Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовой проект по курсу**

**«Операционные системы»**

Студент: Головенко Анатолий Валерьевич

Группа: М8О–212Б–22

Вариант: 31

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

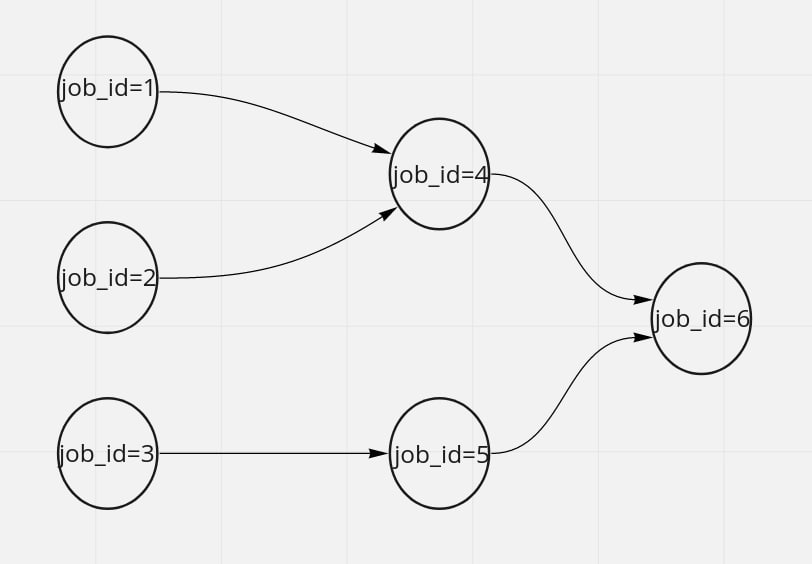
Подпись: \_\_\_\_\_\_\_\_\_\_

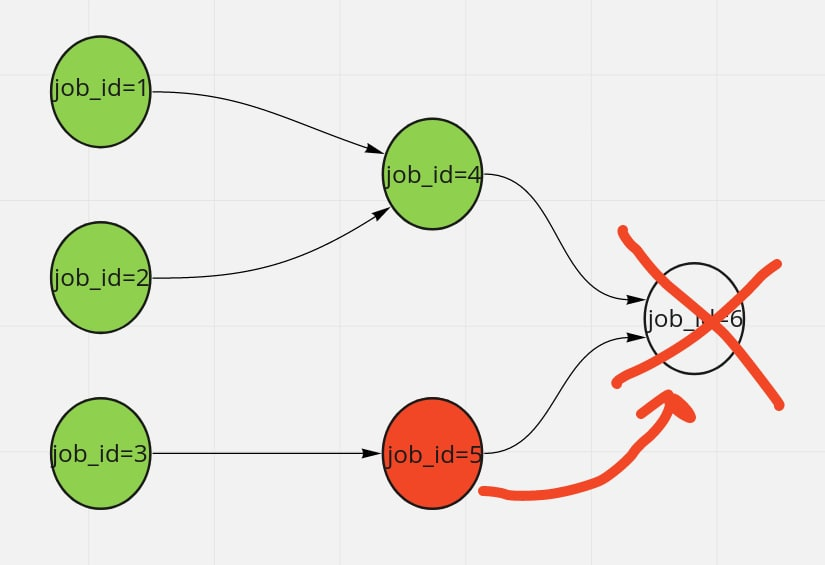
Москва, 2023.

**Постановка задачи**

## Задание

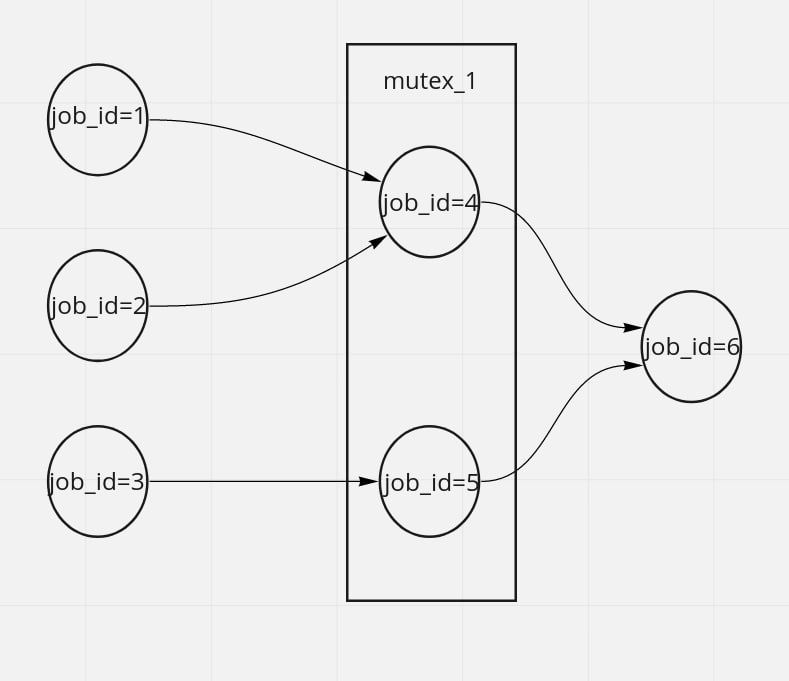
На языке C\C++ написать программу, которая:

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. То есть в конфиге дать возможность определять имена семафоров (с их степенями)\мьютексов\барьеров и указывать их в определение джобов в конфиге. Джобы указанные с одним мьютексом могут выполняться только последовательно (в любом порядке допустимом в DAG). Джобы указанные с одним семафором могут выполнятся параллельно с максимальным числом параллельно выполняемых джоб равным степени семафору. Джобы указанные с одним барьером имеют следующие свойство – зависимые от них джобы начнут выполнятся не раньше того момента времени, когда выполнятся все джобы с указанным барьером.



\* 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).

**Вариант 37:** Yaml\Mutex

**Общие сведения о программе**

Основной файл программы - main.cpp. В программе используются следующие системные вызовы:

1. **fork** – создание дочернего процесса.
2. **kill** – завершает процесс.
3. **execv** – выполняет файл.
4. **waitpid** – ожидает завершения процесса и принимает от него сигнал

**Общий метод и алгоритм решения**.

Для обработки ini файла использовалась библиотека, скаченная с GitHub, yaml-cpp-master.

Для нахождения циклов используется обход графа в глубину. Текущая вершина помечается 1 и обход продолжается вглубь, когда вершина покидается она помечается 2. Таким образом, если на пути встречается вершина, помеченная 1, то в графе есть цикл.

Для проверки на наличие только одной компоненты связности сначала происходит обход графа и помечаются посещённые вершины. Потом запускается ещё один обход графа, если при нём не посещается ни одна помеченная вершина, то в графе больше одной компоненты связности.

При запуске номера процессов берутся из start\_jobs – туда изначально записаны все номера стартовых джобов. Номера запущенных процессов помещаются в очередь. Для ожидания завершения джоб существует отдельный поток, он берет номера процессов из очереди и ждёт их завершения с помощью waitpid и принимает сигнал, который позволяет понять завершился ли процесс с ошибкой.

**Основные файлы программы**

**checker.cpp:**

#include <iostream>

#include <fstream>

#include <yaml-cpp/yaml.h>

#include <unordered\_map>

#include <unordered\_set>

#include <queue>

#include <vector>

#include <sys/wait.h>

#include <thread>

#include <mutex>

#include <string>

#include <signal.h>

#include <unistd.h>

// структура для джоба

struct Job

{

int name;

std::string command;

std::vector<int> depends\_on;

std::string mutex\_name;

};

// достаём инфу, просто собираем джобы в один вектор

std::vector<Job> parser(std::string filename)

{

YAML::Node config = YAML::LoadFile("/Users/anatolii/Desktop/Oper\_Syst/kp/src/new\_test.yaml");

std::vector<Job> jobs;

// Парсинг job'ов

for (const auto &jobNode : config["jobs"])

{

Job job;

job.name = jobNode["name"].as<int>();

job.command = jobNode["command"].as<std::string>();

if (jobNode["mutex"])

job.mutex\_name = jobNode["mutex"].as<std::string>();

// Парсинг зависимостей

const auto dependsOnNode = jobNode["depends\_on"];

if (dependsOnNode.IsSequence())

{

for (const auto &dependency : dependsOnNode)

{

job.depends\_on.push\_back(dependency.as<int>());

}

}

jobs.push\_back(job);

}

return jobs;

}

// создаём дерево, которое показывает переходы от данной вершины к её родителю(ям)

std::vector<std::unordered\_set<int>> create\_tree\_from\_up\_to\_down(const std::vector<Job> &jobs, std::unordered\_set<int> &start\_jobs)

{

std::vector<std::unordered\_set<int>> tree(jobs.size());

for (auto &job : jobs)

{

if (job.depends\_on.empty())

{

start\_jobs.insert(job.name);

}

for (auto &dependence : job.depends\_on)

{

tree[job.name - 1].insert(dependence);

}

}

return tree;

}

// находим конечные джобы

void find\_end\_jobs(const std::vector<std::unordered\_set<int>> &tree, std::unordered\_set<int> &end\_jobs)

{

std::vector<bool> is\_top(tree.size(), true);

for (const auto &children : tree)

{

for (auto &child : children)

{

is\_top[child - 1] = false;

}

}

for (size\_t i = 0; i < is\_top.size(); ++i)

{

if (is\_top[i])

{

end\_jobs.insert(i + 1);

}

}

}

// находим циклы

bool find\_cycle(std::vector<std::unordered\_set<int>> &tree, int u, std::vector<int> &visited)

{

if (visited[u] == 1)

{

return true;

}

if (visited[u] == 2)

{

return false;

}

visited[u] = 1;

bool res = false;

for (int v : tree[u])

{

res |= find\_cycle(tree, v - 1, visited);

}

visited[u] = 2;

return res;

}

// прогоняем функцию нахождения циклов для каждой вершины дерева

bool check\_cycle(std::vector<std::unordered\_set<int>> &tree, std::unordered\_set<int> &starts)

{

std::vector<int> visited(tree.size());

bool res = true;

for (int start : starts)

{

res &= !find\_cycle(tree, start - 1, visited);

}

return res;

}

// обход дерева в глубину и отметка посещённых вершин

void dfs(std::vector<std::unordered\_set<int>> &tree, int u, std::vector<int> &visited)

{

if (visited[u] == 1)

{

return;

}

visited[u] = 1;

bool res = true;

for (int v : tree[u])

{

dfs(tree, v - 1, visited);

}

}

/\* ищет компоненты отделенные от графа

\* @return true, если текущий компонент не связан с графом, false, если связь есть.

\*/

bool find\_components(std::vector<std::unordered\_set<int>> &tree, int u, std::vector<int> &visited)

{

if (visited[u] == 1)

{

return false;

}

if (tree[u].empty())

{

return true;

}

visited[u] = 1;

bool res = true;

for (int v : tree[u])

{

res &= find\_components(tree, v - 1, visited);

}

return res;

}

// проверка графа на одну компоненту связности

bool check\_connectivity(std::vector<std::unordered\_set<int>> &tree, std::unordered\_set<int> &starts)

{

std::vector<int> visited(tree.size());

bool res = true;

size\_t i = 0;

dfs(tree, \*(starts.begin()) - 1, visited);

for (auto itr = std::next(starts.begin(), 1); itr != starts.end(); ++itr)

{

res &= !find\_components(tree, \*itr - 1, visited);

}

return res;

}

void killall\_jobs(std::queue<std::pair<int, pid\_t>> &q)

{

while (!q.empty())

{

auto [id, pid] = q.front();

q.pop();

if (pid != -1)

kill(pid, SIGINT);

}

}

pid\_t create\_job(std::string command)

{

std::stringstream ss{command};

std::vector<std::string> strargs;

std::vector<const char \*> args;

std::string str;

while (ss >> str)

{

strargs.push\_back(str);

args.push\_back(str.c\_str());

}

pid\_t pid = fork();

if (pid == 0)

{

execv(args[0], const\_cast<char \*const \*>(args.data()));

pid = -2;

std::cerr << "exec error" << std::endl;

exit(-2);

}

std::cout << "create " << pid << ": " << command << std::endl;

return pid;

}

int main()

{

std::vector<Job> jobs = parser("/Users/anatolii/Desktop/Oper\_Syst/kp/new\_test.yaml"); // читаем файл

std::unordered\_set<int> start\_jobs, end\_jobs;

// создаём дерево, и сразу находим начальные job'ы

std::vector<std::unordered\_set<int>> tree = create\_tree\_from\_up\_to\_down(jobs, start\_jobs);

// находим конечные job'ы

find\_end\_jobs(tree, end\_jobs);

// проверяем граф по заданию

if (start\_jobs.empty())

{

std::cerr << "Error: Отсутствуют начальные джобы" << std::endl;

return -1;

}

if (end\_jobs.empty())

{

std::cerr << "Error: Отсутствуют завершающие джобы" << std::endl;

return -1;

}

if (!check\_cycle(tree, end\_jobs))

{

std::cerr << "Error: В графе есть циклы" << std::endl;

return -1;

}

if (!check\_connectivity(tree, end\_jobs))

{

std::cerr << "Error: В графе больше одной компоненты связности" << std::endl;

return -1;

}

// читаем мьютексы, указанные у джобов

std::unordered\_map<std::string, bool> mutex\_vals;

std::unordered\_map<int, std::string> mutex\_names;

for (auto &job : jobs)

{

std::string mutex\_name = job.mutex\_name;

if (!mutex\_name.empty())

{

mutex\_names[job.name] = mutex\_name;

mutex\_vals[mutex\_name] = true;

}

}

// начинаем запуск DAG'а

std::queue<std::pair<int, pid\_t>> waitq; // очередь задач

std::mutex qmtx; // мьютекс для доступа к waitq

std::thread wait\_thread([&]()

{

while (!end\_jobs.empty() || !waitq.empty()) {

if (!waitq.empty()) {

pid\_t pid;

int id;

{

std::lock\_guard<std::mutex> lock(qmtx);

id = waitq.front().first;

pid = waitq.front().second;

waitq.pop();

}

// проверка нужно ли ждать этот процесс, -1 значит, что команда была запущена с помощью system()

if (pid != -1) {

int sig;

// Ловим ошибку с помощью waitpid

std::cout << "wait " << pid << std::endl;

waitpid(pid, &sig, 0);

if (WIFSIGNALED(sig)) {

// если ошибка найдена, то завершаем все процессы и переходим к завершению программы

std::cout << "signal is heared from " << id << std::endl;

std::lock\_guard<std::mutex> lock(qmtx);

end\_jobs = {};

killall\_jobs(waitq);

}

}

// Разблокируем мьютекс джобы

{

std::lock\_guard<std::mutex> lock(qmtx);

mutex\_vals[mutex\_names[id - 1]] = true;

}

// добавляем в очередь запуска готовые к выполнению джобы

for (int i = 0; i < tree.size(); ++i) {

if (tree[i].count(id)) {

tree[i].erase(id);

if (tree[i].empty()) {

std::lock\_guard<std::mutex> lock(qmtx);

start\_jobs.insert(i + 1);

}

}

}

end\_jobs.erase(id);

}

} });

while (!end\_jobs.empty())

{

// контейнер для запоминания запущенных джобов, чтобы удалить их из start\_points

std::vector<int> erase\_id;

std::lock\_guard<std::mutex> lock(qmtx);

for (int id : start\_jobs)

{

if ((mutex\_names.count(id) && mutex\_vals[mutex\_names[id]]) || !mutex\_names.count(id))

{

std::string command = jobs[id - 1].command;

if (command.front() != '.' && command.front() != '/')

{

// если команда не является выполнимым файлом, то запускаем её с помощью system()

int success = std::system(command.c\_str());

if (success == -1)

{

killall\_jobs(waitq);

}

erase\_id.push\_back(id);

waitq.push({id, -1});

}

else

{

// иначе делаем fork exec

pid\_t pid = create\_job(command);

// блокируем мьютекс джобы

if (mutex\_names.count(id))

mutex\_vals[mutex\_names[id]] = false;

erase\_id.push\_back(id);

// добавляем процесс в очередь для ожидания

waitq.push({id, pid});

}

}

}

for (int id : erase\_id)

{

start\_jobs.erase(id);

}

}

killall\_jobs(waitq);

wait\_thread.join();

return 0;

}

**Тестирование:**

Для тестирования были написаны ещё два файла: errorjob – процесс завершающийся с ошибкой; timerjob – процесс ожидающий 10 секунд и печатающий время начала и конца работы.

**timerjob.cpp:**

#include <iostream>

#include <chrono>

#include <iomanip>

#include <thread>

using namespace std::chrono\_literals;

int main() {

auto now = std::chrono::system\_clock::now();

std::time\_t now\_time = std::chrono::system\_clock::to\_time\_t(now);

std::tm local\_tm = \*std::localtime(&now\_time);

std::cout << std::put\_time(&local\_tm, "%H:%M:%S - start\n");

std::this\_thread::sleep\_for(10s);

now = std::chrono::system\_clock::now();

now\_time = std::chrono::system\_clock::to\_time\_t(now);

local\_tm = \*std::localtime(&now\_time);

std::cout << std::put\_time(&local\_tm, "%H:%M:%S - end\n");

}

**errorjob.cpp:**

#include <vector>

int main() {

int v[5] = {1, 2, 3, 4, 5};

v[6] = 1;

return 0;

}

**Валидный тест**

anatolii@MacBook-Pro-Anatolii ~/D/O/kp (main)> /Users/anatolii/Desktop/Oper\_Syst/kp/build/src/check

/Users/anatolii/Desktop/Oper\_Syst/kp

Happy New Year

Hello

create 35506: /Users/anatolii/Desktop/OOP\_MAI/OOP\_hometasks/MAI\_OOP\_hometask\_5/build/tests

wait 35506

Running main() from /Users/anatolii/Desktop/OOP\_MAI/OOP\_hometasks/MAI\_OOP\_hometask\_5/googletest/googletest/src/gtest\_main.cc

[==========] Running 6 tests from 1 test suite.

[----------] Global test environment set-up.

[----------] 6 tests from constructors

[ RUN ] constructors.defaulte

[ OK ] constructors.defaulte (13 ms)

[ RUN ] constructors.size\_resize

[ OK ] constructors.size\_resize (13 ms)

[ RUN ] constructors.init\_lst

[ OK ] constructors.init\_lst (13 ms)

[ RUN ] constructors.another\_vect

[ OK ] constructors.another\_vect (22 ms)

[ RUN ] constructors.resize

[ OK ] constructors.resize (13 ms)

[ RUN ] constructors.iters

[ OK ] constructors.iters (13 ms)

[----------] 6 tests from constructors (90 ms total)

[----------] Global test environment tear-down

[==========] 6 tests from 1 test suite ran. (90 ms total)

[ PASSED ] 6 tests.

Running job4

create 35508: /Users/anatolii/Desktop/Oper\_Syst/kp/build/src/timer

wait 35508

12:56:47 - start

12:56:57 - end

anatolii@MacBook-Pro-Anatolii ~/D/O/kp (main)>

**Тест с ошибкой**

anatolii@MacBook-Pro-Anatolii ~/D/O/kp (main)> /Users/anatolii/Desktop/Oper\_Syst/kp/build/src/check

/Users/anatolii/Desktop/Oper\_Syst/kp

Happy New Year

Hello

Running job4

create 35717: /Users/anatolii/Desktop/OOP\_MAI/OOP\_hometasks/MAI\_OOP\_hometask\_5/build/tests

wait 35717

Running main() from /Users/anatolii/Desktop/OOP\_MAI/OOP\_hometasks/MAI\_OOP\_hometask\_5/googletest/googletest/src/gtest\_main.cc

[==========] Running 6 tests from 1 test suite.

[----------] Global test environment set-up.

[----------] 6 tests from constructors

[ RUN ] constructors.defaulte

[ OK ] constructors.defaulte (13 ms)

[ RUN ] constructors.size\_resize

[ OK ] constructors.size\_resize (13 ms)

[ RUN ] constructors.init\_lst

[ OK ] constructors.init\_lst (13 ms)

[ RUN ] constructors.another\_vect

[ OK ] constructors.another\_vect (22 ms)

[ RUN ] constructors.resize

[ OK ] constructors.resize (13 ms)

[ RUN ] constructors.iters

[ OK ] constructors.iters (13 ms)

[----------] 6 tests from constructors (90 ms total)

[----------] Global test environment tear-down

[==========] 6 tests from 1 test suite ran. (90 ms total)

[ PASSED ] 6 tests.

create 35720: /Users/anatolii/Desktop/Oper\_Syst/kp/build/src/error

wait 35720

signal is heared from 6

**Вывод**

Проделав работу, я написал DAF Scheduler и укрепил свои навыки в проектировании программ взаимодействующих с процессами.