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: Старцев Иван Романович
Группа: М8О-201Б-21
Преподаватель: Миронов Евгений Сергеевич
Вариант: 35
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/IvanTvardovsky/OS-labs

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 4 файлов: main.cpp (получает команды от пользователя и отправляет их в вычислительный узел), client.cpp (получает эти команды и выполняет их), tree.cpp и tree.h (реализация бинарного дерева поиска).

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 "../include/tree.h"
#include "zmq.hpp"

using namespace std;

const int TIMER = 500;
const int DEFAULT_PORT = 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 receive_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 << "1. create (id)\n";
cout << "2. exec (id) (text_string, pattern_string)\n";
cout << "3. kill (id)\n";
cout << "4. ping (id)\n";
cout << "5. exit\n" << endl;
while (true) {
```

```
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(DEFAULT_PORT + node_id));
main_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
main_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
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, DEFAULT_PORT + node_id);
} else {
child_id = node_id;
main_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
main_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
send_message(main_socket, "pid");
result = receive_message(main_socket);
}
} else {
main_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
main_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
string msg_s = "create " + to_string(node_id);
```

```
send_message(main_socket, msg_s);
result = receive_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 = receive_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 = receive_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 = receive_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 TIMER = 500;
const int DEFAULT_PORT = 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 receive_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 real_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,DEFAULT_PORT + create_id);
}
else {
id = create_id;
send_message(socket, "pid");
send_message(parent_socket, receive_message(socket));
```

```
}
}

void real_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");
receive_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, receive_message(socket));
}
}

void real_exec(zmq::socket_t& parent_socket, zmq::socket_t& socket, int& id, int&
pid, string& request_string) {
if (pid == 0) {
string receive_message = "Error:" + to_string(id);
receive_message += ": Not found";
send_message(parent_socket, receive_message);
}
else {
send_message(socket, request_string);
string str = receive_message(socket);
if (str == "") str = "Error: Node is unavailable";
```

```
send_message(parent_socket, str);
}
}

void real_ping(zmq::socket_t& parent_socket, zmq::socket_t& socket, int& id, int& pid, string& request_string) {
if (pid == 0) {
string receive_message = "Error:" + to_string(id);
receive_message += ": Not found";
send_message(parent_socket, receive_message);
}
else {
send_message(socket, request_string);
string str = receive_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 receive_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();
receive_message = "Ok:" + to_string(id) + ":";
if (!answer.empty()) {
receive_message += answer;
} else {
receive_message += "-1";
}
send_message(parent_socket, receive_message);
} else if (exec_id < id) {
real_exec(parent_socket, left_socket, exec_id, left_pid, request_string);
} else {
real_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 receive_message;
command_stream >> ping_id;
if (ping_id == id) {
receive_message = "Ok: 1";
send_message(parent_socket, receive_message);
} else if (ping_id < id) {
real_ping(parent_socket, left_socket, ping_id, left_pid, request_string);
}
else {
real_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");
receive_message(left_socket);
kill(left_pid,SIGTERM);
kill(left_pid,SIGKILL);
}
if (right_pid != 0) {
send_message(right_socket, "kill_children");
receive_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, TIMER);
parent_socket.setsockopt(ZMQ_SNDTIMEO, TIMER);
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) {
string request_string = receive_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(DEFAULT_PORT + create_id));
left_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
left_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
left_pid = fork();
real_create(parent_socket, left_socket, create_id, left_id, left_pid);
} else {
send_message(left_socket, request_string);
string str = receive_message(left_socket);
if (str == "") {
left_socket.bind(get_port_name(DEFAULT_PORT + create_id));
left_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
left_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
```

```
left_pid = fork();
real_create(parent_socket, left_socket, create_id, left_id, left_pid);
} else {
send_message(parent_socket, str);
n++;
left_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
left_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
}
}
} else {
if (right_pid == 0) {
right_socket.bind(get_port_name(DEFAULT_PORT + create_id));
right_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
right_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
right_pid = fork();
real_create(parent_socket, right_socket, create_id, right_id, right_pid);
} else {
send_message(right_socket, request_string);
string str = receive_message(right_socket);
if (str == "") {
right_socket.bind(get_port_name(DEFAULT_PORT + create_id));
right_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
right_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
right_pid = fork();
real_create(parent_socket, right_socket, create_id, right_id, right_pid);
} else {
send_message(parent_socket, str);
n++;
right_socket.setsockopt(ZMQ_RCVTIMEO, n * TIMER);
right_socket.setsockopt(ZMQ_SNDTIMEO, n * TIMER);
}
}
}
```

```
} else if (command == "kill") {
int delete_id;
command_stream >> delete_id;
if (delete_id < id) {
real_kill(parent_socket, left_socket, delete_id, left_id, left_pid, request_string);
} else {
real_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

```
#include <iostream>
#include <vector>
#include <algorithm>
#include "../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;
}
```

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*);
};
```

Демонстрация работы программы

```
tvard@tvard-HVY-WXX9:~/os/OS-labs/lab6-8$ ./server
```

Commands:

1. create (id)

2. exec (id) (text_string, pattern_string)

3. kill (id)

4. ping (id)

5. exit

create 2

Ok:4030

create 3

Ok:4035

exec 2 abracadabra abra

Ok:2:0;7

exec 3 abracadabra abra

Ok:3:0;7

create 4

Ok:4176

ping 2

Ok: 1

ping 3

Ok: 1

kill 3

Ok

ping 4

Error:4: Not found

ping 3

Error:3: Not found

ping 2

Ok: 1

ping 1

Error:1: Not found

create 5

Ok:4350

ping 5

Ok: 1

exec 5 bebra beb

Ok:5:0

kill 5

Ok

ping 2

Ok: 1

create 7

Ok:4407

create 8

Ok:4441

create 9

Ok:4446

exec 7 hahahah ha

Ok:7:0;2;4

kill 7

Ok

ping 8

Error:8: Not found

ping 9

Error:9: Not found

ping 2

Ok: 1

exec 2 hehehe he

Ok:2:0;2;4

exit

Выводы

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