Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Лабораторная работа №5-7 по курсу**

**«Операционные системы»**

Группа: М8О-209Б-23

Студент: Борисов Д.С.

Преподаватель: Миронов Е.С.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 18.12.24

Москва, 2024

**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 («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.

**Общий метод и алгоритм решения**

Программа состоит из 7 файлов: main.cpp (получает команды от пользователя и отправляет их в вычислительный узел), client.cpp (получает эти команды и выполняет их), timer.cpp, timer.h (реализация таймера), tree.cpp, tree.h (реализация бинарного дерева поиска), Makefile.

* create id — вставка вычислительного узла в бинарное дерево
* exec id subcommand — отправка подкоманды вычислительному узлу
* kill id — удаление вычислительного узла и всех его дочерних узлов из дерева
* pingall — все вычислительные узлы подтверждают свою работоспособность

**Код программы**

**main.cpp**

|  |
| --- |
| #include <iostream> #include <unistd.h> #include <string> #include <vector> #include <set> #include <sstream> #include <signal.h> #include "zmq.hpp" #include "tree.h"  const int WAIT\_TIME = 1000; const int PORT\_BASE = 5050;  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 = false;  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; }  void create\_node(int id, int port) {  char\* arg0 = strdup("./client");  char\* arg1 = strdup((std::to\_string(id)).c\_str());  char\* arg2 = strdup((std::to\_string(port)).c\_str());  char\* args[] = {arg0, arg1, arg2, NULL};  execv("./client", args); }  std::string get\_port\_name(const int port) {  return "tcp://127.0.0.1:" + std::to\_string(port); }  bool is\_number(std::string val) {  try  {  int tmp = std::stoi(val);  return true;  }  catch(std::exception& e)  {  std::cout << "Error: " << e.what() << "\n";  return false;  } }  int main() {  Tree T;  std::string command;  int child\_pid = 0;  int child\_id = 0;  zmq::context\_t context(1);  zmq::socket\_t main\_socket(context, ZMQ\_REQ);  std::cout << "Commands:\n";  std::cout << "create id\n";  std::cout << "exec id subcommand (start/stop/time)\n";  std::cout << "kill id\n";  std::cout << "pingall\n";  std::cout << "exit\n" << std::endl;  while(1)  {  std::cin >> command;  if (command == "create")  {  size\_t node\_id = 0;  std::string str = "";  std::string result = "";  std::cin >> str;  if (!is\_number(str))  {  continue;  }  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)  {  std::cout << "Unable to create first worker node\n";  child\_pid = 0;  exit(1);  }  else if (child\_pid == 0)  {  create\_node(node\_id, PORT\_BASE + node\_id);  }  else  {  child\_id = node\_id;  send\_message(main\_socket,"pid");  result = recieve\_message(main\_socket);  }  }  else  {  std::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")  {  T.push(node\_id);  }  std::cout << result << "\n";  }  else if (command == "kill")  {  int node\_id = 0;  std::string str = "";  std::cin >> str;  if (!is\_number(str))  {  continue;  }  node\_id = stoi(str);  if (child\_pid == 0)  {  std::cout << "Error: Not found\n";  continue;  }  if (node\_id == child\_id)  {  kill(child\_pid, SIGTERM);  kill(child\_pid, SIGKILL);  child\_id = 0;  child\_pid = 0;  T.kill(node\_id);  std::cout << "Ok\n";  continue;  }  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, std::min<int>(recieved\_message.size(), 2)) == "Ok")  {  T.kill(node\_id);  }  std::cout << recieved\_message << "\n";  }  else if (command == "exec")  {  std::string id\_str = "";  std::string subcommand = "";  int id = 0;  std::cin >> id\_str >> subcommand;  if (!is\_number(id\_str))  {  continue;  }  id = stoi(id\_str);  if ((subcommand != "start") && (subcommand != "stop") && (subcommand != "time"))  {  std::cout << "Wrong subcommandmand\n";  continue;  }  std::string message\_string = "exec " + std::to\_string(id) + " " + subcommand;  send\_message(main\_socket, message\_string);  std::string recieved\_message = recieve\_message(main\_socket);  std::cout << recieved\_message << "\n";  }  else if (command == "pingall")  {  send\_message(main\_socket,"pingall");  std::string recieved = recieve\_message(main\_socket);  std::istringstream is;  if (recieved.substr(0, std::min<int>(recieved.size(), 5)) == "Error")  {  is = std::istringstream("");  }  else  {  is = std::istringstream(recieved);  }  std::set<int> recieved\_T;  int rec\_id;  while (is >> rec\_id)  {  recieved\_T.insert(rec\_id);  }  std::vector<int> from\_tree = T.get\_nodes();  auto part\_it = partition(from\_tree.begin(), from\_tree.end(), [&recieved\_T] (int a)  {  return recieved\_T.count(a) == 0;  });  if (part\_it == from\_tree.begin())  {  std::cout << "Ok:-1\n";  }  else  {  std::cout << "Ok:";  for (auto it = from\_tree.begin(); it != part\_it; ++it)  {  std::cout << \*it << " ";  }  std::cout << "\n";  }  }  else if (command == "exit")  {  int n = system("killall client");  break;  }  }  return 0; } |

**client.cpp**

|  |
| --- |
| #include <iostream> #include <unistd.h> #include <string> #include <sstream> #include <exception> #include <signal.h> #include "zmq.hpp" #include "timer.h"  const int WAIT\_TIME = 1000; const int PORT\_BASE = 5050;  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 = false;  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; }  void create\_node(int id, int port) {  char\* arg0 = strdup("./client");  char\* arg1 = strdup((std::to\_string(id)).c\_str());  char\* arg2 = strdup((std::to\_string(port)).c\_str());  char\* args[] = {arg0, arg1, arg2, NULL};  execv("./client", args); }  std::string get\_port\_name(const int port) {  return "tcp://127.0.0.1:" + std::to\_string(port); }  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 rl\_kill(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& delete\_id, int& id, int& pid, std::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 rl\_exec(zmq::socket\_t& parent\_socket, zmq::socket\_t& socket, int& id, int& pid, std::string& request\_string) {  if (pid == 0)  {  std::string recieve\_message = "Error:" + std::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(std::istringstream& command\_stream, zmq::socket\_t& parent\_socket, zmq::socket\_t& left\_socket,  zmq::socket\_t& right\_socket, int& left\_pid, int& right\_pid, int& id, std::string& request\_string, Timer\* timer) {  std::string subcommand;  int exec\_id;  command\_stream >> exec\_id;  if (exec\_id == id)  {  command\_stream >> subcommand;  std::string recieve\_message = "";  if (subcommand == "start")  {  timer->start\_timer();  recieve\_message = "Ok:" + std::to\_string(id);  send\_message(parent\_socket, recieve\_message);  }  else if (subcommand == "stop")  {  timer->stop\_timer();  recieve\_message = "Ok:" + std::to\_string(id);  send\_message(parent\_socket, recieve\_message);  }  else if (subcommand == "time")  {  recieve\_message = "Ok:" + std::to\_string(id) + ": ";  recieve\_message += std::to\_string(timer->get\_time());  send\_message(parent\_socket, recieve\_message);  }  }  else if (exec\_id < id)  {  rl\_exec(parent\_socket, left\_socket, exec\_id,  left\_pid, request\_string);  }  else  {  rl\_exec(parent\_socket, right\_socket, exec\_id,  right\_pid, request\_string);  } }   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) {  std::ostringstream res;  std::string left\_res;  std::string right\_res;  res << std::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, std::min<int>(left\_res.size(),5) ) != "Error")  {  res << " " << left\_res;  }  if ((!right\_res.empty()) && (right\_res.substr(0, std::min<int>(right\_res.size(),5) ) != "Error"))  {  res << " "<< right\_res;  }  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 = std::stoi(argv[1]);  int parent\_port = std::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;  zmq::socket\_t left\_socket(context, ZMQ\_REQ);  zmq::socket\_t right\_socket(context, ZMQ\_REQ);  while(1)  {  std::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 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));  }  }  }  else if (command == "kill")  {  int delete\_id;  command\_stream >> delete\_id;  if (delete\_id < id)  {  rl\_kill(parent\_socket, left\_socket, delete\_id, left\_id, left\_pid, request\_string);  }  else  {  rl\_kill(parent\_socket, right\_socket, delete\_id, right\_id, right\_pid, request\_string);  }  }  else if (command == "exec")  {  exec(command\_stream, parent\_socket, left\_socket, right\_socket, left\_pid, right\_pid, 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;  }  }  return 0; } |

**timer.h**

|  |
| --- |
| #pragma once #include <chrono>  class Timer {  public:  Timer() = default;  ~Timer() = default;  void start\_timer();  void stop\_timer();  int get\_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(); } |

**tree.h**

|  |
| --- |
| #pragma once #include <vector>  struct Node {  int id;  Node\* left;  Node\* right; };  class Tree {  public:  void push(int);  void kill(int);  std::vector<int> get\_nodes();  ~Tree();  private:  Node\* root = NULL;  Node\* push(Node\* t, int);  Node\* kill(Node\* t, int);  void get\_nodes(Node\*, std::vector<int>&);  void delete\_node(Node\*); }; |

**tree.cpp**

|  |
| --- |
| #include <iostream> #include <vector> #include <algorithm> #include "tree.h"  Tree::~Tree() {  delete\_node(root); }  void Tree::push(int id) {  root = push(root, id); }  void Tree::kill(int id) {  root = kill(root, id); }  void Tree::delete\_node(Node\* node) {  if(node == NULL)  {  return;  }  delete\_node(node->right);  delete\_node(node->left);  delete node; }  std::vector<int> Tree::get\_nodes() {  std::vector<int> result;  get\_nodes(root, result);  return result; }  void Tree::get\_nodes(Node\* node, std::vector<int>& v) {  if (node == NULL)  {  return;  }  get\_nodes(node->left, v);  v.push\_back(node->id);  get\_nodes(node->right, v); }  Node\* Tree::push(Node\* root, int val) {  if (root == NULL)  {  root = new Node;  root->id = val;  root->left = NULL;  root->right = NULL;  return root;  }  else if (val < root->id)  {  root->left = push(root->left, val);  }  else if (val >= root->id)  {  root->right = push(root->right, val);  }  return root; }  Node\* Tree::kill(Node\* root\_node, int val) {  Node\* node;  if (root\_node == NULL)  {  return NULL;  }  else if (val < root\_node->id)  {  root\_node->left = kill(root\_node->left, val);  }  else if (val >root\_node->id)  {  root\_node->right = kill(root\_node->right, val);  }  else  {  node = root\_node;  if (root\_node->left == NULL)  {  root\_node = root\_node->right;  }  else if (root\_node->right == NULL)  {  root\_node = root\_node->left;  }  delete node;  }  if (root\_node == NULL)  {  return root\_node;  }  return root\_node; } |

**Протокол работы программы**

|  |
| --- |
| Commands:  create id  exec id subcommand (start/stop/time)  kill id  pingall  exit  create 4  Ok:10686  create 2  Ok:10691  create 8  Ok:10711  exec 4 start  Ok:4  exec 8 start  Ok:8  exec 4 time  Ok:4: 10283  exec 8 time  Ok:8: 8413  kill 2  Ok  pingall  Ok:-1  Exit |

**Вывод**

Выполняя лабораторную работу, я освоил основы библиотеки ZMQ, а также познакомился с очередями сообщений.