Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №5-7 по курсу**

**«Операционные системы»**

Студент: Корнев Максим Сергеевич

Группа: М8О–212Б–22

Вариант: 37

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2023.

**Постановка задачи**

## Цель работы

Целью является приобретение практических навыков в:

1. Управлении серверами сообщений (№5)

2. Применение отложенных вычислений (№6)

3. Интеграция программных систем друг с другом (№7)

## 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

*Примечания: создание нового управляющего узла осуществляется пользователем программы при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы.*

**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.

*Примечание: выполнение команд должно быть асинхронным. Т.е. пока выполняется команда на одном из вычислительных узлов, то можно отправить следующую команду на другой вычислительный узел.*

*Вариант 37:*

**Топология 1.**

Все вычислительные узлы находятся в списке. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

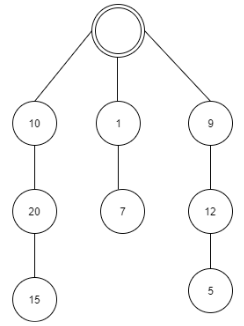
**Набор команд 2 (локальный целочисленный словарь).**

Формат команды сохранения значения: exec id name value

id – целочисленный идентификатор вычислительного узла, на который отправляется команда name – ключ, по которому будет сохранено значение (строка формата [A-Za-z0-9]+)

value – целочисленное значение

Формат команды загрузки значения: exec id name

Пример:

> exec 10 MyVar

Ok:10: 'MyVar' not found

> exec 10 MyVar 5

Ok:10

> exec 12 MyVar

Ok:12: 'MyVar' not found

> exec 10 MyVar

Ok:10: 5

> exec 10 MyVar 7

Ok:10

> exec 10 MyVar

Ok:10: 7

*Примечания: Можно использовать std:map.*

**Команда 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

**Реализация**

**child.cpp**

#include "zmq.hpp"

#include <sstream>

#include <string>

#include <iostream>

#include <zconf.h>

#include <vector>

#include <map>

#include <signal.h>

#include <fstream>

#include <algorithm>

#include <thread>

using namespace std;

string adr, adrChild;

zmq::context\_t context(1); // параметр - количество потоков

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

zmq::context\_t contextChild(1);

zmq::socket\_t childSocket(contextChild, ZMQ\_REP);

vector<int> ChildrenId;

std::map<string, int> dict;

int idThisNode, childNodeId;

void sendMessage(const string& messageString, zmq::socket\_t& socket) {

    zmq::message\_t messageBack(messageString.size());

    memcpy(messageBack.data(), messageString.c\_str(), messageString.size());

    if (!socket.send(messageBack)) {

        cerr << "Error: can't send message from node with pid " << getpid() << endl;

    }

}

void mapAccess(string receivedMessage, string idProcString, int idProc) {

    cout << "Function started in thread: " << std::this\_thread::get\_id() << endl;

    sleep(1);

    int value;

    bool isSpace = false;

    string key, valueStr, returnMessage;

    vector<int> answer;

    for (int i = 6 + idProcString.size(); i < receivedMessage.size(); ++i) {

        if (receivedMessage[i] != ' ' && !isSpace) {

            key += receivedMessage[i];

        } else {

            isSpace = true;

            valueStr += receivedMessage[i];

        }

    }

    if (isSpace) {

        value = stoi(valueStr);

        dict[key] = value;

        returnMessage += idProcString;

    } else {

        if (dict[key]){

            returnMessage += to\_string(dict[key]);

        } else {

            returnMessage = "'" + key + "'" + " not found";

        }

    }

    cout << endl << "OK: " << returnMessage << endl;

    cout << "Function completed in thread: " << std::this\_thread::get\_id() << endl;

}

void funcCreate(string receivedMessage) {

    bool isSpace = false;

    int idNewProc, parentIdNewProc;

    string idNewProcString, parentIdNewProcString;

    for (int i = 7; i < receivedMessage.size(); ++i) {

        if (receivedMessage[i] == ' ') {

            isSpace = true;

        } else if (receivedMessage[i] != ' ' && !isSpace) {

            idNewProcString += receivedMessage[i];

        } else if (receivedMessage[i] != ' ' && isSpace) {

            parentIdNewProcString += receivedMessage[i];

        }

    }

    idNewProc = stoi(idNewProcString);

    parentIdNewProc = stoi(parentIdNewProcString);

    if (idNewProc == idThisNode) {

        sendMessage("Error: Already exists", mainSocket);

    } else {

        if (childNodeId == 0 && parentIdNewProc == idThisNode) {

            childNodeId = idNewProc;

            childSocket.bind(adrChild + to\_string(childNodeId));

            adrChild += to\_string(childNodeId);

            char\* adrChildTmp = new char[adrChild.size() + 1];

            memcpy(adrChildTmp, adrChild.c\_str(), adrChild.size() + 1);

            char\* childIdTmp = new char[to\_string(childNodeId).size() + 1];

            memcpy(childIdTmp, to\_string(childNodeId).c\_str(), to\_string(childNodeId).size() + 1);

            char\* args[] = {"./child", adrChildTmp, childIdTmp, NULL};

            int procesId = fork();

            if (procesId == 0) {

                execv("./child", args);

                ChildrenId.push\_back(idNewProc);

            } else if (procesId < 0) {

                cerr << "Error in forking in node with pid: " << getpid() << endl;

            } else {

                zmq::message\_t messageFromNode;

                if (!childSocket.recv(messageFromNode)) {

                    cerr << "Error: can't receive message from child node in node with pid:" << getpid()

                         << endl;

                }

                if (!mainSocket.send(messageFromNode)) {

                    cerr << "Error: can't send message to main node from node with pid:" << getpid() << endl;

                }

            }

            delete[] adrChildTmp;

            delete[] childIdTmp;

        } else if (childNodeId == 0 && parentIdNewProc != idThisNode) {

            sendMessage("Error: there is no such parent", mainSocket);

        } else if (childNodeId != 0 && parentIdNewProc == idThisNode) {

            sendMessage("Error: this parent already has a child", mainSocket);

        } else {

            sendMessage(receivedMessage, childSocket);

            zmq::message\_t message;

            if (!childSocket.recv(message)) {

                cerr << "Error: can't receive message from child node in node with pid: " << getpid() << endl;

            }

            if (!mainSocket.send(message)) {

                cerr << "Error: can't send message to main node from node with pid: " << getpid() << endl;

            }

        }

    }

}

void funcExec(string receivedMessage) {

    int idProc;

    string idProcString;

    for (int i = 5; i < receivedMessage.size(); ++i) {

        if (receivedMessage[i] != ' ') {

            idProcString += receivedMessage[i];

        } else {

            break;

        }

    }

    idProc = stoi(idProcString);

    if (idProc == idThisNode) {

        thread workThread(mapAccess, receivedMessage, idProcString, idProc);

        workThread.detach();

        string returnMessage = "The child process performs calculations and outputs them when it finishes calculations";

        sendMessage(returnMessage, mainSocket);

    } else {

        if (childNodeId == 0) {

            sendMessage("Error: id: Not found", mainSocket);

        } else {

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

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

            if (!childSocket.send(message)) {

                cerr << "Error: can't send message to child node from node with pid: " << getpid() << endl;

            }

            if (!childSocket.recv(message)) {

                cerr << "Error: can't receive message from child node in node with pid: " << getpid() << endl;

            }

            if (!mainSocket.send(message)) {

                cerr << "Error: can't send message to main node from node with pid: " << getpid() << endl;

            }

        }

    }

}

void funcPing(string receivedMessage) {

    int idProc;

    string idProcString;

    for (int i = 5; i < receivedMessage.size(); ++i) {

        if (receivedMessage[i] != ' ') {

            idProcString += receivedMessage[i];

        } else {

            break;

        }

    }

    idProc = stoi(idProcString);

    if (idProc == idThisNode) {

        sendMessage("OK: 1", mainSocket);

    } else {

        if (childNodeId == 0) {

            sendMessage("OK: 0", mainSocket);

        } else {

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

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

            childSocket.send(message);

            childSocket.recv(message);

            mainSocket.send(message);

        }

    }

}

void funcKill(string receivedMessage) {

    int idProcToKill;

    string idProcToKillString;

    for (int i = 5; i < receivedMessage.size(); ++i) {

        if (receivedMessage[i] != ' ') {

            idProcToKillString += receivedMessage[i];

        } else {

            break;

        }

    }

    idProcToKill = stoi(idProcToKillString);

    if (childNodeId == 0) {

        sendMessage("Error: there isn`t node with this id child", mainSocket);

    } else {

        if (childNodeId == idProcToKill) {

            sendMessage("OK: " + to\_string(childNodeId), mainSocket);

            sendMessage("DIE", childSocket);

            childSocket.unbind(adrChild);

            adrChild = "tcp://127.1.1.1:300";

            childNodeId = 0;

        } else {

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

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

            childSocket.send(message);

            childSocket.recv(message);

            mainSocket.send(message);

        }

    }

}

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

    adr = argv[1];

    mainSocket.connect(argv[1]);

    sendMessage("OK: " + to\_string(getpid()), mainSocket);

    idThisNode = stoi(argv[2]);

    childNodeId = 0;

    adrChild = "tcp://127.1.1.1:300";

    while (true) {

        zmq::message\_t messageMain;

        mainSocket.recv(messageMain);

        string receivedMessage(static\_cast<char\*>(messageMain.data()), messageMain.size());

        string command;

        for (char element: receivedMessage) {

            if (element != ' ') {

                command += element;

            } else {

                break;

            }

        }

        if (command == "exec") {

            funcExec(receivedMessage);

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

            funcCreate(receivedMessage);

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

            funcPing(receivedMessage);

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

            funcKill(receivedMessage);

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

            if (childNodeId != 0) {

                sendMessage("DIE", childSocket);

                childSocket.unbind(adrChild);

            }

            mainSocket.unbind(adr);

            return 0;

        }

    }

}

**parent.cpp**

#include "zmq.hpp"

#include <sstream>

#include <string>

#include <iostream>

#include <zconf.h>

#include <vector>

#include <signal.h>

#include <sstream>

#include <set>

#include <algorithm>

using namespace std;

zmq::context\_t context(1);

string adr = "tcp://127.1.1.1:300";

string command;

vector<int> childesId;

vector<int> allChildrenId;

vector<unique\_ptr<zmq::socket\_t>> sockets;

void createChildFromMainNode(int childId) {

    auto socket = std::make\_unique<zmq::socket\_t>(context, ZMQ\_REP);

    socket->bind(adr + to\_string(childId));

    string new\_adr = adr + to\_string(childId);

    char\* adr\_ = new char[new\_adr.size() + 1];

    memcpy(adr\_, new\_adr.c\_str(), new\_adr.size() + 1);

    char\* id\_ = new char[to\_string(childId).size() + 1];

    memcpy(id\_, to\_string(childId).c\_str(), to\_string(childId).size() + 1);

    char\* args[] = {"./child", adr\_, id\_, NULL};

    int processId = fork();

    if (processId < 0) {

        cerr << "Unable to create first worker node" << endl;

        childId = 0;

        exit(1);

    } else if (processId == 0) {

        execv("./child", args);

    }

    allChildrenId.push\_back(childId);

    childesId.push\_back(childId);

    sockets.push\_back(std::move(socket));

    zmq::message\_t message;

    sockets[sockets.size() - 1]->recv(message);

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

    cout << receiveMessage << endl;

    delete[] adr\_;

    delete[] id\_;

}

void funcCreate() {

    int childId, parentId;

    cin >> childId >> parentId;

    if (childesId.empty()) {

        if (parentId != -1) {

            cerr << "There is no such parent node" << endl;

            return;

        }

        createChildFromMainNode(childId);

    } else {

        if (parentId == -1) {

            bool wasChild = false;

            for (int indexInChildes = 0; indexInChildes < childesId.size(); ++indexInChildes) {

                if (childesId[indexInChildes] == childId) {

                    cout << "This id has already been created" << endl;

                    wasChild = true;

                    break;

                }

            }

            if (wasChild) {

                return;

            }

            createChildFromMainNode(childId);

        } else {

            string messageString = command + " " + to\_string(childId) + " " + to\_string(parentId);

            for (int indexOfSockets{0}; indexOfSockets < sockets.size(); ++indexOfSockets) {

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

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

                sockets[indexOfSockets]->send(message);

                sockets[indexOfSockets]->recv(message);

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

                if (receiveMessage[0] == 'O' && receiveMessage[1] == 'K') {

                    allChildrenId.push\_back(childId);

                    cout << receiveMessage << endl;

                    break;

                } else if (receiveMessage == "Error: Already exists") {

                    cout << receiveMessage << endl;

                    break;

                } else if (receiveMessage == "Error: this parent already has a child") {

                    cout << receiveMessage << endl;

                    break;

                } else if (receiveMessage == "Error: there is no such parent" &&

                           indexOfSockets == sockets.size() - 1) {

                    cout << receiveMessage << endl;

                    break;

                }

            }

        }

    }

}

void funcExec() {

    int id, value, flag = -1;

    string inputLine, key, valueStr, idStr;

    getline(cin, inputLine);

    for (int index{0}; index < inputLine.size(); ++index) {

        if (inputLine[index] == ' ') {

            flag++;

        } else if (inputLine[index] != ' ' && (flag == 0)) {

            idStr += inputLine[index];

        } else if (inputLine[index] != ' ' && (flag == 1)) {

            key += inputLine[index];

        } else if (inputLine[index] != ' ' && (flag == 2)) {

            valueStr += inputLine[index];

        }

    }

    id = stoi(idStr);

    string messageString = command + " " + to\_string(id) + " " + key;

    if (flag == 2) {

        value = stoi(valueStr);

        messageString = messageString + " " + to\_string(value);

    }

    for (int indexOfSockets{0}; indexOfSockets < sockets.size(); ++indexOfSockets) {

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

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

        sockets[indexOfSockets]->send(message);

        sockets[indexOfSockets]->recv(message);

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

        if (receiveMessage[0] == 'T' && receiveMessage[1] == 'h' && receiveMessage[2] == 'e') {

            cout << receiveMessage << endl;

            sleep(2);

            break;

        } else if (receiveMessage == "Error: id: Not found" &&

                   indexOfSockets == sockets.size() - 1) {

            cout << receiveMessage << endl;

            break;

        }

    }

}

int funcPing(int id) {

    int unavailableProc = NULL;

    if (childesId.empty()) {

        cout << "OK: 0" << endl;

    } else {

        command = "ping";

        string messageString = command + " " + to\_string(id);

        for (int indexOfSockets{0}; indexOfSockets < sockets.size(); ++indexOfSockets) {

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

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

            sockets[indexOfSockets]->send(message);

            sockets[indexOfSockets]->recv(message);

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

            if (receiveMessage == "OK: 1") {

                break;

            } else if (receiveMessage == "OK: 0" &&

                       indexOfSockets == sockets.size() - 1) {

                unavailableProc = id;

                break;

            }

        }

        return unavailableProc;

    }

}

void funcPingAll() {

    if (childesId.size() == 0) {

        cout << "Error: there are no processes" << endl;

    } else {

        set<int> unavailableProcs;

        for (int i{0}; i < allChildrenId.size(); ++i) {

            int ProcStatus = funcPing(allChildrenId[i]);

            if(ProcStatus) {

                unavailableProcs.insert(ProcStatus);

            }

        }

        if (unavailableProcs.empty()) {

            cout << "OK: -1" << endl;

        } else {

            cout << "OK: ";

            for (int const &proc : unavailableProcs) {

                cout << proc << "; ";

            }

            cout << endl;

        }

    }

}

void funcKill() {

    int id;

    cin >> id;

    if (childesId.empty()) {

        cout << "Error: there isn't nodes" << endl;

    } else {

        for (int indexOfSockets{0}; indexOfSockets < sockets.size(); ++indexOfSockets) {

            if (childesId[indexOfSockets] == id) {

                string killMessage = "DIE";

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

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

                sockets[indexOfSockets]->send(message);

                sockets[indexOfSockets]->unbind(adr + to\_string(childesId[indexOfSockets]));

                childesId.erase(childesId.begin() + indexOfSockets);

                sockets.erase(sockets.begin() + indexOfSockets);

                cout << "Node deleted successfully" << endl;

                break;

            } else {

                string killMessage = command + " " + to\_string(id);

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

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

                sockets[indexOfSockets]->send(message);

                sockets[indexOfSockets]->recv(message);

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

                if (receiveMessage[0] == 'O' && receiveMessage[1] == 'K') {

                    cout << receiveMessage << endl;

                    break;

                } else if (receiveMessage == "Error: there isn`t node with this id" &&

                           indexOfSockets == sockets.size() - 1) {

                    cout << receiveMessage << endl;

                    break;

                }

            }

        }

    }

}

void funcExit() {

    for (int indexOfSockets{0}; indexOfSockets < sockets.size(); ++indexOfSockets) {

        if (childesId[indexOfSockets]) {

            string killMessage = "DIE";

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

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

            sockets[indexOfSockets]->send(message);

        }

        sockets[indexOfSockets]->close();

    }

    cout << "All node was deleted" << endl;

    context.close();

    exit(0);

}

int main() {

    while (true) {

        cout << "command:";

        cin >> command;

        if (command == "create") {

            funcCreate();

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

            funcExec();

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

            funcPingAll();

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

            funcKill();

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

            funcExit();

        } else {

            cout << "Error: incorrect command" << endl;

        }

    }

}

**CMakeLists.txt**

cmake\_minimum\_required(VERSION 3.10)

project(DistributedSystem)

set(CMAKE\_CXX\_STANDARD 14)

## load in pkg-config support

find\_package(PkgConfig)

## use pkg-config to get hints for 0mq locations

pkg\_check\_modules(PC\_ZeroMQ QUIET zmq)

## use the hint from above to find where 'zmq.hpp' is located

find\_path(ZeroMQ\_INCLUDE\_DIR

NAMES zmq.hpp

PATHS ${PC\_ZeroMQ\_INCLUDE\_DIRS}

)

## use the hint from above to find the location of libzmq

find\_library(ZeroMQ\_LIBRARY

NAMES zmq

PATHS ${PC\_ZeroMQ\_LIBRARY\_DIRS}

)

# Добавляем исполняемый файл main

add\_executable(main parent.cpp)

# Подключаем каталоги и библиотеки ZeroMQ

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

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

add\_executable(child child.cpp)

# Подключаем каталоги и библиотеки ZeroMQ

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

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

**Пример работы**

radioactive@DESKTOP-RNP2IGB:/mnt/d/labs/os\_lab\_5-7$ ./main

command:create 1 -1

OK: 439

command:create 2 1

OK: 445

command:create 3 2

OK: 452

command:create 4 -1

OK: 457

command:create 5 4

OK: 462

command:create 6 4

Error: this parent already has a child

command:pingall

OK: -1

command:kill 4

Node deleted successfully

command:pingall

OK: 4; 5;

command:exec 3 first 10

Function started in thread: 139692066469440

The child process performs calculations and outputs them when it finishes calculations

OK: 3

Function completed in thread: 139692066469440

command:exec 3 second

Function started in thread: 139692066469440

The child process performs calculations and outputs them when it finishes calculations

OK: 'second' not found

Function completed in thread: 139692066469440

command:exit

All node was deleted

radioactive@DESKTOP-RNP2IGB:/mnt/d/labs/os\_lab\_5-7$

**Вывод**

В ходе данной работы я познакомился с очередями сообщений - еще одним способом обмениваться данными между процессами. Я использовал библиотеку **zeromq** для реализации данной лабораторной. На мой взгляд это самая интересная и полезная из всех лабораторных работ. В ней я использовал большое количество технологий и знаний из предыдущих лабораторных работ.