МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ

ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"

Кафедра 806 "Вычислительная математика и программирование"

Лабораторная работа №5-7

По курсу «Операционные системы»

Студент: Степанов Н.Е.

Группа: М8О-208Б-23

Вариант: 15

Преподаватель: Миронов Е. С.

Дата: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Москва, 2024

**Содержание**

1. Репозиторий

2. Постановка задачи

3. Общие сведения о программе

4. Общий метод и алгоритм решения

5. Исходный код

6. Сборка программы

7. Демонстрация работы программы

8. Выводы

**Репозиторий**

https://github.com/n0w3e/os\_labs/tree/lab5

**Постановка задачи**

**Цель работы**

Целью является приобретение практических навыков в:

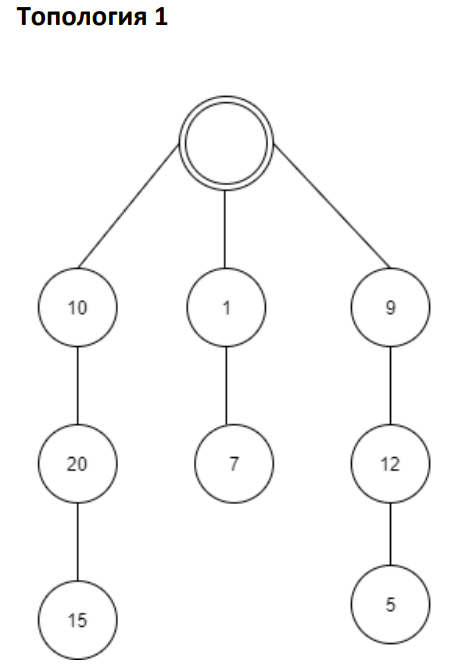
Управлении серверами сообщений (№5)

Применение отложенных вычислений (№6)

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.



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.

**Набора 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

**Команда проверки 3**

Формат команды: heartbit time

Каждый узел начинает сообщать раз в time миллисекунд о том, что он работоспособен. Если от узла нет сигнала в течении 4\*time миллисекунд, то должна выводится пользователю строка: «Heartbit: node id is unavailable now», где id – идентификатор недоступного вычислительного узла.

Tехнология очередей сообщений: ZeroMQ

**Общие сведения о программе**

Программа представляет собой систему управления узлами (рабочими процессами), которая позволяет создавать, управлять и мониторить их состояние. Основные функции включают создание новых узлов, выполнение команд на узлах, проверку их доступности через механизм "heartbit" и обработку пользовательских команд. Узлы взаимодействуют через сетевое соединение с использованием библиотеки ZeroMQ. Программа поддерживает команды для создания узлов, выполнения задач, проверки доступности и остановки мониторинга.

**Общий метод и алгоритм решения**

Основная задача — обеспечить взаимодействие между контроллером и узлами через механизмы сетевого взаимодействия и мониторинга состояния.

Контроллер выступает в роли центрального управляющего элемента. Он обрабатывает команды пользователя, такие как создание узлов, выполнение команд на узлах и проверка их доступности. Узлы — процессы, которые выполняют команды, отправленные контроллером. Каждый узел работает независимо и взаимодействует с контроллером через сетевое соединение.

При получении команды **create**, контроллер создает новый узел с помощью системного вызова **fork**. Новый процесс запускается с помощью **execl**, передавая ему идентификатор и адрес для подключения. Узел инициализирует сетевое соединение и начинает работу, ожидая команд от контроллера. Команды, такие как **exec**, отправляются контроллером на узлы через сокеты **ZeroMQ**. Узел обрабатывает команду и возвращает результат обратно контроллеру. Контроллер также поддерживает таймауты для команд, чтобы избежать зависания при недоступности узла.

Контроллер запускает отдельный поток для проверки доступности узлов через механизм **heartbit**. Узлы периодически отправляют сообщения о своей работоспособности, а контроллер отслеживает время последнего сообщения. Если узел не отвечает в течение заданного времени, контроллер помечает его как недоступный.

Контроллер запускается и ожидает команд от пользователя. Узлы создаются по запросу и подключаются к контроллеру через сетевые сокеты. Пользователь вводит команду, которая разбивается на токены и обрабатывается контроллером. В зависимости от команды, контроллер либо создает новый узел, либо отправляет команду на существующий узел. Контроллер запускает поток для проверки состояния узлов через **heartbit**. Узлы периодически отправляют сообщения о своей работоспособности, а контроллер обновляет информацию о последнем времени ответа. Пользователь может ввести команду **exit**, чтобы завершить работу контроллера.

**Исходный код**

**controller.h:**

#pragma once

#include "utils.h"

#include <thread>

#include <atomic>

#include <unordered\_map>

#include <chrono>

class Controller {

private:

std::map<int, ChildInfo> workers;

std::atomic<bool> heartbitRunning;

std::thread heartbitThread;

std::unordered\_map<int, std::chrono::time\_point<std::chrono::steady\_clock>> lastHeartbit;

void processCommand(const std::string &command);

void runHeartbit(int time);

public:

Controller() : heartbitRunning(false) {}

void run();

};

**utils.cpp:**

#pragma once

#include <string>

#include <utility>

#include <vector>

#include <map>

#include <zmq.hpp>

#include <queue>

#include <utility>

#include "iostream"

inline zmq::context\_t globalContext(1);

struct ChildInfo {

int id;

int pid;

std::string address;

};

class WorkerNodeInfo {

public:

int id;

int pid;

std::string address;

WorkerNodeInfo \*left = nullptr;

WorkerNodeInfo \*right = nullptr;

WorkerNodeInfo \*parent = nullptr;

public:

WorkerNodeInfo(int id, int pid, std::string address) : id(id), pid(pid), address(std::move(address)) {};

WorkerNodeInfo(int id, int pid, std::string address, WorkerNodeInfo \*parent) : id(id), pid(pid),

address(std::move(address)),

parent(parent) {};

};

void sendResponse(zmq::socket\_t &socket, const std::string &response);

std::string receiveRequest(zmq::socket\_t &socket);

bool

sendRequestWithTimeout(zmq::socket\_t &socket, const std::string &request, std::string &response, int timeout = 1000);

void createWorker(int id, ChildInfo &info);

bool isPidAlive(int pid);

**worker\_node.h:**

#pragma once

#include <iostream>

#include <zmq.hpp>

#include <map>

#include <thread>

#include <unistd.h>

#include "utils.h"

class WorkerNode {

public:

WorkerNode(int id, const std::string &address);

~WorkerNode();

void run();

private:

int id;

zmq::context\_t context;

zmq::socket\_t socket;

std::string address;

zmq::socket\_t commandSocket;

zmq::socket\_t heartbitSocket;

std::map<std::string, int> localDict;

void processCommand(const std::string &command);

void sendHeartbit(int time);

};

**controller.cpp:**

#include <iostream>

#include <zmq.hpp>

#include <map>

#include <vector>

#include <unistd.h>

#include <sys/wait.h>

#include <chrono>

#include "../include/controller.h"

void Controller::runHeartbit(int time) {

zmq::context\_t context(1);

zmq::socket\_t subscriber(context, ZMQ\_SUB);

for (const auto &worker : workers) {

subscriber.connect(worker.second.address + "\_pub");

subscriber.set(zmq::sockopt::subscribe, "heartbit");

}

while (heartbitRunning) {

auto now = std::chrono::steady\_clock::now();

for (const auto &worker : workers) {

auto it = lastHeartbit.find(worker.first);

if (it != lastHeartbit.end()) {

auto elapsed = std::chrono::duration\_cast<std::chrono::milliseconds>(now - it->second).count();

if (elapsed > 4 \* time) {

std::cout << "Heartbit: node " << worker.first << " is unavailable now" << std::endl;

} else {

std::cout << "Heartbit: node " << worker.first << " is available" << std::endl;

}

}

}

zmq::message\_t message;

while (subscriber.recv(message, zmq::recv\_flags::dontwait)) {

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

if (msg.find("heartbit") != std::string::npos) {

int id = std::stoi(msg.substr(msg.find("id:") + 3));

lastHeartbit[id] = std::chrono::steady\_clock::now();

}

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(time));

}

}

void Controller::processCommand(const std::string &command) {

std::vector<std::string> tokens;

std::string token;

std::istringstream tokenStream(command);

while (std::getline(tokenStream, token, ' ')) {

tokens.push\_back(token);

}

if (tokens[0] == "create") {

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

if (id == -1) {

std::cout << "Error: Invalid id format" << std::endl;

return;

}

if (workers.find(id) != workers.end()) {

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

return;

}

ChildInfo info;

createWorker(id, info);

workers[id] = info;

lastHeartbit[id] = std::chrono::steady\_clock::now();

} else if (tokens[0] == "exec") {

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

if (workers.find(id) == workers.end()) {

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

return;

}

zmq::context\_t context(1);

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

workerSocket.set(zmq::sockopt::linger, 0);

workerSocket.connect(workers[id].address);

std::string command = tokens[0] + " " + tokens[1];

for (size\_t i = 2; i < tokens.size(); ++i) {

command += " " + tokens[i];

}

std::string response;

if (sendRequestWithTimeout(workerSocket, command, response, 1000)) {

std::cout << response << std::endl;

lastHeartbit[id] = std::chrono::steady\_clock::now();

} else {

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

}

workerSocket.close();

} else if (tokens[0] == "heartbit") {

int time = std::stoi(tokens[1]);

if (heartbitRunning) {

std::cout << "Heartbit is already running. Use 'stop' to stop it." << std::endl;

return;

}

heartbitRunning = true;

std::cout << "Ok" << std::endl;

heartbitThread = std::thread(&Controller::runHeartbit, this, time);

} else if (tokens[0] == "stop") {

if (!heartbitRunning) {

std::cout << "Heartbit is not running." << std::endl;

return;

}

heartbitRunning = false;

heartbitThread.join();

std::cout << "Heartbit stopped." << std::endl;

} else if (tokens[0] == "ping") {

if (tokens.size() < 2) {

std::cout << "Error: Missing node ID" << std::endl;

return;

}

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

auto it = lastHeartbit.find(id);

if (it != lastHeartbit.end()) {

auto now = std::chrono::steady\_clock::now();

auto elapsed = std::chrono::duration\_cast<std::chrono::milliseconds>(now - it->second).count();

if (elapsed <= 4 \* 2000) {

std::cout << "Ok: " << id << ": 1" << std::endl;

} else {

std::cout << "Ok: " << id << ": 0" << std::endl;

}

} else {

std::cout << "Error: Node " << id << " not found" << std::endl;

}

} else {

std::cout << "Error: Unknown command" << std::endl;

}

}

void Controller::run() {

std::string command;

while (true) {

std::cout << "> ";

std::getline(std::cin, command);

if (command == "exit") {

break;

}

processCommand(command);

}

}

**utils.cpp:**

#include <sys/wait.h>

#include "../include/utils.h"

#include <unistd.h>

#include <iostream>

void sendResponse(zmq::socket\_t &socket, const std::string &response) {

zmq::message\_t reply(response.size());

memcpy(reply.data(), response.c\_str(), response.size());

socket.send(reply, zmq::send\_flags::none);

}

std::string receiveRequest(zmq::socket\_t &socket) {

zmq::message\_t request;

socket.recv(request, zmq::recv\_flags::none);

return std::string(static\_cast<char \*>(request.data()), request.size());

}

bool sendRequestWithTimeout(zmq::socket\_t &socket, const std::string &request, std::string &response, int timeout) {

zmq::message\_t req(request.size());

memcpy(req.data(), request.c\_str(), request.size());

socket.send(req, zmq::send\_flags::none);

zmq::pollitem\_t items[] = {{socket, 0, ZMQ\_POLLIN, 0}};

zmq::poll(&items[0], 1, std::chrono::milliseconds(timeout));

if (items[0].revents & ZMQ\_POLLIN) {

response = receiveRequest(socket);

return true;

} else {

return false;

}

}

void createWorker(int id, ChildInfo &info) {

int basePort = 5555;

int port = basePort + id;

std::string address = "tcp://127.0.0.1:" + std::to\_string(port);

pid\_t pid = fork();

if (pid == 0) {

if (execl("/Users/evgenijstepanov/VSCODE/OS/os\_labs/build/lab5/worker", "worker", std::to\_string(id).c\_str(), address.c\_str(), NULL) == -1) {

perror("Child run error");

}

} else if (pid > 0) {

info = {id, pid, address};

std::cout << "Ok: pid: " << pid << " port: " << port << std::endl;

} else {

std::cout << "Error: Fork failed" << std::endl;

}

}

bool isPidAlive(int pid) {

int status = 0;

int result = waitpid(pid, &status, WNOHANG);

if (result == 0) {

return true;

} else {

return false;

}

}

**worker\_node.cpp:**  
#include "../include/worker\_node.h"

#include <thread>

#include <chrono>

#include <iostream>

WorkerNode::WorkerNode(int id, const std::string &address) : id(id), address(address) {

context = zmq::context\_t(1);

socket = zmq::socket\_t(context, ZMQ\_REP);

socket.bind(address);

std::thread heartbitThread(&WorkerNode::sendHeartbit, this, 2000);

heartbitThread.detach();

}

void WorkerNode::sendHeartbit(int time) {

while (true) {

zmq::pollitem\_t items[] = {{socket, 0, ZMQ\_POLLOUT, 0}};

zmq::poll(&items[0], 1, std::chrono::milliseconds(time));

if (items[0].revents & ZMQ\_POLLOUT) {

std::string heartbitMessage = "heartbit id:" + std::to\_string(id);

zmq::message\_t message(heartbitMessage.begin(), heartbitMessage.end());

socket.send(message, zmq::send\_flags::none);

std::cout << "Worker " << id << " sent heartbit" << std::endl;

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(time));

}

}

void WorkerNode::processCommand(const std::string &command) {

std::vector<std::string> tokens;

std::string token;

std::istringstream tokenStream(command);

while (std::getline(tokenStream, token, ' ')) {

tokens.push\_back(token);

}

if (tokens[0] == "exec") {

if (tokens.size() == 3) {

std::string key = tokens[2];

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

sendResponse(socket, "Ok:" + std::to\_string(id) + ": " + std::to\_string(localDict[key]));

} else {

sendResponse(socket, "Ok:" + std::to\_string(id) + ": 'MyVar' not found");

}

} else if (tokens.size() == 4) {

std::string key = tokens[2];

int value = std::stoi(tokens[3]);

localDict[key] = value;

sendResponse(socket, "Ok:" + std::to\_string(id));

}

} else if (tokens[0] == "ping") {

sendResponse(socket, "Ok");

}

}

void WorkerNode::run() {

while (true) {

std::string command = receiveRequest(socket);

processCommand(command);

}

}

WorkerNode::~WorkerNode() {

socket.unbind(address);

}

**worker.cpp:**

#include <iostream>

#include "../include/worker\_node.h"

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

if (argc < 3) {

std::cerr << "Usage: " << argv[0] << " <id> <address>" << std::endl;

return 1;

}

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

std::string address = argv[2];

WorkerNode worker(id, address);

worker.run();

return 0;

}

**main.cpp:**

#include "include/controller.h"

int main() {

auto controller = Controller();

controller.run();

return 0;

}

**Демонстрация работы программы**

n0wee@DESKTOP-8QSPN1P:~/Coding/os\_labs/build/lab5$ ./controller

> create 1

Ok: pid: 12345 port: 5556

> exec 1 set MyVar 42

Ok:1

> exec 1 get MyVar

Ok:1: 42

> ping 1

Ok: 1: 1

> heartbit 2000

Ok

Heartbit: node 1 is available

Heartbit: node 1 is available

> stop

Heartbit stopped.

> exit

Если узел не существует, выводиться сообщение:

**Error: 2: Not found**

Если команда неверная, выводится:

**Error: Unknown command**

Если узел недоступен, выводится:

**Error: 1: Node is unavailable**

Мониторинг **heartbit** выполняется в отдельном потоке, что позволяет контроллеру одновременно обрабатывать команды пользователя и проверять состояние узлов.

**Вывод**

В ходе выполнения лабораторной работы я изучил принципы работы с многопоточностью, сетевым взаимодействием и управлением процессами в операционной системе. Для реализации системы управления узлами использовал библиотеку **ZeroMQ** для организации сетевого взаимодействия и механизмы **fork** и **execl** для создания новых процессов. Была реализована система мониторинга состояния узлов через механизм **heartbit**, что позволило отслеживать их доступность в реальном времени.

Особенно понравилось работать с **ZeroMQ**, так как она предоставляет удобные инструменты для организации сетевого взаимодействия. Также было интересно реализовывать многопоточность для параллельной обработки команд и мониторинга. В целом, работа позволила глубже понять принципы распределенных систем и взаимодействия между процессами.