Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовой проект по курсу**

**«Операционные системы»**

Студент: Климов Иван Павлович

Группа: М8О–212Б–22

Вариант: 4

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2023

1. **Цель работы**
2. Приобретение практических навыков в использовании знаний, полученных в течении курса
3. Проведение исследования в выбранной предметной области
4. **Задание**

Необходимо спроектировать и реализовать программный прототип в соответствии с выбранным вариантом. Произвести анализ и сделать вывод на основании данных, полученных при работе программного прототипа.

Создание планировщика DAG\*’a «джобов» (jobs)\*\*

На языке C\C++ написать программу, которая:

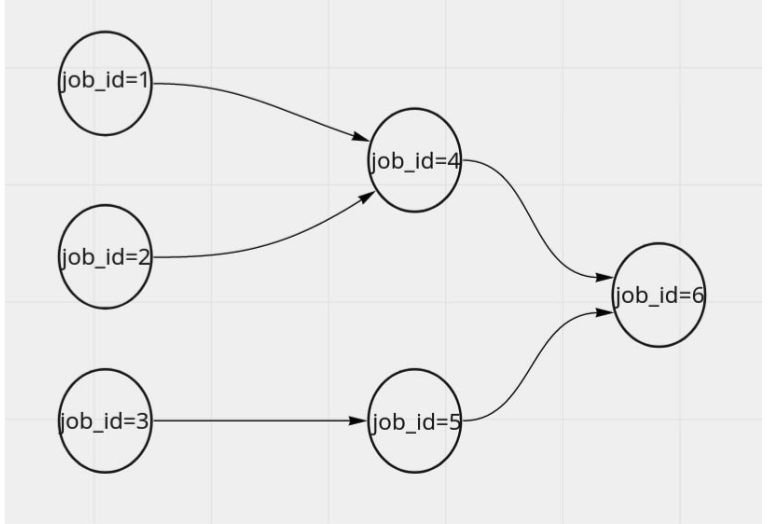
По конфигурационному файлу в формате yaml, json или ini принимает

спроектированный DAG джобов и проверяет на корректность:

отсутствие циклов, наличие только одной компоненты связанности,

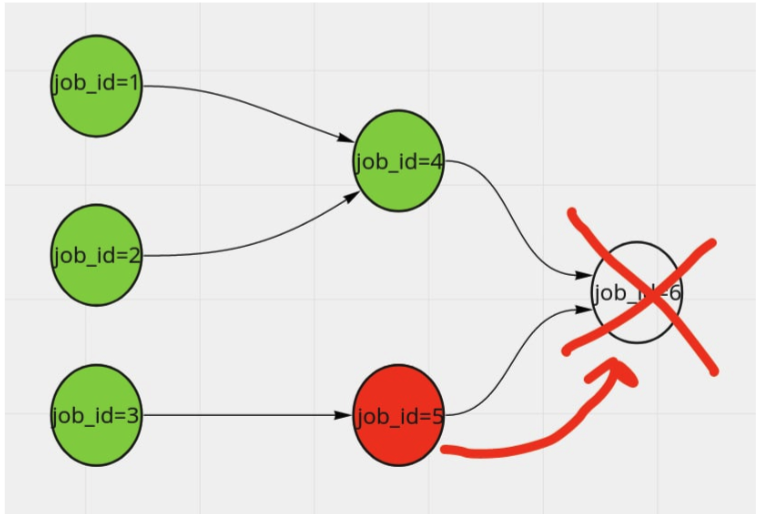
наличие стартовых и завершающих джоб. Структура описания джоб и

их связей произвольная.



При завершении джобы с ошибкой, необходимо прервать выполнение

всего DAG’а и всех запущенных джоб.



(на оценку 4) Джобы должны запускаться максимально параллельно.

Должны быть ограниченны параметром – максимальным числом

одновременно выполняемых джоб.

**Вариант 36: Json\Barrier**

1. **Общий метод и алгоритм решения**

Создаем конфигурационный файл формата json. В нем хранится наш граф, который будем обрабатывать.

В main.cpp проверяем наш DAG на наличие циклов, компонент связности и начального/конечного узла. Потом запускаем сами job`ы. Для этого находим джобы «первого уровня». Запускаем их в многопоточном режиме при помощи потоков. В сами же потоках делаем дочерние процессы, которые выполняют джоб. Далее запускаем уже зависимые узлы. Для этого на каждом этапе выбираем все те джобы, которые зависят от предыдущих. Выполняем джобы аналогично с первым этапом. Барьер реализовал при помощи **thread.join().** В консоль выводим результат работы каждого джоба.

1. **Основные файлы программы**

**main.cpp:**

#include <iostream>

#include <fstream>

#include <json/json.h>

#include <set>

#include <functional>

#include <vector>

#include <string>

#include <sys/wait.h>

#include <thread>

#include <queue>

#include <unordered\_set>

#include <unordered\_map>

#include "unistd.h"

#include "sys/wait.h"

#include "stdio.h"

#include "stdlib.h"

#include "ctype.h"

void executeJob(const std::string& job) {

pid\_t pid = fork();

if (pid == -1) {

throw std::runtime\_error("Ошибка при создании процесса");

}

if (pid == 0) {

//sleep(1);

std::cout << "Работаю " << job << std::endl;

exit(0);

} else {

std::cout << "Выполняется работа: " << job << std::endl;

wait(NULL);

sleep(1);

std::cout << "Работа завершена: " << job << std::endl;

sleep(1);

//wait(NULL);

}

}

void processJob(const std::string& job, const std::unordered\_map<std::string, std::vector<Json::String>>& dependencies,

std::unordered\_set<std::string>& visitedJobs) {

if (visitedJobs.size() == dependencies.size()) {

return;

}

if (visitedJobs.count(job) == 0) {

visitedJobs.insert(job);

}

if (dependencies.count(job) > 0) {

const std::vector<Json::String>& currentDependencies = dependencies.at(job);

if (currentDependencies.size() > 1) {

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

for (const auto& dependency : currentDependencies) {

if (visitedJobs.count(dependency) == 0) {

visitedJobs.insert(dependency);

if (visitedJobs.size() == dependencies.size()) {

threads.emplace\_back(executeJob, dependency);

} else {

threads.emplace\_back(processJob, job, std::ref(dependencies), std::ref(visitedJobs));

}

}

}

for (auto& thread : threads) {

thread.join();

}

} else {

for (const auto& dependency : currentDependencies) {

if (visitedJobs.count(dependency) == 0) {

visitedJobs.insert(dependency);

if (visitedJobs.size() == dependencies.size()) {

executeJob(dependency);

} else {

processJob(dependency, dependencies, visitedJobs);

}

}

}

}

}

executeJob(job);

}

bool hasCycle(const Json::Value& jobs, const std::string& currentJob, std::set<std::string>& visited, std::set<std::string>& recursionStack) {

visited.insert(currentJob);

recursionStack.insert(currentJob);

const Json::Value& dependencies = jobs[currentJob]["dependencies"]; //?

for (const auto& dependency : dependencies) {

const std::string& dependencyJob = dependency.asString();

if (recursionStack.count(dependencyJob) == 1) {

return true;

}

if (!visited.count(dependencyJob) && hasCycle(jobs, dependencyJob, visited, recursionStack)) {

return true;

}

}

recursionStack.erase(currentJob);

return false;

}

bool isValidDAG(const Json::Value& jobs) {

std::set<std::string> visited;

std::set<std::string> recursionStack;

for (const auto& job : jobs.getMemberNames()) {

if (!visited.count(job) && hasCycle(jobs, job, visited, recursionStack)) {

return false;

}

}

return true;

}

bool hasOnlyOneComponent(const Json::Value& jobs) {

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

std::set<std::string> visited;

// Строим список смежности

for (const auto& job : jobs.getMemberNames()) {

const Json::Value& dependencies = jobs[job]["dependencies"];

for (const auto& dependency : dependencies) {

adjacencyList[dependency.asString()].push\_back(job);

adjacencyList[job].push\_back(dependency.asString());

}

}

int componentCount = 0;

// Обходим граф и считаем количество компонент связности

for (const auto& job : jobs.getMemberNames()) {

if (visited.count(job) == 0) {

std::set<std::string> component;

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

q.push(job);

visited.insert(job);

while (!q.empty()) {

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

q.pop();

component.insert(currentJob);

for (const auto& neighbor : adjacencyList[currentJob]) {

if (visited.count(neighbor) == 0) {

q.push(neighbor);

visited.insert(neighbor);

}

}

}

componentCount++;

}

}

return componentCount > 1;

}

bool hasStartAndEndJobs(const Json::Value& jobs) {

std::set<std::string> startJobs;

std::set<std::string> endJobs;

for (const auto& job : jobs.getMemberNames()) {

const Json::Value& dependencies = jobs[job]["dependencies"];

if (dependencies.empty()) {

startJobs.insert(job);

}

for (const auto& dependency : dependencies) {

endJobs.erase(dependency.asString());

}

endJobs.insert(job);

}

return !startJobs.empty() && !endJobs.empty();

}

int main() {

std::ifstream file("dag.json");

Json::Value data;

file >> data;

Json::Value jobb = data["jobs"];

if (!isValidDAG(jobb)) {

throw std::runtime\_error("DAG содержит цикл");

}

if (hasOnlyOneComponent(jobb)) {

throw std::runtime\_error("DAG имеет больше чем один компонент связности");

}

if (!hasStartAndEndJobs(jobb)) {

throw std::runtime\_error("DAG не имеет начального и конечного джоба");

}

std::cout << "DAG валидный, можем продолжать" << std::endl << std::endl;

sleep(1);

int n = data["jobs"].getMemberNames().size();

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

for (const auto& job : data["jobs"].getMemberNames()) {

jobs.push\_back(job);

}

std::unordered\_map<std::string, std::vector<Json::String>> dependencies\_map;

std::unordered\_set<std::string> visitedJobs;

for (const auto& job : data["jobs"].getMemberNames()) {

std::vector<Json::String> dependencies;

for (const auto& dependency : data["jobs"][job]["dependencies"]) {

dependencies.push\_back(dependency.asString());

}

dependencies\_map[job] = dependencies;

}

for (const std::string& job : jobs) {

processJob(job, dependencies\_map, visitedJobs);

}

std::cout << "Все работы завершены" << std::endl;

return 0;

}

**dag.json**

{

"jobs": {

"job1": {

"dependencies": []

},

"job2": {

"dependencies": []

},

"job3": {

"dependencies": ["job1", "job2"]

},

"job4": {

"dependencies": ["job3"]

},

"job5": {

"dependencies": ["job3"]

},

"job6": {

"dependencies": ["job4", "job5"]

},

"job7": {

"dependencies": ["job6"]

}

}

}

1. **Демонстрация работы программы**

ivanklimov@MacBook-Air-Ivan-2 src % ./main

DAG валидный, можем продолжать

Выполняется работа: job1

Работаю job1

Работа завершена: job1

Выполняется работа: job2

Работаю job2

Работа завершена: job2

Выполняется работа: job3

Работаю job3

Работа завершена: job3

Выполняется работа: job4

Работаю job4

Работа завершена: job4

Выполняется работа: job5

Работаю job5

Работа завершена: job5

Выполняется работа: job6

Работаю job6

Работа завершена: job6

Выполняется работа: job7

Работаю job7

Работа завершена: job7

Все работы завершены

**Пример, когда есть цикл**

ivanklimov@MacBook-Air-Ivan-2 src % ./main

libc++abi: terminating with uncaught exception of type std::runtime\_error: DAG содержит цикл

ivanklimov@MacBook-Air-Ivan-2 src %

1. **Вывод**

После выполнения этого курсового проекта я получил опыт работы с файлами json. Создал проект, который обрабатывает **DAG job`ов**. Сам **DAG** мы задаем при помощи конфигурационного файла. Теперь, после компиляции, программа проверяет наличие циклов, компонент связности и начальный/конечный узел. Потом мы последовательно запускаем **job`**ы, которые выполняются в той последовательности, в которой они заданы в конфигурационном файле

Благодаря данному курсовому проекту я закрепил все полученные в курсе операционных систем знания. Я получил большой опыт в работе с json и библиотекой jsoncpp, научился логически строить структуру своего проекта, вспомнил, про работу с графами.