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: Попов Матвей Романович
Группа: М8О-208Б-20
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/.../os_lab6-8

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 7 файлов: main.cpp (получает команды от пользователя и отправляет их в вычислительный узел), client.cpp (получает эти команды и выполняет их), timer.cpp, timer.h (реализация таймера), 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
- pingall — все вычислительные узлы подтверждают свою работоспособность

Исходный код:

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 WAIT_TIME = 1000;
const int PORT_BASE  = 5050;

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

```
}

std::string recieve_message(zmq::socket_t &socket)
{
    zmq::message_t message;
    bool ok = false;
    try
    {
        ok = socket.recv(&message);
    }
    catch (...)
    {
        ok = false;
    }
    std::string recieved_message(static_cast<char*>(message.data()), message.size());
    if (recieved_message.empty() || !ok)
    {
        return "Error: Node is not available";
    }
    return recieved_message;
}

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

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

int main()
{
    Tree T;
    std::string command;
    int child_pid = 0;
    int child_id = 0;
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REQ);
    std::cout << "Commands:\n";
    std::cout << "create id\n";
    std::cout << "exec id subcommand (start/stop/time)\n";
    std::cout << "kill id\n";
    std::cout << "pingall\n";
    std::cout << "exit\n" << std::endl;
```

```
while(1)
{
    std::cin >> command;
    if (command == "create")
    {
        size_t node_id = 0;
        std::string str = "";
        std::string result = "";
        std::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));
            child_pid = fork();
            if (child_pid == -1)
            {
                std::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;
                send_message(main_socket,"pid");
                result = recieve_message(main_socket);
            }
        }
        else
        {
            std::string msg_s = "create " + std::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);
        }
        std::cout << result << "\n";
    }
    else if (command == "kill")
    {
        int node_id = 0;
        std::string str = "";
        std::cin >> str;
        if (!is_number(str))
        {
            continue;
        }
        node_id = stoi(str);
        if (child_pid == 0)
        {
            std::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);
            std::cout << "Ok\n";
            continue;
        }
        std::string message_string = "kill " + std::to_string(node_id);
        send_message(main_socket, message_string);
        std::string recieved_message = recieve_message(main_socket);
        if (recieved_message.substr(0, std::min<int>(recieved_message.size(), 2)) == "Ok")
        {
            T.kill(node_id);
        }
        std::cout << recieved_message << "\n";
    }
    else if (command == "exec")
    {
        std::string id_str = "";
        std::string subcommand = "";
        int id = 0;
        std::cin >> id_str >> subcommand;
        if (!is_number(id_str))
        {
            continue;
        }
        id = stoi(id_str);
        if ((subcommand != "start") && (subcommand != "stop") && (subcommand != "time"))
        {
            std::cout << "Wrong subcommandmand\n";
            continue;
        }
        std::string message_string = "exec " + std::to_string(id) + " " + subcommand;
        send_message(main_socket, message_string);
        std::string recieved_message = recieve_message(main_socket);
        std::cout << recieved_message << "\n";
    }
    else if (command == "pingall")
    {
        send_message(main_socket,"pingall");
        std::string recieved = recieve_message(main_socket);
        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 rec_id;
        while (is >> rec_id)
        {
            recieved_T.insert(rec_id);
        }
        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 (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"
#include "timer.h"

const int WAIT_TIME = 1000;
const int PORT_BASE  = 5050;

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

std::string recieve_message(zmq::socket_t &socket)
{
    zmq::message_t message;
    bool ok = false;
    try
    {
        ok = socket.recv(&message);
    }
    catch (...)
    {
        ok = false;
    }
    std::string recieved_message(static_cast<char*>(message.data()), message.size());
    if (recieved_message.empty() || !ok)
    {
        return "Error: Node is not available";
    }
    return recieved_message;
}

void create_node(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 get_port_name(const int port)
{
    return "tcp://127.0.0.1:" + std::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, std::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,
std::string& request_string)
{
    if (pid == 0)
    {
        std::string recieve_message = "Error:" + std::to_string(id);
        recieve_message += ": Not found";
```

```
        send_message(parent_socket, recieve_message);
    }
    else
    {
        send_message(socket, request_string);
        send_message(parent_socket, recieve_message(socket));
    }
}

void exec(std::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, std::string&
request_string, Timer* timer)
{
    std::string subcommand;
    int exec_id;
    command_stream >> exec_id;
    if (exec_id == id)
    {
        command_stream >> subcommand;
        std::string recieve_message = "";
        if (subcommand == "start")
        {
            timer->start_timer();
            recieve_message = "Ok:" + std::to_string(id);
            send_message(parent_socket, recieve_message);
        }
        else if (subcommand == "stop")
        {
            timer->stop_timer();
            recieve_message = "Ok:" + std::to_string(id);
            send_message(parent_socket, recieve_message);
        }
        else if (subcommand == "time")
        {
            recieve_message = "Ok:" + std::to_string(id) + ": ";
            recieve_message += std::to_string(timer->get_time());
            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 pingall(zmq::socket_t& parent_socket, 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)
    {
        send_message(left_socket, "pingall");
        left_res = recieve_message(left_socket);
```

```
    }
    if (right_pid != 0)
    {
        send_message(right_socket, "pingall");
        right_res = recieve_message(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;
    }
    send_message(parent_socket, res.str());
}

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)
{
    Timer timer;
    int id = std::stoi(argv[1]);
    int parent_port = std::stoi(argv[2]);
    zmq::context_t context(3);
    zmq::socket_t parent_socket(context, ZMQ_REP);
    parent_socket.connect(get_port_name(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(1)
    {
        std::string request_string = recieve_message(parent_socket);
        std::istringstream command_stream(request_string);
        std::string command;
```

```
        command_stream >> command;
        if (command == "id")
        {
            std::string parent_string = "Ok:" + std::to_string(id);
            send_message(parent_socket, parent_string);
        }
        else if (command == "pid")
        {
            std::string parent_string = "Ok:" + std::to_string(getpid());
            send_message(parent_socket, parent_string);
        }
        else if (command == "create")
        {
            int create_id;
            command_stream >> create_id;
            if (create_id == id)
            {
                std::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_pid = fork();
                    rl_create(parent_socket, left_socket, create_id, left_id, left_pid);
                }
                else
                {
                    send_message(left_socket, request_string);
                    send_message(parent_socket, recieve_message(left_socket));
                }
            }
            else
            {
                if (right_pid == 0)
                {
                    right_socket.bind(get_port_name(PORT_BASE + create_id));
                    right_pid = fork();
                    rl_create(parent_socket, right_socket, create_id, right_id, right_pid);
                }
                else
                {
                    send_message(right_socket, request_string);
                    send_message(parent_socket, recieve_message(right_socket));
                }
            }
        }
        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, &timer);
        }
        else if (command == "pingall")
        {
            pingall(parent_socket, id, left_socket, right_socket, left_pid, right_pid);
        }
        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.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;
}
```

timer.h

```
#pragma once
#include <chrono>

class Timer
{
public:
    Timer() = default;
    ~Timer() = default;
    void start_timer();
    void stop_timer();
    int get_time();
private:
    bool is_timer_started = false;
    std::chrono::steady_clock::time_point start_;
    std::chrono::steady_clock::time_point finish_;
};
```

timer.cpp

```
#include "timer.h"

void Timer::start_timer()
{
    is_timer_started = true;
    start_ = std::chrono::steady_clock::now();
}

void Timer::stop_timer()
{
    if (is_timer_started)
    {
        is_timer_started = false;
        finish_ = std::chrono::steady_clock::now();
```

```
    }
}

int Timer::get_time()
{
    if (is_timer_started)
    {
        finish_ = std::chrono::steady_clock::now();
    }
    return std::chrono::duration_cast<std::chrono::milliseconds>(finish_ - start_).count();
}
```

Makefile

```
CC = g++
CFLAGS = -Wno-unused-variable

LDFLAGS = -lrt -lzmq
SRC = main.cpp tree.cpp
OBJ=$(SRC:.cpp=.o)

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

all: main client

main: $(OBJ)
	$(CC) $(CFLAGS) $(OBJ) -o $@ $(LDFLAGS)

client: $(OBJ2)
	$(CC) $(CFLAGS) $(OBJ2) -o $@ $(LDFLAGS)

.cpp.o:
	$(CC) $(CFLAGS) -c $< -o $@

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

clean:
	rm -rf *.o
```

Демонстрация работы программы

```
papey@PAPEY:~/Ubuntu/OS/os_lab6-8/src$ ./main
Commands:
create id
exec id subcommand (start/stop/time)
kill id
pingall
exit

create 4
Ok:10686
create 2
Ok:10691
create 8
Ok:10711
exec 4 start
Ok:4
exec 8 start
```

```
Ok:8
exec 4 time
Ok:4: 10283
exec 8 time
Ok:8: 8413
kill 2
Ok
pingall
Ok:-1
Exit
```

Выводы

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