Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6 по курсу**

**«Операционные системы»**

Группа: М80-207Б-20

Студент: Михеева К.О.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 29.12.21

Москва, 2021.

# Содержание

1. Постановка задачи.
2. Общие сведения о программе.
3. Общий метод и алгоритм решения.
4. Код программы.
5. Демонстрация работы программы.
6. Вывод.

## Постановка задачи

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить свою работоспособность.

## Общие сведения о программе

Программа состоит из шести файлов:

manager.cpp — исполняемый файл для управляющего узла.

worker.cpp — исполняемый файл для вычислительного узла.

lab\_api.hpp — файл, содержащий сигнатуры функций, которые служат для взаимодействия узлов.

lab\_api.cpp — файл, содержащий реализации функций из файла lab\_api.hpp.

tree.hpp — файл, содержащий класс дерева общего вида.

tree.cpp — файл, содержащий реализации методов класса дерева общего вида из файла tree.hpp.

Связь между узлами осуществляется с помощью сокетов и очередей сообщений ZeroMQ.\

## Общий метод и алгоритм решения

В моем варианте топология подразумевает дерево общего вида, т. е. каждый узел может иметь сколько угодно дочерних узлов. Для получения доступа к нужному дочернему узлу за O(logN), где N — количество дочерних узлов, в каждом узле содержатся три std::map: для сокетов, для потоков, для портов. Для общения между узлами я использовал библиотеку ZeroMQ.

1. Управляющий узел принимает команды, обрабатывает их и пересылает их дочерним вычислительным узлам, а они передают сообщение дальше по дереву, если это необходимо. После работы вычислительных узлов, управляющий узел выводит сообщение о результате вычислений. В случае ошибок, управляющий узел выводит сообщение об этом.

2. Вычислительный узел определяет ему ли передан запрос, если не ему, то он передает его дальше одному из своих дочерних узлов. Если ему, то он выполняет некоторую работу и возвращает сообщение родительскому узлу.

3. Если узел недоступен, то по истечении попытки подключиться к узлу будет выдано сообщение о недоступности узла.

4. При удалении узла, все его дочерние узлы также удаляются.

## Код программы

**Manager.cpp:**

#include <iostream>

#include <string>

#include <map>

#include <set>

#include <cctype>

#include <unistd.h>

#include <sys/types.h>

#include <signal.h>

#include <zmq.hpp>

#include "tree.hpp"

#include "lab\_api.hpp"

int main() {

std::string command; //строка для команд

TTree topology; // дерево

int child\_pid; // id дочернего потока

zmq::context\_t context(1); // 1 означает размер пула потоков для передачи сообщений

std::map<int, zmq::socket\_t> sockets; // словарь для сокетов, где

// первый аргумент - id сокета

// второй аргумент - сокет

std::map<int, int> pids; // словарь для id потоков, где

// первый аргумент - id сокета

// второй аргумент - id потока

std::map<int, int> ports; // словарь для портов, где

// первый аргумент - id сокета

// второй аргумент - порт

while (std::cin >> command) {

std::string res; // строка для результата

int linger = 0; // задержка после закрытия сокета

if (command == "create" || command == "c") {

int new\_id, parent\_id; // переменные для id вычислительного и родительского узлов

std::cin >> new\_id >> parent\_id;

if (parent\_id == -1 && pids.count(new\_id) == 0) { // добавление вычислительного узла к управляющему, т.е. к корню

sockets.emplace(new\_id, zmq::socket\_t(context, ZMQ\_REQ)); // в словарь вставляется запрашивающий сокет с id new\_id

sockets.at(new\_id).setsockopt(ZMQ\_LINGER, &linger, sizeof(linger)); // для нового сокета устанавливается опция zmq\_linger

// для задержки после закрытия сокета, чтобы сообщения не отбрасывались в течение linger

sockets.at(new\_id).setsockopt(ZMQ\_SNDTIMEO, 20); // для нового сокета устанавливается опция zmq\_sndtimeo

// для определения интервала ожидания отправки сообщения, если сообщение за этот период не будет отправлено, то вернется ошибка eagain

int port = accept\_connection(sockets.at(new\_id)); // происходит привязка сокета с id new\_id к некоторому порту

child\_pid = fork();

if (child\_pid == -1) {

std::cout << "Unable to create worker node.\n";

exit(1);

} else if (child\_pid == 0) { // дочерний процесс

create\_server(new\_id, parent\_id, port);

} else {

// заполнение словарей

ports[new\_id] = port;

pids[new\_id] = child\_pid;

//

send\_message(sockets.at(new\_id), "pid");

res = receive\_message(sockets.at(new\_id));

}

} else {

if (topology.Search(new\_id) != nullptr) { // если уже есть узел с id new\_id

std::cout << "Error: Already exists\n";

continue;

}

if (topology.Search(parent\_id) == nullptr) { // если нет родителя с нужным id

std::cout << "Error: Parent not found\n";

continue;

}

auto path = topology.GetPath(parent\_id); // получаем вектор пар, путь до нужной вершины. Первый элемент пары - id узла, второй - направление движения по пути (либо к сыну, либо к брату)

std::vector<int> path\_sons; // вектор id узлов, из которых затем происходит движение по сыну

for (auto it : path) {

if (it.second == TO\_SON) {

path\_sons.push\_back(it.first);

}

}

int id\_on\_path = path\_sons.front();

path\_sons.erase(path\_sons.begin());

std::ostringstream msg\_stream;

msg\_stream << "create " << path\_sons.size();

for (auto it : path\_sons) {

msg\_stream << " " << it;

}

msg\_stream << " " << new\_id;

send\_message(sockets.at(id\_on\_path), msg\_stream.str());

res = receive\_message(sockets.at(id\_on\_path));

}

if (res.substr(0, 2) == "Ok") {

topology.Insert(new\_id, parent\_id);

}

std::cout << res << "\n";

} else if (command == "remove" || command == "r") {

int remove\_id;

std::cin >> remove\_id;

if (pids.size() == 0) { // если нет узлов

std::cout << "Error: Not found\n";

continue;

}

if (pids.count(remove\_id) != 0) { // если удаляемый вычислительный узел является ребенком управляющего

send\_message(sockets.at(remove\_id), "kill");

res = receive\_message(sockets.at(remove\_id));

kill(pids.at(remove\_id), SIGTERM); // SIGTERM - сигнал завершения процесса. Происходит запрос остановки работы процесса.

pids.erase(remove\_id);

sockets.erase(remove\_id);

ports.erase(remove\_id);

if (res.substr(0, 2) == "Ok") {

topology.Remove(remove\_id);

}

std::cout << res << "\n";

continue;

}

// иначе

auto path = topology.GetPath(remove\_id); // получаем путь до удаляемой вершины

if (path.empty()) {

std::cout << "Error: Not found\n";

continue;

}

std::vector<int> path\_sons;

for (auto it : path) {

if (it.second == TO\_SON) {

path\_sons.push\_back(it.first);

}

}

int id\_on\_path = path\_sons.front();

path\_sons.erase(path\_sons.begin());

path\_sons.pop\_back(); // выбрасываем из пути удаляемый элемент

std::ostringstream msg\_stream;

msg\_stream << "remove " << path\_sons.size();

for (auto it : path\_sons) {

msg\_stream << " " << it;

}

msg\_stream << " " << remove\_id;

send\_message(sockets.at(id\_on\_path), msg\_stream.str());

res = receive\_message(sockets.at(id\_on\_path));

if (res.substr(0, 2) == "Ok") {

topology.Remove(remove\_id);

}

std::cout << res << "\n";

} else if (command == "exec") {

int id, value;

std::string name, help\_str, id\_str, value\_str;

bool find = true;

std::cin.ignore(256, ' ');

std::getline(std::cin, help\_str);

int i;

for (i = 0; i < help\_str.size(); i++) { // считывание id

if (std::isdigit(help\_str[i])) {

id\_str += help\_str[i];

} else if (help\_str[i] == ' ') {

if (id\_str.empty()) {

continue;

} else {

id = std::stoi(id\_str);

break;

}

} else { // no digit

break;

}

}

for (; i < help\_str.size(); i++) { // считывание name

if (std::isdigit(help\_str[i]) || std::isalpha(help\_str[i]) || help\_str[i] == '+') {

name += help\_str[i];

} else if (help\_str[i] == ' ') {

if (name.empty()) {

continue;

}

break;

} else { // неподходящий символ

break;

}

}

for (; i < help\_str.size(); i++) { // считывание value

if (std::isdigit(help\_str[i])) {

value\_str += help\_str[i];

} else if (help\_str[i] == ' ') {

if (value\_str.empty()) {

continue;

}

break;

} else { // no digit

break;

}

}

if (!value\_str.empty()) {

value = std::stoi(value\_str);

find = false;

}

auto path = topology.GetPath(id);

if (path.empty()) {

std::cout << "Error:" << id << ": Not found\n";

continue;

}

std::vector<int> path\_sons;

for (auto it : path) {

if (it.second == TO\_SON) {

path\_sons.push\_back(it.first);

}

}

auto next\_id = path\_sons.front();

path\_sons.erase(path\_sons.begin());

std::ostringstream msg\_stream;

msg\_stream << "exec " << path\_sons.size();

for (auto it : path\_sons) {

msg\_stream << " " << it;

}

if (find) { // если поиск в словаре

msg\_stream << " find " << name;

} else { // если вствка в словарь

msg\_stream << " save " << name << " " << value;

}

send\_message(sockets.at(next\_id), msg\_stream.str());

res = receive\_message(sockets.at(next\_id));

std::cout << res << "\n";

} else if (command == "pingall") {

if (pids.empty()) {

std::cout << "Error: no nodes\n";

continue;

}

std::string pre\_result;

for (auto it = sockets.begin(); it != sockets.end(); it++) {

send\_message(it->second, "ping");

pre\_result += receive\_message(it->second);

}

std::istringstream result(pre\_result);

std::set<int> responded\_id;

int temp;

while (result >> temp) {

responded\_id.insert(temp);

}

std::set<int> all\_id = topology.GetNodes();

std::set<int> result\_set;

auto it\_resp = responded\_id.begin();

auto it\_all = all\_id.begin();

for (int i = 0; i < all\_id.size(); i++) {

if (\*it\_resp == \*it\_all) {

it\_resp++;

it\_all++;

} else {

result\_set.insert(\*it\_all);

it\_all++;

}

}

std::cout << "Ok: ";

if (result\_set.empty()) {

std::cout << "-1\n";

continue;

}

for (auto it : result\_set) {

std::cout << it << ";";

}

std::cout << "\n";

} else if (command == "print") {

std::cout << "---------------------------------\n";

topology.Print();

std::cout << "---------------------------------\n";

} else if (command == "exit" || command == "quit" || command == "e" || command == "q") {

for (auto it = sockets.begin(); it != sockets.end(); it++) {

send\_message(it->second, "kill");

receive\_message(it->second);

kill(pids.at(it->first), SIGTERM);

}

break;

} else {

std::cout << "Error: incorrect input\n";

}

}

return 0;

}

**worker.cpp:**

#include <iostream>

#include <zmq.hpp>

#include <string>

#include <map>

#include <unistd.h>

#include <sys/types.h>

#include <signal.h>

#include <sstream>

#include <stdexcept>

#include "lab\_api.hpp"

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

/\* argv[1] - id, argv[2] - parent\_id, argv[3] - parent\_port\*/

int id = std::stoi(argv[1]);

int parent\_id = std::stoi(argv[2]);

int parent\_port = std::stoi(argv[3]);

zmq::context\_t context(3); // у Димы написано "сыну, брату и отцу", т.е., скорее всего, для каждого узла будет свой поток обработки сообщений

zmq::socket\_t parent\_socket(context, ZMQ\_REP); // создаем сокет, который отправляет ответы на запросы

parent\_socket.connect(get\_port\_name(parent\_port)); // подключаем созданный сокет к тому же порту, что и переданный по аргументу

std::map<int, zmq::socket\_t> sockets; // словарь для сокетов, где

// первый аргумент - id сокета

// второй аргумент - сокет

std::map<int, int> pids; // словарь для id потоков, где

// первый аргумент - id сокета

// второй аргумент - id потока

std::map<int, int> ports; // словарь для портов, где

// первый аргумент - id сокета

// второй аргумент - порт

std::map<std::string, int> local\_dict; // локальный целочисленный словарь

while (true) {

int linger = 0; // задержка для закрытия сокета

std::string request = receive\_message(parent\_socket); // получение сообщение

std::istringstream command\_stream(request); // создается строчный поток для удобной обработки сообщения

std::string command;

command\_stream >> command; // считывается команда

if (command == "pid") {

std::string answer = "Ok:" + std::to\_string(getpid());

send\_message(parent\_socket, answer);

} else if (command == "create") {

int size, new\_id, port;

command\_stream >> size;

std::vector<int> path(size);

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

command\_stream >> path[i];

}

command\_stream >> new\_id;

if (path.empty()) {

sockets.emplace(new\_id, zmq::socket\_t(context, ZMQ\_REQ));

sockets.at(new\_id).setsockopt(ZMQ\_SNDTIMEO, 20);

sockets.at(new\_id).setsockopt(ZMQ\_LINGER, &linger, sizeof(linger));

port = accept\_connection(sockets.at(new\_id));

int pid = fork();

if (pid == -1) {

send\_message(parent\_socket, "Can't fork");

continue;

} else if (pid == 0) {

create\_server(new\_id, id, port);

} else {

ports[new\_id] = port;

pids[new\_id] = pid;

send\_message(sockets.at(new\_id), "pid");

send\_message(parent\_socket, receive\_message(sockets.at(new\_id)));

}

} else {

int next\_id = path.front();

path.erase(path.begin());

std::ostringstream msg\_stream;

msg\_stream << "create " << path.size();

for (auto it : path) {

msg\_stream << " " << it;

}

msg\_stream << " " << new\_id;

send\_message(sockets.at(next\_id), msg\_stream.str());

send\_message(parent\_socket, receive\_message(sockets.at(next\_id)));

}

} else if (command == "kill") {

for (auto it = sockets.begin(); it != sockets.end(); it++) {

send\_message(it->second, "kill");

receive\_message(it->second);

kill(pids.at(it->first), SIGTERM);

}

send\_message(parent\_socket, "Ok");

} else if (command == "remove") {

int size, remove\_id;

command\_stream >> size;

std::vector<int> path(size);

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

command\_stream >> path[i];

}

command\_stream >> remove\_id;

if (path.empty()) { // если мы находимся в родительском узле для удаляемого

if (sockets.count(remove\_id) == 0) {

send\_message(parent\_socket, "Error: Not found\n");

continue;

}

send\_message(sockets.at(remove\_id), "kill");

std::string res = receive\_message(sockets.at(remove\_id));

kill(pids.at(remove\_id), SIGTERM);

pids.erase(remove\_id);

sockets.erase(remove\_id);

ports.erase(remove\_id);

send\_message(parent\_socket, res);

} else { // если мы находимся на пути к удаляемому

auto next\_id = path.front();

path.erase(path.begin());

std::ostringstream msg\_stream;

msg\_stream << "remove " << path.size();

for (auto it : path) {

msg\_stream << " " << it;

}

msg\_stream << " " << remove\_id;

send\_message(sockets.at(next\_id), msg\_stream.str());

send\_message(parent\_socket, receive\_message(sockets.at(next\_id)));

}

} else if (command == "exec") {

int size;

command\_stream >> size;

std::vector<int> path(size);

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

command\_stream >> path[i];

}

if (size == 0) { // находимся в нужном узле

std::string find\_or\_save, name;

command\_stream >> find\_or\_save >> name;

if (find\_or\_save == "find") {

auto search = local\_dict.find(name);

std::ostringstream msg\_stream;

if (search != local\_dict.end()) { // если элемент найден

msg\_stream << "Ok:" << id << ": " << search->second;

} else {

msg\_stream << "Ok:" << id << ": \'" << name << "\' not found";

}

send\_message(parent\_socket, msg\_stream.str());

} else if (find\_or\_save == "save") {

int value;

command\_stream >> value;

auto search = local\_dict.find(name);

if (search != local\_dict.end()) { // если элемент найден

search->second = value;

} else {

local\_dict.insert(std::make\_pair(name, value));

}

std::string result = "Ok:" + std::to\_string(value);

send\_message(parent\_socket, result);

}

} else {

std::string find\_or\_save, name;

command\_stream >> find\_or\_save >> name;

auto next\_id = path.front();

path.erase(path.begin());

std::ostringstream msg\_stream;

msg\_stream << "exec " << path.size();

for (auto it : path) {

msg\_stream << " " << it;

}

if (find\_or\_save == "find") {

msg\_stream << " find " << name;

} else if (find\_or\_save == "save") {

std::string value;

command\_stream >> value;

msg\_stream << " save " << name << " " << value;

}

send\_message(sockets.at(next\_id), msg\_stream.str());

send\_message(parent\_socket, receive\_message(sockets.at(next\_id)));

}

} else if (command == "ping") {

std::ostringstream stream;

for (auto it = sockets.begin(); it != sockets.end(); it++) {

send\_message(it->second, "ping");

std::string str = receive\_message(it->second);

if (str == "Error: Node is not available") {

continue;

}

stream << str;

}

stream << id << " ";

send\_message(parent\_socket, stream.str());

}

}

return 0;

}

**tree.hpp:**

#ifndef TREE\_HPP

#define TREE\_HPP 1

#include <memory> // shared\_ptr

#include <utility> // pair

#include <vector>

#include <set>

enum Direction {

TO\_SON, TO\_BRO

};

struct TNode {

int id\_{0};

std::shared\_ptr<TNode> son\_{nullptr};

std::shared\_ptr<TNode> brother\_{nullptr};

TNode(int id, const std::shared\_ptr<TNode> &son, const std::shared\_ptr<TNode> &brother) : id\_{id}, son\_{son}, brother\_{brother} {}

TNode(std::shared\_ptr<TNode> node) : id\_{node->id\_}, son\_{node->son\_}, brother\_{node->brother\_} {}

};

class TTree {

public:

using pair\_of\_ptr = std::pair<std::shared\_ptr<TNode>, std::shared\_ptr<TNode>>;

private:

std::shared\_ptr<TNode> root\_{nullptr};

std::shared\_ptr<TNode> create\_node(int id);

std::shared\_ptr<TNode> depth\_search(std::shared\_ptr<TNode> node, int id);

void help\_get\_nodes(std::set<int> &s, std::shared\_ptr<TNode> node);

std::shared\_ptr<TNode> breadth\_search(std::shared\_ptr<TNode> node, int id);

pair\_of\_ptr remove\_help(std::shared\_ptr<TNode> iter\_node, std::shared\_ptr<TNode> prev\_node, int id);

void print\_help(std::shared\_ptr<TNode> node, int right, int down);

bool help\_get\_path(std::shared\_ptr<TNode> node, std::vector<std::pair<int, Direction>> &path, int id);

public:

TTree() = default;

TTree(const TTree &tree) : root\_{tree.root\_} {}

TTree(TTree &&tree) : root\_{std::move(tree.root\_)} {}

std::shared\_ptr<TNode> Search(int id);

bool Insert(int id, int parent\_id);

bool Remove(int id);

std::vector<std::pair<int, Direction>> GetPath(int id);

std::set<int> GetNodes();

void Print();

};

#endif // TREE\_HPP

**tree.cpp:**

#include <iostream>

#include <memory> // shared\_ptr

#include <utility> // pair

#include "tree.hpp"

std::shared\_ptr<TNode> TTree::create\_node(int id) {

return std::make\_shared<TNode>(id, nullptr, nullptr);

}

std::shared\_ptr<TNode> TTree::depth\_search(std::shared\_ptr<TNode> node, int id) {

std::shared\_ptr<TNode> result{nullptr};

if (node != nullptr) {

if (id == node->id\_) {

return node;

}

result = depth\_search(node->son\_, id);

if (result == nullptr) {

result = depth\_search(node->brother\_, id);

}

}

return result;

}

void TTree::help\_get\_nodes(std::set<int> &s, std::shared\_ptr<TNode> node) {

if (node != nullptr) {

s.insert(node->id\_);

help\_get\_nodes(s, node->son\_);

help\_get\_nodes(s, node->brother\_);

}

}

std::shared\_ptr<TNode> TTree::breadth\_search(std::shared\_ptr<TNode> node, int id) {

std::shared\_ptr<TNode> result{nullptr};

if (node != nullptr) {

if (id == node->id\_) {

return node;

}

result = breadth\_search(node->brother\_, id);

if (result == nullptr) {

result = breadth\_search(node->son\_, id);

}

}

return result;

}

TTree::pair\_of\_ptr TTree::remove\_help(std::shared\_ptr<TNode> iter\_node, std::shared\_ptr<TNode> prev\_node, int id) {

TTree::pair\_of\_ptr result = std::make\_pair<std::shared\_ptr<TNode>, std::shared\_ptr<TNode>>(nullptr, nullptr);

if (iter\_node != nullptr) {

if (id == iter\_node->id\_) {

result.first = iter\_node;

result.second = prev\_node;

return result;

}

result = remove\_help(iter\_node->son\_, iter\_node, id);

if (result.first == nullptr && result.second == nullptr) {

result = remove\_help(iter\_node->brother\_, iter\_node, id);

}

}

return result;

}

void TTree::print\_help(std::shared\_ptr<TNode> node, int right, int down) {

if (node != nullptr) {

for (int i = 0; i < down; i++) {

std::cout << "\n";

}

for (int i = 0; i < right; i++) {

std::cout << " ";

}

std::cout << node->id\_;

print\_help(node->son\_, right + 1, 1);

print\_help(node->brother\_, right, 1);

}

}

bool TTree::help\_get\_path(std::shared\_ptr<TNode> iter, std::vector<std::pair<int, Direction>> &path, int id) {

if (iter == nullptr) {

return false;

}

if (iter->id\_ == id && iter->son\_ == nullptr) {

auto p = std::make\_pair(iter->id\_, TO\_SON);

path.push\_back(p);

return true;

}

if (iter->id\_ == id && iter->son\_ != nullptr) {

auto p = std::make\_pair(iter->id\_, TO\_SON);

path.push\_back(p);

iter = iter->son\_;

p = std::make\_pair(iter->id\_, TO\_BRO);

path.push\_back(p);

while (iter->brother\_ != nullptr) {

iter = iter->brother\_;

auto p = std::make\_pair(iter->id\_, TO\_BRO);

path.push\_back(p);

}

return true;

}

auto p = std::make\_pair(iter->id\_, TO\_SON);

path.push\_back(p);

std::shared\_ptr<TNode> iter\_son = iter;

while (iter\_son->son\_ != nullptr) {

iter\_son = iter\_son->son\_;

if (help\_get\_path(iter\_son, path, id)) {

return true;

}

}

path.pop\_back();

p = std::make\_pair(iter->id\_, TO\_BRO);

path.push\_back(p);

std::shared\_ptr<TNode> iter\_bro = iter;

while (iter\_bro->brother\_ != nullptr) {

iter\_bro = iter\_bro->brother\_;

if (help\_get\_path(iter\_bro, path, id)) {

return true;

}

}

path.pop\_back();

return false;

}

std::shared\_ptr<TNode> TTree::Search(int id) {

return depth\_search(root\_, id);

}

bool TTree::Insert(int id, int parent\_id) {

std::shared\_ptr<TNode> new\_node = create\_node(id);

std::shared\_ptr<TNode> parent\_node = depth\_search(root\_, parent\_id);

if (root\_ == nullptr) {

root\_ = new\_node;

return true;

}

if (root\_ != nullptr && parent\_id == -1) {

std::shared\_ptr<TNode> it = root\_;

while (it->brother\_ != nullptr) {

it = it->brother\_;

}

it->brother\_ = new\_node;

}

if (parent\_node == nullptr) {

return false;

}

if (parent\_node->son\_ == nullptr) {

parent\_node->son\_ = new\_node;

return true;

}

std::shared\_ptr<TNode> iter\_node = parent\_node->son\_;

while (iter\_node->brother\_ != nullptr) {

iter\_node = iter\_node->brother\_;

}

iter\_node->brother\_ = new\_node;

return true;

}

bool TTree::Remove(int id) {

TTree::pair\_of\_ptr need\_and\_prev = remove\_help(root\_, nullptr, id);

if (need\_and\_prev.first == nullptr && need\_and\_prev.second == nullptr) { // id not found

return false;

}

if (need\_and\_prev.first == root\_) {

root\_ = nullptr;

return true;

}

if (need\_and\_prev.second->son\_ == need\_and\_prev.first) { // если предыдущий является отцом

need\_and\_prev.second->son\_ = need\_and\_prev.first->brother\_;

need\_and\_prev.first = nullptr;

return true;

}

if (need\_and\_prev.second->brother\_ == need\_and\_prev.first) { // если предыдущий является братом

need\_and\_prev.second->brother\_ = need\_and\_prev.first->brother\_;

need\_and\_prev.first = nullptr;

return true;

}

}

std::vector<std::pair<int, Direction>> TTree::GetPath(int id) {

std::vector<std::pair<int, Direction>> result;

help\_get\_path(root\_, result, id);

return result;

}

std::set<int> TTree::GetNodes() {

std::set<int> s;

help\_get\_nodes(s, root\_);

return s;

}

void TTree::Print() {

print\_help(root\_, 0, 0);

std::cout << "\n";

}

**lab\_api.hpp:**

#ifndef LAB\_API\_HPP

#define LAB\_API\_HPP 1

#include <string>

#include "zmq.hpp"

void create\_server(int id, int parent\_id, int port);

bool send\_message(zmq::socket\_t &socket, const std::string &message\_string);

std::string receive\_message(zmq::socket\_t &socket);

std::string get\_port\_name(int port);

int accept\_connection(zmq::socket\_t &socket);

#endif // LAB\_API\_HPP

**lab\_api.cpp:**

#include "lab\_api.hpp"

#include <string.h> // for strdup

#include <unistd.h> // for execv

void create\_server(int id, int parent\_id, int port) {

char \*arg1 = strdup((std::to\_string(id)).c\_str()); // выделяется память под строку и эта строка копируется в выделенную память

char \*arg2 = strdup((std::to\_string(parent\_id)).c\_str());

char \*arg3 = strdup((std::to\_string(port)).c\_str());

char \*args[] = {"./worker", arg1, arg2, arg3, NULL};

execv("./worker", args);

free(arg1);

free(arg2);

free(arg3);

}

bool send\_message(zmq::socket\_t &socket, const std::string &message\_string) {

zmq::message\_t message(message\_string.size());

memcpy(message.data(), message\_string.c\_str(), message\_string.size());

return socket.send(message);

}

std::string receive\_message(zmq::socket\_t &socket) {

zmq::message\_t message;

bool result\_recv;

try {

result\_recv = socket.recv(&message); // true, если сообщение получено

// false, если сообщение не получено

} catch (...) { // false, если ошибка

result\_recv = false;

}

std::string message\_str(static\_cast<char \*>(message.data()), message.size());

if (message\_str.empty() || !result\_recv) {

return "Error: Node is not available";

}

return message\_str;

}

std::string get\_port\_name(int port) {

return "tcp://127.0.0.1:" + std::to\_string(port);

}

int accept\_connection(zmq::socket\_t &socket) {

int port = 4040;

while (true) {

try {

socket.bind(get\_port\_name(port)); // bind, т.к. привязывается к долгоживущему порту

break;

} catch (...) {

port++;

}

}

return port;

}

## Демонстрация программы

manager

c 1 -1

Ok:12263

c 2 -1

Ok:12268

c 3 -1

Ok:12274

c 4 -1

Ok:12279

c 5 3

Ok:12285

exec 5 h 10

Ok:10

pingall

Ok: -1

e

./manager

c 1 -1

Ok:12967

c 2 -1

Ok:12974

c 3 1

Ok:12979

c 4 1

Ok:12984

c 5 1

Ok:12990

c 6 3

Ok:12996

c 7 3

Ok:13005

c 8 6

Ok:13010

c 9 5

Ok:13016

print

---------------------------------

1

3

6

8

7

4

5

9

2

---------------------------------

exec 8 myvar 10

Ok:10

exec 8 myvar

Ok:8: 10

exec 8 myvar 15

Ok:15

exec 8 myvar

Ok:8: 15

exec 9 myvar

Ok:9: 'myvar' not found

exec 11

Ok:9: '' not found

exec 11 myvar

Error:11: Not found

print

---------------------------------

1

3

6

8

7

4

5

9

2

---------------------------------

remove 3

Ok

print

---------------------------------

1

4

5

9

2

---------------------------------

pingall

Ok: -1

## Вывод

В данной лабораторной работе мы реализовать распределенную систему по асинхронной обработке запросов. Эта работа оказалась довольно интересной, многому научила, особенно тщательному продумыванию архитектуры программы наперед. Также можно было закрепить ранее полученные навыки работ с процессами, потоками и их синхронизацией.