    if (dag.hasCycle()) {
        std::cerr << "Ошибка: Обнаружен цикл в DAG." << std::endl;
        return false;
    }

    if (!dag.isConnected()) {
        std::cerr << "Ошибка: DAG имеет более одной компоненты связности." << std::endl;
        return false;
    }

    auto starts = dag.getStartJobs();
    auto ends = dag.getEndJobs();

    if (starts.empty()) {
        std::cerr << "Ошибка: Отсутствуют стартовые джобы." << std::endl;
        return false;
    }
```

```
    if (ends.empty()) {
        std::cerr << "Ошибка: Отсутствуют завершающие джобы." << std::endl;
        return false;
    }

    std::cout << "DAG прошел валидацию успешно." << std::endl;
    std::cout << "Стартовые джобы: ";
    for (auto id : starts) std::cout << id << " ";
    std::cout << "\nЗавершающие джобы: ";
    for (auto id : ends) std::cout << id << " ";
    std::cout << std::endl;

    return true;
}

// Класс планировщика джоб + создание потоков
class Scheduler {
public:
    Scheduler(DAG& dag) : dag(dag), stopFlag(false) {}

    void execute() {
        std::unordered_map<int, int> dependencyCount;
        for (auto& pair : dag.jobs) {
            dependencyCount[pair.first] = pair.second.dependencies.size();
        }

        std::queue<int> readyQueue;
        for (auto& pair : dependencyCount) {
            if (pair.second == 0)
                readyQueue.push(pair.first);
        }

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

        while (true) {
            {
                std::lock_guard<std::mutex> lock(queueMutex);
                if (readyQueue.empty() && workers.empty()) {
                    break;
                }

                while (!readyQueue.empty()) {
                    int jobId = readyQueue.front();
                    readyQueue.pop();
                    workers.emplace_back(&Scheduler::runJob, this, jobId, std::ref(dependencyCount),
std::ref(readyQueue), std::ref(queueMutex));
                }
            }

            for (auto it = workers.begin(); it != workers.end(); ) {
                if (it->joinable()) {
                    it->join();
```

```
                it = workers.erase(it);
            } else {
                ++it;
            }
        }

        if (stopFlag) {
            std::cerr << "Выполнение прервано из-за ошибки." << std::endl;
            break;
        }
    }
}

private:
    DAG& dag;
    std::atomic<bool> stopFlag;

    void runJob(int jobId, std::unordered_map<int, int>& dependencyCount, std::queue<int>& readyQueue,
std::mutex& queueMutex) {
        if (stopFlag) return;

        {
            std::cout << "Запуск джобы " << jobId << std::endl;
        }

        bool success = executeJob(jobId);
        if (!success) {
            std::cerr << "Джоба " << jobId << " завершилась с ошибкой." << std::endl;
            stopFlag = true;
            return;
        }

        {
            std::cout << "Джоба " << jobId << " завершена успешно." << std::endl;
        }

        std::lock_guard<std::mutex> lock(queueMutex);
        for (auto& dependent : dag.jobs[jobId].dependents) {
            dependencyCount[dependent]--;
            if (dependencyCount[dependent] == 0) {
                readyQueue.push(dependent);
            }
        }
    }

    bool executeJob(int jobId) {
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
        return jobId != 3;
    }
};

int main(int argc, char* argv[]) {
    if(argc < 2) {
```

```
        std::cerr << "Использование: " << argv[0] << " <config.ini>" << std::endl;
        return 1;
    }

    std::string configFile = argv[1];
    DAG dag;

    if(!Parser::parseINI(configFile, dag)) {
        return 1;
    }

    if(!validateDAG(dag)) {
        return 1;
    }

    Scheduler scheduler(dag);
    scheduler.execute();

    return 0;
}
```

файл config.ini:

```
job_id = 1 -> job_id = 4
job_id = 2 -> job_id = 4
job_id = 3 -> job_id = 5
job_id = 4 -> job_id = 6
job_id = 5 -> job_id = 6
```