**Московский авиационный институт (национальный исследовательский университет)**

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

**Лабораторная работа по предмету "Операционные системы" №5-7**

Студент: Федоров М. А.

Преподаватель: Миронов Е.С.

Группа: М8О-207Б-22

Дата: 15.09.2022

Оценка:

Подпись:

Оглавление

[Цель работы 3](#__RefHeading__738_2004454639)

[Постановка задачи 3](#__RefHeading__740_2004454639)

[Общие сведения о программе 4](#__RefHeading__742_2004454639)

[Общий алгоритм решения 5](#__RefHeading__744_2004454639)

[Реализация 5](#__RefHeading__746_2004454639)

[Пример работы 8](#__RefHeading__748_2004454639)

[Вывод 9](#__RefHeading__750_2004454639)

# **Цель работы**

Целью является приобретение практических навыков в:

* Управлении серверами сообщений (№5)
* Применение отложенных вычислений (№6)
* Интеграция программных систем друг с другом (№7)

# **Постановка задачи**

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.

*Вариант 41:*

*Топология 3*: Все вычислительные узлы хранятся в бинарном дереве поиска. [parent] — является необязательным параметром.

*Тип 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

*Тип проверки 1*:

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

# **Общие сведения о программе**

Программа представлена четырьмя файлами **–** tree.h, tree.cpp, client.cpp, main.cpp. Использовалась библиотека ZMQ.

В программе используются следующие системные вызовы:

**context() –** создание контекста для сокета

**connect() –** подключение сокета к указанному порту

**recv()** **–** чтение сообщения по указанному сокету

**send()** **–** отправка сообщения по указанному сокету

**fork() –** создание дочернего процесса

**execv() –** создание процесса с другой программой

# **Общий алгоритм решения**

# **Реализация**

**tree.h**

#pragma once

#include <vector>

struct Node

{

    int id;

    Node\* left;

    Node\* right;

};

class Tree

{

public:

    void push(int);

    void kill(int);

    std::vector<int> get\_nodes();

    ~Tree();

private:

    Node\* root = NULL;

    Node\* push(Node\* t, int);

    Node\* kill(Node\* t, int);

    void get\_nodes(Node\*, std::vector<int>&);

    void delete\_node(Node\*);

};

**-----------------------------------------------------------**

**tree.cpp**

#include <iostream>

#include <vector>

#include <algorithm>

#include "tree.h"

Tree::~Tree()

{

    delete\_node(root);

}

void Tree::push(int id)

{

    root = push(root, id);

}

void Tree::kill(int id)

{

    root = kill(root, id);

}

void Tree::delete\_node(Node\* node)

{

    if(node == NULL)

    {

        return;

    }

    delete\_node(node->right);

    delete\_node(node->left);

    delete node;

}

std::vector<int> Tree::get\_nodes()

{

    std::vector<int> result;

    get\_nodes(root, result);

    return result;

}

void Tree::get\_nodes(Node\* node, std::vector<int>& v)

{

    if (node == NULL)

    {

        return;

    }

    get\_nodes(node->left, v);

    v.push\_back(node->id);

    get\_nodes(node->right, v);

}

Node\* Tree::push(Node\* root, int val)

{

    if (root == NULL)

    {

        root = new Node;

        root->id = val;

        root->left = NULL;

        root->right = NULL;

        return root;

    }

    else if (val < root->id)

    {

        root->left = push(root->left, val);

    }

    else if (val >= root->id)

    {

        root->right = push(root->right, val);

    }

    return root;

}

Node\* Tree::kill(Node\* root\_node, int val)

{

    Node\* node;

    if (root\_node == NULL)

    {

        return NULL;

    }

    else if (val < root\_node->id)

    {

        root\_node->left = kill(root\_node->left, val);

    }

    else if (val >root\_node->id)

    {

        root\_node->right = kill(root\_node->right, val);

    }

    else

    {

        node = root\_node;

        if (root\_node->left == NULL)

        {

            root\_node = root\_node->right;

        }

        else if (root\_node->right == NULL)

        {

             root\_node = root\_node->left;

        }

        delete node;

    }

    if (root\_node == NULL)

    {

         return root\_node;

    }

    return root\_node;

}

**-----------------------------------------------------------**

**client.cpp**

#include <iostream>

#include <unistd.h>

#include <string>

#include <sstream>

#include <exception>

#include <signal.h>

#include "zmq.hpp"

const int PORTBASE  = 5050;

bool sendMessage(zmq::socket\_t& socket, const std::string& message) {

    zmq::message\_t zmqMessage(message.c\_str(), message.size());

    zmq::send\_result\_t result = socket.send(zmqMessage, zmq::send\_flags::none);

    return result.has\_value();

}

std::string recieveMessage(zmq::socket\_t &socket)

{

    zmq::message\_t message;

    bool ok = false;

    try

    {

        zmq::recv\_result\_t result = socket.recv(message, zmq::recv\_flags::none);

        ok = result.has\_value();

    }

    catch (...)

    {

        ok = false;

    }

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

    if (received\_message.empty() || !ok)

    {

        return "Error: Node is not available";

    }

    return received\_message;

}

void createNode(int id, int port)

{

    char\* arg0 = strdup("./client");

    char\* arg1 = strdup((std::to\_string(id)).c\_str());

    char\* arg2 = strdup((std::to\_string(port)).c\_str());

    char\* args[] = {arg0, arg1, arg2, NULL};

    execv("./client", args);

}

std::string getPortName(const int port)

{

    return "tcp://127.0.0.1:" + std::to\_string(port);

}

void create(zmq::socket\_t& parentsocket, zmq::socket\_t& socket, int& create\_id, int& id, int& pid)

{

    if (pid == -1)

    {

        sendMessage(parentsocket, "Error: Cannot fork");

        pid = 0;

    }

    else if (pid == 0)

    {

        createNode(create\_id,PORTBASE + create\_id);

    }

    else

    {

        id = create\_id;

        sendMessage(socket, "pid");

        sendMessage(parentsocket, recieveMessage(socket));

    }

}

void kill(zmq::socket\_t& parentsocket, zmq::socket\_t& socket,  int& delete\_id, int& id, int& pid, std::string& request\_string)

{

    if (id == 0)

    {

        sendMessage(parentsocket, "Error: Not found");

    }

    else if (id == delete\_id)

    {

        sendMessage(socket, "kill\_children");

        recieveMessage(socket);

        kill(pid,SIGTERM);

        kill(pid,SIGKILL);

        id = 0;

        pid = 0;

        sendMessage(parentsocket, "Ok");

    }

    else

    {

        sendMessage(socket, request\_string);

        sendMessage(parentsocket, recieveMessage(socket));

    }

}

void rl\_exec(zmq::socket\_t& parentsocket, zmq::socket\_t& socket,  int& id, int& pid, std::string& request\_string)

{

    if (pid == 0)

    {

        std::string recieveMessage = "Error:" + std::to\_string(id);

        recieveMessage += ": Not found";

        sendMessage(parentsocket, recieveMessage);

    }

    else

    {

        sendMessage(socket, request\_string);

        sendMessage(parentsocket, recieveMessage(socket));

    }

}

void exec(std::istringstream& request\_stream, zmq::socket\_t& parentsocket, zmq::socket\_t& left\_socket,

            zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid, int& id, std::string& request\_string)

{

    std::string size\_str;

    std::string number;

    int exec\_id;

    request\_stream >> exec\_id;

    if (exec\_id == id)

    {

        request\_stream >> size\_str;

        int size=stoi(size\_str);

        int sum=0;

        for (size\_t i = 0; i < size; i++)

        {

            request\_stream >> number;

            sum+=stoi(number);

        }

        std::string recieveMessage = std::to\_string(sum);

        sendMessage(parentsocket, recieveMessage);

    }

    else if (exec\_id < id)

    {

        rl\_exec(parentsocket, left\_socket, exec\_id,

                left\_pid, request\_string);

    }

    else

    {

        rl\_exec(parentsocket, right\_socket, exec\_id,

                right\_pid, request\_string);

    }

}

void pingall(zmq::socket\_t& parentsocket, int& id, zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket,int& left\_pid, int& right\_pid)

{

    std::ostringstream res;

    std::string left\_res;

    std::string right\_res;

    res << std::to\_string(id);

    if (left\_pid != 0)

    {

        sendMessage(left\_socket, "pingall");

        left\_res = recieveMessage(left\_socket);

    }

    if (right\_pid != 0)

    {

        sendMessage(right\_socket, "pingall");

        right\_res = recieveMessage(right\_socket);

    }

    if (!left\_res.empty() && left\_res.substr(0, std::min<int>(left\_res.size(),5) ) != "Error")

    {

        res << " " << left\_res;

    }

    if ((!right\_res.empty()) && (right\_res.substr(0, std::min<int>(right\_res.size(),5) ) != "Error"))

    {

        res << " "<< right\_res;

    }

    sendMessage(parentsocket, res.str());

}

void kill\_children(zmq::socket\_t& parentsocket, zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid)

{

    if (left\_pid == 0 && right\_pid == 0)

    {

        sendMessage(parentsocket, "Ok");

    }

    else

    {

        if (left\_pid != 0)

        {

            sendMessage(left\_socket, "kill\_children");

            recieveMessage(left\_socket);

            kill(left\_pid,SIGTERM);

            kill(left\_pid,SIGKILL);

        }

        if (right\_pid != 0)

        {

            sendMessage(right\_socket, "kill\_children");

            recieveMessage(right\_socket);

            kill(right\_pid,SIGTERM);

            kill(right\_pid,SIGKILL);

        }

        sendMessage(parentsocket, "Ok");

    }

}

int main(int argc, char\*\* argv)

{

    int id = std::stoi(argv[1]);

    int parent\_port = std::stoi(argv[2]);

    zmq::context\_t context(3);

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

    parentsocket.connect(getPortName(parent\_port));

    int left\_pid = 0;

    int right\_pid = 0;

    int left\_id = 0;

    int right\_id = 0;

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

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

    while(true)

    {

        std::string request\_string = recieveMessage(parentsocket);

        std::istringstream request\_stream(request\_string);

        std::string request;

        request\_stream >> request;

        if (request == "id")

        {

            std::string parent\_string = "Ok:" + std::to\_string(id);

            sendMessage(parentsocket, parent\_string);

        }

        else if (request == "pid")

        {

            std::string parent\_string = "Ok:" + std::to\_string(getpid());

            sendMessage(parentsocket, parent\_string);

        }

        else if (request == "create")

        {

            int create\_id;

            request\_stream >> create\_id;

            if (create\_id == id)

            {

                std::string message\_string = "Error: Already exists";

                sendMessage(parentsocket, message\_string);

            }

            else if (create\_id < id)

            {

                if (left\_pid == 0)

                {

                    left\_socket.bind(getPortName(PORTBASE + create\_id));

                    left\_pid = fork();

                    create(parentsocket, left\_socket, create\_id, left\_id, left\_pid);

                }

                else

                {

                    sendMessage(left\_socket, request\_string);

                    sendMessage(parentsocket, recieveMessage(left\_socket));

                }

            }

            else

            {

                if (right\_pid == 0)

                {

                    right\_socket.bind(getPortName(PORTBASE + create\_id));

                    right\_pid = fork();

                    create(parentsocket, right\_socket, create\_id, right\_id, right\_pid);

                }

                else

                {

                    sendMessage(right\_socket, request\_string);

                    sendMessage(parentsocket, recieveMessage(right\_socket));

                }

            }

        }

        else if (request == "kill")

        {

            int delete\_id;

            request\_stream >> delete\_id;

            if (delete\_id < id)

            {

                kill(parentsocket, left\_socket, delete\_id, left\_id, left\_pid, request\_string);

            }

            else

            {

                kill(parentsocket, right\_socket, delete\_id, right\_id, right\_pid, request\_string);

            }

        }

        else if (request == "exec")

        {

            exec(request\_stream, parentsocket, left\_socket, right\_socket, left\_pid, right\_pid, id, request\_string);

        }

        else if (request == "pingall")

        {

            pingall(parentsocket, id, left\_socket, right\_socket, left\_pid, right\_pid);

        }

        else if (request == "kill\_children")

        {

            kill\_children(parentsocket, left\_socket, right\_socket, left\_pid, right\_pid);

        }

        if (parent\_port == 0)

        {

            break;

        }

    }

    return 0;

}

**-----------------------------------------------------------**

**main.cpp**

#include <iostream>

#include <unistd.h>

#include <string>

#include <vector>

#include <set>

#include <sstream>

#include <signal.h>

#include "zmq.hpp"

#include "tree.h"

const int PORTBASE = 5050;

Tree T;

std::string request;

int childProcessId = 0;

int childId = 0;

zmq::context\_t context(1);

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

bool sendMessage(zmq::socket\_t &socket, const std::string &message)

{

    zmq::message\_t zmqMessage(message.c\_str(), message.size());

    zmq::send\_result\_t result = socket.send(zmqMessage, zmq::send\_flags::none);

    return result.has\_value();

}

std::string recieveMessage(zmq::socket\_t &socket)

{

    zmq::message\_t message;

    bool ok = false;

    try

    {

        zmq::recv\_result\_t result = socket.recv(message, zmq::recv\_flags::none);

        ok = result.has\_value();

    }

    catch (...)

    {

        ok = false;

    }

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

    if (receivedMessage.empty() || !ok)

    {

        return "Error: Node is not available";

    }

    return receivedMessage;

}

void createNode(int id, int port)

{

    char \*arg0 = strdup("./client");

    char \*arg1 = strdup((std::to\_string(id)).c\_str());

    char \*arg2 = strdup((std::to\_string(port)).c\_str());

    char \*args[] = {arg0, arg1, arg2, NULL};

    execv("./client", args);

}

std::string getPortName(const int port)

{

    return "tcp://127.0.0.1:" + std::to\_string(port);

}

bool isNumber(std::string value)

{

    try

    {

        int check = std::stoi(value);

        return true;

    }

    catch (std::exception &e)

    {

        std::cout << "Error: " << e.what() << "\n";

        return false;

    }

}

void funcCreate()

{

    size\_t nodeId = 0;

    std::string lineInput = "";

    std::string result = "";

    std::cin >> lineInput;

    if (!isNumber(lineInput))

    {

        return;

    }

    nodeId = stoi(lineInput);

    if (childProcessId == 0)

    {

        mainSocket.bind(getPortName(PORTBASE + nodeId));

        childProcessId = fork();

        if (childProcessId == -1)

        {

            std::cout << "Unable to create first worker node\n";

            childProcessId = 0;

            exit(1);

        }

        else if (childProcessId == 0)

        {

            createNode(nodeId, PORTBASE + nodeId);

        }

        else

        {

            childId = nodeId;

            sendMessage(mainSocket, "pid");

            result = recieveMessage(mainSocket);

        }

    }

    else

    {

        std::string mеssage\_send = "create " + std::to\_string(nodeId);

        sendMessage(mainSocket, mеssage\_send);

        result = recieveMessage(mainSocket);

    }

    if (result.substr(0, 2) == "Ok")

    {

        T.push(nodeId);

    }

    std::cout << result << "\n";

}

void funcKill()

{

    int nodeId = 0;

    std::string lineInput = "";

    std::cin >> lineInput;

    if (!isNumber(lineInput))

    {

        return;

    }

    nodeId = stoi(lineInput);

    if (childProcessId == 0)

    {

        std::cout << "Error: Not found\n";

        return;

    }

    if (nodeId == childId)

    {

        kill(childProcessId, SIGTERM);

        kill(childProcessId, SIGKILL);

        childId = 0;

        childProcessId = 0;

        T.kill(nodeId);

        std::cout << "Ok\n";

        return;

    }

    std::string message = "kill " + std::to\_string(nodeId);

    sendMessage(mainSocket, message);

    std::string recieved\_message = recieveMessage(mainSocket);

    if (recieved\_message.substr(0, std::min<int>(recieved\_message.size(), 2)) == "Ok")

    {

        T.kill(nodeId);

    }

    std::cout << recieved\_message << "\n";

}

void funcExec()

{

    std::string stringID = "";

    std::string amountNumbers = "";

    std::string number = "";

    int id = 0;

    std::cin >> stringID >> amountNumbers;

    if (!isNumber(stringID))

    {

        return;

    }

    if (!isNumber(amountNumbers))

    {

        return;

    }

    std::vector<std::string> inputNumbers;

    for (size\_t i = 0; i < stoi(amountNumbers); i++)

    {

        std::cin >> number;

        inputNumbers.push\_back(number);

    }

    id = stoi(stringID);

    std::string messageLine = "exec " + std::to\_string(id) + " " + amountNumbers;

    for (size\_t i = 0; i < stoi(amountNumbers); i++)

    {

        messageLine += (" " + (inputNumbers[i]));

    }

    sendMessage(mainSocket, messageLine);

    std::string result = recieveMessage(mainSocket);

    std::cout << result << "\n";

}

int main()

{

    std::cout << "requests:\n";

    std::cout << "create id\n";

    std::cout << "exec id amount\_num num1 num2...num\_n\n";

    std::cout << "kill id\n";

    std::cout << "pingall\n";

    std::cout << "exit\n"

              << std::endl;

    while (1)

    {

        std::cin >> request;

        if (request == "create"){

            funcCreate();

        }

        else if (request == "kill"){

            funcKill();

        }

        else if (request == "exec"){

            funcExec();

        }

        else if (request == "pingall")

        {

            sendMessage(mainSocket, "pingall");

            std::string recieved = recieveMessage(mainSocket);

            std::istringstream is;

            if (recieved.substr(0, std::min<int>(recieved.size(), 5)) == "Error")

            {

                is = std::istringstream("");

            }

            else

            {

                is = std::istringstream(recieved);

            }

            std::set<int> recieved\_T;

            int recievedId;

            while (is >> recievedId)

            {

                recieved\_T.insert(recievedId);

            }

            std::vector<int> from\_tree = T.get\_nodes();

            auto part\_it = partition(from\_tree.begin(), from\_tree.end(), [&recieved\_T](int a)

                                     { return recieved\_T.count(a) == 0; });

            if (part\_it == from\_tree.begin())

            {

                std::cout << "Ok:-1\n";

            }

            else

            {

                std::cout << "Ok:";

                for (auto it = from\_tree.begin(); it != part\_it; ++it)

                {

                    std::cout << \*it << " ";

                }

                std::cout << "\n";

            }

        }

        else if (request == "exit")

        {

            int n = system("killall client");

            break;

        }

    }

    return 0;

}

# **Пример работы**

**Test 1**

|  |  |
| --- | --- |
| Input | Output |
| Create 7  Create 4  Create 3  Create 5  Exec 4 3 1 2 3  Pingall  Kill 3  Pingall  Kill 4  Pingall  exit |  |

# **Вывод**