Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6 по курсу**

**«Операционные системы»**

Студент: Меджидли Исмаил Ибрагим оглы

Группа: М8О-201Б-21

Вариант: 9

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2023

**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/imedzhidli/Operational-Systems

**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. Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

**Создание нового вычислительного узла** (Формат команды: create id [parent])

**Исполнение команды на вычислительном узле** (Формат команды: exec id [params])

**Проверка доступности узла** (Формат команды: ping id)

**Удаление узла** (Формат команды remove id)

Топология — список, команда — работа с локальным словарем, проверка доступности — ping id.

**Общие сведения о программе**

Связь между вычислительными узлами будем поддерживать с помощью ZMQ\_PAIR. При инициализации установить время ожидания ZMQ\_SNDTIMEO и ZMQ\_RECVTIMEO, чтобы предусмотреть случай, когда дочерний процесс был убит. Для обмена информацией будем использовать специальную структуру node\_token\_t, в которой есть перечислимое поле actions. Вычислительные узлы обрабатывают каждое сообщение: если идентификатор сообщения не совпадает с идентификатором узла, то он отправляет сообщение дальше и ждёт ответа снизу.

**Общий метод и алгоритм решения**

Используемые методы системные вызовы:

| zmq\_ctx\_new() | Создает новый ØMQ контекст |
| --- | --- |
| void \*zmq\_socket (void \*context, int type); | Создает ØMQ сокет |
| int zmq\_setsockopt (void \*socket, int option\_name, const void \*option\_value, size\_t option\_len); | Устанавливает опции ØMQ сокета |
| int zmq\_msg\_init (zmq\_msg\_t \*msg); | Инициализирует пустое ØMQ сообщение |
| int zmq\_msg\_recv (zmq\_msg\_t \*msg, void \*socket, int flags); | Получает часть сообщения из сокета |
| int zmq\_msg\_close (zmq\_msg\_t \*msg); | Освобождает сообщение ØMQ |
| int zmq\_msg\_init\_size (zmq\_msg\_t \*msg, size\_t size); | Инициализирует ØMQ сообщение определенного размера |
| int zmq\_msg\_init\_data (zmq\_msg\_t \*msg, void \*data, size\_t size, zmq\_free\_fn \*ffn, void \*hint); | Инициализирует сообщение ØMQ из предоставленного буфера. |
| int zmq\_msg\_send (zmq\_msg\_t \*msg, void \*socket, int flags); | Отправляет часть сообщения на сокет |
| int zmq\_bind (void \*socket, const char \*endpoint); | Принимает входящие соединения на сокет |
| int zmq\_close (void \* socket ); | Закрывает сокет ØMQ |
| int zmq\_ctx\_term (void \* context ); | Уничтожает контекст ØMQ |
| assert(expr) | Прекращает работу программы при ложном утверждении |

**Исходный код**

**CMakeLists.txt**

**cmake\_minimum\_required(VERSION 3.16.3)**

**project(6-8\_lab/src)**

**add\_definitions( -Wall -Werror -Wextra)**

**add\_executable(control control\_node.cpp)**

**add\_executable(calculation\_node calculation\_node.cpp)**

**target\_link\_libraries(control zmq)**

**target\_link\_libraries(calculation\_node zmq)**

**topology.h**

#ifndef INC\_6\_8\_LAB\_\_TOPOLOGY\_H\_

#define INC\_6\_8\_LAB\_\_TOPOLOGY\_H\_

#include <iostream>

#include <list>

#include <map>

using namespace std;

template<typename T>

class topology\_t {

public:

using list\_type = list<list<T>>;

using iterator = typename list<T>::iterator;

using list\_iterator = typename list\_type::iterator;

list\_type container;

size\_t container\_size;

topology\_t() : container(), container\_size(0){};

~topology\_t() = default;

void insert(const T &elem) {

list<T> new\_list;

new\_list.emplace\_back(elem);

++container\_size;

container.emplace\_back(new\_list);

}

bool insert(const T &parent, const T &elem) {

for (list\_iterator external\_it = container.begin(); external\_it != container.end(); ++external\_it) {

for (iterator internal\_it = external\_it->begin(); internal\_it != external\_it->end(); ++internal\_it) {

if (\*internal\_it == parent) {

external\_it->insert(++internal\_it, elem);

++container\_size;

return true;

}

}

}

return false;

}

bool erase(const T &elem) {

for (list\_iterator external\_it = container.begin(); external\_it != container.end(); ++external\_it) {

for (iterator internal\_it = external\_it->begin(); internal\_it != external\_it->end(); ++internal\_it) {

if (\*internal\_it == elem) {

if (external\_it->size() > 1) {

external\_it->erase(internal\_it);

} else {

container.erase(external\_it);

}

--container\_size;

return true;

}

}

}

return false;

}

size\_t size() {

return container\_size;

}

int find(const T &elem) {// in which list exists (or not) element with id $id

int ind = 0;

for (auto &external : container) {

for (auto &internal : external) {

if (internal == elem) {

return ind;

}

}

++ind;

}

return -1;

}

template<typename S>

friend ostream &operator<<(ostream &os, const topology\_t<S> &topology) {

for (auto &external : topology.container) {

os << "{";

for (auto &internal : external) {

os << internal << " ";

}

os << "}" << endl;

}

return os;

}

};

#endif//INC\_6\_8\_LAB\_\_TOPOLOGY\_H\_

**my\_zmq.h**

#ifndef INC\_6\_8\_LAB\_\_ZMQ\_H\_

#define INC\_6\_8\_LAB\_\_ZMQ\_H\_

#include <cassert>

#include <cerrno>

#include <cstring>

#include <string>

#include <zmq.h>

#include <random>

enum actions\_t {

fail = 0,

success = 1,

create = 2,

destroy = 3,

bind = 4,

ping = 5,

exec\_check = 6,

exec\_add = 7

};

const char \*NODE\_EXECUTABLE\_NAME = "calculation\_node";

const int PORT\_BASE = 8000;

const int WAIT\_TIME = 1000;

const char SENTINEL = '$';

struct node\_token\_t {

actions\_t action;

long long parent\_id, id;

};

namespace my\_zmq {

void init\_pair\_socket(void \*&context, void \*&socket) {

int rc;

context = zmq\_ctx\_new();

socket = zmq\_socket(context, ZMQ\_PAIR);

rc = zmq\_setsockopt(socket, ZMQ\_RCVTIMEO, &WAIT\_TIME, sizeof(int));

assert(rc == 0);

rc = zmq\_setsockopt(socket, ZMQ\_SNDTIMEO, &WAIT\_TIME, sizeof(int));

assert(rc == 0);

}

template<typename T>

void receive\_msg(T &reply\_data, void \*socket) {

int rc = 0;

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

rc = zmq\_msg\_recv(&reply, socket, 0);

assert(rc == sizeof(T));

reply\_data = \*(T \*)zmq\_msg\_data(&reply);

rc = zmq\_msg\_close(&reply);

assert(rc == 0);

}

template<typename T>

bool receive\_msg\_wait(T &reply\_data, void \*socket) {

int rc = 0;

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

rc = zmq\_msg\_recv(&reply, socket, 0);

if (rc == -1) {

zmq\_msg\_close(&reply);

return false;

}

assert(rc == sizeof(T));

reply\_data = \*(T \*)zmq\_msg\_data(&reply);

rc = zmq\_msg\_close(&reply);

assert(rc == 0);

return true;

}

template<typename T>

void send\_msg(T \*token, void \*socket) {

int rc = 0;

zmq\_msg\_t message;

zmq\_msg\_init(&message);

rc = zmq\_msg\_init\_size(&message, sizeof(T));

assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL);

assert(rc == 0);

rc = zmq\_msg\_send(&message, socket, 0);

assert(rc == sizeof(T));

}

template<typename T>

bool send\_msg\_no\_wait(T \*token, void \*socket) {

int rc;

zmq\_msg\_t message;

zmq\_msg\_init(&message);

rc = zmq\_msg\_init\_size(&message, sizeof(T));

assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL);

assert(rc == 0);

rc = zmq\_msg\_send(&message, socket, ZMQ\_DONTWAIT);

if (rc == -1) {

zmq\_msg\_close(&message);

return false;

}

assert(rc == sizeof(T));

return true;

}

/\* Returns true if T was successfully queued on the socket \*/

template<typename T>

bool send\_msg\_wait(T \*token, void \*socket) {

int rc;

zmq\_msg\_t message;

zmq\_msg\_init(&message);

rc = zmq\_msg\_init\_size(&message, sizeof(T));

assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL);

assert(rc == 0);

rc = zmq\_msg\_send(&message, socket, 0);

if (rc == -1) {

zmq\_msg\_close(&message);

return false;

}

assert(rc == sizeof(T));

return true;

}

/\* send\_msg && receive\_msg \*/

template<typename T>

bool send\_receive\_wait(T \*token\_send, T &token\_reply, void \*socket) {

if (send\_msg\_wait(token\_send, socket)) {

if (receive\_msg\_wait(token\_reply, socket)) {

return true;

}

}

return false;

}

}// namespace my\_zmq

#endif//INC\_6\_8\_LAB\_\_ZMQ\_H\_

**control\_node.cpp**

#include <unistd.h>

#include <vector>

#include <zmq.hpp>

#include "my\_zmq.h"

#include "topology.h"

using namespace std;

using node\_id\_type = long long;

void delete\_control\_node(node\_id\_type id, topology\_t<node\_id\_type> &control\_node, vector<pair<void \*, void \*>> children) {

int ind = control\_node.find(id);

int rc;

bool ok;

if (ind != -1) {

auto \*token = new node\_token\_t({destroy, id, id});

node\_token\_t reply({fail, id, id});

ok = my\_zmq::send\_receive\_wait(token, reply, children[ind].second);

if (reply.action == destroy and reply.parent\_id == id) {

rc = zmq\_close(children[ind].second);

assert(rc == 0);

rc = zmq\_ctx\_destroy(children[ind].first);

assert(rc == 0);

auto it = children.begin();

while (ind--) {

++it;

}

children.erase(it);

} else if (reply.action == bind and reply.parent\_id == id) {

rc = zmq\_close(children[ind].second);

assert(rc == 0);

rc = zmq\_ctx\_term(children[ind].first);

assert(rc == 0);

my\_zmq::init\_pair\_socket(children[ind].first, children[ind].second);

rc = zmq\_bind(children[ind].second, ("tcp://\*:" + to\_string(PORT\_BASE + id)).c\_str());

assert(rc == 0);

}

if (ok) {

control\_node.erase(id);

cout << "OK: " << id << endl;

} else {

cout << "Error: Node " << id << " is unavailable" << endl;

}

} else {

cout << "Error: Not found" << endl;

}

}

int main() {

int rc;

bool ok;

topology\_t<node\_id\_type> control\_node;

vector<pair<void \*, void \*>> children;// [context, socket]

string s;

node\_id\_type id;

while (cin >> s >> id) {

if (s == "create") {

node\_id\_type parent\_id;

cin >> parent\_id;

int ind;

if (parent\_id == -1) {

void \*new\_context = nullptr;

void \*new\_socket = nullptr;

my\_zmq::init\_pair\_socket(new\_context, new\_socket);

rc = zmq\_bind(new\_socket, ("tcp://\*:" + to\_string(PORT\_BASE + id)).c\_str());

assert(rc == 0);

int fork\_id = fork();

if (fork\_id == 0) {

rc = execl(NODE\_EXECUTABLE\_NAME, NODE\_EXECUTABLE\_NAME, to\_string(id).c\_str(), nullptr);

assert(rc != -1);

return 0;

} else {

auto \*token = new node\_token\_t({ping, id, id});

node\_token\_t reply({fail, id, id});

if (my\_zmq::send\_receive\_wait(token, reply, new\_socket) and reply.action == success) {

children.emplace\_back(make\_pair(new\_context, new\_socket));

control\_node.insert(id);

} else {

rc = zmq\_close(new\_socket);

assert(rc == 0);

rc = zmq\_ctx\_destroy(new\_context);

assert(rc == 0);

}

}

} else if ((ind = control\_node.find(parent\_id)) == -1) {

cout << "Error: Not found" << endl;

continue;

} else {

if (control\_node.find(id) != -1) {

cout << "Error: Already exists" << endl;

continue;

}

auto \*token = new node\_token\_t({create, parent\_id, id});

node\_token\_t reply({fail, id, id});

if (my\_zmq::send\_receive\_wait(token, reply, children[ind].second) and reply.action == success) {

control\_node.insert(parent\_id, id);

} else {

cout << "Error: Parent is unavailable" << endl;

}

}

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

delete\_control\_node(id, control\_node, children);

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

int ind = control\_node.find(id);

if (ind == -1) {

cout << "Error: Not found" << endl;

continue;

}

auto \*token = new node\_token\_t({ping, id, id});

node\_token\_t reply({fail, id, id});

if (my\_zmq::send\_receive\_wait(token, reply, children[ind].second) and reply.action == success) {

cout << "OK: 1" << endl;

} else {

cout << "OK: 0" << endl;

}

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

ok = true;

string key;

char c;

int val = -1;

bool add = false;

cin >> key;

if ((c = getchar()) == ' ') {

add = true;

cin >> val;

}

int ind = control\_node.find(id);

if (ind == -1) {

cout << "Error: Not found" << endl;

continue;

}

key += SENTINEL;

if (add) {

for (auto i: key) {

auto \*token = new node\_token\_t({exec\_add, i, id});

node\_token\_t reply({fail, id, id});

if (!my\_zmq::send\_receive\_wait(token, reply, children[ind].second) or reply.action != success) {

cout << "Fail: " << i << endl;

ok = false;

break;

}

}

auto \*token = new node\_token\_t({exec\_add, val, id});

node\_token\_t reply({fail, id, id});

if (!my\_zmq::send\_receive\_wait(token, reply, children[ind].second) or reply.action != success) {

cout << "Fail: " << val << endl;

ok = false;

}

} else {

for (auto i: key) {

auto \*token = new node\_token\_t({exec\_check, i, id});

node\_token\_t reply({fail, i, id});

if (!my\_zmq::send\_receive\_wait(token, reply, children[ind].second) or reply.action != success) {

ok = false;

cout << "Fail: " << i << endl;

break;

}

}

}

if (!ok) {

cout << "Error: Node is unavailable" << endl;

}

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

cout << control\_node;

}

}

cout << control\_node;

for (auto i: control\_node.container) {

for (size\_t size = i.size(); size > 1; --size) {

node\_id\_type last = i.back();

delete\_control\_node(last, control\_node, children);

i.pop\_back();

}

}

vector<node\_id\_type> after\_root;

for (auto i: control\_node.container) {

after\_root.push\_back(i.back());

}

for (auto i: after\_root) {

delete\_control\_node(i, control\_node, children);

}

return 0;

}

**calculation\_node.cpp**

#include "my\_zmq.h"

#include <iostream>

#include <map>

#include <unistd.h>

using namespace std;

long long node\_id;

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

string key;

int val;

map<string, int> dict;

int rc;

assert(argc == 2);

node\_id = stoll(string(argv[1]));

void \*node\_parent\_context = zmq\_ctx\_new();

void \*node\_parent\_socket = zmq\_socket(node\_parent\_context, ZMQ\_PAIR);

rc = zmq\_connect(node\_parent\_socket, ("tcp://localhost:" + to\_string(PORT\_BASE + node\_id)).c\_str());

assert(rc == 0);

long long child\_id = -1;

void \*node\_context = nullptr;

void \*node\_socket = nullptr;

cout << "OK: " << getpid() << endl;

bool has\_child = false, awake = true, add = false;

while (awake) {

node\_token\_t token({fail, 0, 0});

my\_zmq::receive\_msg(token, node\_parent\_socket);

auto \*reply = new node\_token\_t({fail, node\_id, node\_id});

if (token.action == bind and token.parent\_id == node\_id) {

/\*

\* Bind could be recieved when parent created node

\* and this node should bind to parent's child

\*/

my\_zmq::init\_pair\_socket(node\_context, node\_socket);

rc = zmq\_bind(node\_socket, ("tcp://\*:" + to\_string(PORT\_BASE + token.id)).c\_str());

assert(rc == 0);

has\_child = true;

child\_id = token.id;

auto \*token\_ping = new node\_token\_t({ping, child\_id, child\_id});

node\_token\_t reply\_ping({fail, child\_id, child\_id});

if (my\_zmq::send\_receive\_wait(token\_ping, reply\_ping, node\_socket) and reply\_ping.action == success) {

reply->action = success;

}

} else if (token.action == create) {

if (token.parent\_id == node\_id) {

if (has\_child) {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

}

my\_zmq::init\_pair\_socket(node\_context, node\_socket);

rc = zmq\_bind(node\_socket, ("tcp://\*:" + to\_string(PORT\_BASE + token.id)).c\_str());

assert(rc == 0);

int fork\_id = fork();

if (fork\_id == 0) {

rc = execl(NODE\_EXECUTABLE\_NAME, NODE\_EXECUTABLE\_NAME, to\_string(token.id).c\_str(), nullptr);

assert(rc != -1);

return 0;

} else {

bool ok = true;

if (has\_child) {

auto \*token\_bind = new node\_token\_t({bind, token.id, child\_id});

node\_token\_t reply\_bind({fail, token.id, token.id});

ok = my\_zmq::send\_receive\_wait(token\_bind, reply\_bind, node\_socket);

ok = ok and (reply\_bind.action == success);

}

if (ok) {

/\* We should check if child has connected to this node \*/

auto \*token\_ping = new node\_token\_t({ping, token.id, token.id});

node\_token\_t reply\_ping({fail, token.id, token.id});

ok = my\_zmq::send\_receive\_wait(token\_ping, reply\_ping, node\_socket);

ok = ok and (reply\_ping.action == success);

if (ok) {

reply->action = success;

child\_id = token.id;

has\_child = true;

} else {

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_term(node\_context);

assert(rc == 0);

}

}

}

} else if (has\_child) {

auto \*token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == ping) {

if (token.id == node\_id) {

reply->action = success;

} else if (has\_child) {

auto \*token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == destroy) {

if (has\_child){

if (token.id == child\_id){

bool ok;

auto\* token\_down = new node\_token\_t({destroy, node\_id, child\_id});

node\_token\_t reply\_down = {fail, child\_id, child\_id};

ok = my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket);

if (reply\_down.action == destroy){

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_destroy(node\_context);

assert(rc == 0);

has\_child = false;

child\_id = -1;

}

else if(reply\_down.action == bind){

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_destroy(node\_context);

assert(rc == 0);

my\_zmq::init\_pair\_socket(node\_context, node\_socket);

rc = zmq\_bind(node\_socket, ("tcp://\*:" + to\_string(PORT\_BASE + reply\_down.id)).c\_str());

assert(rc == 0);

child\_id = reply\_down.id;

auto \*token\_ping = new node\_token\_t({ping, child\_id, child\_id});

node\_token\_t reply\_ping({fail, child\_id, child\_id});

ok = my\_zmq::send\_receive\_wait(token\_ping, reply\_ping, node\_socket) and (reply\_ping.action == success);

}

if (ok){

reply->action = success;

}

} else if (token.id == node\_id){

rc = zmq\_close(node\_socket);

assert(rc == 0);

rc = zmq\_ctx\_destroy(node\_context);

assert(rc == 0);

awake = false;

reply->action = bind;

reply->id = child\_id;

reply->parent\_id = token.parent\_id;

} else{

auto\* token\_down = new node\_token\_t(token);

node\_token\_t reply\_down = token;

reply\_down.action = fail;

if (my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket) and (reply\_down.action == success)){

\*reply = reply\_down;

}

}

} else if (token.id == node\_id){

reply->action = destroy;

awake = false;

}

} else if (token.action == exec\_check) {

if (token.id == node\_id) {

char c = token.parent\_id;

if (c == SENTINEL) {

if (dict.find(key) != dict.end()) {

cout << "OK:" << node\_id << ":" << dict[key] << endl;

} else {

cout << "OK:" << node\_id << ":'" << key << "' not found" << endl;

}

reply->action = success;

key = "";

} else {

key += c;

reply->action = success;

}

} else if (has\_child) {

auto \*token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

} else if (token.action == exec\_add) {

if (token.id == node\_id) {

char c = token.parent\_id;

if (c == SENTINEL) {

add = true;

reply->action = success;

} else if (add) {

val = token.parent\_id;

dict[key] = val;

cout << "OK:" << node\_id << endl;

add = false;

key = "";

reply->action = success;

} else {

key += c;

reply->action = success;

}

} else if (has\_child) {

auto \*token\_down = new node\_token\_t(token);

node\_token\_t reply\_down(token);

reply\_down.action = fail;

if (my\_zmq::send\_receive\_wait(token\_down, reply\_down, node\_socket) and reply\_down.action == success) {

\*reply = reply\_down;

}

}

}

my\_zmq::send\_msg\_no\_wait(reply, node\_parent\_socket);

}

rc = zmq\_close(node\_parent\_socket);

assert(rc == 0);

rc = zmq\_ctx\_destroy(node\_parent\_context);

assert(rc == 0);

}

**Демонстрация работы программы**

imedzhidli@imedzhidli:~/Desktop/OS/6Laba/build$ ./control

create 10 -1

OK: 26009

create 5 -1

OK: 26037

create 25 10

OK: 26108

ping 10

OK: 1

ping 5

OK: 1

ping 7

Error: Not found

ping 25

OK: 1

exec 10 key

OK:10:'key' not found

exec 10 key 10

OK:10

exec 10 key

OK:10:10

remove 10

OK: 10

ping 10

Error: Not found

ping 5

OK: 1

ping 25

OK: 0

imedzhidli@imedzhidli:~/Desktop/OS/6Laba/build$

**Выводы:**

В ходе выполнения лабораторной работы познакомился с сокетами, очередями сообщений, с высокопроизводительной библиотекой ZMQ, некоторыми паттернами, познакомился с клиент-серверной архитектурой.