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

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

Студент: Меркулов Фёдор Алексеевич
Группа: М8О-207Б-20
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/WhatTheMUCK/OSi/tree/main/LR_6-8

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.

Общие сведения о программе: программа состоит из 5 файлов: main.cpp (получает команды от пользователя и отправляет их в вычислительный узел), client.cpp (получает эти команды и выполняет их), tree.cpp, tree.h (реализация бинарного дерева поиска), Makefile.

Общий метод и алгоритм решения:

- create id - вставка вычислительного узла в бинарное дерево
- exec id subcommand - отправка подкоманды вычислительному узлу
- kill id - удаление вычислительного узла и всех его дочерних узлов из дерева
- ping id – проверка доступности конкретного узла

Исходный код:

main.cpp

```
#include <iostream>
#include <unistd.h>
#include <string>
#include <vector>
#include <sstream>
#include <signal.h>
#include <cassert>
#include "zmq.hpp"
#include "tree.h"

using namespace std;

const int WAIT_TIME = 500;
int n = 2;
const int PORT_BASE  = 5050;


bool send_message(zmq::socket_t &socket, const string &message_string)
{
    zmq::message_t message(message_string.size());
```

```
    memcpy(message.data(), message_string.c_str(), message_string.size());
//Копирует содержимое одной области памяти в другую
    return socket.send(message);
}

string recieve_message(zmq::socket_t &socket)
{
    zmq::message_t message;
    bool ok = false;
    try
    {
        ok = socket.recv(&message);
    }
    catch (...)
    {
        ok = false;
    }
    string recieved_message(static_cast<char*>(message.data()), message.size());
    if (recieved_message.empty() || !ok)
    {
        return "Root is dead!";
    }
    return recieved_message;
}

void create_node(int id, int port)
{
    char* arg0 = strdup("./client");
    char* arg1 = strdup((to_string(id)).c_str());
    char* arg2 = strdup((to_string(port)).c_str());
    char* args[] = {arg0, arg1, arg2, NULL};
    execv("./client", args);
}

string get_port_name(const int port)
{
    return "tcp://127.0.0.1:" + to_string(port);
}

bool is_number(string val)
{
    try
    {
        int tmp = stoi(val);
        return true;
    }
    catch(exception& e)
    {
        cout << "Error: " << e.what() << "\n";
        return false;
    }
}

int main()
{
    Tree T;
    string command;
    int child_pid = 0;
    int child_id = 0;
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REQ);
    cout << "Commands:\n";
    cout << "create id\n";
    cout << "exec id (text_string, pattern_string)\n";
    cout << "kill id\n";
```

```
cout << "ping id\n";
cout << "exit\n" << endl;
while(1)
{
    cin >> command;
    if (command == "create")
    {
      n++;
        size_t node_id = 0;
        string str = "";
        string result = "";
        cin >> str;
        if (!is_number(str))
        {
            continue;
        }
        node_id = stoi(str);
        if (child_pid == 0)
        {
            main_socket.bind(get_port_name(PORT_BASE + node_id));
            main_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
         main_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
               child_pid = fork();
            if (child_pid == -1)
            {
                cout << "Unable to create first worker node\n";
                child_pid = 0;
                exit(1);
            }
            else if (child_pid == 0)
            {
                create_node(node_id, PORT_BASE + node_id);
            }
            else
            {
                child_id = node_id;
                main_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
             main_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
                send_message(main_socket,"pid");
                result = recieve_message(main_socket);
            }
        }
        else
        {
         main_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
         main_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
            string msg_s = "create " + to_string(node_id);
            send_message(main_socket, msg_s);
            result = recieve_message(main_socket);
        }
        if (result.substr(0, 2) == "Ok")
        {
            T.push(node_id);
        }
        cout << result << "\n";
    }
    else if (command == "kill")
    {
        int node_id = 0;
        string str = "";
        cin >> str;
        if (!is_number(str))
        {
            continue;
        }
```

```
            node_id = stoi(str);
            if (child_pid == 0)
            {
                cout << "Error: Not found\n";
                continue;
            }
            if (node_id == child_id)
            {
                kill(child_pid, SIGTERM);
                kill(child_pid, SIGKILL);
                child_id = 0;
                child_pid = 0;
                T.kill(node_id);
                cout << "Ok\n";
                continue;
            }
            string message_string = "kill " + to_string(node_id);
            send_message(main_socket, message_string);
            string recieved_message;
          recieved_message = recieve_message(main_socket);
            if (recieved_message.substr(0, min<int>(recieved_message.size(), 2)) ==
"Ok")
            {
                T.kill(node_id);
            }
            cout << recieved_message << "\n";
        }
        else if (command == "exec")
        {
            string id_str = "";
            string text_string = "";
          string pattern_string = "";
            int id = 0;
            cin >> id_str >> text_string >> pattern_string;
            if (!is_number(id_str))
            {
                continue;
            }
            id = stoi(id_str);
            string message_string = "exec " + to_string(id) + " " + text_string + "
" + pattern_string;
            send_message(main_socket, message_string);
            string recieved_message = recieve_message(main_socket);
            cout << recieved_message << "\n";
        }
      else if (command == "ping")
      {
          string id_str = "";
            int id = 0;
            cin >> id_str;
            if (!is_number(id_str))
            {
                continue;
            }
            id = stoi(id_str);
            string message_string = "ping " + to_string(id);
            send_message(main_socket, message_string);
          string recieved_message = recieve_message(main_socket);
            cout << recieved_message << "\n";
      }
        else if (command == "exit")
        {
            int n = system("killall client");
            break;
        }
```

```
    }
    return 0;

}
```

client.cpp

```
#include <iostream>
#include <unistd.h>
#include <string>
#include <sstream>
#include <exception>
#include <signal.h>
#include "zmq.hpp"

using namespace std;

const int WAIT_TIME = 500;
const int PORT_BASE  = 5050;
int n = 2;

bool send_message(zmq::socket_t &socket, const string &message_string)
{
    zmq::message_t message(message_string.size());
    memcpy(message.data(), message_string.c_str(), message_string.size());
    return socket.send(message);
}

string recieve_message(zmq::socket_t &socket)
{
    zmq::message_t message;
    bool ok = false;
    try
    {
        ok = socket.recv(&message);
    }
    catch (...)
    {
        ok = false;
    }
    string recieved_message(static_cast<char*>(message.data()), message.size());
    if (recieved_message.empty() || !ok)
    {
        return "";
    }
    return recieved_message;
}

void create_node(int id, int port)
{
    char* arg0 = strdup("./client");
    char* arg1 = strdup((to_string(id)).c_str());
    char* arg2 = strdup((to_string(port)).c_str());
    char* args[] = {arg0, arg1, arg2, NULL};
    execv("./client", args);
}

string get_port_name(const int port)
{
    return "tcp://127.0.0.1:" + to_string(port);
}

void rl_create(zmq::socket_t& parent_socket, zmq::socket_t& socket, int& create_id,
int& id, int& pid)
{
```

```
    if (pid == -1)
    {
        send_message(parent_socket, "Error: Cannot fork");
        pid = 0;
    }
    else if (pid == 0)
    {
        create_node(create_id,PORT_BASE + create_id);
    }
    else
    {
        id = create_id;
        send_message(socket, "pid");
        send_message(parent_socket, recieve_message(socket));
    }
}

void rl_kill(zmq::socket_t& parent_socket, zmq::socket_t& socket,  int& delete_id,
int& id, int& pid, string& request_string)
{
    if (id == 0)
    {
        send_message(parent_socket, "Error: Not found");
    }
    else if (id == delete_id)
    {
        send_message(socket, "kill_children");
        recieve_message(socket);
        kill(pid,SIGTERM);
        kill(pid,SIGKILL);
        id = 0;
        pid = 0;
        send_message(parent_socket, "Ok");
    }
    else
    {
        send_message(socket, request_string);
        send_message(parent_socket, recieve_message(socket));
    }
}

void rl_exec(zmq::socket_t& parent_socket, zmq::socket_t& socket,  int& id, int&
pid, string& request_string)
{
    if (pid == 0)

    {
        string recieve_message = "Error:" + to_string(id);
        recieve_message += ": Not found";
        send_message(parent_socket, recieve_message);
    }
    else
    {
        send_message(socket, request_string);
        string str = recieve_message(socket);
        if (str == "")
             str = "Error: Node is unavailable";
        send_message(parent_socket, str);
    }
}

void rl_ping(zmq::socket_t& parent_socket, zmq::socket_t& socket,  int& id, int&
pid, string& request_string)
{
    if (pid == 0)
    {
```

```
        string recieve_message = "Error:" + to_string(id);
        recieve_message += ": Not found";
        send_message(parent_socket, recieve_message);
    }
    else
    {
        send_message(socket, request_string);
        string str = recieve_message(socket);
        if (str == "")
            str = "Ok: 0";
        send_message(parent_socket, str);
    }
}

void exec(istringstream& command_stream, zmq::socket_t& parent_socket,
zmq::socket_t& left_socket,
            zmq::socket_t& right_socket, int& left_pid, int& right_pid, int& id,
string& request_string)
{
    string text_string, pattern_string;
    int exec_id;
    command_stream >> exec_id;
    if (exec_id == id)
    {
        command_stream >> text_string;
      command_stream >> pattern_string;
        string recieve_message = "";
      string answer = "";
        int index = 0;
          while ((index = text_string.find(pattern_string, index)) !=
string::npos){
             answer += to_string(index) + ";";
             index += pattern_string.length();
      }
      if (!answer.empty())
            answer.pop_back();
        recieve_message = "Ok:" + to_string(id) + ":";
      if (!answer.empty()){
            recieve_message += answer;
      } else {
            recieve_message += "-1";
      }
        send_message(parent_socket, recieve_message);
    }
    else if (exec_id < id)
    {
        rl_exec(parent_socket, left_socket, exec_id, left_pid, request_string);
    }
    else
    {
        rl_exec(parent_socket, right_socket, exec_id, right_pid, request_string);
    }
}

void ping(istringstream& command_stream, zmq::socket_t& parent_socket,
zmq::socket_t& left_socket,
            zmq::socket_t& right_socket, int& left_pid, int& right_pid, int& id,
string& request_string)
{
    int ping_id;
    string recieve_message;
    command_stream >> ping_id;
    if (ping_id == id)
    {
        recieve_message = "Ok: 1";
```

```
            send_message(parent_socket, recieve_message);
        }
        else if (ping_id < id)
        {
            rl_ping(parent_socket, left_socket, ping_id, left_pid, request_string);
        }
        else
        {
            rl_ping(parent_socket, right_socket, ping_id, right_pid, request_string);
        }
    }

    void kill_children(zmq::socket_t& parent_socket, zmq::socket_t& left_socket,
    zmq::socket_t& right_socket, int& left_pid, int& right_pid)
    {
        if (left_pid == 0 && right_pid == 0)
        {
            send_message(parent_socket, "Ok");
        }
        else
        {
            if (left_pid != 0)
            {
                send_message(left_socket, "kill_children");
                recieve_message(left_socket);
                kill(left_pid,SIGTERM);
                kill(left_pid,SIGKILL);
            }
            if (right_pid != 0)
            {
                send_message(right_socket, "kill_children");
                recieve_message(right_socket);
                kill(right_pid,SIGTERM);
                kill(right_pid,SIGKILL);
            }
            send_message(parent_socket, "Ok");
        }
    }

    int main(int argc, char** argv)
    {
        int id = stoi(argv[1]);
        int parent_port = stoi(argv[2]);
        zmq::context_t context(3);
        zmq::socket_t parent_socket(context, ZMQ_REP);
        parent_socket.connect(get_port_name(parent_port));
        parent_socket.setsockopt(ZMQ_RCVTIMEO, WAIT_TIME);
        parent_socket.setsockopt(ZMQ_SNDTIMEO, WAIT_TIME);
        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(1)
        {
            string request_string = recieve_message(parent_socket);
            istringstream command_stream(request_string);
            string command;
            command_stream >> command;
            if (command == "id")
            {
                string parent_string = "Ok:" + to_string(id);
                send_message(parent_socket, parent_string);
            }
```

```
        else if (command == "pid")
        {
            string parent_string = "Ok:" + to_string(getpid());
            send_message(parent_socket, parent_string);
        }
        else if (command == "create")
        {
            int create_id;
            command_stream >> create_id;
            if (create_id == id)
            {
                string message_string = "Error: Already exists";
                send_message(parent_socket, message_string);
            }
            else if (create_id < id)
            {
                if (left_pid == 0)
                {
                    left_socket.bind(get_port_name(PORT_BASE + create_id));
                    left_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
                  left_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
                    left_pid = fork();
                    rl_create(parent_socket, left_socket, create_id, left_id,
left_pid);
                }
                else
                {
                    send_message(left_socket, request_string);
                    string str = recieve_message(left_socket);
                    if (str == "")
                    {
                        left_socket.bind(get_port_name(PORT_BASE + create_id));
                        left_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
                        left_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
                        left_pid = fork();
                        rl_create(parent_socket, left_socket, create_id, left_id,
left_pid);
                    }
                    else
                    {
                        send_message(parent_socket, str);
                        n++;
                        left_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
                        left_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
                    }
                }
            }
            else
            {
                if (right_pid == 0)
                {
                    right_socket.bind(get_port_name(PORT_BASE + create_id));
                    right_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
                  right_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
                    right_pid = fork();
                    rl_create(parent_socket, right_socket, create_id, right_id,
right_pid);
                }
                else
                {
                    send_message(right_socket, request_string);
                    string str = recieve_message(right_socket);
                    if (str == "")
                    {
                        right_socket.bind(get_port_name(PORT_BASE + create_id));
```

```
                        right_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
                        right_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
                        right_pid = fork();
                        rl_create(parent_socket, right_socket, create_id, right_id,
right_pid);
                    }
                    else
                    {
                        send_message(parent_socket, str);
                        n++;
                        right_socket.setsockopt(ZMQ_RCVTIMEO, n * WAIT_TIME);
                        right_socket.setsockopt(ZMQ_SNDTIMEO, n * WAIT_TIME);
                    }
                }
            }
        }
        else if (command == "kill")
        {
            int delete_id;
            command_stream >> delete_id;
            if (delete_id < id)
            {
                rl_kill(parent_socket, left_socket, delete_id, left_id, left_pid,
request_string);
            }
            else
            {
                rl_kill(parent_socket, right_socket, delete_id, right_id, right_pid,
request_string);
            }
        }
        else if (command == "exec")
        {
            exec(command_stream, parent_socket, left_socket, right_socket, left_pid,
right_pid, id, request_string);
        }
      else if (command == "ping")
      {
          ping(command_stream, parent_socket, left_socket, right_socket, left_pid,
right_pid, id, request_string);
      }
        else if (command == "kill_children")
        {
            kill_children(parent_socket, left_socket, right_socket, left_pid,
right_pid);
        }
        if (parent_port == 0)
        {
            break;
        }
    }
    return 0;
}
```

tree.cpp

```
{
        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;
}
```

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*);
};
```

Makefile

```
SRC = main.cpp tree.cpp
OBJ = $(SRC:.cpp=.o)

SRC2 = client.cpp
OBJ2 = $(SRC2:.cpp=.o)

all: main client

main: $(OBJ)
	g++ -Wno-unused-variable $(OBJ) -o $@ -lrt -lzmq -g

client: $(OBJ2)
	g++ -Wno-unused-variable $(OBJ2) -o $@ -lrt -lzmq

.cpp.o:
	g++ -Wno-unused-variable -c $< -o $@

client.o: tree.h
main.o: tree.h
tree.o: tree.h

clean:
	rm client.o main.o tree.o
```

Демонстрация работы программы

papik@papik-VirtualBox:~/OSlaba6-8/build$ cmake ..

-- Configuring done

-- Generating done

-- Build files have been written to: /home/papik/OSlaba6-8/build

papik@papik-VirtualBox:~/OSlaba6-8/build$ make

[50%] Built target client

[100%] Built target server

papik@papik-VirtualBox:~/OSlaba6-8/build$ ./server

Commands:

create id

exec id (text_string, pattern_string)

kill id

ping id

exit

create 2

Ok:6274

ping 2

Ok: 1

exec 2

abracabra abra

Ok:2:0;5

create 3

Ok:6279

create 4

Ok:6284

create 5

Ok:6289

create 6

Ok:6294

create 7

Ok:6299

create 8

Ok:6304

ping 8

Ok: 1

ping 8

Ok: 0

ping 7

Ok: 0

ping 6

Ok: 0

ping 5

Ok: 0

ping 4

Ok: 0

ping 3

Ok: 0

ping 2

Ok: 1

create 9

Ok:6311

ping 9

Ok: 1

exec 9 papa ap

Ok:9:1

create 10

Ok:6316

create 11

Ok:6321

ping 10

Ok: 1

ping 11

Ok: 1

kill 10

Ok

ping 10

Error:10: Not found

ping 11

Error:11: Not found

exit

Выводы

Выполняя лабораторную работу, я освоил основы библиотеки ZMQ, а также познакомился с очередями сообщений.