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: Матвеев Семен Романович Преподаватель: Миронов Евгений Сергеевич

Оценка:

Дата: 13.03.25

Москва, 2025

**Постановка задачи**

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.

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).

Реализация через ini.

**Общий метод и алгоритм решения**

Программа представляет собой симулятор выполнения Directed Acyclic Graph (DAG) — направленного ациклического графа задач (джоб). Джобы зависят друг от друга, и их выполнение должно происходить в определённом порядке, заданном зависимостями. Программа использует многопоточность для параллельного выполнения задач на одном уровне графа.

Алгоритм работы программы:

1. Парсинг INI-файла. Программа считывает конфигурационный файл (по умолчанию config.ini), в котором в секции [DAG] заданы:

* jobs: список всех джоб (задач)
* dependencies: зависимости между джобами в формате A->B (джоба B зависит от джобы A)

Если файл не удалось открыть или данные некорректны, программа завершается с ошибкой.

2. Построение графа зависимостей:

* На основе данных из INI-файла строится граф зависимостей:
* Каждая джоба — это узел графа.
* Зависимости A->B — это направленные рёбра от узла A к узлу B.
* Также вычисляются входящие степени (количество зависимостей) для каждой джобы.

3. Топологическая сортировка:

* Программа выполняет топологическую сортировку графа, чтобы определить порядок выполнения джоб.
* Если в графе обнаружен цикл или недостижимые узлы, программа завершается с ошибкой.

4. Поиск стартовых и завершающих джоб:

* Стартовые джобы — это джобы с нулевой входящей степенью (не зависят ни от чего).
* Завершающие джобы — джобы, от которых ничего не зависит (нет исходящих рёбер).

5. Параллельное выполнение джоб:

* Джобы выполняются по уровням: уровень 0: стартовые джобы; уровень N: джобы, которые могут быть выполнены только после завершения всех джоб уровня N-1.
* На каждом уровне джобы выполняются параллельно с использованием потоков (std::thread).
* После завершения всех джоб на текущем уровне программа переходит к следующему уровню (барьерная синхронизация).

6. Обработка ошибок:

* Если одна из джоб завершается с ошибкой, выполнение всего DAG прерывается.

7. Завершение программы:

* Если все джобы выполнены успешно, программа завершается с сообщением об успехе.
* В случае ошибки программа завершается с соответствующим сообщением.

**Код программы**

**kp.cpp:**

#include <iostream>

#include <fstream>

#include <sstream>

#include <string>

#include <vector>

#include <queue>

#include <unordered\_map>

#include <stdexcept>

#include <algorithm>

#include <thread>

#include <atomic>

// Парсер INI для секции [DAG]

bool parseIniFile(const std::string& filename,

std::vector<std::string>& jobs,

std::vector<std::pair<std::string, std::string> >& dependencies)

{

std::ifstream file(filename);

if (!file.is\_open()) {

std::cerr << "Не удалось открыть файл: " << filename << std::endl;

return false;

}

bool inDagSection = false;

std::string line;

while (std::getline(file, line)) {

while(!line.empty() && (line.front() == ' ' || line.front() == '\t'))

line.erase(line.begin());

while(!line.empty() && (line.back() == ' ' || line.back() == '\t'))

line.pop\_back();

if (line.empty() || line[0] == '#' || line[0] == ';')

continue;

if (line.size() >= 5 && line.front() == '[' && line.back() == ']') {

std::string sectionName = line.substr(1, line.size() - 2);

inDagSection = (sectionName == "DAG");

continue;

}

if (inDagSection) {

const std::string jobsPrefix = "jobs=";

const std::string depsPrefix = "dependencies=";

if (line.find(jobsPrefix) == 0) {

std::string jobsLine = line.substr(jobsPrefix.size());

std::stringstream ss(jobsLine);

while (ss.good()) {

std::string job;

if (!std::getline(ss, job, ','))

break;

while(!job.empty() && (job.front() == ' ' || job.front() == '\t'))

job.erase(job.begin());

while(!job.empty() && (job.back() == ' ' || job.back() == '\t'))

job.pop\_back();

if (!job.empty())

jobs.push\_back(job);

}

}

else if (line.find(depsPrefix) == 0) {

std::string depsLine = line.substr(depsPrefix.size());

std::stringstream ss(depsLine);

while (ss.good()) {

std::string dep;

if (!std::getline(ss, dep, ';'))

break;

while(!dep.empty() && (dep.front() == ' ' || dep.front() == '\t'))

dep.erase(dep.begin());

while(!dep.empty() && (dep.back() == ' ' || dep.back() == '\t'))

dep.pop\_back();

if (!dep.empty()) {

size\_t arrowPos = dep.find("->");

if (arrowPos == std::string::npos) {

std::cerr << "Неверный формат зависимости: " << dep << std::endl;

return false;

}

std::string from = dep.substr(0, arrowPos);

std::string to = dep.substr(arrowPos + 2);

while(!from.empty() && (from.front() == ' ' || from.front() == '\t'))

from.erase(from.begin());

while(!from.empty() && (from.back() == ' ' || from.back() == '\t'))

from.pop\_back();

while(!to.empty() && (to.front() == ' ' || to.front() == '\t'))

to.erase(to.begin());

while(!to.empty() && (to.back() == ' ' || to.back() == '\t'))

to.pop\_back();

if (!from.empty() && !to.empty())

dependencies.push\_back(std::make\_pair(from, to));

}

}

}

}

}

file.close();

return true;

}

// Алгоритм топологической сортировки (проверка отсутствия циклов и единственной компоненты)

std::vector<std::string> topologicalSort(

const std::vector<std::string>& jobs,

const std::unordered\_map<std::string, std::vector<std::string> >& graph,

const std::unordered\_map<std::string, int>& inDegree)

{

std::unordered\_map<std::string, int> deg = inDegree;

std::queue<std::string> q;

for (const auto& job : jobs) {

if (deg[job] == 0)

q.push(job);

}

std::vector<std::string> result;

while (!q.empty()) {

std::string current = q.front();

q.pop();

result.push\_back(current);

auto it = graph.find(current);

if (it != graph.end()) {

for (const auto& neighbor : it->second) {

deg[neighbor]--;

if (deg[neighbor] == 0)

q.push(neighbor);

}

}

}

if (result.size() != jobs.size())

throw std::runtime\_error("В графе обнаружен цикл или недостижимая часть (неодносвязный DAG).");

return result;

}

// Поиск стартовых и завершающих джоб

void findStartAndEndJobs(

const std::vector<std::string>& jobs,

const std::unordered\_map<std::string, std::vector<std::string> >& graph,

const std::unordered\_map<std::string, int>& inDegree,

std::vector<std::string>& startJobs,

std::vector<std::string>& endJobs)

{

for (const auto& job : jobs) {

if (inDegree.at(job) == 0)

startJobs.push\_back(job);

auto it = graph.find(job);

if (it == graph.end() || it->second.empty())

endJobs.push\_back(job);

}

if (startJobs.empty())

throw std::runtime\_error("Нет стартовых джоб (inDegree=0).");

if (endJobs.empty())

throw std::runtime\_error("Нет завершающих джоб (нет джоб без исходящих рёбер).");

}

// Симуляция выполнения джобы

void executeJob(const std::string& jobName)

{

std::cout << "Запуск джобы " << jobName << "..." << std::endl;

// Искусственная ошибка для демонстрации

// if (jobName.find("C") != std::string::npos)

// throw std::runtime\_error("Ошибка при выполнении " + jobName);

std::cout << "Джоба " << jobName << " успешно завершена." << std::endl;

}

void executeDAGWithBarrier(

const std::vector<std::string>& jobs,

const std::vector<std::string>& topoOrder,

const std::unordered\_map<std::string, std::vector<std::string> >& graph)

{

// Вычисляем уровни для каждой джобы

std::unordered\_map<std::string, int> levels;

for (const auto& job : jobs)

levels[job] = 0;

for (const auto& job : topoOrder) {

auto it = graph.find(job);

if (it != graph.end()) {

for (const auto& child : it->second) {

levels[child] = std::max(levels[child], levels[job] + 1);

}

}

}

// Группируем джобы по уровням

std::unordered\_map<int, std::vector<std::string> > levelGroups;

int maxLevel = 0;

for (const auto& job : jobs) {

int lvl = levels[job];

levelGroups[lvl].push\_back(job);

if (lvl > maxLevel)

maxLevel = lvl;

}

std::atomic<bool> errorOccurred(false);

// Выполняем по уровням

for (int lvl = 0; lvl <= maxLevel; ++lvl) {

std::vector<std::string> jobsAtLevel = levelGroups[lvl];

std::vector<std::thread> threads;

std::cout << "Выполнение джоб уровня " << lvl << " (" << jobsAtLevel.size() << " шт.)" << std::endl;

for (const auto& job : jobsAtLevel) {

threads.push\_back(std::thread([job, &errorOccurred]() {

try {

executeJob(job);

} catch (const std::exception& ex) {

std::cerr << "Выполнение прервано из-за ошибки: " << ex.what() << std::endl;

errorOccurred = true;

}

}));

}

// Barrier: ожидаем завершения всех потоков данного уровня

for (auto& t : threads) {

t.join();

}

if (errorOccurred)

throw std::runtime\_error("Один из джоб завершился с ошибкой, прерывание DAG.");

}

}

int main(int argc, char\* argv[])

{

std::string iniFile = (argc > 1) ? argv[1] : "config.ini";

std::vector<std::string> jobs;

std::vector<std::pair<std::string, std::string> > deps;

if (!parseIniFile(iniFile, jobs, deps)) {

std::cerr << "Ошибка парсинга INI-файла." << std::endl;

return 1;

}

if (jobs.empty()) {

std::cerr << "Секция [DAG], параметр jobs= ... не заполнен." << std::endl;

return 1;

}

// Формируем граф и считаем входящие степени

std::unordered\_map<std::string, std::vector<std::string> > graph;

std::unordered\_map<std::string, int> inDegree;

for (const auto& job : jobs) {

inDegree[job] = 0;

}

for (const auto& d : deps) {

const std::string& from = d.first;

const std::string& to = d.second;

if (inDegree.find(from) == inDegree.end() || inDegree.find(to) == inDegree.end()) {

std::cerr << "Зависимость ссылается на несуществующую джобу: " << from << "->" << to << std::endl;

return 1;

}

graph[from].push\_back(to);

inDegree[to]++;

}

try {

std::vector<std::string> topoOrder = topologicalSort(jobs, graph, inDegree);

std::vector<std::string> startJobs, endJobs;

findStartAndEndJobs(jobs, graph, inDegree, startJobs, endJobs);

std::cout << "Стартовые джобы: ";

for (const auto& sj : startJobs)

std::cout << sj << " ";

std::cout << std::endl;

std::cout << "Завершающие джобы: ";

for (const auto& ej : endJobs)

std::cout << ej << " ";

std::cout << std::endl;

// Выполнение DAG с использованием параллельных джоб и барьера

executeDAGWithBarrier(jobs, topoOrder, graph);

std::cout << "Все джобы DAG успешно выполнены!" << std::endl;

}

catch (const std::exception& except) {

std::cerr << "Ошибка выполнения DAG: " << except.what() << std::endl;

return 1;

}

return 0;

}

**config.ini:**

[DAG]

jobs=JobA, JobB, JobC, JobD, JobE

dependencies=JobA->JobB; JobA->JobC; JobB->JobD; JobC->JobD; JobD->JobE

**Протокол работы программы**

./kp config.ini

//DAG прошел валидацию успешно.

Стартовые джобы: JobA

Завершающие джобы: JobE

Выполнение джоб уровня 0 (1 шт.)

Запуск джобы JobA...

Джоба JobA успешно завершена.

Выполнение джоб уровня 1 (2 шт.)

Запуск джобы JobB...

Джоба JobB успешно завершена.

Запуск джобы JobC...

Джоба JobC успешно завершена.

Выполнение джоб уровня 2 (1 шт.)

Запуск джобы JobD...

Джоба JobD успешно завершена.

Выполнение джоб уровня 3 (1 шт.)

Запуск джобы JobE...

Джоба JobE успешно завершена.

Все джобы DAG успешно выполнены!

**Вывод**

В ходе выполнения данной лабораторной работы я освежил знания в области дискретной математики и теории графов. Кроме того, я получил опыт работы с .ini файлами.