Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6-8 по курсу**

**«Операционные системы»**

Студент: Старцев Иван Романович

Группа: М8О-201Б-21

Преподаватель: Миронов Евгений Сергеевич

Вариант: 35

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2022

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

https://github.com/IvanTvardovsky/OS-labs

**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 4 файлов: main.cpp (получает команды от пользователя и отправляет их в вычислительный узел), client.cpp (получает эти команды и выполняет их), tree.cpp и tree.h (реализация бинарного дерева поиска).

**Общий метод и алгоритм решения:**

* create id - вставка вычислительного узла в бинарное дерево
* exec id subcommand - отправка подкоманды вычислительному узлу
* kill id - удаление вычислительного узла и всех его дочерних узлов из дерева
* ping id – проверка доступности конкретного узла

**Исходный код:**

**main.cpp**

#include <iostream>

#include <unistd.h>

#include <string>

#include <vector>

#include <sstream>

#include <signal.h>

#include <cassert>

#include "../include/tree.h"

#include "zmq.hpp"

using namespace std;

const int TIMER = 500;

const int DEFAULT\_PORT = 5050;

int n = 2;

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

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

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

return socket.send(message);

}

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

zmq::message\_t message;

bool ok = false;

try {

ok = socket.recv(&message);

}

catch (...) {

ok = false;

}

string recieved\_message(static\_cast<char\*>(message.data()), message.size());

if (recieved\_message.empty() || !ok) {

return "Root is dead";

}

return recieved\_message;

}

void create\_node(int id, int port) {

char\* arg0 = strdup("./client");

char\* arg1 = strdup((to\_string(id)).c\_str());

char\* arg2 = strdup((to\_string(port)).c\_str());

char\* args[] = {arg0, arg1, arg2, NULL};

execv("./client", args);

}

string get\_port\_name(const int port) {

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

}

bool is\_number(string val) {

try {

int tmp = stoi(val);

return true;

}

catch(exception& e) {

cout << "Error: " << e.what() << "\n";

return false;

}

}

int main() {

Tree T;

string command;

int child\_pid = 0;

int child\_id = 0;

zmq::context\_t context(1);

zmq::socket\_t main\_socket(context, ZMQ\_REQ);

cout << "Commands:\n";

cout << "1. create (id)\n";

cout << "2. exec (id) (text\_string, pattern\_string)\n";

cout << "3. kill (id)\n";

cout << "4. ping (id)\n";

cout << "5. exit\n" << endl;

while (true) {

cin >> command;

if (command == "create") {

n++;

size\_t node\_id = 0;

string str = "";

string result = "";

cin >> str;

if (!is\_number(str)) {

continue;

}

node\_id = stoi(str);

if (child\_pid == 0) {

main\_socket.bind(get\_port\_name(DEFAULT\_PORT + node\_id));

main\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

main\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

child\_pid = fork();

if (child\_pid == -1) {

cout << "Unable to create first worker node\n";

child\_pid = 0;

exit(1);

} else if (child\_pid == 0) {

create\_node(node\_id, DEFAULT\_PORT + node\_id);

} else {

child\_id = node\_id;

main\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

main\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

send\_message(main\_socket, "pid");

result = receive\_message(main\_socket);

}

} else {

main\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

main\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

string msg\_s = "create " + to\_string(node\_id);

send\_message(main\_socket, msg\_s);

result = receive\_message(main\_socket);

}

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

T.push(node\_id);

}

cout << result << "\n";

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

int node\_id = 0;

string str = "";

cin >> str;

if (!is\_number(str)) {

continue;

}

node\_id = stoi(str);

if (child\_pid == 0) {

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

continue;

}

if (node\_id == child\_id) {

kill(child\_pid, SIGTERM);

kill(child\_pid, SIGKILL);

child\_id = 0;

child\_pid = 0;

T.kill(node\_id);

cout << "Ok\n";

continue;

}

string message\_string = "kill " + to\_string(node\_id);

send\_message(main\_socket, message\_string);

string recieved\_message;

recieved\_message = receive\_message(main\_socket);

if (recieved\_message.substr(0, min<int>(recieved\_message.size(), 2)) == "Ok") {

T.kill(node\_id);

}

cout << recieved\_message << "\n";

}

else if (command == "exec") {

string id\_str = "";

string text\_string = "";

string pattern\_string = "";

int id = 0;

cin >> id\_str >> text\_string >> pattern\_string;

if (!is\_number(id\_str)) {

continue;

}

id = stoi(id\_str);

string message\_string = "exec " + to\_string(id) + " " + text\_string + " " + pattern\_string;

send\_message(main\_socket, message\_string);

string recieved\_message = receive\_message(main\_socket);

cout << recieved\_message << "\n";

}

else if (command == "ping") {

string id\_str = "";

int id = 0;

cin >> id\_str;

if (!is\_number(id\_str)) {

continue;

}

id = stoi(id\_str);

string message\_string = "ping " + to\_string(id);

send\_message(main\_socket, message\_string);

string recieved\_message = receive\_message(main\_socket);

cout << recieved\_message << "\n";

}

else if (command == "exit") {

int n = system("killall client");

break;

}

}

return 0;

}

**client.cpp**

#include <iostream>

#include <unistd.h>

#include <string>

#include <sstream>

#include <exception>

#include <signal.h>

#include "zmq.hpp"

using namespace std;

const int TIMER = 500;

const int DEFAULT\_PORT = 5050;

int n = 2;

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

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

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

return socket.send(message);

}

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

zmq::message\_t message;

bool ok = false;

try {

ok = socket.recv(&message);

}

catch (...) {

ok = false;

}

string recieved\_message(static\_cast<char\*>(message.data()), message.size());

if (recieved\_message.empty() || !ok) {

return "";

}

return recieved\_message;

}

void create\_node(int id, int port) {

char\* arg0 = strdup("./client");

char\* arg1 = strdup((to\_string(id)).c\_str());

char\* arg2 = strdup((to\_string(port)).c\_str());

char\* args[] = {arg0, arg1, arg2, NULL};

execv("./client", args);

}

string get\_port\_name(const int port) {

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

}

void real\_create(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& create\_id, int& id, int& pid) {

if (pid == -1) {

send\_message(parent\_socket, "Error: Cannot fork");

pid = 0;

}

else if (pid == 0) {

create\_node(create\_id,DEFAULT\_PORT + create\_id);

}

else {

id = create\_id;

send\_message(socket, "pid");

send\_message(parent\_socket, receive\_message(socket));

}

}

void real\_kill(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& delete\_id, int& id, int& pid, string& request\_string) {

if (id == 0) {

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

}

else if (id == delete\_id) {

send\_message(socket, "kill\_children");

receive\_message(socket);

kill(pid, SIGTERM);

kill(pid, SIGKILL);

id = 0;

pid = 0;

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

}

else {

send\_message(socket, request\_string);

send\_message(parent\_socket, receive\_message(socket));

}

}

void real\_exec(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& id, int& pid, string& request\_string) {

if (pid == 0) {

string receive\_message = "Error:" + to\_string(id);

receive\_message += ": Not found";

send\_message(parent\_socket, receive\_message);

}

else {

send\_message(socket, request\_string);

string str = receive\_message(socket);

if (str == "") str = "Error: Node is unavailable";

send\_message(parent\_socket, str);

}

}

void real\_ping(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& id, int& pid, string& request\_string) {

if (pid == 0) {

string receive\_message = "Error:" + to\_string(id);

receive\_message += ": Not found";

send\_message(parent\_socket, receive\_message);

}

else {

send\_message(socket, request\_string);

string str = receive\_message(socket);

if (str == "") str = "Ok: 0";

send\_message(parent\_socket, str);

}

}

void exec(istringstream& command\_stream, zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket,

zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid, int& id, string& request\_string) {

string text\_string, pattern\_string;

int exec\_id;

command\_stream >> exec\_id;

if (exec\_id == id) {

command\_stream >> text\_string;

command\_stream >> pattern\_string;

string receive\_message = "";

string answer = "";

int index = 0;

while ((index = text\_string.find(pattern\_string, index)) != string::npos) {

answer += to\_string(index) + ";";

index += pattern\_string.length();

}

if (!answer.empty()) answer.pop\_back();

receive\_message = "Ok:" + to\_string(id) + ":";

if (!answer.empty()) {

receive\_message += answer;

} else {

receive\_message += "-1";

}

send\_message(parent\_socket, receive\_message);

} else if (exec\_id < id) {

real\_exec(parent\_socket, left\_socket, exec\_id, left\_pid, request\_string);

} else {

real\_exec(parent\_socket, right\_socket, exec\_id, right\_pid, request\_string);

}

}

void ping(istringstream& command\_stream, zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket,

zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid, int& id, string& request\_string) {

int ping\_id;

string receive\_message;

command\_stream >> ping\_id;

if (ping\_id == id) {

receive\_message = "Ok: 1";

send\_message(parent\_socket, receive\_message);

} else if (ping\_id < id) {

real\_ping(parent\_socket, left\_socket, ping\_id, left\_pid, request\_string);

}

else {

real\_ping(parent\_socket, right\_socket, ping\_id, right\_pid, request\_string);

}

}

void kill\_children(zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket,

zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid) {

if (left\_pid == 0 && right\_pid == 0) {

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

} else {

if (left\_pid != 0) {

send\_message(left\_socket, "kill\_children");

receive\_message(left\_socket);

kill(left\_pid,SIGTERM);

kill(left\_pid,SIGKILL);

}

if (right\_pid != 0) {

send\_message(right\_socket, "kill\_children");

receive\_message(right\_socket);

kill(right\_pid,SIGTERM);

kill(right\_pid,SIGKILL);

}

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

}

}

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

int id = stoi(argv[1]);

int parent\_port = stoi(argv[2]);

zmq::context\_t context(3);

zmq::socket\_t parent\_socket(context, ZMQ\_REP);

parent\_socket.connect(get\_port\_name(parent\_port));

parent\_socket.setsockopt(ZMQ\_RCVTIMEO, TIMER);

parent\_socket.setsockopt(ZMQ\_SNDTIMEO, TIMER);

int left\_pid = 0;

int right\_pid = 0;

int left\_id = 0;

int right\_id = 0;

zmq::socket\_t left\_socket(context, ZMQ\_REQ);

zmq::socket\_t right\_socket(context, ZMQ\_REQ);

while(true) {

string request\_string = receive\_message(parent\_socket);

istringstream command\_stream(request\_string);

string command;

command\_stream >> command;

if (command == "id") {

string parent\_string = "Ok:" + to\_string(id);

send\_message(parent\_socket, parent\_string);

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

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

send\_message(parent\_socket, parent\_string);

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

int create\_id;

command\_stream >> create\_id;

if (create\_id == id) {

string message\_string = "Error: Already exists";

send\_message(parent\_socket, message\_string);

} else if (create\_id < id) {

if (left\_pid == 0) {

left\_socket.bind(get\_port\_name(DEFAULT\_PORT + create\_id));

left\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

left\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

left\_pid = fork();

real\_create(parent\_socket, left\_socket, create\_id, left\_id, left\_pid);

} else {

send\_message(left\_socket, request\_string);

string str = receive\_message(left\_socket);

if (str == "") {

left\_socket.bind(get\_port\_name(DEFAULT\_PORT + create\_id));

left\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

left\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

left\_pid = fork();

real\_create(parent\_socket, left\_socket, create\_id, left\_id, left\_pid);

} else {

send\_message(parent\_socket, str);

n++;

left\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

left\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

}

}

} else {

if (right\_pid == 0) {

right\_socket.bind(get\_port\_name(DEFAULT\_PORT + create\_id));

right\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

right\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

right\_pid = fork();

real\_create(parent\_socket, right\_socket, create\_id, right\_id, right\_pid);

} else {

send\_message(right\_socket, request\_string);

string str = receive\_message(right\_socket);

if (str == "") {

right\_socket.bind(get\_port\_name(DEFAULT\_PORT + create\_id));

right\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

right\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

right\_pid = fork();

real\_create(parent\_socket, right\_socket, create\_id, right\_id, right\_pid);

} else {

send\_message(parent\_socket, str);

n++;

right\_socket.setsockopt(ZMQ\_RCVTIMEO, n \* TIMER);

right\_socket.setsockopt(ZMQ\_SNDTIMEO, n \* TIMER);

}

}

}

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

int delete\_id;

command\_stream >> delete\_id;

if (delete\_id < id) {

real\_kill(parent\_socket, left\_socket, delete\_id, left\_id, left\_pid, request\_string);

} else {

real\_kill(parent\_socket, right\_socket, delete\_id, right\_id, right\_pid, request\_string);

}

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

exec(command\_stream, parent\_socket, left\_socket, right\_socket, left\_pid, right\_pid, id, request\_string);

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

ping(command\_stream, parent\_socket, left\_socket, right\_socket, left\_pid, right\_pid, id, request\_string);

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

kill\_children(parent\_socket, left\_socket, right\_socket, left\_pid, right\_pid);

}

if (parent\_port == 0) {

break;

}

}

return 0;

}

**tree.cpp**

#include <iostream>

#include <vector>

#include <algorithm>

#include "../include/tree.h"

Tree::~Tree() {

delete\_node(root);

}

void Tree::push(int id) {

root = push(root, id);

}

void Tree::kill(int id) {

root = kill(root, id);

}

void Tree::delete\_node(Node\* node) {

if (node == NULL) {

return;

}

delete\_node(node->right);

delete\_node(node->left);

delete node;

}

std::vector<int> Tree::get\_nodes() {

std::vector<int> result;

get\_nodes(root, result);

return result;

}

void Tree::get\_nodes(Node\* node, std::vector<int>& v) {

if (node == NULL) {

return;

}

get\_nodes(node->left, v);

v.push\_back(node->id);

get\_nodes(node->right, v);

}

Node\* Tree::push(Node\* root, int val) {

if (root == NULL) {

root = new Node;

root->id = val;

root->left = NULL;

root->right = NULL;

return root;

}

else if (val < root->id) {

root->left = push(root->left, val);

}

else if (val >= root->id) {

root->right = push(root->right, val);

}

return root;

}

Node\* Tree::kill(Node\* root\_node, int val) {

Node\* node;

if (root\_node == NULL) {

return NULL;

}

else if (val < root\_node->id) {

root\_node->left = kill(root\_node->left, val);

}

else if (val >root\_node->id) {

root\_node->right = kill(root\_node->right, val);

}

else {

node = root\_node;

if (root\_node->left == NULL) {

root\_node = root\_node->right;

}

else if (root\_node->right == NULL) {

root\_node = root\_node->left;

}

delete node;

}

if (root\_node == NULL) {

return root\_node;

}

return root\_node;

}

**tree.h**

#pragma once

#include <vector>

struct Node {

int id;

Node\* left;

Node\* right;

};

class Tree {

public:

void push(int);

void kill(int);

std::vector<int> get\_nodes();

~Tree();

private:

Node\* root = NULL;

Node\* push(Node\* t, int);

Node\* kill(Node\* t, int);

void get\_nodes(Node\*, std::vector<int>&);

void delete\_node(Node\*);

};

**Демонстрация работы программы**

tvard@tvard-HVY-WXX9:~/os/OS-labs/lab6-8$ ./server

Commands:

1. create (id)

2. exec (id) (text\_string, pattern\_string)

3. kill (id)

4. ping (id)

5. exit

create 2

Ok:4030

create 3

Ok:4035

exec 2 abracadabra abra

Ok:2:0;7

exec 3 abracadabra abra

Ok:3:0;7

create 4

Ok:4176

ping 2

Ok: 1

ping 3

Ok: 1

kill 3

Ok

ping 4

Error:4: Not found

ping 3

Error:3: Not found

ping 2

Ok: 1

ping 1

Error:1: Not found

create 5

Ok:4350

ping 5

Ok: 1

exec 5 bebra beb

Ok:5:0

kill 5

Ok

ping 2

Ok: 1

create 7

Ok:4407

create 8

Ok:4441

create 9

Ok:4446

exec 7 hahahah ha

Ok:7:0;2;4

kill 7

Ok

ping 8

Error:8: Not found

ping 9

Error:9: Not found

ping 2

Ok: 1

exec 2 hehehe he

Ok:2:0;2;4

exit

**Выводы**

Выполняя лабораторную работу, я освоил основы библиотеки ZeroMQ, а также познакомился с очередями сообщений.