Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6 по курсу**

**«Операционные системы»**

**Управление серверами сообщений**

Студент: Петрухин Дмитрий Олегович

Группа: М8О –201Б-18

Вариант: 13

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2019

**Содержание**

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Основные файлы программы
5. Тестирование
6. Демонстрация работы программы
7. 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.   
Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

**Задание согласно варианту 26:**

Тип топологии: 3 (общее дерево).

Тип вычислительных комманд: 3 (таймер).

тип команд доступности узлов 2 (ping id).

**Общие сведения о программе**

Программа состоит из двух основных файлов и библиотеки, реализующей взаимодействия с узлами. Помимо этого используется библиотека zmq, которая реализует очередь сообщений.

1) main.cpp - программа, которая является управляющим узлом

2) child\_main.сpp - программа, которая подгружает необходимые данные из библиотеки во время исполнения программы.

3) os06\_api.cpp и os06\_api.h - реализация библиотеки для взаимодействия между узлами.

4) zmq.hpp - статическая библиотека для межпроцессорного общения.

Очередь сообщений - [компонент](https://ru.wikipedia.org/wiki/%25D0%259A%25D0%25BE%25D0%25BC%25D0%25BF%25D0%25BE%25D0%25BD%25D0%25B5%25D0%25BD%25D1%2582%25D0%25BD%25D0%25BE-%25D0%25BE%25D1%2580%25D0%25B8%25D0%25B5%25D0%25BD%25D1%2582%25D0%25B8%25D1%2580%25D0%25BE%25D0%25B2%25D0%25B0%25D0%25BD%25D0%25BD%25D0%25BE%25D0%25B5_%25D0%25BF%25D1%2580%25D0%25BE%25D0%25B3%25D1%2580%25D0%25B0%25D0%25BC%25D0%25BC%25D0%25B8%25D1%2580%25D0%25BE%25D0%25B2%25D0%25B0%25D0%25BD%25D0%25B8%25D0%25B5), используемый для [межпроцессного](https://ru.wikipedia.org/wiki/%25D0%259C%25D0%25B5%25D0%25B6%25D0%25BF%25D1%2580%25D0%25BE%25D1%2586%25D0%25B5%25D1%2581%25D1%2581%25D0%25BD%25D0%25BE%25D0%25B5_%25D0%25B2%25D0%25B7%25D0%25B0%25D0%25B8%25D0%25BC%25D0%25BE%25D0%25B4%25D0%25B5%25D0%25B9%25D1%2581%25D1%2582%25D0%25B2%25D0%25B8%25D0%25B5) или [межпотокового](https://ru.wikipedia.org/wiki/%25D0%259F%25D0%25BE%25D1%2582%25D0%25BE%25D0%25BA_%25D0%25B2%25D1%258B%25D0%25BF%25D0%25BE%25D0%25BB%25D0%25BD%25D0%25B5%25D0%25BD%25D0%25B8%25D1%258F) взаимодействия внутри одного [процесса](https://ru.wikipedia.org/wiki/%25D0%259F%25D1%2580%25D0%25BE%25D1%2586%25D0%25B5%25D1%2581%25D1%2581_(%25D0%25B8%25D0%25BD%25D1%2584%25D0%25BE%25D1%2580%25D0%25BC%25D0%25B0%25D1%2582%25D0%25B8%25D0%25BA%25D0%25B0)). Для [обмена сообщениями](https://ru.wikipedia.org/wiki/%25D0%259E%25D0%25B1%25D0%25BC%25D0%25B5%25D0%25BD_%25D1%2581%25D0%25BE%25D0%25BE%25D0%25B1%25D1%2589%25D0%25B5%25D0%25BD%25D0%25B8%25D1%258F%25D0%25BC%25D0%25B8) используется очередь. Очереди сообщений предоставляют асинхронный протокол передачи данных, означая, что отправитель и получатель сообщения не обязаны взаимодействовать с очередью сообщений одновременно. Размещённые в очереди сообщения хранятся до тех пор, пока получатель не получит их.

Очереди сообщений имеют неявные или явные ограничения на размер данных, которые могут передаваться в одном сообщении, и количество сообщений, которые могут оставаться в очереди.

Многие реализации очередей сообщений функционируют внутренне: внутри операционной системы или внутри приложения. Такие очереди существуют только для целей этой системы.

Другие реализации позволяют передавать сообщения между различными компьютерными системами, потенциально подключая несколько приложений и несколько операционных систем. Эти системы очередей сообщений обычно обеспечивают расширенную функциональность для обеспечения устойчивости, чтобы гарантировать, что сообщения не будут «потеряны» в случае сбоя системы.

ZMQ - это высокопроизводительная библиотека асинхронных сообщений, предназначенная для использования в распределенных или параллельных приложениях. Он обеспечивает очередь сообщений, но в отличие от промежуточного программного обеспечения, ориентированного на сообщения, система ZMQ может работать без выделенного посредника сообщений.

ZMQ поддерживает общие шаблоны обмена сообщениями (публикация / подпрограмма, запрос / ответ, клиент / сервер и т.д.) По различным транспортам (TCP, внутрипроцессный, межпроцессный, многоадресный, WebSocket и т.д.), что делает межпроцессный обмен сообщениями простым как обмен сообщениями между потоками.

**Сокеты**— название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Основные функции и вызовы:

1. **context\_t::context\_t(int io\_threads) -** Функция инициализирует контекст ZMQ. Аргумент io\_threads указывает размер пула потоков ZMQ для обработки операций ввода-вывода.

2) **socket\_t::socket\_t(context\_t &context, int type)** - Функция должна создать сокет ZMQ в указанном контексте и вернуть непрозрачный дескриптор вновь созданному сокету. Аргумент type указывает тип сокета, который определяет семантику связи через сокет.

Вновь созданный сокет изначально не связан и не связан ни с какими конечными точками. Чтобы установить поток сообщений, сокет должен быть сначала подключен по крайней мере к одной конечной точке с помощью zmq\_connect (3), или по крайней мере одна конечная точка должна быть создана для приема входящих соединений с помощью zmq\_bind (3).

Сокет типа ZMQ\_REQ используется клиентом для отправки запросов и получения ответов от службы.

ZMQ\_REP используется службой для получения запросов и отправки ответов клиенту.

3) **int execv(const char \**path*, char \*const *argv*[])** - функция **execv**() заменяет текущий образ процесса новым. **execv()** предоставляет новой программе список аргументов в виде массива указателей на строки, заканчивающиеся null. Первый аргумент, по соглашению, должен указать на имя, ассоциированное с файлом, который необходимо запустить. Массив указателей *должен* заканчиваться указателем null.

4) **void socket\_t::bind(const char \*endpoint)** - Функция должна создать конечную точку для приема соединений и связать ее с сокетом, на который ссылается аргумент сокета.

Аргумент endpoint - это строка, состоящая из двух частей: transport: // address. Транспортная часть определяет базовый транспортный протокол для использования. Значение адресной части зависит от выбранного основного транспортного протокола. Один из них TCP - это вездесущий, надежный, одноадресный транспорт.

5) **bool socket\_t::recv(message\_t \*msg, int flags = 0)** - Функция должна получить сообщение от сокета, на который ссылается аргумент сокета, и сохранить его в сообщении, на которое ссылается аргумент msg. Любой контент, ранее сохраненный в msg, должен быть надлежащим образом освобожден. Если в указанном сокете нет доступных сообщений, функция zmq\_recv () блокируется до тех пор, пока запрос не будет удовлетворен. Аргумент flags можно опустить, либо указать **ZMQ\_NOBLOCK** - Указывает, что операция должна выполняться в неблокирующем режиме. Если в указанном сокете нет доступных сообщений, функция zmq\_recv () должна завершиться с ошибкой, когда для errno установлено значение EAGAIN.

6) **zmq::message\_t msg(size\_t size)** - создается экземпляр класса msg размера size, который имеет доступ к функциям для создания, уничтожения и управления сообщениями ZMQ.

Функция должна распределять любые ресурсы, необходимые для хранения длинного байта сообщения, и инициализировать объект сообщения, на который ссылается msg, для представления вновь выделенного сообщения.

Реализация должна выбрать, хранить ли содержимое сообщения в стеке (маленькие сообщения) или в куче (большие сообщения). По соображениям производительности не должен очищать данные сообщения.

7) **bool socket\_t::send(message\_t &msg, int flags = 0)** - Функция помещает в очередь сообщение, на которое ссылается аргумент msg, для отправки в сокет, на который ссылается аргумент socket. Аргумент flags представляет собой комбинацию флагов, определенных ниже:

ZMQ\_NOBLOCK

Указывает, что операция должна выполняться в неблокирующем режиме. Если сообщение не может быть поставлено в очередь в сокете, функция socket\_t::send () должна завершиться с ошибкой, когда errno установлено в EAGAIN.

ZMQ\_SNDMORE

Указывает, что отправляемое сообщение является сообщением, состоящим из нескольких частей, и что последующие части сообщения должны следовать.

8) **void \*memcpy(void \*dest, const void \*source, size\_t count)** - Функция memcpy() копирует count символов из массива, на который указывает source, в массив, на который указывает dest. Если массивы перекрываются, поведение memcpy() не определено.

**Общий метод и алгоритм решения**.

Имеется управляющий узел и выполняющие узлы

1. Управляющий узел принимает команды, обрабатывает их и пересылает их дочерним узлам, а они передают сообщение дальше, если это необходим. В случае ошибок, управляющий узел выводит сообщение об этом.
2. Дочерние узлы проверяют им ли это сообщение передано, либо необходимо сообщение передать следующим дочерним узлам. Затем из дочерних узлов возвращается некоторое сообщение(об успехе либо ошибки выполнения переданной команды), которое пересылается вверх по дереву к управляющему узлу.
3. Если узел недоступен, то по истечении попытки подключиться к узлу будет выдано сообщение о недоступности узла и оно будет передано вверх по дереву.
4. При удалении узла, все его потомки рекурсивно удаляются.

**Листинг программы.**

**Файл main.cpp**

#include "os06\_api.h"

#include <iostream>

#include <unistd.h>

#include <zmq.hpp>

#include <string>

#include <zconf.h>

#include <csignal>

#include <sstream>

#include <set>

#include <algorithm>

#include <map>

#include <utility>

#include <memory> //shared\_ptr

#include <vector> //vector

struct TNode {

TNode(int id, std::shared\_ptr<TNode> son, std::shared\_ptr<TNode> brother)

: id\_(id), son\_(son), brother\_(brother) {}

int id\_;

std::shared\_ptr<TNode> son\_;

std::shared\_ptr<TNode> brother\_;

};

class TreeFunc {

public:

TreeFunc() = default;

~TreeFunc() = default;

bool Insert(int new\_id, int parent\_id) {

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

root\_ = std::make\_shared<TNode> (parent\_id, std::make\_shared<TNode> (new\_id, nullptr, nullptr), nullptr);

return true;

}

std::vector<std::pair<int, int>> path = GetPath(parent\_id);

if (path.empty()) {

return false;

}

int p;

path.erase(path.begin());//delete -1 root

std::shared\_ptr<TNode> node = root\_->son\_;

for (auto i : path) {

if(path.back().first == i.first){

p = i.second;///????

break;

}

if (i.second == 0 && node->son\_ != nullptr) {

node = node->son\_;

} else if(i.second == 1 && node->brother\_ != nullptr) {

node = node->brother\_;

}

}

if(p == 0){

node->son\_ = std::make\_shared<TNode>(new\_id, nullptr, nullptr);//new node аля

return true;

} else if (p == 1){

node->brother\_ = std::make\_shared<TNode>(new\_id, nullptr, nullptr);

return true;

}

return false;

}

bool Erase(int elem) {

std::vector<std::pair<int, int>> path = GetPath(elem);

if (path.empty()) {

return false;

}

path.erase(path.begin());

while(path.back().first != elem){

path.pop\_back();

}

path.pop\_back();

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

node = node->son\_;

for (auto i : path) {

if (i.second == 0){

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

std::shared\_ptr<TNode> ptr = node;

ptr = ptr->son\_;

node->son\_ = ptr->brother\_;

}//сохранить ссылку на node

node = node->son\_;

} else if(i.second == 1){

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

std::shared\_ptr<TNode> ptr = node;

ptr = ptr->brother\_;

node->brother\_ = ptr->brother\_;

}//сохранить ссылку на node

node = node->brother\_;

} else {

return false;

}

}

return true;

}

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

std::vector<std::pair<int, int>> v;

if (Search(root\_, id, v)) {

return v;

} else return v;

}

private:

bool Search(std::shared\_ptr<TNode> node, int id, std::vector<std::pair<int, int>>& v) {//b=0

if (node == nullptr) {

return false;

}

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

auto p = std::make\_pair(node->id\_, 0);//to son

v.push\_back(p);

return true;

}

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

auto p = std::make\_pair(node->id\_, 0);//положили сына, затем от этого сына идем по братьям

v.push\_back(p);

node = node->son\_;

p = std::make\_pair(node->id\_, 1);

v.push\_back(p);

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

node = node->brother\_;

auto p = std::make\_pair(node->id\_, 1);

v.push\_back(p);

}

return true;

}

auto p = std::make\_pair(node->id\_, 0);

v.push\_back(p);

std::shared\_ptr<TNode> son\_node = node;

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

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

if (Search(son\_node, id, v)) {

return true;

}

}

v.pop\_back();

p = std::make\_pair(node->id\_, 1);

v.push\_back(p);

std::shared\_ptr<TNode> brother\_node = node;

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

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

if (Search(brother\_node, id, v)) {

return true;

}

}

v.pop\_back();

return false;

}

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

};

int main() {

std::string command;

TreeFunc Topology;

int child\_pid;

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

std::map<int, zmq::socket\_t> sockets;

std::map<int, int> pids;

std::map<int, int> ports;//id and ports

while (std::cin >> command) {

std::string res;

int linger = 0;

if (command == "create") {

int new\_id, parent\_id;

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

if (parent\_id == -1 && pids.count(new\_id) == 0) {//добавляем в корень

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));

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

child\_pid = fork();

if(child\_pid == -1) {

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

child\_pid = 0;

exit(1);

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

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

} else {//родитель

//parent\_id = 0; лишнее

ports[new\_id] = port;

pids[new\_id] = child\_pid;

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

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

}

} else {

if(!Topology.GetPath(new\_id).empty()) {

std::cout << "Error: Node already exists\n";

continue;

}

std::vector<std::pair<int, int>> path = Topology.GetPath(parent\_id);

path.erase(path.begin());//убираем -1 корень

if (path.empty()){

std::cout << "Error: No parent node\n";

continue;

}

std::vector <int> path1;

for (auto i : path) {

if(i.second == 0){

path1.push\_back(i.first);

}

}

int id = path1.front();

path1.erase(path1.begin());

std::ostringstream msg\_stream;

msg\_stream << "create " << path1.size();

for (auto i : path1) {

msg\_stream << " " << i;

}

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

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

res = recieve\_message(sockets.at(id));

}

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

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

}

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

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

if (pids.size() == 0) {

std::cout << "Error: No such node\n";

continue;

}

int remove\_id;

std::cin >> remove\_id;

if (pids.count(remove\_id) != 0) {//child\_id у родителя хранить PID потомка

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

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

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

//SIGTERM - этот сигнал запрашивает остановку работы процесса.

//Сигнал SIGTERM завершает лишь те процессы, которые не обрабатывают его приход

kill(pids.at(remove\_id), SIGKILL); //немедленное завершение процесса

//eckb SIGTERM не сработал, то SIGKILL сработает точно

child\_pid = 0;

pids.erase(remove\_id);

sockets.at(remove\_id);

ports.erase(remove\_id);

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

Topology.Erase(remove\_id);

}

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

continue;

}

std::vector<std::pair<int, int>> path = Topology.GetPath(remove\_id);

if (path.empty()) {

std::cout << "Error: No such node\n";

continue;

}

path.erase(path.begin());

std::vector <int> path1;

for (auto i : path) {

if(i.second == 0){

path1.push\_back(i.first);

}

}

std::ostringstream msg\_stream;

int next\_id = path1.front();

path1.erase(path1.begin());

path1.pop\_back();

msg\_stream << "remove " << path1.size();

for (int i : path1) {

msg\_stream << " " << i;

}

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

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

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

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

Topology.Erase(remove\_id);

}

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

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

int id;

std::string subcommand;

std::cin >> id >> subcommand;

std::vector<std::pair<int, int>> path = Topology.GetPath(id);

if (path.empty()) {

std::cout << "Error: No such node\n";

continue;

}

path.erase(path.begin());

std::vector <int> path1;

for (auto i : path) {

if(i.second == 0){

path1.push\_back(i.first);

}

}

int next\_id = path1.front();

path1.erase(path1.begin());

std::ostringstream msg\_stream;

msg\_stream << "exec " << id << " " << subcommand << " "<< path1.size();

for (int i : path1) {

msg\_stream << " " << i;

}

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

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

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

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

int ping\_id;

std::cin >> ping\_id;

std::vector<std::pair<int, int>> path = Topology.GetPath(ping\_id);

if(path.empty()) {

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

continue;

}

path.erase(path.begin());

std::vector <int> path1;

for (auto i : path) {

if(i.second == 0){

path1.push\_back(i.first);

}

}

std::ostringstream msg\_stream;

int next\_id = path1.front();

path1.erase(path1.begin());

msg\_stream << "ping " << path1.size();

for (int i : path1) {

msg\_stream << " " << i;

}

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

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

if(res.substr(0,5) == "Error") {

std::cout << "Ok: 0\n";

continue;

}

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

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

break;

}

}//while

return 0;

}//main

**child\_main.cpp**

#include <iostream>

#include <zmq.hpp>

#include <string>

#include <sstream>

#include <zconf.h>

#include <exception>

#include <csignal>

#include <map>

#include <ctime>

#include <utility>

#include "os06\_api.h"

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

time\_t seconds, now;

time\_t stop = 0;

bool bool\_time;

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);// сыну, брату и отцу

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

parent\_socket.connect(get\_port\_name(parent\_port)); // подключаемся к сокету родителя

std::map<int, zmq::socket\_t> sockets;

std::map<int, int> pids;

std::map<int, int> ports;

while (true) {

int linger = 0;

std::string request;

request = recieve\_message(parent\_socket);

std::istringstream command\_stream(request);

std::string command;

command\_stream >> command;

if (command == "pid"){

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

send\_message(parent\_socket, answer);

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

int new\_id, size;

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.size() == 0) {

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));

int port = accept\_connection(sockets.at(new\_id));//создаем точку для соединений с с сыном

int pid = fork();

if (pid == -1) {

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

continue;

} else if (pid == 0) {

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

} else {

//сохраняем порт нового процесса и его pid

ports[new\_id] = port;

pids[new\_id] = pid;

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

send\_message(parent\_socket, recieve\_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(int i : path) {

msg\_stream << " " << i;

}

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

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

//std::string res = recieve\_message(sockets.at(next\_id));

//res = res + " " + msg\_stream.str();

//send\_message(parent\_socket, res);

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

}

} 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 (size == 0) {

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

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

continue;

}

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

recieve\_message(sockets.at(remove\_id));

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

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

pids.erase(remove\_id);

sockets.at(remove\_id).disconnect(get\_port\_name(ports[remove\_id]));

ports.erase(remove\_id);

sockets.erase(remove\_id);

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

} else {

int next\_id = path.front();

path.erase(path.begin());

std::ostringstream msg\_stream;

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

if(path.size() != 0){

for (int i : path) {

msg\_stream << " " << i;

}

}

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

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

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

}

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

int size;

std::string subcommand;

command\_stream >> id >> subcommand >> size;

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

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

command\_stream >> path[i];

}

if (size == 0) { //дошли до нужного узла

std::ostringstream msg\_stream;

if (subcommand == "start") {

seconds = time(NULL);

bool\_time = true;

msg\_stream << "Ok:" << id;

} else if (subcommand == "time") {

if(!bool\_time){

now = stop;

} else {

now = time(NULL) - seconds + stop;

}

int time = now \* 1000;

msg\_stream << "Ok:" << id << ": " << std::to\_string(time);

} else if (subcommand == "stop"){

stop = time(NULL) - seconds;

bool\_time = false;

msg\_stream << "Ok:" << id;

}

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

} else {

int next\_id = path.front();

path.erase(path.begin());

std::ostringstream msg\_stream;

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

for(int i : path) {

msg\_stream << " " << i;

}

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

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

}

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

int size;

command\_stream >> size;

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

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

command\_stream >> path[i];

}

if(size == 0){//дошли до нужного нам сокета

std::ostringstream msg\_stream;

msg\_stream << "Ok:" << id;

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

} else {

int next\_id = path.front();

path.erase(path.begin());

std::ostringstream msg\_stream;

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

for(int i: path) {

msg\_stream << " " << i;

}

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

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

}

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

for (auto& [child\_id, child\_socket] : sockets) {

send\_message(child\_socket, "kill");

recieve\_message(child\_socket);

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

kill(pids.at(child\_id), SIGKILL);

}

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

} else {

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

}

if (parent\_port == 0) {//если будет кол-во узлов больше int

break;

}

}

}

**os06\_api.cpp**

#include <string>

#include "os06\_api.h"

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

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

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

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

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

execv("./child\_main", args);

}

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 recieve\_message(zmq::socket\_t& socket) {

zmq::message\_t message;

bool ok;

try {

ok = socket.recv(&message);

} catch (...) {

ok = false;

}

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

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

return "Error: Node is not available";

}

return recv\_message;

}

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));

break;

} catch(...) {

port++;

}

}

return port;

}

**CMakeLists.txt**

cmake\_minimum\_required(VERSION 3.14)

project(os\_lab\_06)

set(CMAKE\_BUILD\_TYPE Debug)

set(CMAKE\_CXX\_STANDARD 17)

find\_package(PkgConfig)

find\_path(ZeroMQ\_INCLUDE\_DIR

NAMES zmq.hpp

PATHS ${PC\_ZeroMQ\_INCLUDE\_DIRS}

)

find\_library(ZeroMQ\_LIBRARY

NAMES zmq

PATHS ${PC\_ZeroMQ\_LIBRARY\_DIRS}

)

add\_library(api os06\_api.cpp os06\_api.h)

target\_link\_libraries(api PUBLIC ${ZeroMQ\_LIBRARY})

target\_include\_directories(api PUBLIC ${ZeroMQ\_INCLUDE\_DIR})

add\_executable(child\_main child\_main.cpp)

target\_include\_directories(child\_main PUBLIC ${ZeroMQ\_INCLUDE\_DIR})

target\_link\_libraries(child\_main PUBLIC ${ZeroMQ\_LIBRARY})

target\_link\_libraries(child\_main PUBLIC api)

add\_executable(main main.cpp)

target\_include\_directories(main PUBLIC ${ZeroMQ\_INCLUDE\_DIR})

target\_link\_libraries(main PUBLIC ${ZeroMQ\_LIBRARY})

target\_link\_libraries(main PUBLIC api)

**Тестирование.**

**test1:** Запуск программы, которая использует библиотеку уже на этапе компиляции.

create 1 -1

create 2 1

create 3 -1

create 4 3

create 5 3

create 6 5

exec 6 start

create 7 6

exec 6 time

exec 6 stop

remove 7

exec 6 time

create 8 3

create 9 8

exec 9 start

create 10 9

create 11 9

create 12 11

create 13 11

create 14 13

create 15 14

ping 5

exec 15 start

exec 11 start

exec 9 time

remove 6

ping 6

exec 11 time

remove 11

exec 15 time

remove 3

ping 15

**test2:**

**create 1 -1**

**create 2 -1**

**create 3 2**

**create 4 3**

**ping 4**

**kill 46489(4 узел)**

**ping 4**

create 4 2

**Демонстрация работы программы.**

**test1:**

➜ src git:(master) ✗ ./main

create 1 -1

Ok:46446

create 2 1

Ok:46447

create 3 -1

Ok:46448

create 4 3

Ok:46449

create 5 3

Ok:46450

create 6 5

Ok:46451

exec 6 start

Ok:6

create 7 6

Ok:46452

exec 6 time

Ok:6: 10000

exec 6 stop

Ok:6

remove 7

Ok

exec 6 time

Ok:6: 15000

create 8 3

Ok:46453

create 9 8

Ok:46454

exec 9 start

Ok:9

create 10 9

Ok:46455

create 11 9

Ok:46456

create 12 11

Ok:46457

create 13 11

Ok:46458

create 14 13

Ok:46459

create 15 14

Ok:46460

ping 5

Ok:6

exec 15 start

Ok:15

exec 11 start

Ok:11

exec 9 time

Ok:9: 44000

remove 6

Ok

ping 6

Error: Not found

exec 11 time

Ok:11: 13000

remove 11

Ok

exec 15 time

Error: No such node

remove 3

Ok

ping 15

Error: Not found

**test2:**

create 1 -1

Ok:46485

create 2 -1

Ok:46487

create 3 2

Ok:46488

create 4 3

Ok:46489

ping 4

Ok:4

ping 4

Ok: 0

create 4 2

Error: Node already exists

**Вывод.**

Очередь сообщений очень полезная технология для различных ситуаций. Они являются важным компонентом в приложениях и архитектурах программного обеспечения за счет масштабируемости(хорошо распределяют процессы обработки информации), асинхронности и удобного интерфейса обмена данными между процессами.