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

Лабораторная работа №6-7-8 по курсу «Операционные системы»

Студент: Казанцев Данила Игоревич
Группа: М8О-207Б-21
Вариант: 26
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/thgdanilaya/mai_os_labs/lab6

Постановка задачи

Цель работы

Целью является приобретение практических навыков в: • Управлении серверами сообщений (№6) • Применение отложенных вычислений (№7) • Интеграция программных систем друг с другом (№8)

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

Общие сведения о программе

Программа состоит из двух файлов(main.cpp , child.cpp) и двух библеотек(tree.hpp , server.hpp).

Исходный код

```
\#include "server.hpp"
#include "tree.hpp"
#include <algorithm>
#include <csignal>
#include <iostream>
#include <set>
#include <string>
#include <unistd.h>
#include <vector>
#include <zmq.hpp>


using namespace std;

int main()
{

    zmq::context_t context(1);
    zmq::socket_t mainSocket(context, ZMQ_REQ);

    mainSocket.setsockopt(ZMQ_SNDTIMEO, 2000);
    int linger = 0;
    mainSocket.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
    int port = bindSocket(mainSocket);

    Tree tree;

    int childPid = 0;
    int childId = 0;
```

```
    int createNodeId;

    int id;
    char excmd;
    string word;
    int val;

    string sendingMsg;
    string receivedMsg;

    string cmd;
    while (cout << "> " && cin >> cmd) {
        if (cmd == "create") {
            cin >> createNodeId;
            if (childPid == 0) {
                childPid = fork();
                if (childPid == -1) {
                    cout << "Error: fork fails\n";
                    childPid = 0;
                    exit(1);
                }
                else if (childPid == 0) {
                    createNode(createNodeId, port);
                }
                else {
                    childId = createNodeId;
                    sendMessage(mainSocket, "pid");
                    receivedMsg = receiveMessage(mainSocket);
                }
            }
            else {
                ostringstream sendingMsgStream;
                sendingMsgStream << "create " << createNodeId;
                sendMessage(mainSocket, sendingMsgStream.str());
                receivedMsg = receiveMessage(mainSocket);
            }

            if (receivedMsg.substr(0, 2) == "OK") {
                tree.insert(createNodeId);
            }

            cout << receivedMsg << "\n";
        }
        else if (cmd == "remove") {
            if (childPid == 0) {
                cout << "Error: Not found\n";
                continue;
            }
            cin >> createNodeId;
            if (createNodeId == childId) {
                kill(childPid, SIGTERM);
                kill(childPid, SIGKILL);
                childId = childPid = 0;
                cout << "OK\n";
                tree.erase(createNodeId);
                continue;
            }
            sendingMsg = "remove " + to_string(createNodeId);
            sendMessage(mainSocket, sendingMsg);
            receivedMsg = receiveMessage(mainSocket);
            if (receivedMsg.substr(0, 2) == "OK")
                tree.erase(createNodeId);
            cout << receivedMsg << "\n";
```

```
        }
        else if (cmd == "exec") {
            cin >> id >> excmd >> word;
            if (excmd == '+')
                cin >> val;

            sendingMsg = "exec " + to_string(id);
            sendMessage(mainSocket, sendingMsg);

            receivedMsg = receiveMessage(mainSocket);
            if (receivedMsg == "Node is available") {
                if (excmd == '+') {
                    tree.dictInsertWord(id, word, val);
                    cout << "OK:" << id << endl;
                }
                if (excmd == '?') {
                    cout << "OK:" << id << ": ";
                    tree.dictGetWord(id, word);
                }
            }
            else {
                cout << receivedMsg << endl;
            }
        }
        else if (cmd == "ping") {
            cin >> id;
            vector<int> nodesList = tree.getNodesList();
            bool nodeExists = binary_search(nodesList.begin(),
nodesList.end(), id);
            if (nodeExists == 0) {
                cout << "Error: Not found\n";
            }
            else {
                sendMessage(mainSocket, "exec " + to_string(id));
                receivedMsg = receiveMessage(mainSocket);
                istringstream is;
                if (receivedMsg.substr(0, 5) == "Error")
                    cout << "OK:0\n";
                else
                    cout << "OK:1\n";
            }
        }
        else if (cmd == "pingall") {
            vector<int> nodesList = tree.getNodesList();
            if (nodesList.empty()) {
                cout << "Error: Tree is empty\n";
                continue;
            }

            sendMessage(mainSocket, "pingall");
            receivedMsg = receiveMessage(mainSocket);
            istringstream is;
            if (receivedMsg.substr(0, 5) == "Error")
                is = istringstream("");
            else
                is = istringstream(receivedMsg);

            set<int> receivedNodes;
            int rec_id;
            while (is >> rec_id) {
                receivedNodes.insert(rec_id);
            }
```

```
            cout << "Received nodes: ";
            for (const int &i : receivedNodes)
                cout << i << " ";

            cout << "\nNodes list:     ";
            for (const int &i : nodesList)
                cout << i << " ";

            cout << "\n";
        }
        else if (cmd == "exit") {
            break;
        }
        else {
            cout << "Unknown command\n";
        }
    }
    return 0;
}
```

```
#include "server.hpp"
#include <csignal>
#include <string>
#include <unistd.h>
#include "iostream"

using namespace std;
int main(int argc, char **argv)
{
    // айди и номер порта, к к-рым нужно подключиться
    int id = stoi(argv[1]);
    int parentPort = stoi(argv[2]);

    // подключение
    zmq::context_t context(2);
    zmq::socket_t parentSocket(context, ZMQ_REP);

    parentSocket.connect(getPortName(parentPort));

    zmq::socket_t leftSocket(context, ZMQ_REQ);
    zmq::socket_t rightSocket(context, ZMQ_REQ);

    int linger = 0;
    leftSocket.setsockopt(ZMQ_SNDTIMEO, 2000);
    leftSocket.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
    rightSocket.setsockopt(ZMQ_SNDTIMEO, 2000);
    rightSocket.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));

    int leftPort = bindSocket(leftSocket);
    int rightPort = bindSocket(rightSocket);

    // вспомогательные переменные
    int leftPid = 0;
    int rightPid = 0;
    int leftId = 0;
    int rightId = 0;

    string request;
    string cmd;

    while (true) {
```

```
        request = receiveMessage(parentSocket);
        istringstream cmdStream(request);
        cmdStream >> cmd;
        if (cmd == "id") {
            printf("debug\n");
            string parentString = "OK:" + to_string(id);
            sendMessage(parentSocket, parentString);
        }
        else if (cmd == "pid") {
            string parentString = "OK:" + to_string(getpid());
            sendMessage(parentSocket, parentString);
        }
        else if (cmd == "create") {
            int idToCreate;
            cmdStream >> idToCreate;
            if (idToCreate == id) {
                string msgString = "Error: Already exists";
                sendMessage(parentSocket, msgString);
            }
            else if (idToCreate < id) {
                if (leftPid == 0) {
                    leftPid = fork();
                    if (leftPid == -1) {
                        sendMessage(parentSocket, "Error: fork fails");
                        leftPid = 0;
                    }
                    else if (leftPid == 0) {
                        createNode(idToCreate, leftPort);
                    }
                    else {
                        leftId = idToCreate;
                        sendMessage(leftSocket, "pid");
                        sendMessage(parentSocket, receiveMes-
sage(leftSocket));
                    }
                }
                else {
                    sendMessage(leftSocket, request);
                    sendMessage(parentSocket, receiveMessage(leftSocket));
                }
            }
            else {
                if (rightPid == 0) {
                    rightPid = fork();
                    if (rightPid == -1) {
                        sendMessage(parentSocket, "Error: fork fails");
                        rightPid = 0;
                    }
                    else if (rightPid == 0) {
                        createNode(idToCreate, rightPort);
                    }
                    else {
                        rightId = idToCreate;
                        sendMessage(rightSocket, "pid");
                        sendMessage(parentSocket, receiveMes-
sage(rightSocket));
                    }
                }
                else {
                    sendMessage(rightSocket, request);
                    sendMessage(parentSocket, receiveMessage(rightSocket));
                }
            }
```

```
        }
        else if (cmd == "remove") {
            int idToDelete;
            cmdStream >> idToDelete;
            if (idToDelete < id) {
                if (leftId == 0) {
                    sendMessage(parentSocket, "Error: Node not found");
                }
                else if (leftId == idToDelete) {
                    sendMessage(leftSocket, "recursiveKilling");
                    receiveMessage(leftSocket);
                    kill(leftPid, SIGTERM);
                    kill(leftPid, SIGKILL);
                    leftId = 0;
                    leftPid = 0;
                    sendMessage(parentSocket, "OK");
                }
                else {
                    sendMessage(leftSocket, request);
                    sendMessage(parentSocket, receiveMessage(leftSocket));
                }
            }
            else {
                if (rightId == 0) {
                    sendMessage(parentSocket, "Error: Node not found");
                }
                else if (rightId == idToDelete) {
                    sendMessage(rightSocket, "recursiveKilling");
                    receiveMessage(rightSocket);
                    kill(rightPid, SIGTERM);
                    kill(rightPid, SIGKILL);
                    rightId = 0;
                    rightPid = 0;
                    sendMessage(parentSocket, "OK");
                }
                else {
                    sendMessage(rightSocket, request);
                    sendMessage(parentSocket, receiveMessage(rightSocket));
                }
            }
        }
        else if (cmd == "exec") {
            int execNodeId;
            cmdStream >> execNodeId;
            if (execNodeId == id) {
                string receiveMessage = "Node is available";
                sendMessage(parentSocket, receiveMessage);
            }
            else if (execNodeId < id) {
                if (leftPid == 0) {
                    string receiveMessage = "Error:" + to_string(execNodeId)
+ ": Not found";
                    sendMessage(parentSocket, receiveMessage);
                }
                else {
                    sendMessage(leftSocket, request);
                    sendMessage(parentSocket, receiveMessage(leftSocket));
                }
            }
            else {
                if (rightPid == 0) {
                    string receiveMessage = "Error:" + to_string(execNodeId)
+ ": Not found";
```

```
                    sendMessage(parentSocket, receiveMessage);
                }
                else {
                    sendMessage(rightSocket, request);
                    sendMessage(parentSocket, receiveMessage(rightSocket));
                }
            }
        }
        else if (cmd == "pingall") {
            ostringstream res;
            string leftRes;
            string rightRes;
            res << id << " ";
            if (leftPid != 0) {
                sendMessage(leftSocket, "pingall");
                leftRes = receiveMessage(leftSocket);
            }
            if (rightPid != 0) {
                sendMessage(rightSocket, "pingall");
                rightRes = receiveMessage(rightSocket);
            }
            if (!leftRes.empty() && leftRes.substr(0, 5) != "Error") {
                res << leftRes << " ";
            }
            if (!rightRes.empty() && rightRes.substr(0, 5) != "Error") {
                res << rightRes << " ";
            }
            sendMessage(parentSocket, res.str());
        }
        else if (cmd == "recursiveKilling") {
            if (leftPid == 0 && rightPid == 0) {
                sendMessage(parentSocket, "OK");
            }
            else {
                if (leftPid != 0) {
                    sendMessage(leftSocket, "recursiveKilling");
                    receiveMessage(leftSocket);
                    kill(leftPid, SIGTERM);
                    kill(leftPid, SIGKILL);
                }
                if (rightPid != 0) {
                    sendMessage(rightSocket, "recursiveKilling");
                    receiveMessage(rightSocket);
                    kill(rightPid, SIGTERM);
                    kill(rightPid, SIGKILL);
                }
                sendMessage(parentSocket, "OK");
            }
        }
        if (parentPort == 0) {
            break;
        }
    }
}
```

```
#pragma once

#include <iostream>
#include <vector>
#include <unordered_map>
```

```
class Tree {
private:
    struct Node;

public:
    Tree() = default;

    ~Tree()
    {
        deleteTree(root);
    }

    bool find(const int &id)
    {
        Node *temp = root;
        while (temp != nullptr) {
            if (temp->id == id)
                break;
            if (id > temp->id)
                temp = temp->right;
            if (id < temp->id)
                temp = temp->left;
        }
        return temp != nullptr;
    }

    void insert(int id)
    {
        if (root == nullptr) {
            root = new Node(id);
            return;
        }
        Node *temp = root;
        while (temp != nullptr) {
            if (id == temp->id)
                break;
            if (id < temp->id) {
                if (temp->left == nullptr) {
                    temp->left = new Node(id);
                    break;
                }
                temp = temp->left;
            }
            if (id > temp->id) {
                if (temp->right == nullptr) {
                    temp->right = new Node(id);
                    break;
                }
                temp = temp->right;
            }
        }
    }

    void erase(int id)
    {
        Node *prev id = nullptr;
        Node *temp = root;
        while (temp != nullptr) {
            if (id == temp->id) {
                if (prev_id == nullptr) {
                    root = nullptr;
                }
                else {
```

```
                    if (prev_id->left == temp)
                        prev_id->left = nullptr;
                    else
                        prev_id->right = nullptr;
                }
                deleteTree(temp);
            }
            else if (id < temp->id) {
                prev_id = temp;
                temp = temp->left;
            }
            else if (id > temp->id) {
                prev_id = temp;
                temp = temp->right;
            }
        }
    }

    std::vector<int> getNodesList() const
    {
        std::vector<int> result;
        getNodesList(root, result);
        return result;
    }

    void dictInsertWord(int id, std::string word, int value)
    {
        Node *node = getNodeById(root, id);
        node->dictionary[word] = value;
    }

    void dictGetWord(int id, std::string word)
    {
        Node *node = getNodeById(root, id);
        if (node->dictionary.find(word) == node->dictionary.end())
            std::cout << "'" << word << "' not found" << '\n';
        else
            std::cout << node->dictionary[word] << '\n';
    }

private:
    struct Node {
        Node(int id) : id(id) {}
        int id = 0;
        Node *left = nullptr;
        Node *right = nullptr;
        std::unordered_map<std::string, int> dictionary;
    };

    Node *root = nullptr;

    Node *getNodeById(Node *root, int id)
    {
        if (root == nullptr || root->id == id) {
            return root;
        }

        if (root->id < id) {
            return getNodeById(root->right, id);
        }

        return getNodeById(root->left, id);
    }
```

```
    void getNodesList(Node *node, std::vector<int> &v) const
    {
        if (node == nullptr)
            return;
        getNodesList(node->left, v);
        v.push_back(node->id);
        getNodesList(node->right, v);
    }

    void deleteTree(Node *node)
    {
        if (node == nullptr)
            return;
        deleteTree(node->left);
        deleteTree(node->right);
        delete node;
    }
};
```

```
#pragma once

#include <cstdlib>
#include <string>
#include <unistd.h>
#include <zmq.hpp>

// send message to the particular socket
bool sendMessage(zmq::socket_t &socket, const std::string &message_string)
{
    // message size init
    zmq::message_t message(message_string.size());
    // message content init
    memcpy(message.data(), message_string.c_str(), message_string.size());
    return socket.send(message);
}

std::string receiveMessage(zmq::socket_t &socket)
{
    zmq::message_t message;
    int recResult;
    // receiving message from socket
    try {
        recResult = (int)socket.recv(&message);
        if (recResult < 0) {
            perror("socket.recv()");
            exit(1);
        }
    }
    catch (...) {
        recResult = 0;
    }
    // transform to string
    std::string recieved_message((char *)message.data(), message.size());
    if (recieved_message.empty() || !recResult) {
        return "Error: Node is not available";
    }
    return recieved_message;
}

std::string getPortName(int port)
```

```
{
    return "tcp://127.0.0.1:" + std::to_string(port);
}
int bindSocket(zmq::socket_t &socket)
{
    int port = 8080;
    // create endpoint and bind it to the socket
    while (true) {
        try {
            socket.bind(getPortName(port));
            break;
        }
        catch (...) {
            port++;
        }
    }
    return port;
}

void createNode(int id, int port)
{
    // new node process
    execl("./child", "child", std::to_string(id).c_str(),
std::to_string(port).c_str(), NULL);
}
```

Демонстрация работы программы

```
OK:193
> create 2
OK:248
> create 3
OK:252
> pingall
Received nodes: 1 2
Nodes list:     1 2 3
> pingall
Received nodes: 1 3
Nodes list:     1 2 3
> create 4
OK:256
> pingall
Received nodes: 1 3
Nodes list:     1 2 3 4
> remove 2
OK
> pingall
Received nodes: 1 3
Nodes list:     1
> pingall
Received nodes: 1
Nodes list:     1
> ping 1
OK:1
```

Выводы

Я ознакомился и научился работать с очередью сообщений(ZMQ). Используя эту библеотеку, я реализовал взаимодействие между двумя разными программами.