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

Институт №8 «Информационные технологии и прикладная математика» Кафедра 806 «Вычислительная математика и программирование» Дисциплина «Операционные системы»

**Лабораторная работа №6-8 по курсу**

**«Операционные системы»**

**Управление серверами сообщений, применение отложенных вычислений, интеграция программных систем друг с другом**

## Студент: Кондратьев Егор Алексеевич

## Группа: М80-206Б-19

Преподаватель: Миронов Е.С.

Дата:

## Оценка:

Москва, 2021

**Постановка задачи**

**Цель работы**

Целью является приобретение практических навыков в:

* Управлении серверами сообщений (№6)
* Применение отложенных вычислений (№7)
* Интеграция программных систем друг с другом (№8)

**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.

топология – 4, команда для вычислительных узлов - exec id subcommand(таймер), проверка доступности узлов – pingall

**Общие сведения о программе**

Для работы с очередями используется ZMQ, программа собирается при помощи Makefile. Управляющий узел – server, вычислительные узлы – client. В программе используются следующие системные вызовы:

1. **kill** – убивает процесс с pid – первый аргумент и посылает сигнал – второй аргумент.
2. **socket.setsockopt** – устанавливает флаги для сокета.
3. **zmq::context\_t –** создает ZMQ контекст.
4. **zmq::socket\_t –** создает ZMQ сокет.
5. **zmq::message\_t** – создает ZMQ сообщение.
6. **socket.send** – отправляет ZMQ сообщение на socket.
7. **socket.bind** – принимает соединение к сокету.
8. **execv –** выполняет указанный файл.
9. **fork** – создает копию процесса.

**Общий метод и алгоритм решения**.

Создаем сервер – исполняющий узел, дальше делаем fork, в дочернем процессе при помощи execv запускаем client, а с родителя с сервера отсылаем сообщение, внутри клиента также создаются сокеты – левый и правый и на них отправляются сообщения с родителя, а родитель получает сообщения от детей и так по всему дереву. Исполняющий узел получает сообщение выполняет команду и отправляет ответ вверх по дереву до управляющего узла.

**Основные файлы программы**

**client.cpp:**

#include <iostream>

#include "zmq.hpp"

#include <string>

#include <sstream>

#include <zconf.h>

#include <exception>

#include <signal.h>

#include "timer.h"

#include "zmq\_func.hpp"

using namespace std;

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 create(istringstream& command\_stream, zmq::socket\_t& parent\_socket,

            zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket,

            int& left\_pid, int& left\_id, int& right\_pid, int& right\_id, int& id,

            string& request\_string){

    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));

        }

    }

}

void rl\_remove(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 remove(std::istringstream& command\_stream, zmq::socket\_t& parent\_socket,

            zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket,

            int& left\_pid, int& left\_id, int& right\_pid, int& right\_id, int& id,  std::string& request\_string){

    int delete\_id;

    command\_stream >> delete\_id;

    if (delete\_id < id) {

        rl\_remove(parent\_socket, left\_socket, delete\_id, left\_id, left\_pid, request\_string);

    }

    else {

        rl\_remove(parent\_socket, right\_socket, delete\_id, right\_id, right\_pid, request\_string);

    }

}

void rl\_exec(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket,  int& id, int& pid, string& request\_string, Timer\* timer){

    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);

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

    }

}

void exec(istringstream& command\_stream, zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket, zmq::socket\_t& right\_socket,

          int& left\_pid, int& left\_id, int& right\_pid, int& right\_id, int& id, string& request\_string, Timer\* timer){

    string subcom;

    int exec\_id;

    command\_stream >> exec\_id;

    if (exec\_id == id) {

        command\_stream >> subcom;

        //cout << (subcom) << endl;

        string recieve\_message = "";

        if(subcom == "start") {

            timer->start\_timer();

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

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

        } else if(subcom == "stop") {

            timer->stop\_timer();

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

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

        } else if(subcom == "time") {

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

            recieve\_message += to\_string(timer->get\_time());

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

        } else {

            throw logic\_error("Not implemented message command");

        }

    }

    else if(exec\_id < id) {

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

                left\_pid, request\_string, timer);

    }

    else {

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

                right\_pid, request\_string, timer);

    }

}

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) {

            ostringstream res;

            string left\_res;

            string right\_res;

            res << 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, min<int>(left\_res.size(),5) ) != "Error") {

                res << " " << left\_res;

         //       res << left\_res << std::to\_string(left\_pid);

            }

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

                res << " "<< right\_res;

          //      res << right\_res << std::to\_string(right\_pid);

            }

            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 = 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));

    int left\_pid = 0;

    int right\_pid = 0;

    int left\_id = 0;

    int right\_id = 0;

    int linger = 0;

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

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

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

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

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

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

    while(true) {

        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") {

            create(command\_stream, parent\_socket, left\_socket, right\_socket,

                   left\_pid, left\_id, right\_pid, right\_id, id, request\_string);

        }

        else if(command == "remove") {

            remove(command\_stream, parent\_socket, left\_socket, right\_socket,

                   left\_pid, left\_id, right\_pid, right\_id, id, request\_string);

        }

        else if(command == "exec") {

            exec(command\_stream, parent\_socket, left\_socket, right\_socket,

                 left\_pid, left\_id, right\_pid, right\_id, 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; }

    }

}

**server.cpp:**

#include <iostream>

#include <sstream>

#include <set>

#include <vector>

#include <signal.h>

#include "zmq.hpp"

#include <string>

#include "zmq\_func.hpp"

#include "avl.h"

#include <unistd.h>

using namespace std;

bool is\_number(string val) {

    try{

        int tmp = stoi(val);

        return true;

    }

    catch(std::exception& e) {

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

        return false;

    }

}

void create(AvlTreeId& ids, zmq::socket\_t& main\_socket, int& child\_pid, int& child\_id){

    size\_t node\_id = 0;

    string str = "";

    string result = "";

    cin >> str;

    if(!is\_number(str)) { return; }

    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) {

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

            child\_pid = 0;

            exit(1);

        }

        else if(child\_pid == 0) {   //дочерний процесс

            create\_node(node\_id, PORT\_BASE + node\_id);

        }

        else {      //родительский

            child\_id = node\_id;

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

            result = recieve\_message(main\_socket);

        }

    }

    else {

        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") { ids.insert(node\_id); }

    cout << result << "\n";

}

void remove(AvlTreeId& ids, zmq::socket\_t& main\_socket, int& child\_pid, int& child\_id){

    int node\_id = 0;

    string str = "";

    cin >> str;

    if(!is\_number(str)) { return; }

    node\_id = stoi(str);

    if(ids.find(node\_id) == nullptr) {

        cout << "Error: id doesn't exist\n";

        return;

    }

    if(child\_pid == 0) {

        cout << "Error: Not found\n";

        return;

    }

    if(node\_id == child\_id) {

        kill(child\_pid, SIGTERM);

        kill(child\_pid, SIGKILL);

        child\_id = 0;

        child\_pid = 0;

        ids.erase(node\_id);

        cout << "Ok\n";

        return;

    }

    string message\_string = "remove " + to\_string(node\_id);

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

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

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

        ids.erase(node\_id);

    }

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

}

void exec(AvlTreeId& ids, zmq::socket\_t& main\_socket,  int& child\_pid, int& child\_id) {

    string id\_str = "";

    string subcom = "";

    int id = 0;

    cin >> id\_str >> subcom;

    if (!is\_number(id\_str)) { return; }

    id = stoi(id\_str);

    if(ids.find(id) == nullptr){

        cout << "Error: id doesn't exist\n";

        return;

    }

    if((subcom != "start") && (subcom != "stop") && (subcom != "time")){

        cout << "Wrong subcommand\n";

        return;

    }

    string message\_string = "exec " + to\_string(id) + " " + subcom;

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

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

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

}

void pingall(AvlTreeId& ids, zmq::socket\_t& main\_socket, int& child\_pid, int& child\_id) {

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

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

            istringstream is;

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

                is = istringstream("");

            } else {

                is = istringstream(recieved);

            }

            set<int> recieved\_ids;

            int rec\_id;

            while (is >> rec\_id) {

                recieved\_ids.insert(rec\_id);

            }

            vector<int> from\_tree = ids.get\_nodes();

            auto part\_it = partition(from\_tree.begin(), from\_tree.end(), [&recieved\_ids] (int a) {  //разделение по условию

                return recieved\_ids.count(a) == 0;

            });

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

                cout << "Ok:-1\n";

            } else {

                cout << "Ok:";

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

                    cout  << \*it << " ";

                }

                cout << "\n";

            }

}

void help(){

    cout << "create <id> \n"

         << "exec <id> <n> <k0...kn> \n"

         << "remove <id> \n"

         << "pingall\n"

         << "help\n"

         << "exit\n\n";

}

int main() {

    AvlTreeId ids;

    string command = "";

    int child\_pid = 0;

    int child\_id = 0;

    int linger = 0;

    zmq::context\_t context(1);

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

    main\_socket.setsockopt(ZMQ\_SNDTIMEO, WAIT\_TIME);

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

    //main\_socket.bind(get\_port\_name(PORT\_BASE));

    help();

    while(cin >> command){

        if(command == "create") {

            create(ids, main\_socket, child\_pid, child\_id);

        }

        else if(command == "remove") {

            remove(ids, main\_socket, child\_pid, child\_id);

        }

        else if(command == "exec") {

            exec(ids, main\_socket, child\_pid, child\_id);

        }

        else if(command == "pingall"){

            pingall(ids, main\_socket, child\_pid, child\_id);

        }

        else if(command == "help"){

            help();

        }

        else if(command == "exit") {

            system("killall client");

            break;

        }

    }

    return 0;

}

**timer.h:**

#pragma once

#include <chrono>

class Timer {

 public:

  Timer() = default;

  ~Timer() = default;

  void start\_timer();

  void stop\_timer();

  int get\_time();

  void heartbit(int 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();

}

**zmq\_func.hpp:**

#include "zmq.hpp"

#include "avl.h"

#include <string>

#define CLIENTNAME "./client"

const int WAIT\_TIME = 1000;

const int PORT\_BASE  = 5050;

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

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

void create\_node(int, int);

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

**zmq\_func.cpp:**

#include <iostream>

#include "zmq\_func.hpp"

#include <unistd.h>

using namespace std;

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 "Error: Node is not available";

    }

    return recieved\_message;

}

void create\_node(int id, int port) {

    char\* arg0 = strdup(CLIENTNAME);

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

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

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

    execv(CLIENTNAME, args);

}

string get\_port\_name(const int port) {

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

}

**Makefile:**

CC=g++

CFLAGS=-std=c++11 -g -O2 -Wall -Wno-unused-variable

LDFLAGS=-lrt -lzmq

SRC=server.cpp zmq\_func.cpp avl.cpp

OBJ=$(SRC:.cpp=.o)

SRC2=client.cpp zmq\_func.cpp timer.cpp

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

all: server client

server: $(OBJ)

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

client: $(OBJ2)

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

.cpp.o:

    $(CC) $(CFLAGS) -c $< -o $@

client.o: zmq\_func.hpp avl.h timer.h

server.o: zmq\_func.hpp avl.h

timer.o: timer.h

avl.o: avl.h

clean:

    rm -f \*.o

**Пример работы**

root@Du:/mnt/c/Users/egork/Desktop/os/lab\_6# ./server

create 0

OK: 166

create 0

Error: Already exists

create 1

OK: 169

create 2

OK: 172

create 3

OK: 175

create 4

OK: 178

remove 4

OK

exec 1 start

OK:1

exec 1 time

OK:1: 20912

exec 1 stop

OK:1

exec 1 time

OK:1: 25960

exec 1 time

OK:1: 25960

pingall

OK:-1

**Вывод**

По моему мнению , это была самая интересная лабораторная работа. При её написании я научился основам работы с серверами сообщений на примере ZMQ. Технология действительно удобна для решения ряда задач, а конкретно ZMQ обладает достаточно высокой скоростью передачи данных. Конечно, у ZMQ есть ряд недостатков, но особых трудностей при написании лабораторной это не вызвало. Использование C++ оказалось оправданным в этой работе, так как это позволило быстро разработать ряд классов-абстракций и не держать в голове архитектуру целиком. Задача показалась достаточно интересной и необычной, поэтому решать её было удовольствием.