Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу
«Операционные системы»

Студент: Скрипачев Фёдор Михайлович
Группа: М8О-209Б-23
Вариант: 16
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2025

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/gthcbr25/osi/tree/main/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. Список основных поддерживаемых команд:

- **Создание нового вычислительного узла** (Формат команды: create id [parent])
- **Исполнение команды на вычислительном узле** (Формат команды: exec id [params])
- **Проверка доступности узла** (Формат команды: ping id)
- **Удаление узла** (Формат команды remove id)

Вариант №16: топология — идеально сбалансированное бинарное дерево, команда — локальный таймер, проверка доступности — ping id.

Общие сведения о программе

Программа представляет собой менеджер для управления распределёнными узлами через ZeroMQ. Она использует AVL-дерево для хранения информации об узлах, где каждый узел содержит уникальный идентификатор, PID процесса и адрес для связи через ZeroMQ. Основные команды программы: create id — создаёт новый узел с указанным идентификатором, создавая новый процесс. exec id [cmd] — отправляет команду на указанный узел и получает ответ. ping id — проверяет доступность узла. Используются библиотеки ZeroMQ для сетевого взаимодействия и функции управления процессами через fork и waitpid.

Исходный код

Node.cpp

```
#include <iostream>
#include <string>
#include <chrono>
#include <zmq.hpp>

class Timer {
private:
    bool running = false;
    std::chrono::steady_clock::time_point start_time;
    std::chrono::milliseconds elapsed{0};

public:
    void start() {
        if (!running) {
            running = true;
            start_time = std::chrono::steady_clock::now();
```

```
        }
    }

    void stop() {
        if (running) {
            elapsed += std::chrono::duration_cast<std::chrono::milliseconds>(
                std::chrono::steady_clock::now() - start_time);
            running = false;
        }
    }

    long long time() {
        if (running) {
            return elapsed.count() +
std::chrono::duration_cast<std::chrono::milliseconds>(
                                std::chrono::steady_clock::now() - start_time)
                                .count();
        }
        return elapsed.count();
    }
};

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "Error: Invalid arguments\n";
        return 1;
    }

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

```
Timer timer;

zmq::context_t context(1);
zmq::socket_t socket(context, ZMQ_REP);
std::ostringstream endpoint;
endpoint << "tcp://*:" << 5555 + id;
socket.bind(endpoint.str());

std::cout << "Node " << id << " started\n";

while (true) {
    zmq::message_t request;
    socket.recv(request, zmq::recv_flags::none);
    std::string cmd(static_cast<char*>(request.data()), request.size());

    std::string response;
    if (cmd == "start") {
        timer.start();
        response = "Ok: start";
    } else if (cmd == "stop") {
        timer.stop();
        response = "Ok: stop";
    } else if (cmd == "time") {
        response = "Ok: time " + std::to_string(timer.time()) + "ms";
    } else {
        response = "Error: Unknown command";
    }

    zmq::message_t reply(response.size());
```

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

    return 0;
}
```

Server.cpp

```
#include <iostream>
#include <string>
#include <sstream>
#include <algorithm>
#include <sys/wait.h>
#include <zmq.hpp>
#include <memory>

using namespace std;

struct NodeInfo {
    int id;             // Идентификатор узла
    int pid;            // PID процесса
    std::string endpoint; // Адрес ZeroMQ
};

template <typename Key, typename Value>
class AVLNode {
public:
    Key key;
    Value value;
    std::unique_ptr<AVLNode> left;
    std::unique_ptr<AVLNode> right;
```

```
    int height;

    AVLNode(Key k, Value v)
        : key(k), value(v), left(nullptr), right(nullptr), height(1) {}
};

template <typename Key, typename Value>
class AVLTree {
private:
    std::unique_ptr<AVLNode<Key, Value>> root;

    int height(const AVLNode<Key, Value>* node) const {
        return node ? node->height : 0;
    }

    int balanceFactor(const AVLNode<Key, Value>* node) const {
        return node ? height(node->left.get()) - height(node->right.get()) : 0;
    }

    std::unique_ptr<AVLNode<Key, Value>>
rightRotate(std::unique_ptr<AVLNode<Key, Value>> y) {
        auto x = std::move(y->left);
        auto T2 = std::move(x->right);

        x->right = std::move(y);
        x->right->left = std::move(T2);

        x->right->height = std::max(height(x->right->left.get()), height(x->right-
>right.get())) + 1;
        x->height = std::max(height(x->left.get()), height(x->right.get())) + 1;
```

```
        return x;
    }

    std::unique_ptr<AVLNode<Key, Value>>
leftRotate(std::unique_ptr<AVLNode<Key, Value>> x) {
        auto y = std::move(x->right);
        auto T2 = std::move(y->left);

        y->left = std::move(x);
        y->left->right = std::move(T2);

        y->left->height = std::max(height(y->left->left.get()), height(y->left-
>right.get())) + 1;
        y->height = std::max(height(y->left.get()), height(y->right.get())) + 1;

        return y;
    }

    AVLNode<Key, Value>* findMin(AVLNode<Key, Value>* node) const {
        while (node->left) {
            node = node->left.get();
        }
        return node;
    }

    std::unique_ptr<AVLNode<Key, Value>>
insert(std::unique_ptr<AVLNode<Key, Value>> node, Key key, Value value) {
        if (!node) return std::make_unique<AVLNode<Key, Value>>(key, value);
```

```
        if (key < node->key)
            node->left = insert(std::move(node->left), key, value);
        else if (key > node->key)
            node->right = insert(std::move(node->right), key, value);
        else {
            node->value = value; // Обновление значения при существующем ключе
            return node;
        }

        node->height = 1 + std::max(height(node->left.get()), height(node->right.get()));
        int balance = balanceFactor(node.get());

        if (balance > 1 && key < node->left->key)
            return rightRotate(std::move(node));

        if (balance < -1 && key > node->right->key)
            return leftRotate(std::move(node));

        if (balance > 1 && key > node->left->key) {
            node->left = leftRotate(std::move(node->left));
            return rightRotate(std::move(node));
        }

        if (balance < -1 && key < node->right->key) {
            node->right = rightRotate(std::move(node->right));
            return leftRotate(std::move(node));
        }
```

```
        return node;
    }

    std::unique_ptr<AVLNode<Key, Value>>
erase(std::unique_ptr<AVLNode<Key, Value>> node, Key key) {
        if (!node) return nullptr;

        if (key < node->key) {
            node->left = erase(std::move(node->left), key);
        } else if (key > node->key) {
            node->right = erase(std::move(node->right), key);
        } else {
            // Case 1: Node with only one child or no child
            if (!node->left) {
                return std::move(node->right);
            } else if (!node->right) {
                return std::move(node->left);
            }

            // Case 2: Node with two children
            AVLNode<Key, Value>* minNode = findMin(node->right.get());
            node->key = minNode->key;
            node->value = minNode->value;
            node->right = erase(std::move(node->right), minNode->key);
        }

        node->height = 1 + std::max(height(node->left.get()), height(node-
>right.get()));

        int balance = balanceFactor(node.get());
```

```
        if (balance > 1 && balanceFactor(node->left.get()) >= 0)
            return rightRotate(std::move(node));

        if (balance > 1 && balanceFactor(node->left.get()) < 0) {
            node->left = leftRotate(std::move(node->left));
            return rightRotate(std::move(node));
        }

        if (balance < -1 && balanceFactor(node->right.get()) <= 0)
            return leftRotate(std::move(node));

        if (balance < -1 && balanceFactor(node->right.get()) > 0) {
            node->right = rightRotate(std::move(node->right));
            return leftRotate(std::move(node));
        }

        return node;
    }

    AVLNode<Key, Value>* search(AVLNode<Key, Value>* node, Key key)
const {
        if (!node || node->key == key) return node;

        if (key < node->key)
            return search(node->left.get(), key);
        else
            return search(node->right.get(), key);
    }
```

```
    void inorder(const AVLNode<Key, Value>* node) const {
        if (node) {
            inorder(node->left.get());
            cout << node->key << " ";
            inorder(node->right.get());
        }
    }

public:
    void insert(Key key, Value value) {
        root = insert(std::move(root), key, value);
    }

    void erase(Key key) {
        root = erase(std::move(root), key);
    }

    Value* search(Key key) {
        auto node = search(root.get(), key);
        return node ? &node->value : nullptr;
    }

    void printInorder() const {
        inorder(root.get());
        cout << endl;
    }
};
```

```
AVLTree<int, NodeInfo> tree;

bool isProcessAlive(int pid, int id) {
    int status;
    pid_t result = waitpid(pid, &status, WNOHANG); // Проверяем состояние процесса

    if (result == 0) {
        // Процесс живой, но не завершен
        return kill(pid, 0) == 0;
    } else if (result == pid) {
        // Процесс завершился, удаляем его из системы
        if (WIFEXITED(status) || WIFSIGNALED(status)) {
            tree.erase(id);
            cout << "Process " << pid << " became a zombie and was reaped." << endl;
            return false;
        }
    }

    // Ошибка или процесс больше не существует
    return false;
}

std::string createNode(int id) {
    if (tree.search(id)) {
        return "Error: Node already exists";
    }
```

```
    int pid = fork();
    if (pid == -1) {
        return "Error: Fork failed";
    }

    if (pid == 0) {
        execl("./node", "./node", std::to_string(id).c_str(), NULL);
        exit(0);
    } else {
        std::ostringstream endpoint;
        endpoint << "tcp://127.0.0.1:" << 5555 + id;

        NodeInfo node_info = {id, pid, endpoint.str()};
        tree.insert(id, node_info);
        return "Ok: Created node " + std::to_string(id) + " with PID " +
std::to_string(pid);
    }
}

std::string execCommand(int id, const std::string& command) {
    NodeInfo* node_info = tree.search(id);
    if (!node_info) {
        return "Error: Node not found";
    }

    if (!isProcessAlive(node_info->pid, node_info->id)){
        return "Error: Node process not alive";
    }
```

```
    try {
        zmq::context_t context(1);
        zmq::socket_t socket(context, ZMQ_REQ);
        socket.connect(node_info->endpoint);

        zmq::message_t request(command.size());
        memcpy(request.data(), command.data(), command.size());
        socket.send(request, zmq::send_flags::none);

        zmq::message_t reply;
        socket.recv(reply, zmq::recv_flags::none);

        return std::string(static_cast<char*>(reply.data()), reply.size());
    } catch (const zmq::error_t& e) {
        return "Error: Node is unavailable (" + std::string(e.what()) + ")";
    } catch (...) {
        return "Error: Unexpected failure";
    }
}

std::string pingNode(int id) {
    NodeInfo* node_info = tree.search(id);
    if (!node_info) {
        return "Error: Node not found";
    }

    if (!isProcessAlive(node_info->pid, node_info->id)){
        return "Error: Node process not alive";
    }
```

```
    try {
        zmq::context_t context(1);
        zmq::socket_t socket(context, ZMQ_REQ);



        socket.connect(node_info->endpoint);

        zmq::message_t ping("ping", 4);
        auto send_result = socket.send(ping, zmq::send_flags::none);
        if (!send_result) {
            return "Ok: 0 (send failed)";
        }

        zmq::message_t reply;
        auto recv_result = socket.recv(reply, zmq::recv_flags::none);

        if (recv_result.value()) {
            return "Ok: 1";
        } else {
            return "Ok: 0 (no reply)";
        }
    } catch (const zmq::error_t& e) {
        return "Ok: 0 (exception: " + std::string(e.what()) + ")";
    } catch (...) {
        return "Ok: 0 (unknown error)";
    }
}
```

```
int main() {
    cout << "Manager started. Commands: create id, exec id [cmd], ping id" << endl;

    std::string input;
    while (getline(cin, input)) {
        std::istringstream iss(input);
        std::string cmd;
        int id;
        iss >> cmd >> id;

        if (cmd == "create") {
            cout << createNode(id) << endl;
            tree.printInorder();
        } else if (cmd == "exec") {
            std::string subcmd;
            iss >> subcmd;
            cout << execCommand(id, subcmd) << endl;
        } else if (cmd == "ping") {
            cout << pingNode(id) << endl;
        } else {
            cout << "Error: Invalid command" << endl;
        }
    }

    return 0;
}
```

Демонстрация работы программы

./server Manager started.
Commands: create id, exec id [cmd], ping id
create 1
Ok: Created node 1 with PID 16779
1
Node 1 started
create 2
Ok: Created node 2 with PID 16783
1 2
Node 2 started
create 3
Ok: Created node 3 with PID 16790
1 2 3 Node
3 started
ping 1
Ok: 1
ping 2
Ok: 1
ping 3
Ok: 1
ping 4
Error: Node not found
exec 1 start
Ok: start
exec 1 stop
Ok: stop
exec 1 time
Ok: time 4685ms

Выводы

В лабораторной работе я освоил базовые принципы работы с очередями сообщений ZeroMQ и создал программу на основе этой библиотеки. Очереди позволяют организовать взаимодействие между устройствами в распределённой сети. Для повышения надёжности системы я тестировал различные методы связи. Приобретённые навыки помогут в разработке собственной отказоустойчивой инфраструктуры распределённых вычислений.