Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовой проект по курсу**

**«Операционные системы»**

Студент: Куценко Максим Дмитриевич

Группа: М8О–212Б–22

Вариант: 39

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

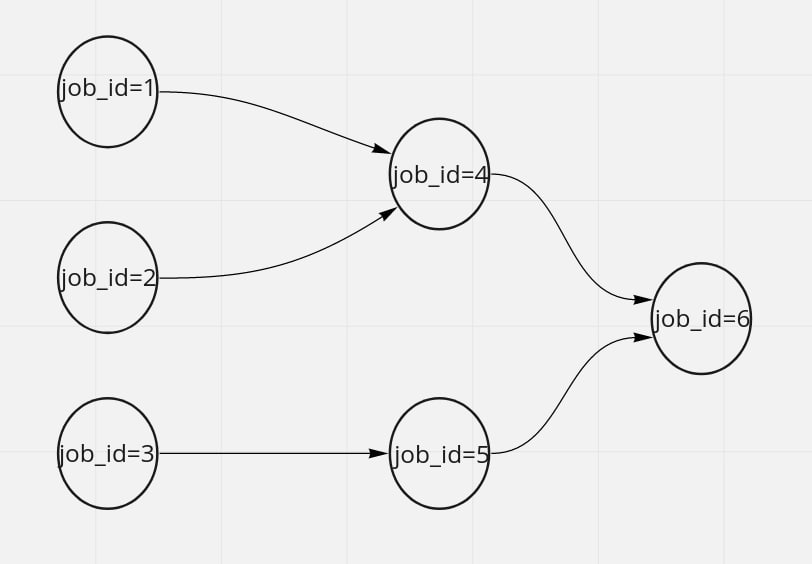
Подпись: \_\_\_\_\_\_\_\_\_\_

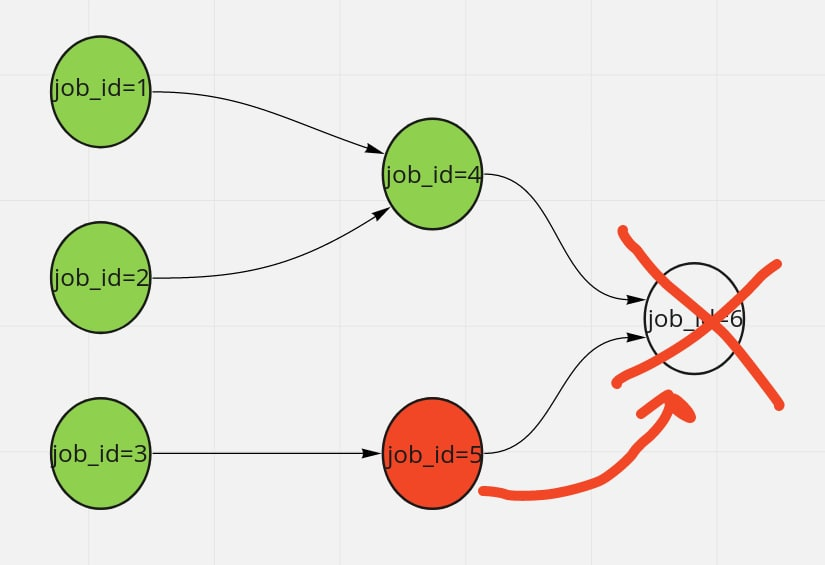
**Москва, 2023.**

**Постановка задачи**

## Задание

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

1. По конфигурационному файлу в формате 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.

\* 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 39:** Ini\Barrier (Ini)

**Общие сведения о программе**

Основной файл программы - main.cpp. Файл с программой джоба — job.cpp. Также используется заголовочные файлы: unistd.h, wait.h, iostream, sstream, vector, string, unordered\_set, unordered\_map, queue, mutex, thread, signal.h, random.

В программе используются следующие системные вызовы:

1. **fork** – создание дочернего процесса.
2. **kill** – отправляет сигнал процессу для завершения.
3. **execv** – выполняет файл.
4. **waitpid** – ожидает завершения процесса и принимает от него сигнал
5. **raise** — используется для остановки дочернего процесса.

**Общий метод и алгоритм решения**.

Для работы с конфигурационным файлом формата Ini используется заголовочный файл SimpleIni.h (<https://github.com/brofield/simpleini>).

Граф мы храним в unordered\_map, где в соответствие каждой вершине ставится вектор с вершинами, в которые есть путь из данной.

Для нахождения циклов используется модификация обхода в глубину, в которой отслеживается, находится ли вершина, в которую совершается переход, в текущем пути.

Для проверки на наличие только одной компоненты связности сначала в граф «достраивается» до неориентированного, после чего происходит поиск в глубину для одной из вершин. Если после этого хотя бы одна вершина не посещена, то компонента связности не одна.

Далее мы проходимся по графу и ищем вершины, в которые не входят и из которых не выходят дуги. На основе результатов составляем вектор стартовых джобов start\_jobs и вектор конечных джобов final\_jobs.

Элементы из вектора стартовых джобов заносятся в очередь на исполнение planned\_jobs. Далее мы для каждой такой джобы вызываем соответственно создание процесса через fork() и исполнение программы через exec(), после чего мы заносим номер джобы и её pid в очередь исполняемых джобов running\_jobs.

Очередь исполняемых джобов просматривается потоком control\_thread, в котором проверяется, что джоба заканчивается нормально. В случае нормального завершения проверяется, находится ли данная джоба в векторе конечных джобов, в зависимости от этого удаляется. В случае какого-либо странного поведения джобы (в примере: остановки через SIGSTOP) мы пишем, что не ней совершилась ошибка, после чего через kill останавливаем все находящиеся в очереди исполняемых джобов процессы по их pid и прерываем выполнение программы.

Если ошибок на протяжении выполнения программы не происходит, мы доходим до всех конечных джобов, выходим из циклов и заканчиваем работу программы.

Число запущенных на данный момент джобов регулируется отдельной переменной active\_amount с максимальным значением max\_jobs. Переменная увеличивается при создании джоба и уменьшается при считывании его статуса через waitpid() с учётом нормального завершения.

**Основные файлы программы**

**CmakeLists.txt**

cmake\_minimum\_required(VERSION 3.10)

project(CW)

set(CMAKE\_CXX\_STANDARD 20)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

include\_directories(${CMAKE\_CURRENT\_SOURCE\_DIR}/include)

add\_executable(main main.cpp)

add\_executable(job job.cpp)

set(SOURCE\_FILE "${CMAKE\_SOURCE\_DIR}/config.ini")

set(DESTINATION\_DIR "${CMAKE\_BINARY\_DIR}")

configure\_file(${SOURCE\_FILE} ${DESTINATION\_DIR} COPYONLY)

**main.cpp:**

#include <unistd.h>

#include <wait.h>

#include <iostream>

#include <sstream>

#include <vector>

#include <string>

#include <unordered\_set>

#include <unordered\_map>

#include <queue>

#include <mutex>

#include <thread>

#include "SimpleIni.h"

bool has\_cycle(const int &node,

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

std::unordered\_map<int, bool> &visited,

std::unordered\_map<int, bool> &path\_st) {

if (!visited[node]) {

visited[node] = true;

path\_st[node] = true;

for (const auto &neighbor: graph.at(node)) {

if (!visited[neighbor] && has\_cycle(neighbor, graph, visited, path\_st)) {

return true;

}

else if (path\_st[neighbor]) {

return true;

}

}

}

path\_st[node] = false;

return false;

}

bool has\_cycle(const std::unordered\_map<int, std::vector<int>> &graph) {

std::unordered\_map<int, bool> visited;

std::unordered\_map<int, bool> path\_st;

for (const auto &node: graph) {

if (has\_cycle(node.first, graph, visited, path\_st)) {

return true;

}

}

return false;

}

void DFS(const int &node,

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

std::unordered\_map<int, bool> &visited) {

visited[node] = true;

for (const auto &neighbor: graph.at(node)) {

if (!visited[neighbor]) {

DFS(neighbor, graph, visited);

}

}

}

bool one\_component(std::unordered\_map<int, std::vector<int>> graph) {

for (const auto &node : graph) {

for (const auto &edge : node.second) {

if (graph.find(edge) == graph.end()) {

std::vector<int> nv;

nv.push\_back(node.first);

graph[edge] = nv;

} else {

graph[edge].push\_back(node.first);

}

}

}

std::unordered\_map<int, bool> visited;

int components = 0;

for (const auto &node: graph) {

if (!visited[node.first]) {

DFS(node.first, graph, visited);

components++;

if (components > 1) {

return false;

}

}

}

return true;

}

std::vector<int> starting(const std::unordered\_map<int, std::vector<int>> &graph) {

std::unordered\_set<int> nodes\_in;

for (const auto &node: graph) {

if (!node.second.empty()) {

for (const auto &edge: node.second) {

nodes\_in.insert(edge);

}

}

}

std::vector<int> start\_nodes;

for (const auto &node: graph) {

if (nodes\_in.find(node.first) == nodes\_in.end()) {

start\_nodes.push\_back(node.first);

}

}

return start\_nodes;

}

std::vector<int> ending(const std::unordered\_map<int, std::vector<int>> &graph) {

std::unordered\_set<int> nodes\_out;

for (const auto &node: graph) {

if (!node.second.empty()) {

nodes\_out.insert(node.first);

}

}

std::vector<int> end\_nodes;

for (const auto &node: graph) {

if (nodes\_out.find(node.first) == nodes\_out.end()) {

end\_nodes.push\_back(node.first);

}

}

return end\_nodes;

}

int main() {

//std::unordered\_map<int, std::string> bars;

//std::unordered\_map<std::string, std::vector<int>> bar\_nodes;

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

CSimpleIniA ini;

ini.LoadFile("config.ini");

CSimpleIniA::TNamesDepend sections;

ini.GetAllSections(sections);

for (const auto &section: sections) {

std::string key = section.pItem;

std::string edge\_str = ini.GetValue(key.c\_str(), "edges", "");

std::istringstream edge\_input(edge\_str);

std::vector<int> edges;

int num;

while (edge\_input >> num) {

edges.push\_back(num);

if (edge\_input.peek() == ',') {

edge\_input.ignore();

}

}

int num\_key = stoi(key);

graph[num\_key] = edges;

/\*std::string bar = ini.GetValue(key.c\_str(), "bars", "");

if (bar!= "") {

bars[num\_key] = bar;

if (bar\_nodes.find(bar) == bar\_nodes.end()) {

std::vector<int> nv;

nv.push\_back(num\_key);

bar\_nodes[bar] = nv;

} else {

bar\_nodes[bar].push\_back(num\_key);

}

}\*/

}

bool stop = false;

if (has\_cycle(graph)) {

std::cout << "Есть цикл!" << std::endl;

stop = true;

}

if (!one\_component(graph)) {

std::cout << "Число компонент связности графа не равно 1!" << std::endl;

stop = true;

}

std::vector<int> start\_jobs = starting(graph);

if (start\_jobs.empty()) {

std::cout << "Нет начальных точек у графа" << std::endl;

stop = true;

}

std::vector<int> final\_jobs = ending(graph);

if (final\_jobs.empty()) {

std::cout << "Нет конечных точек у графа" << std::endl;

stop = true;

}

if (stop) return 1;

int max\_jobs = 3;

/\*std::cout << "Введите максимальное число одновременных процессов: " << std::endl;

std::cin >> max\_jobs;

if (max\_jobs<=0) {

std::cout << "Неверное число!" << std::endl;

return 2;

}\*/

std::unordered\_map<int, int> enter\_step;

for (const auto &node: graph) {

enter\_step[node.first] = 0;

}

for (const auto &node: graph) {

for (const auto &edge: node.second) {

enter\_step[edge]++;

}

}

std::queue<int> planned\_jobs;

for (const auto &elem: start\_jobs) {

planned\_jobs.push(elem);

}

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

std::mutex active\_mutex, end\_mutex;

int active\_amount = 0;

bool finished = false;

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

while (!finished) {

while (!running\_jobs.empty()) {

int id;

pid\_t pid;

{

std::lock\_guard<std::mutex> lock(active\_mutex);

id = running\_jobs.front().first;

pid = running\_jobs.front().second;

running\_jobs.pop();

}

int status;

waitpid(pid, &status, WUNTRACED);

if (!WIFEXITED(status)) {

std::lock\_guard<std::mutex> lock(end\_mutex);

std::cout << "Job " << id << " завершился с ошибкой" << std::endl;

finished = true;

kill(pid,SIGKILL);

while (!running\_jobs.empty()) {

pid = running\_jobs.front().second;

running\_jobs.pop();

kill(pid,SIGKILL);

}

} else {

if (graph[id].size() == 0) {

final\_jobs.erase(std::remove(final\_jobs.begin(), final\_jobs.end(), id), final\_jobs.end());

} else {

for (const auto &edge: graph.at(id)) {

enter\_step[edge]--;

if (enter\_step[edge] == 0) {

planned\_jobs.push(edge);

}

}

}

std::lock\_guard<std::mutex> lock(active\_mutex);

active\_amount--;

}

}

}

});

while (!final\_jobs.empty()) {

if (finished) break;

while (!planned\_jobs.empty() && active\_amount < max\_jobs) {

std::lock\_guard<std::mutex> lock(end\_mutex);

if (finished) break;

int job = planned\_jobs.front();

planned\_jobs.pop();

pid\_t pid = fork();

if (pid == -1) {

return 3;

} else if (pid == 0) {

execl("job", "job", std::to\_string(job).c\_str(), NULL);

exit(-1);

}

{

std::lock\_guard<std::mutex> lock(active\_mutex);

running\_jobs.emplace(job, pid);

active\_amount++;

}

}

}

finished = true;

capture\_thread.join();

}

**Конфигурационный файл config.ini:**

[0]

edges = 10

bars =

[5]

edges = 4,1

bars =

[2]

edges = 4,1

bars =

[4]

edges = 7

bars =

[1]

edges = 6,7,8

bars =

[8]

edges = 0,10

bars =

[6]

edges = 8

bars =

[10]

edges =

bars =

[7]

edges = 0

bars =

[3]

edges = 1,6

bars =

**job.cpp:**

#include <iostream>

#include <unistd.h>

#include <signal.h>

#include <random>

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

std::cout << "Запущена русская рулетка " << argv[1] << std::endl;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<int> distribution(0, 9);

if (distribution(gen) == 0) {

raise(SIGSTOP);

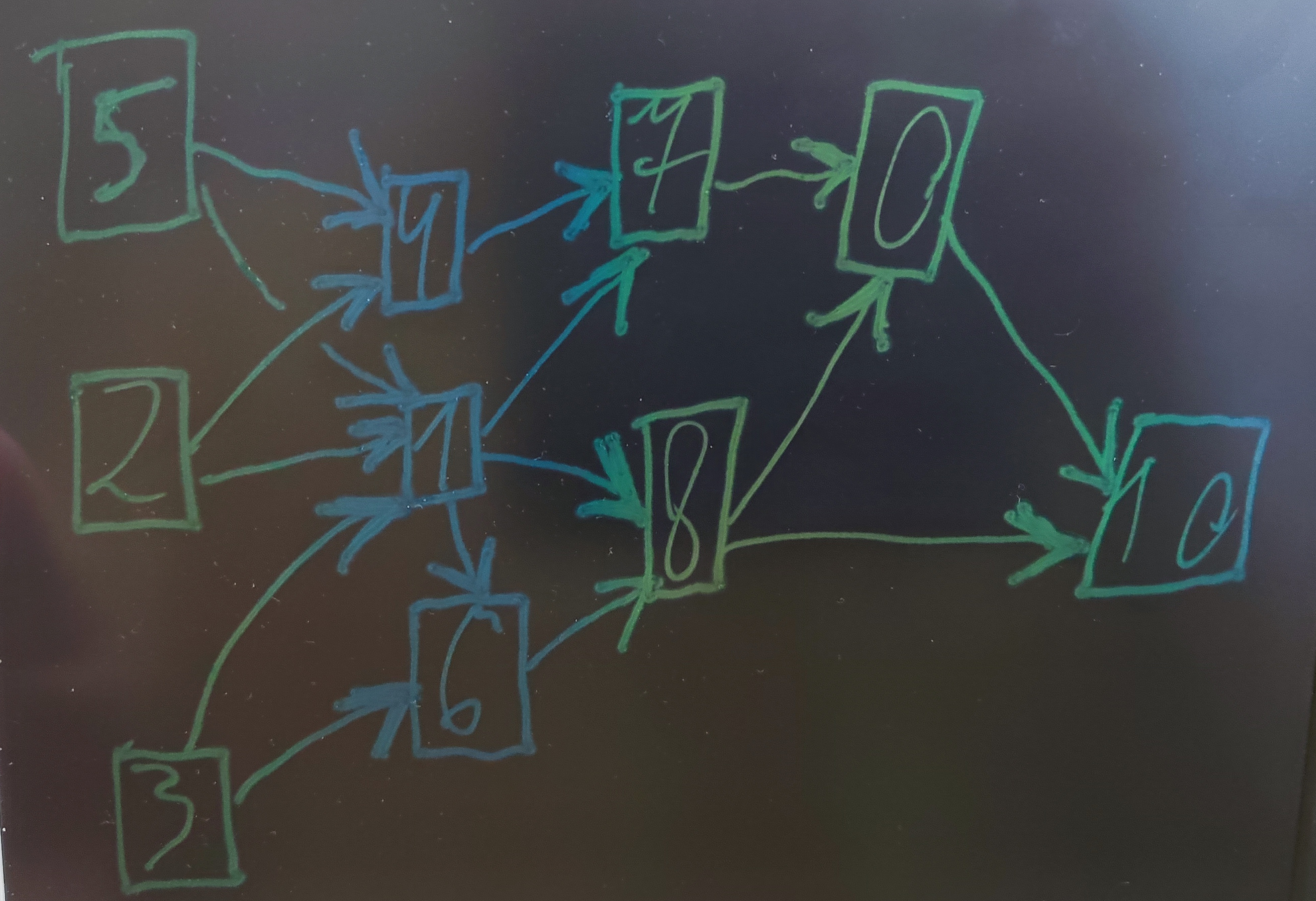
}

**}**

**Примеры работы**

Тестирование производилось при помощи программы “русской рулетки”, которая с указываемым шансом останавливает текущий процесс.

*Структура графа, конфигурация которого указана выше:*

Запущена русская рулетка 3

Запущена русская рулетка 5

Запущена русская рулетка 2

Запущена русская рулетка 4

Запущена русская рулетка 1

Запущена русская рулетка 6

Запущена русская рулетка 7

Запущена русская рулетка 8

Запущена русская рулетка 0

Запущена русская рулетка 10

**[Maxim@HONOR-MB15 src]$**

Запущена русская рулетка 5

Запущена русская рулетка 3

Запущена русская рулетка 2

Запущена русская рулетка 4

Запущена русская рулетка 1

Запущена русская рулетка 6

Запущена русская рулетка 7

Job 7 завершился с ошибкой

**[Maxim@HONOR-MB15 src]$**

**Примеры с ошибками**

[0]

edges = 1

bars =

[1]

edges = 2

bars =

[2]

edges = 0

bars =

Есть цикл!

Нет начальных точек у графа

Нет конечных точек у графа

**[Maxim@HONOR-MB15 src]$**

[0]

edges =

bars =

[1]

edges = 2,3

bars =

[2]

edges = 4

bars =

[3]

edges = 4

bars =

[4]

edges =

bars =

Число компонент связности графа не равно 1!

**[Maxim@HONOR-MB15 src]$**

**Вывод**

Научился работать с файлом Ini, изучил его основную структуру. В процессе выполнения курсового проекта вспомнил алгоритмы на графах, научился запускать процессы по определённой схеме.

Наибольшую сложность представила синхронизация джобов и проверка их на нормальное выполнение. Изначально предполагалось использовать только цикл для всего процесса запуска джобов, но в итоге пришлось добавить поток, а также использовать мьютексы.