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»**

**Тема работы**

**“Очереди сообщений”**

Студент: Зинин Владислав Владимирович

Группа: М8О-208Б-20

Вариант: 34

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

**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/frankeloff

**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.  
  
Вариант 34. Команды:  
create id  
exec id time (локальный таймер)  
ping id

**Общие сведения о программе**

Для выполнения данной лабораторной работы я предварительно реализовал 6 файлов с кодом:  
  
N\_Tree.h – Дерево общего вида, необходимое по заданию. Это, скорее, вспомогательный файл.  
  
my\_zmq.h - отдельный файл для функций zero-message queue, сделанный для удобства работы и во избежание загрязнения кода.  
  
calculation\_node.h - файл с основной реализацией команд, выполняемых программой.  
  
control\_node.cpp – мэйн и по совместительству родитель вычислительной ноды.

CMakeList.txt – файл для сборки.

**Общий метод и алгоритм решения**

Сборка происходит с помощью cmake.  
Сначала запускается control\_node, далее мы создаем главную вычислительную ноду, родителем которой будет являться control\_node, т.е. мы делаем fork(), один процесс (дочерний) кидаем в calculation\_node, второй(родитель) остается в control\_node. Потом calculation\_node отсылает сообщение control\_node об успешном создании и ждет последующих команд, а control\_node, в свою очередь, добавляет ноду в дерево общего вида. Следующие ноды создаются по следующему принципы: contrlon\_node посылает письмо calculation\_node о том, что необходимо создать ноду, та в свою очередь либо посылает это сообщение дальше вниз к другим нодам, либо же создает сама (в зависимости от того, заняты ли все возможные связи для данной ноды (parent, child, brother)). Локальный таймер и pind id реализованы схожим образом. Ping id отсылает сообщение необходимой ноде, которая должна ответить тем, что приняла сообщение. Локальный таймер так же идет сообщением до необходимой ноды и в зависимости от команды либо стартует, либо останавливается, либо выводится.

**Исходный код**

**calculation\_node.cpp**

#include "my\_zmq.h"

#include <iostream>

#include <map>

#include <ctime>

#include <unistd.h>

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

assert(argc == 2);

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

void \*parent\_context = zmq\_ctx\_new(); //создаем контекст

void \*parent\_socket = zmq\_socket(parent\_context, ZMQ\_PAIR); //создаем сокет

if (zmq\_connect(parent\_socket, ("tcp://localhost:" + std::to\_string(PORT\_BASE + id)).c\_str()) != 0) {

std::cout << id << ": Can't connect to parent" << std::endl;

};

std::pair<void \*, void \*> brother, child; // <ctx, sock>

int child\_id = -1, brother\_id = -1;

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

clock\_t start = 0, now = 0;

bool has\_child = false, has\_brother = false, awake = true, timer = false;

while (awake) {

msg\_t token({fail, 0, 0});

zmq::receive\_msg(token, parent\_socket);

auto \*reply = new msg\_t({fail, id, id});

if (token.action == create) {

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

if (!has\_child) {

zmq::init\_ctx\_socket(child.first, child.second);

if (zmq\_bind(child.second, ("tcp://\*:" + std::to\_string(PORT\_BASE + token.id)).c\_str()) != 0) {

perror("Bind:");

exit(EXIT\_FAILURE);

}

int fork\_id = fork();

if (fork\_id == 0) {

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

exit(EXIT\_FAILURE);

} else if (fork\_id > 0) {

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

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

bool ok = zmq::send\_receive\_wait(msg\_ping, reply\_ping, child.second);

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

if (ok) {

reply->action = success;

child\_id = token.id;

has\_child = true;

} else {

if (zmq\_close(child.second) != 0) {

exit(EXIT\_FAILURE);

}

if (zmq\_ctx\_destroy(child.first) != 0) {

exit(EXIT\_FAILURE);

}

}

} else {

perror("Fork error:");

exit(EXIT\_FAILURE);

}

} else {

auto \*son\_msg = new msg\_t({create\_brother, child\_id, token.id});

zmq::send\_receive\_wait(son\_msg, \*reply, child.second);

}

} else {

if (has\_child) {

zmq::send\_receive\_wait(&token, \*reply, child.second);

if (reply->action == success) {

zmq::send\_msg\_no\_wait(reply, parent\_socket);

continue;

}

}

if (has\_brother) {

zmq::send\_receive\_wait(&token, \*reply, brother.second);

}

}

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

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

if (!has\_brother) {

zmq::init\_ctx\_socket(brother.first, brother.second);

if (zmq\_bind(brother.second, ("tcp://\*:" + std::to\_string(PORT\_BASE + token.id)).c\_str()) != 0) {

perror("Bind:");

exit(EXIT\_FAILURE);

}

int fork\_id = fork();

if (fork\_id == 0) {

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

perror("Fork");

exit(EXIT\_FAILURE);

} else if (fork\_id > 0) {

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

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

bool ok = zmq::send\_receive\_wait(msg\_ping, reply\_ping, brother.second);

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

if (ok) {

reply->action = success;

brother\_id = token.id;

has\_brother = true;

} else {

if (zmq\_close(child.second) != 0) {

exit(EXIT\_FAILURE);

}

if (zmq\_ctx\_destroy(child.first) != 0) {

exit(EXIT\_FAILURE);

}

}

} else {

perror("Fork");

exit(EXIT\_FAILURE);

}

} else {

auto \*brother\_msg = new msg\_t({create\_brother, brother\_id, token.id});

zmq::send\_receive\_wait(brother\_msg, \*reply, brother.second);

}

} else {

if (has\_child) {

zmq::send\_receive\_wait(&token, \*reply, child.second);

if (reply->action == success) {

zmq::send\_msg\_no\_wait(reply, parent\_socket);

continue;

}

}

if (has\_brother) {

zmq::send\_receive\_wait(&token, \*reply, brother.second);

}

}

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

if (token.id == id) {

reply->action = success;

} else {

if (has\_child) {

zmq::send\_receive\_wait(&token, \*reply, child.second);

if (reply->action == success) {

zmq::send\_msg\_no\_wait(reply, parent\_socket);

continue;

}

}

if (has\_brother) {

zmq::send\_receive\_wait(&token, \*reply, brother.second);

}

}

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

if (token.id == id) {

switch (token.parent\_id) {

case 1: {

timer = true;

start = clock();

break;

}

case -1: {

timer = false;

break;

}

case 0: {

now = clock();

timer ? reply->parent\_id = (int) (now - start) : reply->parent\_id = -1;

break;

}

}

reply->action = success;

} else{

if (has\_child) {

zmq::send\_receive\_wait(&token, \*reply, child.second);

if (reply->action == success) {

zmq::send\_msg\_no\_wait(reply, parent\_socket);

continue;

}

}

if (has\_brother) {

zmq::send\_receive\_wait(&token, \*reply, brother.second);

}

}

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

reply->action = kill;

if(has\_child){

zmq::send\_msg\_no\_wait(reply, child.second);

zmq\_close(child.second);

zmq\_ctx\_destroy(child.first);

}

if(has\_brother){

zmq::send\_msg\_no\_wait(reply, brother.second);

zmq\_close(brother.second);

zmq\_ctx\_destroy(child.first);

}

zmq\_close(parent\_socket);

zmq\_ctx\_destroy(parent\_context);

exit(0);

}

zmq::send\_msg\_no\_wait(reply, parent\_socket);

}

}

**control\_node.cpp**

#include <unistd.h>

#include <vector>

#include <zmq.h>

#include "my\_zmq.h"

#include "N\_Tree.h"

using node\_id\_type = int;

int main() {

auto \*topology = new N\_Tree<node\_id\_type>(-1);

void \*socket, \*ctx;// [context, socket]

std::string s;

node\_id\_type id;

std::cout << "\t\tUsage" << std::endl;

std::cout << "Create id parent: create calculation node (use parent = -1 if parent is control node)" << std::endl;

std::cout << "Ping id: ping calculation node with id $id" << std::endl;

std::cout << "Exec id n k1,k2,...,kn: sum k1,k2,...,kn" << std::endl;

std::cout << "Print end 1: exit" << std::endl;

std::cout << "Print print 1: print topology" << std::endl;

while (std::cin >> s >> id) {

if (s == "create") {

node\_id\_type parent\_id;

std::cin >> parent\_id;

if (parent\_id == -1) {

if (topology->get\_data() != -1) {

std::cout << "Root already exists" << std::endl;

continue;

}

zmq::init\_ctx\_socket(ctx, socket);

zmq\_bind(socket, ("tcp://\*:" + std::to\_string(PORT\_BASE + id)).c\_str()); //связываем сокет с локальной конечной точкой

int fork\_id = fork();

if (fork\_id == 0) {

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

return 0;

} else {

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

msg\_t reply({fail, id, id});

if (zmq::send\_receive\_wait(token, reply, socket) and reply.action == success) {

topology->set\_data(id);

} else {

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

exit(EXIT\_FAILURE);

}

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

exit(EXIT\_FAILURE);

}

}

}

} else if (search\_tree(parent\_id, topology) == nullptr) {

std::cout << "Error: Parent not found" << std::endl;

continue;

} else {

if (search\_tree(id, topology) != nullptr) {

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

continue;

}

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

msg\_t reply({fail, id, id});

if (zmq::send\_receive\_wait(msg, reply, socket) and reply.action == success) {

topology->insert(id, parent\_id);

} else {

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

}

}

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

if (search\_tree(id, topology) == nullptr) {

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

continue;

}

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

msg\_t reply({fail, id, id});

if (zmq::send\_receive\_wait(token, reply, socket) and reply.action == success) {

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

} else {

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

}

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

std::string cmd;

std::cin >> cmd;

if (cmd == "start" or cmd == "stop") {

auto \*token = new msg\_t({exec, 2, id});

(cmd == "start") ? token->parent\_id = 1 : token->parent\_id = -1;

msg\_t reply({fail, id, id});

if (zmq::send\_receive\_wait(token, reply, socket) and reply.action == success) {

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

} else {

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

}

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

auto \*token = new msg\_t({exec, 0, id});

msg\_t reply({fail, 0, 0});

if (zmq::send\_receive\_wait(token, reply, socket) and reply.action == success){

if (reply.parent\_id != -1) {

std::cout << "OK: " << id << ": " << reply.parent\_id << std::endl;

} else{

std::cout << "Error: Timer isn't started" << std::endl;

}

}

}

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

if(id == 1){

auto \*token = new msg\_t({kill, 0, id});

zmq::send\_msg\_no\_wait(token, socket);

delete\_topology(topology);

break;

}

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

if (id == 1) traverse\_inorder(topology);

}

}

zmq\_close(socket);

zmq\_ctx\_destroy(ctx);

return 0;

}

**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,

create\_brother = 3,

ping = 4,

exec = 5,

kill = 6

};

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

const int PORT\_BASE = 8000;

const int WAIT\_TIME = 5000;

struct msg\_t {

actions\_t action;

int parent\_id, id;

};

namespace zmq {

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

int rc;

context = zmq\_ctx\_new();

socket = zmq\_socket(context, ZMQ\_PAIR); // Сокет типа ZMQ\_PAIR может быть подключен только к одному узлу одновременно, вовзращает десриптор

rc = zmq\_setsockopt(socket, ZMQ\_RCVTIMEO, &WAIT\_TIME, sizeof(int)); // устанавливаем параметры ожидания получения и отправки

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 rec = 0;

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

rec = zmq\_msg\_recv(&reply, socket, 0); // сохраняем ответ из сокета, возвращаем кол-во байт

assert(rec == sizeof(T));

reply\_data = \*(T \*)zmq\_msg\_data(&reply); // возвращаем указатель на содержимое reply

rec = zmq\_msg\_close(&reply); // sdfsd

assert(rec == 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);// сохраняем ответ из сокета, возвращаем кол-во байт

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)); // выделяем любые ресурсы, необходимые для хранения сообщения размером в байтах

assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL); //инициализировать объект сообщения, на который ссылается msg, для представления содержимого, на которое ссылается буфер

assert(rc == 0);

rc = zmq\_msg\_send(&message, socket, ZMQ\_DONTWAIT); // отправляем сообщение в сокет (операция должна выполняться в неблокирующем режиме)

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); // инициализаци пустого сообщения

rc = zmq\_msg\_init\_size(&message, sizeof(T)); // выделяем любые ресурсы, необходимые для хранения сообщения размером в байтах

assert(rc == 0);

rc = zmq\_msg\_init\_data(&message, token, sizeof(T), NULL, NULL); // инициализировать объект сообщения, на который ссылается msg, для представления содержимого, на которое ссылается буфер

assert(rc == 0);

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

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\_

**N\_Tree.h**

#ifndef N\_TREE\_H

#define N\_TREE\_H

#include<iostream>

template<typename T>

class N\_Tree {

public:

N\_Tree() = default;

explicit N\_Tree(T dat) : data(dat), son(nullptr), brother(nullptr) {};

void insert(T id, T parent) { // добавление узла по родителю

if (data == parent){

if (son == nullptr){

son = new N\_Tree<T>(id);

return;

}

auto \*tree = son;

while (tree->brother != nullptr){

tree = tree->brother;

}

tree->brother = new N\_Tree<T>(id);

return;

}

if (son != nullptr){

son->insert(id, parent);

}

if (brother != nullptr){

brother->insert(id, parent);

}

}

T get\_data() {

return this->data;

}

void set\_data(T dat) {

this->data = dat;

}

bool remove(T id) {

if (this->son->data == id) {

this->son = nullptr;

return true;

}

if (this->brother->data == id) {

this->brother = nullptr;

return true;

}

bool ok = false;

if (this->son != nullptr) {

ok = this->son->remove(id);

}

if (this->brother != nullptr) {

ok = ok || this->brother->remove(id);

}

return ok;

}

template<typename S>

friend N\_Tree<S> \*search\_tree(S search, N\_Tree<S> \*t);

template<typename F>

friend void traverse\_inorder(N\_Tree<F> \*t);

template<typename H>

friend void delete\_topology(N\_Tree<H> \*t);

private:

T data;

N\_Tree<T> \*son = nullptr, \*brother = nullptr;

};

template<typename T>

N\_Tree<T> \*search\_tree(T search, N\_Tree<T> \*t) {// поиск узла по значению

if (t == nullptr){

return t;

}

if (t->data == search) {

return t;

}

N\_Tree<T> \*tree = nullptr;

if (t->son != nullptr) {

tree = search\_tree(search, t->son);

}

if (t->brother != nullptr and tree == nullptr ) {

tree = search\_tree(search, t->brother);

}

return tree;

}

template<typename F>

void traverse\_inorder(N\_Tree<F> \*t){

if (t->son != nullptr)

traverse\_inorder(t->son);

std::cout << t->data << std::endl;

if (t->brother != nullptr)

traverse\_inorder(t->brother);

}

template<typename H>

void delete\_topology(N\_Tree<H> \*t){

if (t->son != nullptr){

delete\_topology(t->son);

}

if(t->brother != nullptr){

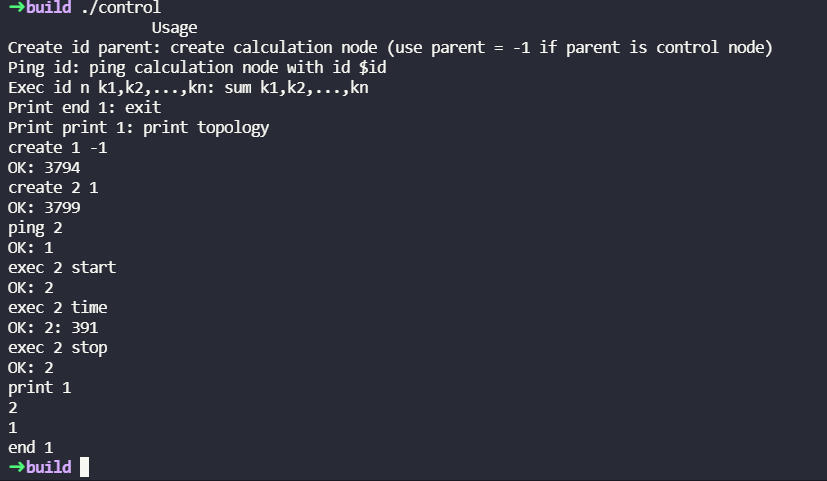
delete\_topology(t->brother);

}

delete t;

}

#endif //N\_TREE\_H

**Демонстрация работы программы**

**Вывод**Данная лабораторная работа была очень и очень непростой. Во время ее выполнения я полностью осознал концепцию очередей сообщений на основе zero message queue. На мой взгляд, это достойное завершение курса “Операционных систем”.