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: Снетков Н. С.
Группа: М8О-203Б-23
Вариант: 44
Преподаватель: Миронов Е. С.

Дата: _____

Оценка: _____

Подпись: _____

Москва, 2025

Содержание

Репозиторий

https://github.com/Sapfir7/labs_os_snet/tree/main/os_labs/lab5-7

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.

Топология 4

Аналогично топологии 4, но узлы находятся в идеально сбалансированном бинарном дереве.

Каждый следующий узел должен добавляться в самое наименьшее поддерево.

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 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: 6

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.

Технология 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

ControllerNode.h

```
#ifndef CONTROLLERNODE_H

#define CONTROLLERNODE_H


#include <zmq.hpp>

#include <string>

#include <unordered_map>

#include <thread>

#include <memory>

#include <queue>

#include <iostream>


struct TreeNode {

    int id;

    TreeNode* left;

    TreeNode* right;


    TreeNode(int id) : id(id), left(nullptr), right(nullptr) {}

};


class BalancedBinaryTree {

public:
```

```
    BalancedBinaryTree() : root(nullptr) {}

    void insert(int id, int parent = -1) {
        if (parent == -1) {
            root = insertBalanced(root, id);
        } else {
            TreeNode* parentNode = findNode(root, parent);
            if (parentNode) {
                if (!parentNode->left) {
                    parentNode->left = new TreeNode(id);
                } else if (!parentNode->right) {
                    parentNode->right = new TreeNode(id);
                } else {
                    std::cerr << "Error: Parent node already has two children" << std::endl;
                }
            } else {
                std::cerr << "Error: Parent node not found" << std::endl;
            }
        }
    }

    // Поиск узла по id
    TreeNode* findNode(int id) {
        return findNode(root, id);
    }

private:
    TreeNode* root;
```

```
// Рекурсивная вставка в самое наименьшее поддерево
TreeNode* insertBalanced(TreeNode* node, int id) {
    if (node == nullptr) {
        return new TreeNode(id);
    }

    // Вычисляем высоту левого и правого поддеревьев
    int leftHeight = getHeight(node->left);
    int rightHeight = getHeight(node->right);

    // Добавляем в самое наименьшее поддерево
    if (leftHeight <= rightHeight) {
        node->left = insertBalanced(node->left, id);
    } else {
        node->right = insertBalanced(node->right, id);
    }

    return node;
}

// Рекурсивный поиск узла
TreeNode* findNode(TreeNode* node, int id) {
    if (node == nullptr || node->id == id) {
        return node;
    }

    TreeNode* leftResult = findNode(node->left, id);
    if (leftResult) {
        return leftResult;
```

```
        }

        return findNode(node->right, id);
    }

    // Вычисление высоты дерева
    int getHeight(TreeNode* node) {
        if (node == nullptr) {
            return 0;
        }
        return 1 + std::max(getHeight(node->left), getHeight(node->right));
    }
};

class ControllerNode {
public:
    ControllerNode();
    void start();
    void createNode(int id, int parent = -1);
    void execCommand(int id, const std::string& params);
    void pingNode(int id);
    void heartbit(int time);

private:
    void sendCommand(const std::string& command);
    void launchWorkerNode(int id, int parentId = -1);

    zmq::context_t context_;
    zmq::socket_t socket_;
```

```
    std::unordered_map<int, std::thread> workerThreads_;
    BalancedBinaryTree tree_;
};



#endif
```

WorkerNode.h

```
#ifndef WORKERNODE_H
#define WORKERNODE_H

#include <zmq.hpp>
#include <string>
#include <atomic>

class WorkerNode {
public:
    WorkerNode(int id, int parentId = -1);
    void run();

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

    int id_;
    int parentId_;
    zmq::context_t context_;
    zmq::socket_t socket_;
    std::atomic<bool> running_;
};
```

```
#endif
```

ControllerNode.cpp

```
#include "../include/ControllerNode.h"
#include "../include/WorkerNode.h"
#include <iostream>
#include <sstream>
#include <cstdlib>
#include <unistd.h>

ControllerNode::ControllerNode() : context_(1), socket_(context_, ZMQ_REQ) {
    socket_.connect("tcp://localhost:5555");
}

void ControllerNode::start() {
    std::string command;
    while (true) {
        std::cout << "> ";
        std::getline(std::cin, command);

        if (command.empty()) continue;

        if (command.find("create") == 0) {
            int id, parent = -1;
            std::istringstream iss(command);
            std::string cmd;
            iss >> cmd >> id >> parent;

            createNode(id, parent);
```

```
        } else if (command.find("exec") == 0) {
            int id;
            std::string params;
            std::istringstream iss(command);
            std::string cmd;
            iss >> cmd >> id;
            std::getline(iss, params);
            execCommand(id, params);
        } else if (command.find("ping") == 0) {
            int id;
            std::istringstream iss(command);
            std::string cmd;
            iss >> cmd >> id;
            pingNode(id);
        } else if (command.find("heartbit") == 0) {
            int time;
            std::istringstream iss(command);
            std::string cmd;
            iss >> cmd >> time;
            heartbit(time);
        } else {
            std::cout << "Error: Unknown command" << std::endl;
        }
    }
}

void ControllerNode::createNode(int id, int parent) {
    if (workerThreads_.find(id) != workerThreads_.end()) {
        std::cout << "Error: Already exists" << std::endl;
```

```
        return;
    }

    if (parent != -1 && workerThreads_.find(parent) == workerThreads_.end()) {
        std::cout << "Error: Parent not found" << std::endl;
        return;
    }
    tree_.insert(id, parent);
    launchWorkerNode(id, parent);
    std::cout << "Ok: Node " << id << " created" << std::endl;
}

void ControllerNode::launchWorkerNode(int id, int parentId) {
    workerThreads_[id] = std::thread([id, parentId]() {
        WorkerNode worker(id, parentId);
        worker.run();
    });
    workerThreads_[id].detach();
}

void ControllerNode::execCommand(int id, const std::string& params) {
    std::string command = "exec " + std::to_string(id) + " " + params;
    sendCommand(command);
}

void ControllerNode::pingNode(int id) {
    if (workerThreads_.find(id) == workerThreads_.end()) {
        throw std::runtime_error("Node not found");
    }
```

```
    std::string command = "ping " + std::to_string(id);
    sendCommand(command);
}

void ControllerNode::heartbit(int time) {
    auto start = std::chrono::steady_clock::now();
    while (true) {
        // Проверяем, прошло ли заданное время
        auto now = std::chrono::steady_clock::now();
        if (std::chrono::duration_cast<std::chrono::milliseconds>(now - start).count() >= time) {
            break; // Завершаем heartbit через указанное время
        }

        // Имитируем выполнение heartbit
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        if (false) {
            std::cout << "Error: Heartbit failed!" << std::endl;  // Сообщение об ошибке
        }
    }
}

void ControllerNode::sendCommand(const std::string& command) {
    // Извлекаем id узла из команды
    std::istringstream iss(command);
    std::string cmd;
    int id;
    iss >> cmd >> id;
```

```
    // Подключаемся к узлу
    zmq::socket_t nodeSocket(context_, ZMQ_REQ);
    std::string address = "tcp://localhost:" + std::to_string(5555 + id);
    nodeSocket.connect(address);

    // Отправляем команду
    zmq::message_t request(command.size());
    memcpy(request.data(), command.data(), command.size());
    nodeSocket.send(request, zmq::send_flags::none);

    // Получаем ответ
    zmq::message_t reply;
    auto recv_result = nodeSocket.recv(reply, zmq::recv_flags::none);
    if (!recv_result) {
        std::cerr << "Error: Failed to receive reply from node " << id << std::endl;
        return;
    }

    std::string replyStr(static_cast<char*>(reply.data()), reply.size());
    std::cout << replyStr << std::endl;
}
```

WorkerNode.cpp

```
#include "../include/WorkerNode.h"
#include <iostream>

WorkerNode::WorkerNode(int id, int parentId) : id_(id), parentId_(parentId), context_(1), socket_(context_, ZMQ_REP), running_(true) {
```

```
    // Узел слушает сообщения на уникальном адресе
    std::string address = "tcp://*:" + std::to_string(5555 + id);
    socket_.bind(address);
    std::cout << "Node " << id_ << " started and listening on " << address << std::endl;
}

void WorkerNode::run() {
    std::cout << "Node " << id_ << " started" << std::endl;
    while (true) {
        zmq::message_t request;
        auto recv_result = socket_.recv(request, zmq::recv_flags::none);
        if (!recv_result) {
            std::cerr << "Node " << id_ << ": Failed to receive message" << std::endl;
            continue;
        }

        std::string command(static_cast<char*>(request.data()), request.size());
        std::string response = processCommand(command);

        zmq::message_t reply(response.size());
        memcpy(reply.data(), response.data(), response.size());
        socket_.send(reply, zmq::send_flags::none);
    }
}

std::string WorkerNode::processCommand(const std::string& command) {
    if (command.find("exec") == 0) {
        std::istringstream iss(command);
        std::string cmd;
```

```
        int requestedId, n;
        iss >> cmd >> requestedId >> n;

        if (requestedId != id_) {
            return "Error:" + std::to_string(requestedId) + ": Node not found";
        }

        int sum = 0;
        for (int i = 0; i < n; ++i) {
            int num;
            iss >> num;
            sum += num;
        }

        return "Ok:" + std::to_string(id_) + ": " + std::to_string(sum);
    } else if (command.find("ping") == 0) {
        int requestedId;
        std::istringstream iss(command);
        std::string cmd;
        iss >> cmd >> requestedId;

        if (requestedId != id_) {
            return "Error:" + std::to_string(requestedId) + ": Node not found";
        }

        return "Ok:1";
    } else if (command.find("heartbit") == 0) {
        return "Ok";
    }
```

```
    return "Error: Unknown 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);
}
```

main.cpp:

```
#include "include/ControllerNode.h"


int main() {
    ControllerNode controller;
    controller.start();
    return 0;
```

}

Демонстрация работы программы

unix@DESKTOP-MPQDBS2:~/labs/osLabs/build/lab5-7$ ./lab5-7

> create 10

Ok: Node 10 created

> Node 10 started and listening on tcp://*:5565

Node 10 started

> create 20 10

Ok: Node 20 created

> Node 20 started and listening on tcp://*:5575

Node 20 started

> ping 10

Ok:1

>

> ping 20

Ok:1

> heartbit

Ok

> exec 10 3 1 2 3

Ok:10: 6

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.