Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторные работы №5-7 по курсу**

**«Операционные системы»**

**УПРАВЛЕНИЕ СЕРВЕРАМИ СООБЩЕНИЙ  
ПРИМЕНЕНИЕ ОТЛОЖЕННЫХ ВЫЧИСЛЕНИЙ  
ИНТЕГРАЦИЯ ПРОГРАММНЫХ СИСТЕМ ДРУГ С ДРУГОМ**

Студент: Юрков Евгений Юрьевич

Группа: М8О–212Б–22

Вариант: 21

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2023.

**Постановка задачи**

## Цель работы

Целью является приобретение практических навыков в:

* Управлении серверами сообщений (No5)
* Применение отложенных вычислений (No6)
* Интеграция программных систем друг с другом (No7)

## 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.

**Вариант 21:**

Топология 4: Узлы находятся в идеально сбалансированном бинарном дереве.

Набор команд 3: Локальный таймер.

Команда проверки 2: Формат команды «ping id». Команда проверяет доступность конкретного узла.

**Общие сведения о программе**

Основной файл программы (управляющий узел) - main.cpp, timer.cpp – программа, обрабатывающая запросы (локальный таймер). Также используется заголовочные файлы: iostream, chrono, string, map, zmqpp/zmqpp.hpp.

В программе используются следующие системные вызовы:

1. **fork** – создание дочернего процесса.
2. **kill** – завершает процесс.
3. **execl** – выполняют файл.

В программе работа с очередью сообщений происходит с помощью:

1. **context** – потокобезопасный класс представляющий внутренний контекст ZMQ.
2. **socket** – класс, осуществляющий связь между программами и обмен данными между ними.
3. **socket.bind()** – связывается с портом.
4. **socket.connect()** – подключается к порту.
5. **socket.send()** – отправляет сообщение.
6. **socket.recieve()** – получает сообщение.
7. **socket.disconnect() –** отключается от порта.
8. **socket.close()** – закрывает сокет.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы с ZeroMQ.
2. Написать программу-клиент, которая будет получать запросы от пользователя и взаимодействовать с серверами.
3. Написать программу-сервер, которая будет получать команды и давать на них соответствующие ответы.

**Основные файлы программы**

**main.cpp:**

#include <zmqpp/zmqpp.hpp>

#include <iostream>

#include <string>

#include <map>

std::string get\_port(int id) {

std::string port = "tcp://localhost:";

port += std::to\_string(50000 + id);

return port;

}

pid\_t create\_timer(std::map<int, pid\_t>& conteiner, int& id) {

pid\_t pid = fork();

if (pid == -1) {

std::cout << "Error: can't fork" << std::endl;

}

if (pid == 0) {

execl("/home/kruyneg/Programming/OS/build/timer", "./timer", get\_port(id).c\_str(), nullptr);

std::cout << "Error: execl" << std::endl;

}

else {

conteiner.insert({id, pid});

}

return pid;

}

int main() {

zmqpp::context context;

zmqpp::socket socket(context, zmqpp::socket\_type::req);

socket.set(zmqpp::socket\_option::receive\_timeout, 1000);

socket.set(zmqpp::socket\_option::linger, 1000);

std::map<int, pid\_t> servers;

std::string query;

do {

std::cin >> query;

if (query == "create") {

int id;

std::cin >> id;

create\_timer(servers, id);

}

else if (query == "exec") {

int id;

std::cin >> id;

std::string subcommand;

std::cin >> subcommand;

if (servers.count(id)) {

try {

socket.connect(get\_port(id));

socket.send(subcommand);

std::string buffer;

socket.receive(buffer);

std::cout << "Ok:" << id;

if (!buffer.empty())

std::cout << ": " << buffer;

std::cout << std::endl;

socket.disconnect(get\_port(id));

}

catch(zmqpp::exception& exc) {

std::cout << "Exception: " << exc.what() << std::endl;

}

}

else {

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

}

}

else if (query == "ping") {

int id;

std::cin >> id;

if (servers.count(id)) {

try {

socket.connect(get\_port(id));

socket.send("alive?");

std::string buffer;

if (socket.receive(buffer, false)) {

std::cout << "Ok:" << id << ": 1" << std::endl;

socket.disconnect(get\_port(id));

}

else {

std::cout << "Ok:" << id << ": 0" << std::endl;

socket.close();

socket = zmqpp::socket(context, zmqpp::socket\_type::req);

socket.set(zmqpp::socket\_option::receive\_timeout, 1000);

socket.set(zmqpp::socket\_option::linger, 1000);

}

}

catch (zmqpp::exception& exc) {

std::cout << "Ok:" << id << " 0 " << exc.what() << std::endl;

}

}

else {

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

}

}

else if (query == "kill") {

int id;

std::cin >> id;

if (servers.count(id)) {

if (kill(servers[id], SIGINT) == -1) {

std::cout << "Error: can't kill" << std::endl;

}

servers.erase(id);

}

else {

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

}

}

} while (query != "exit" && query != "quit");

for (auto& elem : servers) {

kill(elem.second, SIGINT);

}

socket.close();

return 0;

}

**timer.cpp:**

#include <iostream>

#include <chrono>

#include <zmqpp/zmqpp.hpp>

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

std::string my\_port = argv[1];

std::cout << "Hi! My port is " << my\_port << std::endl;

zmqpp::context context;

zmqpp::socket socket(context, zmqpp::socket\_type::reply);

try {

socket.bind(my\_port);

}

catch (zmqpp::exception e) {

std::cerr << "Exception: " << e.what() << std::endl;

}

std::chrono::milliseconds time {0};

std::chrono::time\_point<std::chrono::steady\_clock> start\_time;

bool is\_stopped = true;

while (true) {

std::string message;

socket.receive(message);

std::stringstream response;

if (message == "start") {

start\_time = std::chrono::steady\_clock::now();

is\_stopped = false;

}

else if (message == "stop") {

auto end\_time = std::chrono::steady\_clock::now();

time += std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time);

is\_stopped = true;

}

else if (message == "time") {

auto end\_time = std::chrono::steady\_clock::now();

if (is\_stopped)

response << time;

else

response << time + std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time);

}

else if (message == "alive?") {

response << "yes";

}

else {

response << "Unknown subcommand";

}

// std::cout << my\_port << " got: " << message << std::endl;

socket.send(response.str());

}

}

**Тестирование:**

[kruyneg@matebook14 build]$ /home/kruyneg/Programming/OS/build/main  
create 5  
Hi! My port is tcp://localhost:50005  
ping 5  
Ok:5: 0  
create 1  
Hi! My port is tcp://localhost:50001  
exec 1 time  
Ok:1: 0ms  
exec 1 start  
Ok:1  
ping 1  
Ok:1: 1  
exec 1 stop  
Ok:1  
exec 1 time  
Ok:1: 13218ms  
exec 1 time  
Ok:1: 13218ms  
ping 1  
Ok:1: 1  
ping 5  
Ok:5: 0  
kill 5  
ping 5  
Error: Not found  
ping 1  
Ok:1: 1  
ping 1  
Ok:1: 0  
kill 1  
create 2  
Hi! My port is tcp://localhost:50002  
exec 2 time  
Ok:2: 0ms  
exec 2 start  
Ok:2  
exec 2 time  
Ok:2: 3724ms  
ping 2  
Ok:2: 1  
kill 2  
ping 1  
Error: Not found  
ping 2  
Error: Not found  
quit

**Вывод**

Проделав работу, я успешно написал программу, которая использовала очередь сообщений для взаимодействия между процессами. Я получил практические навыки работы с ZeroMQ. Разработанная программа позволила осуществить контроль таймерами, включая запуск и остановку, предоставив возможность корректного завершения работы определенного таймера при помощи команды kill, а также проверку его доступности с использованием команды ping. Одной из основных проблем являлось зависание программы при ожидании ответа на запрос.

Очереди сообщений необходимы для обеспечения асинхронного обмена данными между процессами или компонентами системы. Они позволяют разделить работу между различными частями приложения, обеспечивают устойчивость к перегрузкам, что делает их важным инструментом для создания отказоустойчивых и эффективных программных систем.