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: Фролов М.А.

Группа: М8О–206Б–20

Вариант: 41

Преподаватель: Соколов Андрей Алексеевич

Оценка: 5

Дата: 18.12.21

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Постановка задачи**

## Цель работы

Целью является приобретение практических навыков в:

* Управлении серверами сообщений (№ 6)
* Применение отложенных вычислений (№ 7)
* Интеграция программных систем друг с другом (№ 8)

## Задание

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]  
id – целочисленный идентификатор нового вычислительного узла

parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)

Формат вывода:

«Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла

«Error: Already exists» - вычислительный узел с таким идентификатором уже существует

«Error: Parent not found» - нет такого родительского узла с таким идентификатором

«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не удается связаться

«Error: [Custom error]» - любая другая обрабатываемая ошибка Пример:  
> create 10 5  
Ok: 3128

Удаление существующего вычислительного узла

Формат команды: remove id  
id – целочисленный идентификатор удаляемого вычислительного узла

Формат вывода:

«Ok» - успешное удаление

«Error: Not found» - вычислительный узел с таким идентификатором не найден

«Error: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

«Error: [Custom error]» - любая другая обрабатываемая ошибка

Пример:

> remove 10

Ok

Исполнение команды на вычислительном узле

Формат команды: exec id [params]

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

Формат вывода:

«Ok:id: [result]», где result – результат выполненной команды

«Error:id: Not found» - вычислительный узел с таким идентификатором не найден

«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

«Error:id: [Custom error]» - любая другая обрабатываемая ошибка  
Пример:  
Можно найти в описании конкретной команды, определенной вариантом задания.

Вариант 41.

Топология 3: все вычислительные узлы хранятся в бинарном дереве поиска. [parent] — является необязательным параметром.

Набор команд 1 (подсчет суммы n чисел).

Формат команды: exec id n k1 … kn

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

n – количество складываемых чисел (от 1 до 108)

k1 … kn – складываемые числа

Пример:

> exec 10 3 1 2 3

Ok:10: 6Команда проверки 1.

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

Пример:

> pingall

Ok: -1 // Все узлы доступны

> pingall

Ok: 7;10;15 // узлы 7, 10, 15 — недоступны

**Общие сведения о программе**

Система состоит из двух программ: сервер и клиент. Используется сервер сообщений ZeroMQ.

Программа сервер компилируется из файла server.c. Программа клиент из файла client.c. Обе программы используют заголовочный файл wrap\_zmq.h, который содержит обертки над вызовами из библиотеки ZeroMQ.

В программе используются следующие системные вызовы:

1. **fork –** создает новый процесс, который является копией родительского процесса, за исключением разных process ID и parent process ID. В случае успеха fork() возвращает 0 для ребенка, число больше 0 для родителя – child ID, в случае ошибки возвращает -1.
2. **execl** — используется для выполнения другой программы. Эта другая программа, называемая процессом-потомком (child process), загружается поверх программы, содержащей вызов exec. Имя файла, содержащего процесс-потомок, задано с помощью первого аргумента. Какие-либо аргументы, передаваемые процессу-потомку, задаются либо с помощью параметров от arg0 до argN, либо с помощью массива arg[].

Также были использованы следующие вызовы из библиотеки ZMQ:

1. **zmq\_ctx\_new –** создает новый контекст ZMQ.
2. **zmq\_connect –** создает входящее соединение на сокет.
3. **zmq\_disconnect –** отсоединяет сокет от заданного endpoint’a.
4. **zmq\_socket** **–** создает ZMQ сокет.
5. **zmq\_setsockopt** **–** задает параметры ZMQ сокета.
6. **zmq\_close –** закрывает ZMQ сокет.
7. **zmq\_ctx\_destroy –** уничтожает контекст ZMQ.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы с ZMQ.
2. Проработать принцип общения между клиентскими узлами и между первым клиентом и сервером и алгоритм выполнения команд клиентами.
3. Реализовать необходимые функции-обертки над вызовами функций библиотеки ZMQ.
4. Написать программу сервера и клиента

**Основные файлы программы**

Client.cpp:

#include <string.h>

#include <iostream>

#include <unistd.h>

#include <vector>

#include <algorithm>

#include <csignal>

#include "wrap\_zmq.h"

#include "socket.h"

#include "client.h"

#include <signal.h>

#include <sys/types.h>

using namespace std;

Client::Client(int new\_id, string parent\_endpoint, int new\_parent\_id){

id = new\_id;

parent\_id = new\_parent\_id;

context = create\_zmq\_ctx();

string endpoint = create\_endpoint(EndpointType::CHILD\_PUB\_LEFT, getpid());

child\_publisher\_left = new Socket(context, SocketType::PUBLISHER, endpoint);

endpoint = create\_endpoint(EndpointType::CHILD\_PUB\_RIGHT, getpid());

child\_publisher\_right = new Socket(context, SocketType::PUBLISHER, endpoint);

endpoint = create\_endpoint(EndpointType::PARENT\_PUB, getpid());

parent\_publisher = new Socket(context, SocketType::PUBLISHER, endpoint);

parent\_subscriber = new Socket(context, SocketType::SUBSCRIBER, parent\_endpoint);

left\_subscriber = nullptr;

right\_subscriber = nullptr;

terminated = false;

}

Client::~Client(){

if(terminated) return;

terminated = true;

try{

delete child\_publisher\_left;

delete child\_publisher\_right;

delete parent\_publisher;

delete parent\_subscriber;

if(left\_subscriber)

delete left\_subscriber;

if(right\_subscriber)

delete right\_subscriber;

destroy\_zmq\_ctx(context);

} catch(runtime\_error& err){

cout << "Server wasn't stopped " << err.what() << endl;

}

}

bool& Client::get\_status(){

return terminated;

}

void Client::send\_up(Message msg){

msg.to\_up = true;

parent\_publisher->send(msg);

}

void Client::send\_down(Message msg){

msg.to\_up = false;

child\_publisher\_left->send(msg);

child\_publisher\_right->send(msg);

}

int Client::get\_id(){

return id;

}

int Client::add\_child(int new\_id){

pid\_t pid = fork();

if(pid == -1) throw runtime\_error("Can not fork.");

if(!pid){

string endpoint;

if(new\_id < id){

endpoint = child\_publisher\_left->get\_endpoint();

} else{

endpoint = child\_publisher\_right->get\_endpoint();

}

execl("client", "client", to\_string(new\_id).data(),

endpoint.data(), to\_string(id).data(), nullptr);

throw runtime\_error("Can not execl.");

}

string endpoint = create\_endpoint(EndpointType::PARENT\_PUB, pid);

size\_t timeout = 10000;

if(id > new\_id){

left\_subscriber = new Socket(context, SocketType::SUBSCRIBER, endpoint);

zmq\_setsockopt(left\_subscriber->get\_socket(), ZMQ\_RCVTIMEO, &timeout, sizeof(timeout));

} else{

right\_subscriber = new Socket(context, SocketType::SUBSCRIBER, endpoint);

zmq\_setsockopt(right\_subscriber->get\_socket(), ZMQ\_RCVTIMEO, &timeout, sizeof(timeout));

}

return pid;

}

void process\_msg(Client& client, Message msg){

switch(msg.command){

case CommandType::ERROR:

throw runtime\_error("Error message received.");

case CommandType::RETURN:{

msg.get\_to\_id() = SERVER\_ID;

client.send\_up(msg);

break;

}

case CommandType::CREATE\_CHILD:{

msg.get\_create\_id() = client.add\_child(msg.get\_create\_id());

msg.get\_to\_id() = SERVER\_ID;

client.send\_up(msg);

break;

}

case CommandType::REMOVE\_CHILD:{

if(msg.to\_up){

client.send\_up(msg);

break;

}

if(msg.to\_id != client.get\_id() && msg.to\_id != UNIVERSAL\_MSG){

client.send\_down(msg);

break;

}

msg.get\_to\_id() = PARENT\_SIGNAL;

msg.get\_create\_id() = client.get\_id();

client.send\_up(msg);

msg.get\_to\_id() = UNIVERSAL\_MSG;

client.send\_down(msg);

client.~Client();

throw invalid\_argument("Exiting child...");

break;

}

case CommandType::EXEC\_CHILD: {

double res = 0.0;

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

res += msg.value[i];

}

msg.get\_to\_id() = SERVER\_ID;

msg.get\_create\_id() = client.get\_id();

msg.value[0] = res;

client.send\_up(msg);

break;

}

default:

throw runtime\_error("Undefined command.");

}

}

Client\* client\_ptr = nullptr;

void TerminateByUser(int) {

if (client\_ptr != nullptr) {

client\_ptr->~Client();

}

cout << to\_string(getpid()) + " Terminated by user" << endl;

exit(0);

}

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

{

if(argc != 4){

cout << "-1" << endl;

return -1;

}

try{

if (signal(SIGINT, TerminateByUser) == SIG\_ERR) {

throw runtime\_error("Can not set SIGINT signal");

}

if (signal(SIGSEGV, TerminateByUser) == SIG\_ERR) {

throw runtime\_error("Can not set SIGSEGV signal");

}

if (signal(SIGTERM, TerminateByUser) == SIG\_ERR) {

throw runtime\_error("Can not set SIGTERM signal");

}

Client client(stoi(argv[1]), string(argv[2]), stoi(argv[3]));

client\_ptr = &client;

cout << getpid() << ": " "Client started. " << "Id:" << client.get\_id() << endl;

for(;;){

//cout << client.get\_id() << ": " "Client ready. " << client.parent\_subscriber->get\_endpoint() << endl;

Message msg = client.parent\_subscriber->receive();

if(msg.to\_id != client.get\_id() && msg.to\_id != UNIVERSAL\_MSG){

if(msg.to\_up){

client.send\_up(msg);

} else{

try{

if(client.get\_id() < msg.to\_id){

msg.to\_up = false;

client.child\_publisher\_right->send(msg);

msg = client.right\_subscriber->receive();

} else{

msg.to\_up = false;

client.child\_publisher\_left->send(msg);

msg = client.left\_subscriber->receive();

}

if(msg.command == CommandType::REMOVE\_CHILD && msg.to\_id == PARENT\_SIGNAL){

msg.to\_id = SERVER\_ID;

if(client.get\_id() < msg.get\_create\_id()){

delete client.right\_subscriber;

client.right\_subscriber = nullptr;

} else{

delete client.left\_subscriber;

client.left\_subscriber = nullptr;

}

}

client.send\_up(msg);

}catch(...){

client.send\_up(Message());

}

}

} else{

process\_msg(client, msg);

}

}

} catch(runtime\_error& err){

cout << getpid() << ": " << err.what() << '\n';

} catch(invalid\_argument& inv){

cout << getpid() << ": " << inv.what() << '\n';

}

return 0;

}

Client.h:

#ifndef \_CLIENT\_H

#define \_CLIENT\_H

#include <string.h>

#include <iostream>

#include <unistd.h>

#include "wrap\_zmq.h"

#include "socket.h"

using namespace std;

class Client{

private:

int id;

void\* context;

bool terminated;

public:

Socket\* child\_publisher\_left;

Socket\* child\_publisher\_right;

Socket\* parent\_publisher;

Socket\* parent\_subscriber;

Socket\* left\_subscriber;

Socket\* right\_subscriber;

Client(int new\_id, string parent\_endpoint, int new\_parent\_id);

~Client();

bool& get\_status();

void send\_up(Message msg);

void send\_down(Message msg);

Message receive();

int get\_id();

int add\_child(int id);

int parent\_id;

};

#endif

Server.cpp:

#include <string.h>

#include <iostream>

#include <vector>

#include <unistd.h>

#include <csignal>

#include "server.h"

#include "socket.h"

#include "wrap\_zmq.h"

void\* subscriber\_thread(void\* server){

Server\* server\_ptr = (Server\*) server;

pid\_t serv\_pid = server\_ptr->get\_pid();

try{

pid\_t child\_pid = fork();

if(child\_pid == -1) throw runtime\_error("Can not fork.");

if(child\_pid == 0){

execl("client", "client", "0", server\_ptr->get\_publisher()->get\_endpoint().data(), "-1", nullptr);

throw runtime\_error("Can not execl");

server\_ptr->~Server();

return (void\*)-1;

}

string endpoint = create\_endpoint(EndpointType::PARENT\_PUB, child\_pid);

server\_ptr->get\_subscriber() = new Socket(server\_ptr->get\_context(), SocketType::SUBSCRIBER, endpoint);

server\_ptr->get\_tree().insert(0);

for(;;){

Message msg = server\_ptr->get\_subscriber()->receive();

if(msg.command == CommandType::ERROR){

//todo

}

server\_ptr->last\_msg = msg;

//cout << "Message on server: " << int

switch(msg.command){

case CommandType::CREATE\_CHILD:

cout << "OK:" << msg.get\_create\_id() << endl;

break;

case CommandType::REMOVE\_CHILD:

cout << "OK" << endl;

server\_ptr->get\_tree().delete\_el(msg.get\_create\_id());

break;

case CommandType::RETURN:

break;

case CommandType::EXEC\_CHILD:

cout << "OK:" << msg.get\_create\_id() << ":" << msg.value[0] << endl;

break;

default:

break;

}

}

} catch(runtime\_error& err){

cout << "Server wasn't started " << err.what() << endl;

}

return nullptr;

}

Server::Server(){

context = create\_zmq\_ctx();

pid = getpid();

string endpoint = create\_endpoint(EndpointType::CHILD\_PUB\_LEFT, getpid());

publisher = new Socket(context, SocketType::PUBLISHER, endpoint);

if(pthread\_create(&receive\_msg, 0, subscriber\_thread, this) != 0){

throw runtime\_error("Can not run second thread.");

}

working = true;

}

Server::~Server(){

if(!working) return;

working = false;

send(Message(CommandType::REMOVE\_CHILD, 0, 0));

try{

delete publisher;

delete subscriber;

publisher = nullptr;

subscriber = nullptr;

destroy\_zmq\_ctx(context);

sleep(2);

} catch (runtime\_error &err){

cout << "Server wasn't stopped " << err.what() << endl;

}

}

pid\_t Server::get\_pid(){

return pid;

}

void Server::print\_tree(){

t.print();

}

bool Server::check(int id){

Message msg(CommandType::RETURN, id, 0);

send(msg);

sleep(msg\_wait\_time);

msg.get\_to\_id() = SERVER\_ID;

//cout << msg.to\_id << " " << msg.create\_id << " " << msg.uniq\_num << " " << endl;

//cout << last\_msg.to\_id << " " << last\_msg.create\_id << " " << last\_msg.uniq\_num << " " << endl;

return last\_msg == msg;

}

void Server::send(Message msg){

msg.to\_up = false;

publisher->send(msg);

}

Message Server::receive(){

return subscriber->receive();

}

Socket\*& Server::get\_publisher(){

return publisher;

}

Socket\*& Server::get\_subscriber(){

return subscriber;

}

void\* Server::get\_context(){

return context;

}

tree& Server::get\_tree(){

return t;

}

void Server::create\_child(int id){

if(t.find(id)){

throw runtime\_error("Error:" + to\_string(id) + ":Node with that number already exists.");

}

if(t.get\_place(id) && !check(t.get\_place(id))){ //not zero node

throw runtime\_error("Error:" + to\_string(id) + ":Parent node is unavailable.");

}

send(Message(CommandType::CREATE\_CHILD, t.get\_place(id), id));

t.insert(id);

}

void Server::remove\_child(int id){

if(!t.find(id)){

throw runtime\_error("Error:" + to\_string(id) + ":Node with that number doesn't exist.");

}

if(!check(id)){

throw runtime\_error("Error:" + to\_string(id) + ":Node is unavailable.");

}

send(Message(CommandType::REMOVE\_CHILD, id, 0));

}

void Server::exec\_child(int id, int n){

double nums[n];

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

int cur;

cin >> cur;

nums[i] = cur;

}

if(!t.find(id)){

throw runtime\_error("Error:" + to\_string(id) + ":Node with that number doesn't exist.");

}

if(!check(id)){

throw runtime\_error("Error:" + to\_string(id) + ":Node is unavailable.");

}

send(Message(CommandType::EXEC\_CHILD, id, n, nums, 0));

}

void process\_cmd(Server& server, string cmd){

if(cmd == "create"){

int id;

cin >> id;

server.create\_child(id);

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

int id;

cin >> id;

server.remove\_child(id);

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

int id;

cin >> id;

int n;

cin >> n;

server.exec\_child(id, n);

} else if(cmd == "exit"){

throw invalid\_argument("Exiting...");

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

vector<int> tmp = server.get\_tree().get\_all\_elems();

//tmp.pop\_back();

cout << "OK:";

bool fnd = false;

for(int& i : tmp){

//cout << "(()()" << i << endl;

if(!server.check(i)){

if(fnd) cout << ';';

cout << i;

fnd = true;

}

}

if(!fnd) cout << "-1";

cout << endl;

} else if(cmd == "status"){

int id;

cin >> id;

if(!server.get\_tree().find(id)){

throw runtime\_error("Error:" + to\_string(id) + ":Node with that number doesn't exist.");

}

if(server.check(id)){

cout << "OK" << endl;

} else{

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

}

} else {

cout << "It is not a command!\n";

}

}

Server\* server\_ptr = nullptr;

void TerminateByUser(int) {

if (server\_ptr != nullptr) {

server\_ptr->~Server();

}

cout << to\_string(getpid()) + " Terminated by user" << endl;

exit(0);

}

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

{

try{

if (signal(SIGINT, TerminateByUser) == SIG\_ERR) {

throw runtime\_error("Can not set SIGINT signal");

}

if (signal(SIGSEGV, TerminateByUser) == SIG\_ERR) {

throw runtime\_error("Can not set SIGSEGV signal");

}

if (signal(SIGTERM, TerminateByUser) == SIG\_ERR) {

throw runtime\_error("Can not set SIGTERM signal");

}

Server server;

server\_ptr = &server;

cout << getpid() << " server started correctly!\n";

for(;;){

try{

string cmd;

while(cin >> cmd){

process\_cmd(server, cmd);

}

} catch(const runtime\_error& arg){

cout << arg.what() << endl;

}

}

} catch(const runtime\_error& arg){

cout << arg.what() << endl;

} catch(...){}

sleep(5);

return 0;

}

Server.h:

#ifndef \_SERVER\_H

#define \_SERVER\_H

#include "tree.h"

#include "socket.h"

#include "wrap\_zmq.h"

#define msg\_wait\_time 1

class Server{

public:

Server();

~Server();

void print\_tree();

void send(Message msg);

Message receive();

void create\_child(int id);

void remove\_child(int id);

void exec\_child(int id, int n);

pid\_t get\_pid();

bool check(int id);

Socket\*& get\_publisher();

Socket\*& get\_subscriber();

void\* get\_context();

tree& get\_tree();

Message last\_msg;

private:

pid\_t pid;

tree t;

void \*context = nullptr;

Socket\* publisher;

Socket\* subscriber;

bool working;

pthread\_t receive\_msg;

};

#endif

Socket.cpp:

#include <string>

#include <iostream>

#include <stdexcept>

#include "wrap\_zmq.h"

#include "socket.h"

using namespace std;

Socket::Socket(void\* context, SocketType new\_socket\_type, string new\_endpoint)

: socket\_type(new\_socket\_type), endpoint(new\_endpoint) {

socket = create\_zmq\_socket(context, new\_socket\_type);

switch (socket\_type) {

case SocketType::PUBLISHER:

bind\_zmq\_socket(socket, new\_endpoint);

break;

case SocketType::SUBSCRIBER:

connect\_zmq\_socket(socket, new\_endpoint);

break;

default:

throw logic\_error("Undefined connection type");

}

}

Socket::~Socket() {

try {

switch(socket\_type){

case SocketType::PUBLISHER:

//cout << "unbind: " << endpoint << endl;

unbind\_zmq\_socket(socket, endpoint);

break;

case SocketType::SUBSCRIBER:

//cout << "disconnect: " << endpoint << endl;

disconnect\_zmq\_socket(socket, endpoint);

break;

}

close\_zmq\_socket(socket);

} catch (exception& ex) {

cout << "Socket wasn't closed: " << ex.what() << endl;

}

}

void Socket::send(Message message) {

if (socket\_type == SocketType::PUBLISHER) {

send\_zmq\_msg(socket, message);

} else {

throw logic\_error("SUB socket can't send messages");

}

}

Message Socket::receive() {

if (socket\_type == SocketType::SUBSCRIBER) {

return get\_zmq\_msg(socket);

} else {

throw logic\_error("PUB socket can't receive messages");

}

}

void Socket::subscribe(string new\_endpoint) {

if (socket\_type == SocketType::SUBSCRIBER) {

connect\_zmq\_socket(socket, new\_endpoint);

endpoint = new\_endpoint;

} else {

throw logic\_error("Subscribe is only for SUB sockets");

}

}

string Socket::get\_endpoint() const{

return endpoint;

}

void\*& Socket::get\_socket(){

return socket;

}

Socket.h:

#ifndef \_SOCKET\_H

#define \_SOCKET\_H

#include <string>

#include "wrap\_zmq.h"

using namespace std;

class Socket {

public:

Socket(void\* context, SocketType new\_socket\_type, string new\_endpoint);

~Socket();

void send(Message message);

Message receive();

void subscribe(string new\_endpoint);

string get\_endpoint() const;

void\*& get\_socket();

private:

void\* socket;

SocketType socket\_type;

string endpoint;

};

#endif

Tree.cpp:

#include <iostream>

#include <vector>

#include "tree.h"

using namespace std;

tree\_el::tree\_el(int new\_val) : value(new\_val), left(nullptr), right(nullptr){}

int& tree\_el::get\_value(){

return value;

}

tree\_el\*& tree\_el::get\_left(){

return left;

}

tree\_el\*& tree\_el::get\_right(){

return right;

}

bool operator<(tree\_el& lhs, tree\_el& rhs){

return lhs.get\_value() < rhs.get\_value();

}

bool operator==(tree\_el& lhs, tree\_el& rhs){

return lhs.get\_value() == rhs.get\_value();

}

void tree::delete\_tree(tree\_el\*& cur){

if(!cur) return;

delete\_tree(cur->get\_left());

delete\_tree(cur->get\_right());

delete cur;

}

void tree::print\_tree(tree\_el\*& cur, int h){

if(!cur) return;

print\_tree(cur->get\_left(), h + 1);

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

cout << " ";

}

cout << cur->get\_value() << endl;

print\_tree(cur->get\_right(), h + 1);

}

void tree::get\_all(tree\_el\* cur, vector<int>& tmp){

if(!cur) return;

get\_all(cur->get\_left(), tmp);

tmp.push\_back(cur->get\_value());

get\_all(cur->get\_right(), tmp);

}

vector<int> tree::get\_all\_elems(){

vector<int> tmp;

get\_all(root, tmp);

return tmp;

}

bool tree::find\_el(tree\_el\* cur, int val){

if(cur == nullptr){

return false;

}

if(val < cur->get\_value()){

return find\_el(cur->get\_left(), val);

} else if (val > cur->get\_value()){

return find\_el(cur->get\_right(), val);

} else {

return true;

}

}

void tree::insert\_el(tree\_el\*& cur, int new\_val){

if(cur == nullptr){

cur = new tree\_el(new\_val);

} else if(new\_val < cur->get\_value()){

return insert\_el(cur->get\_left(), new\_val);

} else if (new\_val > cur->get\_value()){

return insert\_el(cur->get\_right(), new\_val);

}

}

tree\_el\*& tree::delete\_rightmost(tree\_el\*& cur, int\* replacement){

if(cur->get\_right()){

cur->get\_right() = delete\_rightmost(cur->get\_right(), replacement);

return cur;

} else{

\*replacement = cur->get\_value();

return cur->get\_left();

}

}

void tree::remove\_el(tree\_el\*& cur, int val){

if(!cur) return;

if(val < cur->get\_value()){

remove\_el(cur->get\_left(), val);

} else if(val > cur->get\_value()){

remove\_el(cur->get\_right(), val);

} else{

if(!cur->get\_right()){

tree\_el\* tmp = cur;

cur = cur->get\_left();

delete tmp;

} else if (!cur->get\_left()){

tree\_el\* tmp = cur;

cur = cur->get\_right();

delete tmp;

} else{

int new\_val;

cur->get\_left() = delete\_rightmost(cur->get\_left(), &new\_val);

cur->get\_value() = new\_val;

}

}

}

tree::tree():root(nullptr){}

tree::tree(int new\_val){

root = new tree\_el(new\_val);

}

void tree::insert(int new\_val){

insert\_el(root, new\_val);

}

bool tree::find(int val){

return find\_el(root, val);

}

void tree::print(){

print\_tree(root);

}

void tree::delete\_el\_tree(tree\_el\*& cur, int val){

if(!cur) return;

if(val < cur->get\_value()){

delete\_el\_tree(cur->get\_left(), val);

} else if(val > cur->get\_value()){

delete\_el\_tree(cur->get\_right(), val);

} else{

delete\_tree(cur);

cur = nullptr;

}

}

void tree::delete\_el(int val){

//remove\_el(root, val);

delete\_el\_tree(root, val);

}

int tree::get\_parent(tree\_el\*& cur, int val){

if(val < cur->get\_value()){

if(cur->get\_left() == nullptr) return cur->get\_value();

return get\_parent(cur->get\_left(), val);

} else if(val > cur->get\_value()){

if(cur->get\_right() == nullptr) return cur->get\_value();

return get\_parent(cur->get\_right(), val);

}

return -1;

}

int tree::get\_place(int val){

return get\_parent(root, val);

}

tree::~tree() {

delete\_tree(root);

}

Tree.cpp:

#ifndef \_TREE\_H

#define \_TREE\_H

#include <vector>

using namespace std;

class tree\_el{

tree\_el\* left;

tree\_el\* right;

int value;

public:

tree\_el(int new\_val);

int& get\_value();

tree\_el\*& get\_left();

tree\_el\*& get\_right();

friend bool operator<(tree\_el& lhs, tree\_el& rhs);

friend bool operator==(tree\_el& lhs, tree\_el& rhs);

};

class tree {

private:

tree\_el\* root;

void delete\_tree(tree\_el\*& cur);

static void print\_tree(tree\_el\*& cur, int h = 0);

bool find\_el(tree\_el\* cur, int val);

void insert\_el(tree\_el\*& cur, int new\_val);

tree\_el\*& delete\_rightmost(tree\_el\*& cur, int\* replacement);

void remove\_el(tree\_el\*& cur, int val);

void delete\_el\_tree(tree\_el\*& cur, int val);

int get\_parent(tree\_el\*& cur, int val);

public:

tree();

tree(int new\_val);

void insert(int new\_val);

bool find(int val);

void print();

void delete\_el(int val);

int get\_place(int val);

void get\_all(tree\_el\* cur, vector<int>& tmp);

vector<int> get\_all\_elems();

~tree();

};

#endif

Wrap\_zmq.cpp:

#include <tuple>

#include <cstring>

#include <iostream>

#include "socket.h"

#include "wrap\_zmq.h"

#include <unistd.h>

using namespace std;

void\* create\_zmq\_ctx(){

void\* context = zmq\_ctx\_new();

if(context == nullptr){

throw runtime\_error("Can not create new context.");

}

return context;

}

void destroy\_zmq\_ctx(void\* context){

sleep(1);

if(zmq\_ctx\_destroy(context) != 0){

throw runtime\_error("Can not destroy context.");

}

}

int get\_zmq\_socket\_type(SocketType type){

switch(type){

case SocketType::PUBLISHER:

return ZMQ\_PUB;

case SocketType::SUBSCRIBER:

return ZMQ\_SUB;

default:

throw runtime\_error("Undefined socket type.");

}

}

void\* create\_zmq\_socket(void\* context, SocketType type){

int zmq\_type = get\_zmq\_socket\_type(type);

void\* socket = zmq\_socket(context, zmq\_type);

if(socket == nullptr){

throw runtime\_error("Can not create socket.");

}

return socket;

}

void close\_zmq\_socket(void\* socket){

sleep(1);

if(zmq\_close(socket) != 0){

throw runtime\_error("Can not close socket.");

}

}

string create\_endpoint(EndpointType type, pid\_t id){

switch(type){

case EndpointType::PARENT\_PUB:

return "ipc:///tmp/parrent\_pub\_" + to\_string(id);

case EndpointType::CHILD\_PUB\_LEFT:

return "ipc:///tmp/child\_pub\_left\_" + to\_string(id);

case EndpointType::CHILD\_PUB\_RIGHT:

return "ipc:///tmp/child\_pub\_right" + to\_string(id);

default:

throw runtime\_error("Wrong Endpoint type.");

}

}

void bind\_zmq\_socket(void\* socket, string endpoint){

if(zmq\_bind(socket, endpoint.data()) != 0){

//zmq\_connect(socket, endpoint.data());

throw runtime\_error("Can not bind socket.");

}

}

void unbind\_zmq\_socket(void\* socket, string endpoint){

sleep(1);

if(zmq\_unbind(socket, endpoint.data()) != 0){

throw runtime\_error("Can not unbind socket.");

}

}

void connect\_zmq\_socket(void\* socket, string endpoint){

if(zmq\_connect(socket, endpoint.data()) != 0){

throw runtime\_error("Can not connect socket.");

}

zmq\_setsockopt(socket, ZMQ\_SUBSCRIBE, 0, 0);

}

void disconnect\_zmq\_socket(void\* socket, string endpoint){

if(zmq\_disconnect(socket, endpoint.data()) != 0){

throw runtime\_error("Can not disconnect socket.");

}

}

atomic<int> Message::counter;

Message::Message(){

command = CommandType::ERROR;

uniq\_num = counter++;

to\_up = false;

}

Message::Message(CommandType new\_command, int new\_to\_id, int new\_size, double\* new\_value, int new\_id)

: command(new\_command), to\_id(new\_to\_id), size(new\_size), uniq\_num(counter++), to\_up(false), create\_id(new\_id){

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

value[i] = new\_value[i];

}

}

Message::Message(CommandType new\_command, int new\_to\_id, int new\_id)

: command(new\_command), to\_id(new\_to\_id), uniq\_num(counter++), to\_up(false), create\_id(new\_id){}

bool operator==(const Message& lhs, const Message& rhs){

return tie(lhs.command, lhs.to\_id, lhs.create\_id, lhs.uniq\_num) ==

tie(rhs.command, rhs.to\_id, rhs.create\_id, rhs.uniq\_num);

}

int& Message::get\_create\_id(){

return create\_id;

}

int& Message::get\_to\_id(){

return to\_id;

}

void create\_zmq\_msg(zmq\_msg\_t\* zmq\_msg, Message& msg){

zmq\_msg\_init\_size(zmq\_msg, sizeof(msg));

memcpy(zmq\_msg\_data(zmq\_msg), &msg, sizeof(msg));

}

void send\_zmq\_msg(void\* socket, Message& msg){

zmq\_msg\_t zmq\_msg;

create\_zmq\_msg(&zmq\_msg, msg);

if(!zmq\_msg\_send(&zmq\_msg, socket, 0)){

throw runtime\_error("Can not send message.");

}

zmq\_msg\_close(&zmq\_msg);

}

Message get\_zmq\_msg(void\* socket){

zmq\_msg\_t zmq\_msg;

zmq\_msg\_init(&zmq\_msg);

if(zmq\_msg\_recv(&zmq\_msg, socket, 0) == -1){

return Message(); //ERROR message

}

Message msg;

memcpy(&msg, zmq\_msg\_data(&zmq\_msg), sizeof(msg));

zmq\_msg\_close(&zmq\_msg);

return msg;

}

Wrap\_zmq.h:

#ifndef \_WRAP\_ZMQ\_H

#define \_WRAP\_ZMQ\_H

#include <tuple>

#include <vector>

#include <atomic>

#include "zmq.h"

#include "wrap\_zmq.h"

using namespace std;

#define UNIVERSAL\_MSG -1

#define SERVER\_ID -2

#define PARENT\_SIGNAL -3

enum struct SocketType{

PUBLISHER,

SUBSCRIBER,

};

enum struct CommandType {

ERROR,

RETURN,

CREATE\_CHILD,

REMOVE\_CHILD,

EXEC\_CHILD,

};

enum struct EndpointType{

CHILD\_PUB\_LEFT,

CHILD\_PUB\_RIGHT,

PARENT\_PUB,

};

void\* create\_zmq\_ctx();

void destroy\_zmq\_ctx(void\* context);

int get\_zmq\_socket\_type(SocketType type);

void\* create\_zmq\_socket(void\* context, SocketType type);

void close\_zmq\_socket(void\* socket);

string create\_endpoint(EndpointType type, pid\_t id);

void bind\_zmq\_socket(void\* socket, string endpoint);

void unbind\_zmq\_socket(void\* socket, string endpoint);

void connect\_zmq\_socket(void\* socket, string endpoint);

void disconnect\_zmq\_socket(void\* socket, string endpoint);

#define MAX\_CAP 1000

class Message {

protected:

static std::atomic<int> counter;

public:

CommandType command = CommandType::ERROR;

int to\_id;

int create\_id;

int uniq\_num;

bool to\_up;

int size = 0;

double value[MAX\_CAP] = {0};

Message();

Message(CommandType new\_command, int new\_to\_id, int size, double\* new\_value, int new\_id);

Message(CommandType new\_command, int new\_to\_id, int new\_id);

friend bool operator==(const Message& lhs, const Message& rhs);

int& get\_create\_id();

int& get\_to\_id();

};

void create\_zmq\_msg(zmq\_msg\_t\* zmq\_msg, Message& msg);

void send\_zmq\_msg(void\* socket, Message& msg);

Message get\_zmq\_msg(void\* socket);

#endif

**Пример работы**

michael@michael-VirtualBox:~/Desktop/os/os\_lab\_6/src$ make

make: Nothing to be done for 'all\_done'.

michael@michael-VirtualBox:~/Desktop/os/os\_lab\_6/src$ ./server

17850 server started correctly!

17854: Client started. Id:0

create 5

OK:17857

17857: Client started. Id:5

create 1

OK:17860

17860: Client started. Id:1

create 9

OK:17863

17863: Client started. Id:9

exec 9 3 4 7 9

OK:9:20

remove 9

OK

pingall

OK:-1

17863: Exiting child...

create 9

OK:17867

17867: Client started. Id:9

create 8

OK:17871

17871: Client started. Id:8

create 10

OK:17874

17874: Client started. Id:10

exec 10 2 3 7

OK:10:10

exit

OK

17874: Exiting child...

17860: Exiting child...

17871: Exiting child...

17854: Exiting child...

17857: Exiting child...

17867: Exiting child...

michael@michael-VirtualBox:~/Desktop/os/os\_lab\_6/src$

**Вывод**

В ходе лабораторной работы я познакомился с системами очередей сообщений в целом и с библиотекой ZeroMQ в частности, узнал о протоколе ipc и сокетах для взаимодействия программ.

Для большей гибкости архитектуры программы пришлось отказаться от паттерна REP-REQ в пользу PUB-SUB, и написать “обёртки” над стандартными функциями библиотеки ZeroMQ для большего удобства работы с ними.