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»**

**Тема работы**

**“Очереди сообщений”**

Студент: Ханнанов Руслан Маратович

Группа: М8О-208Б-20

Вариант: 4

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

https://github.com/Naksen/OS

**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 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.  
  
Вариант 4. Команды:  
create id [parent\_id] – создать новый узел [id], родителем которого является узел [parent\_id]. Если [parent\_id] = - 1, то родительский узел – управляющий.  
exec id n n1 n2… ni (набор чисел, требуется посчитать сумму)  
ping id – проверка доступности узла

exit – выход из программы

kill id – удалить узел

**Общие сведения о программе**

Для выполнения данной лабораторной работы я предварительно реализовал 6 файлов с кодом:  
  
topology.h – реализация топологии, в соответствии с вариантом – список из списков.  
  
zmq\_function.h - отдельный файл для функций zero-message queue, сделанный для удобства работы и во избежание загрязнения кода.  
  
control.cpp - реализация программы клиента.

counting.cpp - реализация программы сервера.

**Общий метод и алгоритм решения**

Makefile:

all: control counting

control:

g++ control.cpp -lzmq –o control -Wall -pedantic

counting:

g++ control.cpp -lzmq –o counting -Wall -pedantic

clean:

rm -rf control counting

В программе используется тип соединения Request-Response. Узлы передают информацию друг другу при помощи очереди сообщений. Все сообщений имеют следующий вид:

[id узла, которому предназначено сообщение] [команда] [аргументы]

Управляющий узел хранит структуру список списков, в которую записывает id существующих узлов. При помощи этой структуры он определяет, в какой список нужно направить сообщение.

Вычислительный узел, получи сообщение, сравнивает свой id и id из сообщения. Если они совпадают, то узел начинает обрабатывать запрос, в противном случае узел направляет это же сообщение своему ребенку и ждет от него ответа.

Для удобства функции отправки и получения сообщений, а также функции для подключения к сокетам вынесены в отдельный заголовочный файл, который подключается к программам узлов.

**Исходный код**

**Topology.h**

#include <list>

#include <stdexcept>

class topology {

private:

    std::list<std::list<int>> container;

public:

    void insert(int id, int parent\_id) {

        if (parent\_id == -1) {

            std::list<int> new\_list;

            new\_list.push\_back(id);

            container.push\_back(new\_list);

        }

        else {

            int list\_id = find(parent\_id);

            if (list\_id == -1) {

                throw std::runtime\_error("Wrong parent id");

            }

            auto it1 = container.begin();

            std::advance(it1, list\_id);

            for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {

                if (\*it2 == parent\_id) {

                    it1->insert(++it2, id);

                    return;

                }

            }

        }

    }

    int find(int id) {

        int cur\_list\_id = 0;

        for (auto it1 = container.begin(); it1 != container.end(); ++it1) {

            for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {

                if (\*it2 == id) {

                    return cur\_list\_id;

                }

            }

            ++cur\_list\_id;

        }

        return -1;

    }

    void erase(int id) {

        int list\_id = find(id);

        if (list\_id == -1) {

            throw std::runtime\_error("Wrong id");

        }

        auto it1 = container.begin();

        std::advance(it1, list\_id);

        for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {

            if (\*it2 == id) {

                it1->erase(it2, it1->end());

                if (it1->empty()) {

                    container.erase(it1);

                }

                return;

            }

        }

    }

    int get\_first\_id(int list\_id) {

        auto it1 = container.begin();

        std::advance(it1, list\_id);

        if (it1->begin() == it1->end()) {

            return -1;

        }

        return \*(it1->begin());

    }

    friend std::ostream &operator<<(std::ostream &os, const topology &topology) {

    for (auto &external : topology.container) {

      os << "{";

      for (auto &internal : external) {

        os << internal << " ";

      }

      os << "}" << std::endl;

    }

    return os;

    }

};

**Zmq\_functions.h**

#include <zmq.hpp>

#include <iostream>

const int MAIN\_PORT = 4040;

void send\_message(zmq::socket\_t& socket, const std::string& msg) {

    zmq::message\_t message(msg.size());

    memcpy(message.data(), msg.c\_str(), msg.size());

    socket.send(message);

}

std::string receive\_message(zmq::socket\_t& socket) {

    zmq::message\_t message;

    int chars\_read;

    try {

        chars\_read = (int)socket.recv(&message);

    }

    catch (...) {

        chars\_read = 0;

    }

    if (chars\_read == 0) {

        return "Error: node is unavailable [zmq\_func]";

    }

    std::string received\_msg(static\_cast<char\*>(message.data()), message.size());

    return received\_msg;

}

void connect(zmq::socket\_t& socket, int id) {

    std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);

    socket.connect(address);

}

void disconnect(zmq::socket\_t& socket, int id) {

    std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);

    socket.disconnect(address);

}

void bind(zmq::socket\_t& socket, int id) {

    std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);

    socket.bind(address);

}

void unbind(zmq::socket\_t& socket, int id) {

    std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);

    socket.unbind(address);

}

**control.cpp**#include <unistd.h>

#include <sstream>

#include <set>

#include "zmq\_functions.h"

#include "topology.h"

int main() {

    topology network;

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

    zmq::context\_t context;

    std::string cmd;

    while (std::cin >> cmd) {

        if (cmd == "create") {

            int node\_id, parent\_id;

            std::cin >> node\_id >> parent\_id;

            if (network.find(node\_id) != -1) {

                std::cout << "Error: already exists" << std::endl;

            }

            else if (parent\_id == -1) {

                pid\_t pid = fork();

                if (pid < 0) {

                    perror("Can't create new process");

                    return -1;

                }

                if (pid == 0) {

                    execl("./counting", "./counting", std::to\_string(node\_id).c\_str(), NULL);

                    perror("Can't execute new process");

                    return -2;

                }

                branches.emplace\_back(context, ZMQ\_REQ);

                branches[branches.size() - 1].setsockopt(ZMQ\_SNDTIMEO, 5000);

                bind(branches[branches.size() - 1], node\_id);

                send\_message(branches[branches.size() - 1], std::to\_string(node\_id) + "pid");

                std::string reply = receive\_message(branches[branches.size() - 1]);

                std::cout << reply << std::endl;

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

            }

            else if (network.find(parent\_id) == -1) {

                std::cout << "Error: parent not found" << std::endl;

            }

            else {

                int branch = network.find(parent\_id);

                send\_message(branches[branch], std::to\_string(parent\_id) + "create " + std::to\_string(node\_id));

                std::string reply = receive\_message(branches[branch]);

                std::cout << reply << std::endl;

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

            }

        }

        else if (cmd == "exec") {

            int dest\_id;

            int n;

            std::cin >> dest\_id >> n;

            std::vector<int> v(n);

            std::string s;

            int branch = network.find(dest\_id);

            for (int i = 0; i < n; ++i) {

                std::cin >> v[i];

                s = s + std::to\_string(v[i]) + ' ';

            }

            if (branch == -1) {

                std::cout << "ERROR: incorrect node id" << std::endl;

            }

            else {

                send\_message(branches[branch], std::to\_string(dest\_id) + "exec " + std::to\_string(n) + ' ' + s);

                std::string reply = receive\_message(branches[branch]);

                std::cout << reply << std::endl;

            }

        }

        else if (cmd == "kill") {

            int id;

            std::cin >> id;

            int branch = network.find(id);

            if (branch == -1) {

                std::cout << "ERROR: incorrect node id" << std::endl;

            }

            else {

                bool is\_first = (network.get\_first\_id(branch) == id);

                send\_message(branches[branch], std::to\_string(id) + " kill");

                std::string reply = receive\_message(branches[branch]);

                std::cout << reply << std::endl;

                network.erase(id);

                if (is\_first) {

                    unbind(branches[branch], id);

                    branches.erase(branches.begin() + branch);

                }

            }

        }

        else if (cmd == "print") {

            std::cout << network;

        }

        else if (cmd == "pingall") {

            std::set<int> available\_nodes;

            for (size\_t i = 0; i < branches.size(); ++i) {

                int first\_node\_id = network.get\_first\_id(i);

                //std::cout << first\_node\_id << std::endl;

                send\_message(branches[i], std::to\_string(first\_node\_id) + " pingall");

                std::string received\_message = receive\_message(branches[i]);

                std::istringstream reply(received\_message);

                int node;

                while(reply >> node) {

                    available\_nodes.insert(node);

                }

            }

            std::cout << "OK: ";

            if (available\_nodes.empty()) {

                std::cout << "no available nodes" << std::endl;

            }

            else {

                for (auto v : available\_nodes) {

                    std::cout << v << " ";

                }

                std::cout << std::endl;

            }

        }

        else if (cmd == "ping") {

            int id;

            std::cin >> id;

            int branch = network.find(id);

            if (branch == -1) {

                std::cout << "Error: Not found" << std::endl;

                continue;

            }

            send\_message(branches[branch], std::to\_string(id) + " ping");

            std::string reply = receive\_message(branches[branch]);

            std::cout << reply << std::endl;

        }

        else if (cmd == "exit") {

            for (size\_t i = 0; i < branches.size(); ++i) {

                int first\_node\_id = network.get\_first\_id(i);

                send\_message(branches[i], std::to\_string(first\_node\_id) + " kill");

                std::string reply = receive\_message(branches[i]);

                if (reply != "OK") {

                    std::cout << reply << std::endl;

                }

                else {

                    unbind(branches[i], first\_node\_id);

                }

            }

            exit(0);

        }

        else {

            std::cout << "Incorrect cmd" << std::endl;

        }

    }

}

**Counting.cpp**#include <unordered\_map>

#include <unistd.h>

#include <sstream>

#include <unordered\_map>

#include "zmq\_functions.h"

int main(int argc, char\* argv[]) {

    if (argc != 2 && argc != 3) {

        throw std::runtime\_error("Wrong args for counting node");

    }

    int cur\_id = std::atoi(argv[1]);

    int child\_id = -1;

    if (argc == 3) {

        child\_id = std::atoi(argv[2]);

    }

    //std::cout << child\_id << std::endl;

    std::unordered\_map<std::string, int> dictionary;

    zmq::context\_t context;

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

    connect(parent\_socket, cur\_id);

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

    child\_socket.setsockopt(ZMQ\_SNDTIMEO, 5000);

    if (child\_id != -1) {

        bind(child\_socket, child\_id);

    }

    std::string message;

    while (true) {

        message = receive\_message(parent\_socket);

        std::istringstream request(message);

        int dest\_id;

        request >> dest\_id;

        std::string cmd;

        request >> cmd;

        if (dest\_id == cur\_id) {

            if (cmd == "pid") {

                send\_message(parent\_socket, "OK: " + std::to\_string(getpid()));

            }

            else if (cmd == "create") {

                int new\_child\_id;

                request >> new\_child\_id;

                if (child\_id != -1) {

                    unbind(child\_socket, child\_id);

                }

                bind(child\_socket, new\_child\_id);

                pid\_t pid = fork();

                if (pid < 0) {

                    perror("Can't create new process");

                    return -1;

                }

                if (pid == 0) {

                    execl("./counting", "./counting", std::to\_string(new\_child\_id).c\_str(), std::to\_string(child\_id).c\_str(), NULL);

                    perror("Can't execute new process");

                    return -2;

                }

                send\_message(child\_socket, std::to\_string(new\_child\_id) + "pid");

                child\_id = new\_child\_id;

                send\_message(parent\_socket, receive\_message(child\_socket));

            }

            else if (cmd == "exec") {

                int n;

                request >> n;

                int sum = 0;

                int x;

                for (int i = 0; i < n; ++i) {

                    request >> x;

                    sum += x;

                }

                send\_message(parent\_socket, "OK: " + std::to\_string(sum));

            }

            else if (cmd == "pingall") {

                std::string reply;

                if (child\_id != -1) {

                    send\_message(child\_socket, std::to\_string(child\_id) + " pingall");

                    std::string msg = receive\_message(child\_socket);

                    reply += " " + msg;

                }

                send\_message(parent\_socket, std::to\_string(cur\_id) + reply);

            }

            else if (cmd == "ping") {

                send\_message(parent\_socket, "OK 1");

            }

            else if (cmd == "kill") {

                //std::cout << child\_id << std::endl;

                if (child\_id != -1) {

                    send\_message(child\_socket, std::to\_string(child\_id) + " kill");

                    std::string msg = receive\_message(child\_socket);

                    if (msg == "OK") {

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

                    }

                    unbind(child\_socket, child\_id);

                    disconnect(parent\_socket, cur\_id);

                    break;

                }

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

                disconnect(parent\_socket, cur\_id);

                break;

            }

        }

        else if (child\_id != -1) {

            //std::cout << dest\_id << " " << message << std::endl;

            send\_message(child\_socket, message);

            send\_message(parent\_socket, receive\_message(child\_socket));

            if (child\_id == dest\_id && cmd == "kill") {

                std::cout << "SUCCESS" << child\_id << std::endl;

                child\_id = -1;

            }

        }

        else {

            send\_message(parent\_socket, "Error: node is unavailable");

        }

    }

}

**Демонстрация работы программы**naksan@LAPTOP-TL9L61MA:~/cprog/my\_6-8\_lab$ ./control

create 5 -1

OK: 31463

create 10 -1

OK: 31490

create 4 -1

OK: 31527

print

{5 }

{10 }

{4 }

ping 10

OK 1

create 3 4

OK: 31627

print

{5 }

{10 }

{4 3 }

exec 3 3 1 2 3

OK: 6

kill 3

OK

exit

**Выводы**Данная лабораторная работа была направлена на изучение технологии очереди сообщений, на основе которой необходимо было построить сеть с заданной топологией.

Наряду с каналами и отображаемыми файлами, очереди сообщений являются достаточно удобным способом для взаимодействия между процессами. ZeroMQ предоставляет достаточно просто интерфейс для передачи сообщений, а также поддерживает все возможные типы соединений.

Полученные мной навыки работы с очередями сообщений можно использовать при проектировании мессенджеров, многопользовательских игр, да и вообще для любых мультипроцессорных программ.