Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовой проект по курсу**

**«Операционные системы»**

Студент: Юрков Евгений Юрьевич

Группа: М8О–212Б–22

Вариант: 37

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

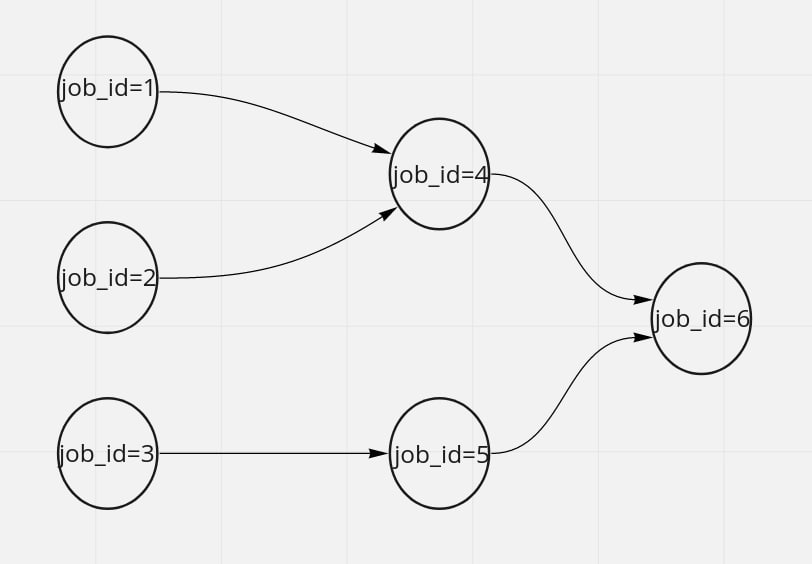
Москва, 2023.

**Постановка задачи**

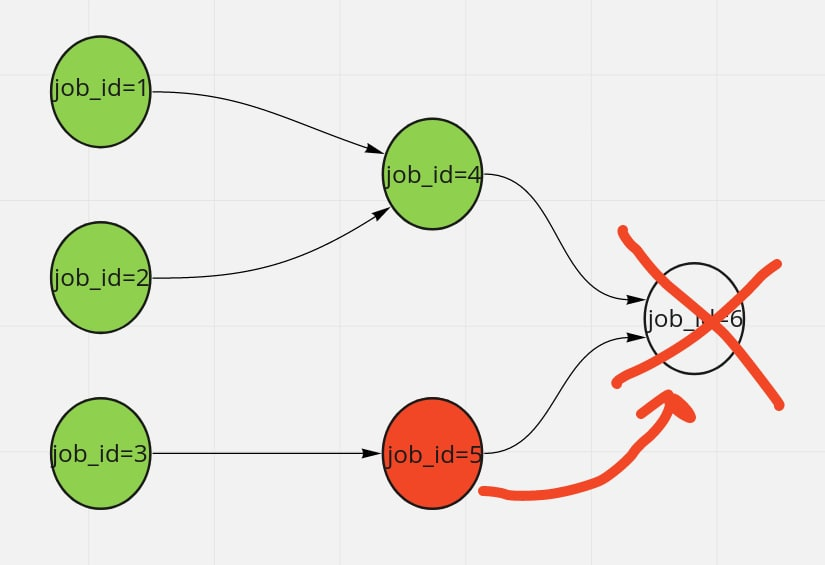
## Задание

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.

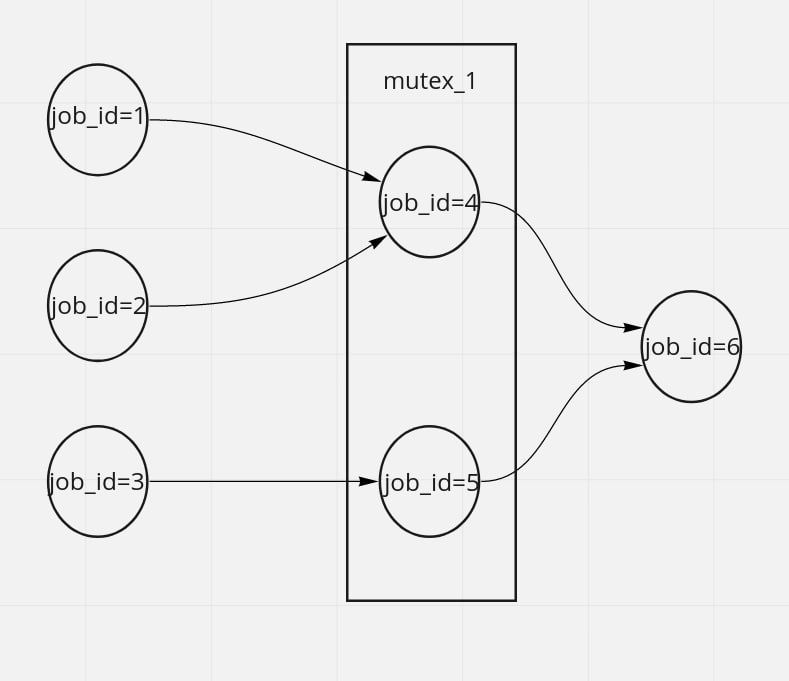


2. При завершении джобы с ошибкой, необходимо прервать выполнение всего DAG’а и всех запущенных джоб.



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:** Ini\Mutex

**Общие сведения о программе**

Основной файл программы - main.cpp. Также используется заголовочные файлы: iostream, thread, mutex, vector, queue, unordered\_map, unordered\_set, sstream, boost/property\_tree/ptree.hpp, boost/property\_tree/ini\_parser.hpp, wait.h.

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 boost/property\_tree.

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\_points – 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**

**main.cpp:**

#include <vector>

#include <queue>

#include <unordered\_map>

#include <unordered\_set>

#include <sstream>

#include <boost/property\_tree/ptree.hpp>

#include <boost/property\_tree/ini\_parser.hpp>

#include <wait.h>

#include <iostream>

#include <thread>

#include <mutex>

using graph = std::vector<std::unordered\_set<int>>;

// Создание графа и множества конечных джобов из property tree

graph create\_graph(boost::property\_tree::ptree& tree, std::unordered\_set<int>& ends) {

graph res(tree.size());

for (auto [id, child] : tree) {

std::stringstream s {child.get<std::string>("next", "")};

int u = std::stoi(id) - 1;

if (s.str().empty()) {

ends.insert(u);

continue;

}

int v;

while (s >> v) {

--v;

res[v].insert(u);

if (s.peek() == ',') {

s.ignore();

}

}

}

return res;

}

/\* поиск цикла в графе

\* @return true если нашел, false если не нашел.

\*/

bool find\_cycle(graph& g, 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 : g[u]) {

res |= find\_cycle(g, v, visited);

}

visited[u] = 2;

return res;

}

// обход дерева в глубину и отметка посещённых вершин

void dfs(graph& g, int u, std::vector<int>& visited) {

if (visited[u] == 1) {

return;

}

visited[u] = 1;

bool res = true;

for (int v : g[u]) {

dfs(g, v, visited);

}

}

/\* ищет компоненты отделенные от графа

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

\*/

bool find\_components(graph& g, int u, std::vector<int>& visited) {

if (visited[u] == 1) {

return false;

}

if (g[u].empty()) {

return true;

}

visited[u] = 1;

bool res = true;

for (int v : g[u]) {

res &= find\_components(g, v, visited);

}

return res;

}

// проверка графа на отсутствие циклов

bool check\_cycle(graph& g, std::unordered\_set<int>& starts) {

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

bool res = true;

for (int start : starts) {

res &= !find\_cycle(g, start, visited);

}

return res;

}

// проверка графа на одну компоненту связности

bool check\_connectivity(graph& g, std::unordered\_set<int>& starts) {

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

bool res = true;

dfs(g, \*(starts.begin()), visited);

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

res &= !find\_components(g, \*itr, visited);

}

return res;

}

/\* создание джобы (процесса)

\* @param Команда путь к выполняемому файлу, который необходимо запустить.

\* @return pid созданного процесса.

\*/

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;

}

/\* убивает все запущенные процессы

\* используется в случае ошибки

\*/

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);

}

}

int main(int argc, char\*\* argv) {

// execl("/bin/sh", "sh", "-c", "ls ../src", nullptr);

// execl("/bin/sh", "sh", "-c", "/home/kruyneg/Programming/OOP/build/lab\_07", nullptr);

std::string dagfilename = argv[1];

// чтение из файла

boost::property\_tree::ptree dag\_ptree;

try {

boost::property\_tree::ini\_parser::read\_ini(dagfilename/\* "/home/kruyneg/Programming/OS/course\_work/src/dag.ini" \*/, dag\_ptree);

}

catch (boost::property\_tree::ini\_parser\_error e) {

std::cerr << e.message() << std::endl;

return 0;

}

#ifdef \_DEBUG

for (auto elem : dag\_ptree) {

std::cout << elem.first << std::endl;

for (auto el : elem.second) {

std::cout << " " << el.first << std::endl;

std:: cout << " " << elem.second.get<std::string>(el.first) << std::endl;

std::cout << std::endl;

}

}

#endif

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

graph g = create\_graph(dag\_ptree, end\_points);

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

if (g[i].empty())

start\_points.insert(i);

}

if (start\_points.empty()) {

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

return -1;

}

if (end\_points.empty()) {

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

return -1;

}

if (!check\_cycle(g, end\_points)) {

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

return -1;

}

if (!check\_connectivity(g, end\_points)) {

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

return -1;

}

// Считываем мьютексы каждой джобы

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

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

for (auto [id, node] : dag\_ptree) {

std::string mutex\_name = node.get<std::string>("mutex", "");

if (!mutex\_name.empty()) {

mutex\_names[std::stoi(id) - 1] = mutex\_name;

mutex\_vals[mutex\_name] = true;

}

}

// начинаем запуск DAG'а

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

std::mutex qmtx;

// этот поток ждёт завершения джоб и ловит их ошибки

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

while (!end\_points.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 + 1 << std::endl;

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

end\_points = {};

killall\_jobs(waitq);

}

}

// Разблокируем мьютекс джобы

{

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

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

}

// добавляем в очередь запуска готовые к выполнению джобы

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

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

g[i].erase(id);

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

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

start\_points.insert(i);

}

}

}

end\_points.erase(id);

}

}

});

while (!end\_points.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\_points) {

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

std::string command = dag\_ptree.get\_child(std::to\_string(id + 1)).get<std::string>("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\_points.erase(id);

}

}

killall\_jobs(waitq);

wait\_thread.join();

}

**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;

}

[kruyneg@matebook14 build]$ cat ./course\_work/tests/error.ini

[1]

command = /home/kruyneg/Programming/OS/build/timerjob

next = 2

[2]

command = /home/kruyneg/Programming/OS/build/errorjob

next = 3, 4

[3]

command = /home/kruyneg/Programming/OS/build/timerjob

[4]

command = /home/kruyneg/Programming/OS/build/timerjob

[kruyneg@matebook14 build]$ ./build/scheduler ./course\_work/tests/error.ini

20:01:05 - start

20:01:15 - end

\*\*\* stack smashing detected \*\*\*: terminated

signal is heared from 2

[kruyneg@matebook14 build]$ cat ./course\_work/tests/timers2.ini

[1]

command = pwd

next = 2

[2]

command = ls

next = 3, 4

[3]

command = /home/kruyneg/Programming/OS/build/timerjob

next = 5

mutex = mt

[4]

command = /home/kruyneg/Programming/OS/build/timerjob

next = 5

mutex = mt

[5]

command = echo DONE!

[kruyneg@matebook14 build]$ ./build/scheduler ./course\_work/tests/timers2.ini

/home/kruyneg/Programming/OS

build course\_work lab1 lab2 lab3 lab4 lab5-7 lab8 output.txt vgcore.35615

create 3953: /home/kruyneg/Programming/OS/build/timerjob

wait 3953

19:55:29 - start

19:55:39 - end

create 3983: /home/kruyneg/Programming/OS/build/timerjob

wait 3983

19:55:39 - start

19:55:49 - end

DONE!

**Вывод**

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