**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»**

**Темы работы**

**“Управлении серверами сообщений”**

**“Применение отложенных вычислений”**

**“Интеграция программных систем друг с другом”**

Студент: Абросимов Алексей Дмитриевич

Группа: М8О-207Б-20

Преподаватель: Е. С. Миронов

Вариант:

Дата:

Оценка:

Москва, 2021

## **1. Описание**

Данная лабораторная работа будет выполняться в ОС Unix.

**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. 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. Список основных поддерживаемых команд:

Создание нового узла  
Удаление существующего узла  
Выполнение функции

Проверка доступности узлов

## **2. Исходный код**

#include <iostream>

#include "zmq.hpp"

#include <string>

#include <zconf.h>

#include <vector>

#include <signal.h>

#include <sstream>

#include <set>

#include <algorithm>

#include "server\_functions.h"

struct treenode

{

int data;

treenode \*parent;

treenode \*son;

treenode \*brother;

};

class IdTree

{

public:

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

{

std::vector<int> result;

get\_nodes(root, result);

return result;

}

bool contains(int id)

{

treenode \*res = nullptr;

res = find(id);

return res != nullptr;

}

void erase(int id)

{

if (root->data == id)

{

destroy();

}

treenode \*finds = nullptr;

finds = find(id);

if (finds == nullptr)

return;

deleteunder(finds);

deletenode(root, id);

}

void createtree(int id)

{

root = new treenode;

root->data = id;

root->brother = NULL;

root->son = NULL;

root->parent = NULL;

}

void Print\_Tree()

{

print\_tree(root, 0);

}

treenode \*find(int id)

{

return find(root, id);

}

void insert(int parent, int id)

{

treenode \*node = nullptr;

node = find(parent);

if (node == nullptr)

return;

treenode \*par = node;

if (node->son == NULL)

{

node->son = createnode(id);

node->son->parent = node;

return;

}

node = node->son;

while (node->brother != NULL)

{

node = node->brother;

}

node->brother = createnode(id);

node->brother->parent = par;

}

private:

treenode \*root;

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

{

if (node == nullptr)

{

return;

}

get\_nodes(node->brother, v);

v.push\_back(node->data);

get\_nodes(node->son, v);

}

void destroy()

{

destroy(root);

}

void destroy(treenode \*node)

{

if (node->son)

{

destroy(node->son);

}

if (node->brother)

{

destroy(node->brother);

}

free(node);

node = NULL;

}

void deleteunder(treenode \*node)

{

if (node == NULL)

{

return;

}

else

{

if (node->son)

{

node = node->son;

}

else

{

return;

}

if (node->son)

{

deleteunder(node->son);

}

treenode \*next = node;

treenode \*prev = NULL;

while (next->brother)

{

prev = next;

next = next->brother;

if (next->son)

{

deleteunder(next->son);

}

free(prev);

}

delete next;

}

}

void deletenode(treenode \*node, int id)

{

if (node != NULL)

{

if (node->son != NULL)

{

if (node->son->data == id)

{

treenode \*prev = root->son;

node->son = node->son->brother;

free(prev);

prev = NULL;

}

else

{

deletenode(node->son, id);

}

}

if (node->brother != NULL)

{

if (node->brother->data == id)

{

treenode \*prev = node->brother;

node->brother = node->brother->brother;

free(prev);

prev = NULL;

}

else

{

deletenode(node->brother, id);

}

}

}

}

void print\_tree(treenode \*tree, int count)

{

if (tree != NULL)

{

for (int i = 0; i < count; i++)

{

printf(" ");

}

printf("%d", tree->data);

printf("\n");

}

if (tree->son != NULL)

{

print\_tree(tree->son, count + 2);

}

if (tree->brother != NULL)

{

print\_tree(tree->brother, count);

}

}

treenode \*createnode(int id)

{

treenode \*node = new treenode;

node->data = id;

node->son = NULL;

node->brother = NULL;

node->parent = NULL;

return node;

}

treenode \*find(treenode \*node, int id)

{

if (node == NULL)

return NULL;

if (node->data == id)

{

return node;

}

treenode \*findnode;

if (node->son != NULL)

{

findnode = find(node->son, id);

if (findnode != NULL)

return findnode;

}

if (node->brother != NULL)

{

findnode = find(node->brother, id);

if (findnode != NULL)

return node;

}

}

};

int main()

{

std::string command;

IdTree ids;

ids.createtree(-1);

size\_t child\_pid = 0;

int child\_id = 0;

zmq::context\_t context(1);

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

int linger = 0;

main\_socket.setsockopt(ZMQ\_SNDTIMEO, 2000);

main\_socket.setsockopt(ZMQ\_RCVTIMEO, 2000);

//main\_socket.setsockopt(ZMQ\_RCVTIMEO, 2000);

main\_socket.setsockopt(ZMQ\_LINGER, &linger, sizeof(linger));

//main\_socket.connect(get\_connect\_name(30000));

int port = bind\_socket(main\_socket);

std::vector<zmq::socket\_t> sons;

while (true)

{

std::cin >> command;

if (command == "create")

{

size\_t node\_id;

int parent\_id;

std::string result;

std::cin >> node\_id;

std::cin >> parent\_id;

std::cout << node\_id << " " << parent\_id << std::endl;

if (ids.find(parent\_id) != nullptr && ids.find(node\_id) == nullptr)

{

if (child\_pid == 0)

{

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);

}

else

{

child\_id = node\_id;

send\_message(main\_socket, "pid");

result = recieve\_message(main\_socket);

}

}

else

{

std::ostringstream msg\_stream;

msg\_stream << "create " << parent\_id << " " << node\_id;

send\_message(main\_socket, msg\_stream.str());

result = recieve\_message(main\_socket);

}

if (result.substr(0, 2) == "Ok")

{

ids.insert(parent\_id, node\_id);

}

std::cout << result << "\n";

}

else

{

std::cout << "Can't create : already exist or doesnt exist\n";

}

}

else if (command == "remove")

{

size\_t node\_id;

std::cin >> node\_id;

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, 2) == "Ok")

{

ids.erase(node\_id);

}

else{

std::cout<<recieved\_message.substr(0,2)<<std::endl;

}

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

}

else if (command == "exec")

{

int id;

std::string example;

std::string search;

std::cin >> id>>example>>search;

std::string message\_string = "exec " + std::to\_string(id)+" "+ example + " " + search;

send\_message(main\_socket, message\_string);

std::string recieved\_message = recieve\_message(main\_socket);

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

}

else if (command == "ping")

{

int pingid;

std::cin>>pingid;

send\_message(main\_socket, "ping " +std::to\_string(pingid));

std::string recieved = recieve\_message(main\_socket);

std::cout<<recieved<<std::endl;

}

else if (command == "exit")

{

send\_message(main\_socket,"kill 0");

recieve\_message(main\_socket);

//sleep(2);

zmq\_close(main\_socket);

close(port);

exit(1);

break;

}

else

{

std::cout << "Try again\n";

continue;

}

}

return 0;

}

---------------------------------

#include <iostream>

#include "zmq.hpp"

#include <string>

#include <sstream>

#include <zconf.h>

#include <exception>

#include <fstream>

#include <vector>

#include <signal.h>

#include "server\_functions.h"

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

{

int id = std::stoi(argv[1]);

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

zmq::context\_t context;

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

parent\_socket.connect(get\_port\_name(parent\_port));

int son\_pid = 0;

int son\_id = 0;

int linger = 0;

int son\_counter = 0;

std::string answ;

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

son\_socket.setsockopt(ZMQ\_SNDTIMEO, 2000);

son\_socket.setsockopt(ZMQ\_LINGER, &linger, sizeof(linger));

int son\_port = bind\_socket(son\_socket);

while (true)

{

std::string request\_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 id\_to\_create;

int parent\_id;

command\_stream >> parent\_id;

if (parent\_id == id)

{

son\_counter++;

command\_stream >> id\_to\_create;

son\_pid = fork();

if (son\_pid == -1)

{

send\_message(parent\_socket, "Error: Cannot fork");

son\_pid = 0;

}

else if (son\_pid == 0)

{

create\_node(id\_to\_create, son\_port);

}

else

{

son\_id = id\_to\_create;

for (int i = 0; i < son\_counter; i++)

{

send\_message(son\_socket, " pid");

answ = recieve\_message(son\_socket);

if (answ.substr(0, 2) == "Ok")

break;

}

send\_message(parent\_socket, answ);

}

}

else

{

send\_message(son\_socket, request\_string);

send\_message(parent\_socket, recieve\_message(son\_socket));

}

}

else if (command == "exec")

{

int exec\_id;

command\_stream >> exec\_id;

if (exec\_id == id)

{

std::string example;

std::string search;

std::vector<int> res;

command\_stream >> example >> search;

for (int i = example.find(search, i++); i != std::string::npos; i = example.find(search, i + 1))

res.push\_back(i);

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

for (int i = 0; i < res.size(); i++)

{

recieve\_message += std::to\_string(res[i]) + ";";

}

send\_message(parent\_socket, recieve\_message);

}

else

{

for (int i = 0; i < son\_counter; i++)

{

send\_message(son\_socket, request\_string);

answ = recieve\_message(son\_socket);

if (answ.substr(0, 2) == "Ok")

break;

}

send\_message(parent\_socket, answ);

}

}

else if (command == "ping")

{

int pingid;

command\_stream >> pingid;

if (pingid == id)

{

send\_message(parent\_socket, "Ok:" + std::to\_string(id));

}

else

{

for (int i = 0; i < son\_counter; i++)

{

send\_message(son\_socket, request\_string);

answ = recieve\_message(son\_socket);

if (answ.substr(0, 2) == "Ok")

break;

}

send\_message(parent\_socket, answ);

}

}

else if (command == "kill")

{ // УБИТЬ ВСЕХ ДЕТЕЙ

int killid;

command\_stream >> killid;

if (!killid)

{

send\_message(parent\_socket, "OK");

for (int i = 0; i < son\_counter; i++)

{

send\_message(son\_socket, "kill 0");

recieve\_message(son\_socket);

}

sleep(2);

zmq\_close(son\_socket);

close(son\_port);

exit(1);

}

else

{

if (killid == id)

{

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

send\_message(parent\_socket, answ);

for (int i = 0; i < son\_counter; i++)

{

send\_message(son\_socket, "kill 0");

recieve\_message(son\_socket);

}

sleep(2);

zmq\_close(son\_socket);

close(son\_port);

exit(1);

}

else

{

for (int i = 0; i < son\_counter; i++)

{

send\_message(son\_socket, request\_string);

answ = recieve\_message(son\_socket);

if (answ.substr(0, 2) == "Ok")

break;

}

send\_message(parent\_socket, answ);

}

}

}

if (parent\_port == 0)

{

break;

}

}

}

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

#include "server\_functions.h"

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;

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;

}

std::string get\_port\_name(int port) {

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

}

int bind\_socket(zmq::socket\_t& socket) {

int port = 30000;

while (true) {

try {

socket.bind(get\_port\_name(port));

break;

} catch(...) {

port++;

}

}

return port;

}

void create\_node(int id, int port) {

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

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

char\* args[] = {"./child\_node", arg1, arg2, NULL};

execv("./child\_node", args);

}

---------------------------------------

#pragma once

#include <string>

#include <zconf.h>

#include "zmq.hpp"

bool send\_message(zmq::socket\_t& socket, const std::string& message\_string);

std::string recieve\_message(zmq::socket\_t& socket);

std::string get\_port\_name(int port);

int bind\_socket(zmq::socket\_t& socket);

void create\_node(int id, int port);

**3. Вывод**

Данная лабораторная работа научила меня пользоваться библиотекой ZMQ, познакомила с такой технологией как очереди сообщений. Это была невероятно сложная, но в то же время и самая интересная лабораторная работа. Думаю, что использовать очередь сообщений мне придётся в дальнейшей работе ещё не раз.