Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №6 по курсу**

**«Операционные системы»**

Студент: Кирилин Иван Олегович

Группа: М8О-301Б-21

Вариант: 3

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2024

**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/Bonartze/OS

**Постановка задачи**

**Цель работы**

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.

**Вариант 6**: 1, , 3.

**Общие сведения о программе**

Программы компилируются из файлов main.cpp, node.cpp с использованием заголовочных файлов topology.h и zmq\_f.h с помощью cmake и CMakeFile. Также используется заголовочные файлы: unistd.h, iostream, vector, zmq.hpp, sstream, string, map.

**Общий метод и алгоритм решения**

В файле node.cpp находится реализация ноды. На вход подаётся current\_id и child\_id. Далее нода создается и обрабатывает входящие сообщения, содержащие команды, согласно условию задания: heartbit, create, remove, exec. После обработки команды и выполнения необходимых действий, она отсылает сообщение с результатом работы.

В файле zmq\_f.h находится реализация необходимого функционала для работы с ZMQ: отправка сообщений, получение сообщений, коннект и дисконнект, бинд и анбинд.

В файле topology.h находится реализация топологии согласно варианту: 1 управляющий узел, вычислительные узлы находятся в списках. Класс Topology содержит контейнер нод, представляющий из себя список списков, а так же необходимые методы для работы: вставка ноды, поиск ноды по id, удаление ноды по id, получение id первой ноды в списке.

В файле main.cpp реализована основная работа по распределению комманд, подающихся на вход. В начале происходит необходимая подготовка (создание основного сокета, подготовка вектора сокетов и т.д.). Далее происходит обработка команд, перечисленных выше, необходимые подготовительные действия перед отправкой сообщения, сама отправка сообщение, получение сообщения с результатом выполнения команды и вывод в стандартный поток вывода.

**Исходный код**

**topology.h**

|  |
| --- |
| #ifndef LAB6\_TOPOLOGY\_H  #define LAB6\_TOPOLOGY\_H  #include <list>  #include <stdexcept>  class Topology { // Топология  private:  std::list<std::list<int>> container; // список списков, наши ноды  public:  void Insert(int id, int parent\_id) { // вставить новую ноду. id - ид ноды, parent\_id - ид родителя  if (parent\_id == -1) { // если родитель - управляющий узел  std::list<int> new\_list; // подготавливаем новый список  new\_list.push\_back(id); // вставляем туда ноду  container.push\_back(new\_list); // и отправляем в хранилище нод  } else {  int list\_id = Find(parent\_id); // ищем родителя по айди  if (list\_id == -1) {  throw std::runtime\_error("Wrong parent id"); // если не нашли - ошибка  }  auto it1 = container.begin();  std::advance(it1, list\_id);  for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {  if (\*it2 == parent\_id) { // иттерируемся по хранилищу и ищем родителя.  it1->insert(++it2, id); // если нашли родителя - вставляем.  return;  }  }  }  }  int Find(int id) { // ищем ноду по айди  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) { // иттерируемся по хранилищу и ищем нужную ноду по айди  return cur\_list\_id; // если нашли - возвращаем её  }  }  ++cur\_list\_id;  }  return -1; // если не нашли - возвращаем -1  }  void Erase(int id) { // удалить ноду по айди  int list\_id = Find(id); // ищем ноду по айди  if (list\_id == -1) {  throw std::runtime\_error("Wrong id"); // если не нашли - пробрасываем ошибку.  }  auto it1 = container.begin();  std::advance(it1, list\_id);  for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {  if (\*it2 == id) { // иттерируемся по списку и ищем ноду  it1->erase(it2, it1->end()); // если нашли - удаляем её  if (it1->empty()) { // если список после удаления оказался пустым  container.erase(it1); // удаляем список  }  return;  }  }  }  int GetFirstId(int list\_id) { // получить первый id в списке  auto it1 = container.begin(); // итератор  std::advance(it1, list\_id); // смещаем итератор  if (it1->begin() == it1->end()) { // если итератор попал на конец коллекции  return -1; // возвращаем -1  }  return \*(it1->begin()); // возвращаем первый элемент  }  };  #endif // LAB6\_TOPOLOGY\_H |

**zmq\_f.h**

|  |
| --- |
| #ifndef LAB6\_ZMQ\_F\_H  #define LAB6\_ZMQ\_F\_H  #include <iostream>  #include <string>  #include <zmq.hpp>  const int MAIN\_PORT = 4040;  void send\_message(zmq::socket\_t& socket, const std::string& msg) { // отправить сообщение  zmq::message\_t message(msg.size()); // подготавливаем переменную  memcpy(message.data(), msg.c\_str(), msg.size()); // копирует size байтов из msg.c\_str() в message.data()  socket.send(message); // возврат результата, отсылает очередь сообщений, созданных в message  }  std::string receive\_message(zmq::socket\_t& socket) { // получить сообщение  zmq::message\_t message; // подготавливаем переменную  int chars\_read; // переменная для количества считанных символов  try {  chars\_read = (int)socket.recv(&message); // получение сообщений  }  catch (...) {  chars\_read = 0; // отлавливаем исключение  }  if (chars\_read == 0) {  return "Error"; // если поймали исключение или считали 0 символов - ошибка  }  std::string received\_msg(static\_cast<char\*>(message.data()), message.size());  return received\_msg;  }  void connect(zmq::socket\_t& socket, int id) {  std::string adress = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);  socket.connect(adress);  }  void disconnect(zmq::socket\_t& socket, int id) {  std::string adress = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);  socket.disconnect(adress);  }  void bind(zmq::socket\_t& socket, int id) {  std::string adress = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);  socket.bind(adress); // Асинхронно привязывается к конечной точке (endpoint).  }  void unbind(zmq::socket\_t& socket, int id) {  std::string adress = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);  socket.unbind(adress); // Отменяет привязку к ранее привязанной конечной точке.  }  #endif // LAB6\_ZMQ\_F\_H |

**node.cpp**

|  |
| --- |
| #include <unistd.h>  #include <iostream>  #include <string>  #include <vector>  #include <map>  #include "zmq\_f.h"  int main(int argc, char\* argv[])  {  if (argc != 2 && argc != 3) { // если нам не передали аргументы - ошибка  std::cout << "Wrong arguments. Not enough parameters!\n";  exit(1);  }  int current\_id = std::atoi(argv[1]); // считываем первый аргумент - айди ноды  int child\_id = -1; // айди ноды-потомка  if (argc == 3) { // если нам передали два аргумента, значит передали айди потомка  child\_id = std::atoi(argv[2]); // считываем его  }  std::string adr = argv[1];  zmq::context\_t context; // контекст  zmq::socket\_t parent\_socket(context, ZMQ\_REP); // создаем сокет  connect(parent\_socket, current\_id); // коннектимся к сокету  zmq::socket\_t child\_socket(context, ZMQ\_REQ); // создаем сокет потомка  child\_socket.setsockopt(ZMQ\_SNDTIMEO, 5000); // Maximum time before a send operation returns with EAGAIN  parent\_socket.setsockopt(ZMQ\_SNDTIMEO, 5000); // Maximum time before a send operation returns with EAGAIN  std::string message;  std::map<std::string, int> storage; // локальный словарь ноды  while (1) {  zmq::message\_t message\_main;  message = receive\_message(parent\_socket); // получаем сообщение  std::string recieved\_message(static\_cast<char\*>(message\_main.data()), message\_main.size());  std::istringstream request(message); // отправляем его в строковый поток  int dest\_id;  request >> dest\_id; // считываем айди ноды  std::string command;  request >> command; // считываем комманду  if (command == "heartbit") {  std::string ans = std::to\_string(current\_id) + ":Ok; "; // подготавливаем сообщение - ответ  if (child\_id != -1) {  int timeout;  request >> timeout; // считываем таймаут  int fl = 0;  for (int i = 0; i < 4; i++) { // 4 раза пингуем ноду  send\_message(child\_socket, message);  std::string repl = receive\_message(child\_socket);  if (repl != "Error") { // если не получили ошибку  ans += repl;  fl = 1;  break; // прекращаем  }  sleep(timeout / 1000); // ждём timeout / 1000 секунд  }  if (fl == 0) {  ans += "Node " + std::to\_string(child\_id) + " is not avail"; // если нода возвращает Error  }  }  send\_message(parent\_socket, ans); // отправляем ответ  } else if (dest\_id == current\_id) {  if (command == "pid") { // если команда - pid  send\_message(parent\_socket, "OK: " + std::to\_string(getpid()));  } else if (command == "create") { // если команда - create  int new\_child\_id;  request >> new\_child\_id; // считываем id новой ноды  if (child\_id != -1) {  unbind(child\_socket, child\_id); // анбиндим сокет  }  bind(child\_socket, new\_child\_id); // биндим сокет на новый id  pid\_t pid = fork(); // создаем новый процесс  if (pid < 0) { // если ошибка  perror("Can't create new process!\n");  exit(1);  }  if (pid == 0) {  // запускаем новую ноду  execl("node", "node", std::to\_string(new\_child\_id).c\_str(), std::to\_string(child\_id).c\_str(), NULL);  perror("Can't create new process!\n");  exit(1);  }  send\_message(child\_socket, std::to\_string(new\_child\_id) + "pid"); // отправляем сообщение с новым id  child\_id = new\_child\_id;  send\_message(parent\_socket, receive\_message(child\_socket)); // отправляем сообщение родителю  } else if (command == "remove") { // если команда - удалить ноду  send\_message(parent\_socket, "OK"); // отправляем сообщение  disconnect(parent\_socket, current\_id); // и дисконнектимся от сокета  break;  } else if (command == "exec") { // если команда - exec  std::string msg = "OK:" + std::to\_string(dest\_id); // подготавливаем сообщение  // request >> mapKey;  // request >> mapValue;  size\_t count = 0; // количество параметров (1 или 2)  std::string mapKey; // ключ  int mapValue; // значение  request >> count; // считываем количество параметров  request >> mapKey; // считываем ключ (он по-любому придёт)  if (count == 2) { // если пришло 2 параметра  request >> mapValue; // считываем значение  }  if (count == 1) { // если пришёл только ключ  if (storage.count(mapKey) > 0) { // если ключ присутствует в словаре  msg += ": " + std::to\_string(storage.at(mapKey)); // добавляем в сообщение значение  } else {  msg += ": '" + mapKey + "' not found"; // иначе возвращаем ошибку  }  } else if (count == 2) { // если пришло 2 параметра  storage[mapKey] = mapValue; // добавляем в словарь по ключу значение  }  send\_message(parent\_socket, msg); // отправляем сообщение  }  } else if (child\_id != -1) {  send\_message(child\_socket, message);  send\_message(parent\_socket, receive\_message(child\_socket));  if (child\_id == dest\_id && command == "remove") {  child\_id = -1;  }  } else {  send\_message(parent\_socket, "Error: node is unavailable!\n"); // ошибка, что нода не доступна  }  }  } |

**main.cpp**

|  |
| --- |
| #include <unistd.h>  #include <iostream>  #include <vector>  #include <zmq.hpp>  #include <sstream>  #include "topology.h"  #include "zmq\_f.h"  int main() {  Topology network; // наша топология  std::vector<zmq::socket\_t> branches; // вектор сокетов  zmq::context\_t context; // контекст. Класс context\_t инкапсулирует функции, связанные с инициализацией и завершением контекста  std::string command; // переменная для комманды  zmq::socket\_t main\_socket(context, ZMQ\_REP); // главный сокет  std::string message;  while (std::cin >> command) { // пока мы получаем на вход комманды  if (command == "create") { // если команда - создать ноду  int node\_id, parent\_id;  std::cin >> node\_id >> parent\_id; // считываем айди ноды и айди ноды-родителя  if (network.Find(node\_id) != -1) { // Поиск id ноды среди существующих  std::cout << "Error: already exists!\n"; // если существует - ошибка  } else if (parent\_id == -1) { // если родитель - управляющий узел  pid\_t pid = fork(); // Создание дочернего узла  if (pid < 0) {  perror("Can't create new process!\n");  exit(EXIT\_FAILURE);  } if (pid == 0) {  execl("node", "node", std::to\_string(node\_id).c\_str(), NULL); // запускаем ноду  perror("Can't execute new process!\n");  exit(EXIT\_FAILURE);  }  branches.emplace\_back(context, ZMQ\_REQ); // добавляем в вектор  branches[branches.size() - 1].setsockopt(ZMQ\_SNDTIMEO, 5000); // добавляем опцию таймаута  bind(branches[branches.size() - 1], node\_id); // биндим сокет  send\_message(branches[branches.size() - 1], std::to\_string(node\_id) + "pid"); // отправляем сообщение  std::string reply = receive\_message(branches[branches.size() - 1]); // и получаем ответ - pid  std::cout << reply << "\n"; // выводим его  network.Insert(node\_id, parent\_id); // и вставляем в топологию новую ноду  } else if (network.Find(parent\_id) == -1) { // если не нашли родителя  std::cout << "Error: parent not found!\n";  } else { // если родитель - не вычислительынй узел  int branch = network.Find(parent\_id); // ищем ноду  // и отправляем сообщение о создании ноды  send\_message(branches[branch], std::to\_string(parent\_id) + "create " + std::to\_string(node\_id));  std::string reply = receive\_message(branches[branch]); // получаем в ответ pid  std::cout << reply << "\n"; // выводим  network.Insert(node\_id, parent\_id); // вставляем в топологию ноду  }  } else if (command == "remove") { // если команда - удалить ноду  int id;  std::cin >> id; // считываем id  int branch = network.Find(id); // ищем ноду по айди  if (branch == -1) {  std::cout << "Error: incorrect node id!\n";  } else {  bool is\_first = (network.GetFirstId(branch) == id); // проверяем, первая ли нода  send\_message(branches[branch], std::to\_string(id) + " remove"); // оправляем сообщение о удалении  std::string reply = receive\_message(branches[branch]);  std::cout << reply << std::endl; // получаем и выводим ответ  network.Erase(id); // удаляем ноду из топологии  if (is\_first) { // если это первая нода  unbind(branches[branch], id); // анбиндим сокет  branches.erase(std::next(branches.begin(), branch)); // удаляем всё что после  }  }  } else if (command == "exec") { // если команда - exec  size\_t count = 0; // количество параметров  int destId;  std::cin >> destId; // считываем id ноды, на которой запускаем  std::string s;  std::getline(std::cin,s); // считываем последующие параметры  std::istringstream iss(s);  // тут мы парсим команды для универсальности (чтобы можно было 1 или 2 параметра передавать)  std::vector<std::string> params;  std::string param;  while (iss >> param) {  count++;  params.push\_back(param);  }  int branch = network.Find(destId); // ищем ноду по айди  if (branch == -1) {  std::cout << "Error: incorrect node id!\n";  } else {  if (params.size() == 2) { // если нам передали 2 параметра  send\_message(branches[branch], std::to\_string(destId) + "exec " + std::to\_string(count) + " " + params[0] + " " + params[1]);  } else if (params.size() == 1) { // и если 1 параметр  send\_message(branches[branch], std::to\_string(destId) + "exec " + std::to\_string(count) + " " + params[0]);  }  std::string reply = receive\_message(branches[branch]);  std::cout << reply << "\n"; // получаем и выводим ответ  }  } else if (command == "heartbit") { // если команда - heartbit  int TIME;  std::cin >> TIME; // считываем таймаут  for (int i = 0; i < 10; i++) { // для примера - ограничим количество пингов до 10  for (int i = 0; i < branches.size(); ++i) { // идем по всем сокетам  // отправляем команду на пинг ноды  send\_message(branches[i], std::to\_string(i) + " heartbit " + std::to\_string(TIME));  std::cout << receive\_message(branches[i]) << "\n---\n"; // выводим ответ  }  sleep(TIME / 1000); // ждём TIME / 1000 секунд  }  } else if (command == "exit") { // если команда - выйти  for (size\_t i = 0; i < branches.size(); ++i) { // идём по всем сокетам  int first\_node\_id = network.GetFirstId(i); // получаем id первой ноды  send\_message(branches[i], std::to\_string(first\_node\_id) + " remove"); // удаляем её  std::string reply = receive\_message(branches[i]); // получаем ответ  if (reply != "OK") {  std::cout << reply << "\n"; // если не всё ок - выводим  } else {  unbind(branches[i], first\_node\_id); // иначе - анбиндим сокет  }  }  exit(0);  } else {  std::cout << "Incorrect command: " << command << "<!\n"; // если ввели херню - вежливо скажем об этом  }  }  } |

**Демонстрация работы программы**

**ivan@MiWiFi-R4AC-srv:/home/ivan/os/lab5$** ./main

create 1 -1

OK: 166

create 2 -1

OK: 169

exec 1 myVal

OK:1: 'myVal' not found

exec 1 myVal 1

OK:1

exec 1 myVal

OK:1: 1

exec 2 myVal

OK:2: 'myVal' not found

heartbit 500

1:Ok;

2:Ok;

1:Ok;

2:Ok;

1:Ok;

2:Ok;

1:Ok;

2:Ok;

1:Ok;

2:Ok;

1:Ok;

2:Ok;

1:Ok;

2:Ok;

1:Ok;

2:Ok;

1:Ok;

2:Ok;

1:Ok;

2:Ok;

remove 2

OK

heartbit 1000

1:Ok;

1:Ok;

1:Ok;

1:Ok;

1:Ok;

1:Ok;

1:Ok;

1:Ok;

1:Ok;

1:Ok;

exit

**Выводы**

Проделав данную работу, я научилсь принципам управлении серверами сообщений, применения отложенных вычислений, интеграции программных систем друг с другом. В частности, познакомилсь с библиотекой ZeroMQ.