Московский авиационный институт
(Национальный исследовательский университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №5-7 по курсу “Операционные системы”

Студент: Немкова Анастасия Романовна

Группа: М8О-208Б-22

Преподаватель: Миронов Евгений Сергеевич

Вариант: 9

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2023

Содержание

Репозиторий

`https://github.com/anastasia-nemkova/OS_labs`

Постановка задачи

Цель работы:

Приобретение практических навыков в:

- Управлении серверами сообщений (№5)
- Применение отложенных вычислений (№6)
- Интеграция программных систем друг с другом (№7)

Задание:

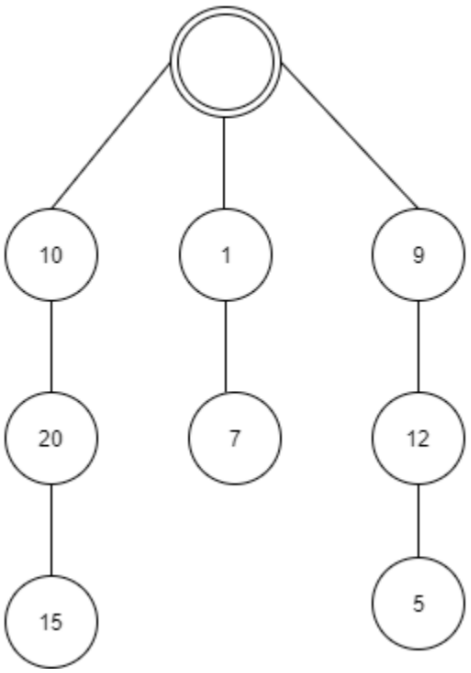
Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить 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.

Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

Создание нового вычислительного узла
Формат команды: create id [parent]
id – целочисленный идентификатор нового вычислительного узла
parent – целочисленный идентификатор родительского узла
Формат вывода:
«Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла
«Error: Already exists» - вычислительный узел с таким идентификатором уже существует
«Error: Parent not found» - нет такого родительского узла с таким идентификатором
«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не удается связаться
«Error: [Custom error]» - любая другая обрабатываемая ошибка
Пример:
> create 10 5
Ok: 3128

Исполнение команды на вычислительном узле
Формат команды: exec id [params]
id – целочисленный идентификатор вычислительного узла, на который отправляется команда
Формат вывода:
«Ok:id: [result]», где result – результат выполненной команды
«Error:id: Not found» - вычислительный узел с таким идентификатором не найден
«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом
«Error:id: [Custom error]» - любая другая обрабатываемая ошибка

Вариант 9:
Топология



Все вычислительные узлы находятся в списке. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

Тип команд на вычислительных узлах

Набор команд 2 (локальный целочисленный словарь)

Формат команды сохранения значения: exec id name value

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

name – ключ, по которому будет сохранено значение (строка формата [A-Za-z0-9]+)

value – целочисленное значение

Формат команды загрузки значения: exec id name

Пример:

> exec 10 MyVar

Ok:10: 'MyVar' not found

> exec 10 MyVar 5

Ok:10

> exec 12 MyVar

Ok:12: 'MyVar' not found

> exec 10 MyVar

Ok:10: 5

> exec 10 MyVar 7

Ok:10

> exec 10 MyVar

Ok:10: 7

Тип проверки доступности узлов
Команда проверки 2
Формат команды: ping id
Команда проверяет доступность конкретного узла. Если узла нет, то необходимо выводить ошибку: «Error: Not found»
Пример:
> ping 10
Ok: 1 // узел 10 доступен
> ping 17
Ok: 0 // узел 17 недоступен

Общие сведения о программе

Программа компилируется из файлов socket.cpp с функциями для работы с сокетами, topology.cpp с топологией списка, control_node.cpp с управляющим узлом, calcutation_node.cpp с вычислительным узлом. Также имеются заголовочные файлы socket.hpp и topology.hpp и тестовый файл lab5-7_test.cpp. В программе работы были использованы следующие системные вызовы:

- fork() - создание нового процесса
- execl() - замена текустщего процесса на новый
- bind() - привязка сокета к локальному адресу и порту
- coonect() - установка соединения сокета с удаленным адресом и портом
- unbind() - отключение сокета от локального адреса и порта
- disconnect() - разрыв соединения с удаленным сокетом
- send() - отправка сообщений через сокет
- revc() - приём сообщений через сокет
- set() - установка опций сокета

Общий метод и алгоритм решения

Управляющий узел отвечает за создание, управление и завершение работы вычислительных узлов. При создании нового узла происходит создание нового процесса, к которому подключается новый сокет для обмена сообщениями с родительским узлом. Для коммуникации между узлами используются ZeroMQ сокеты, которые позволяют отправлять и принимать сообщения через TCP протокол.

Каждый вычислительный узел имеет свой идентификатор, который используется для обращения к нему из управляющего узла. Узлы могут создавать другие узлы, передавать им команды на выполнение определенных действий, такие как добавление новых данных в хранилище или проверка существующих данных. Также они отвечают на запросы проверки их доступности.

При завершении работы узел отключается от родительского узла и завершает свою работу, освобождая ресурсы.

Исходный код

socket.hpp

```cpp
#pragma once

#include <iostream>
#include <zmq.hpp>
#include <string>
#include <unistd.h>
#include <sstream>
#include <set>
#include <unordered_map>
#include <optional>

constexpr int MAIN_PORT = 4040;

void sendMessage(zmq::socket_t& socket, const std::string& msg);
std::string receiveMessage(zmq::socket_t& socket);
void connect(zmq::socket_t& socket, int id);
void disconnect(zmq::socket_t& socket, int id);
void bind(zmq::socket_t& socket, int id);
void unbind(zmq::socket_t& socket, int id);
```

socket.cpp

```cpp
#include <socket.hpp>

void sendMessage(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, zmq::send_flags::none);
}


std::string receiveMessage(zmq::socket_t& socket) {
    zmq::message_t msg;
    int msgReceiv;
    try {
        std::optional<size_t> result = socket.recv(msg);
        if (result) {
            msgReceiv = static_cast<int>(*result);
        }
    }
    catch (...) {
        msgReceiv = 0;
    }
    if (msgReceiv == 0) {
        return "Error: Node is unavailable";
    }
    std::string receivedMsg(static_cast<char*>(msg.data()), msg.size());
    return receivedMsg;

}

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);
}

```

```cpp
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);
}
```

topology.hpp

```cpp
#pragma once

#include <iostream>
#include <list>

class Topology {
private:
    using listType = std::list<std::list<int>>;

    listType list;
public:
    Topology() : list() {}

    void insert(int id, int parentId);
    int find(int id);
    void erase(int id);
    int getFirstId(int listId);
};
```

topology.cpp

```cpp
#include "topology.hpp"

void Topology::insert(int id, int parentId) {
    if (parentId == -1) {
        std::list<int> newList;
        newList.push_back(id);
        list.push_back(newList);
        return;
    }
    int listId = find(parentId);
    if (listId == -1) {
        throw std::runtime_error("Wrong parent id");
    }
    auto it1 = list.begin();
    std::advance(it1, listId);
    for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {
        if (*it2 == parentId) {
            it1->insert(++it2, id);
            return;
        }
    }
}

int Topology::find(int id) {
    int curListId = 0;
    for (auto it1 = list.begin(); it1 != list.end(); ++it1) {
        for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {
            if (*it2 == id) {
                return curListId;
            }
        }
        ++curListId;
    }
    return -1;
}

void Topology::erase(int id) {
    int listId = find(id);
    if (listId == -1) {
        throw std::runtime_error("Wrong id");
    }
    auto it1 = list.begin();
```

```
    std::advance(it1, listId);
    for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {
        if (*it2 == id) {
            it1->erase(it2, it1->end());
            if (it1->empty()) {
                list.erase(it1);
            }
            return;
        }
    }
}

int Topology::getFirstId(int listId) {
    auto it1 = list.begin();
    std::advance(it1, listId);
    if (it1->begin() == it1->end()) {
        return -1;
    }
    return *(it1->begin());
}
```

calculation_node.cpp

```
#include "socket.hpp"

int main(int argc, char* argv[]) {
    if (argc != 2 && argc != 3) {
        throw std::runtime_error("Wrong args for counting node");
    }

    int curId = atoi(argv[1]);
    int childId = -1;
    if (argc == 3) {
        childId = atoi(argv[2]);
    }
    std::string adr;
    std::string path = getenv("PATH_TO_CLIENT");

    std::unordered_map<std::string, int> dictionary;

    zmq::context_t context;
    zmq::socket_t parentSocket(context, ZMQ_REP);

    connect(parentSocket, curId);

    zmq::socket_t childSocket(context, ZMQ_REQ);
    if (childId != -1) {
        bind(childSocket, childId);
    }
    childSocket.set(zmq::sockopt::sndtimeo, 5000);

    std::string message;
    while (true) {
        message = receiveMessage(parentSocket);
        std::istringstream request(message);
        int destId;
        request >> destId;

        std::string command;
        request >> command;

        if (destId == curId) {

            if (command == "pid") {
                sendMessage(parentSocket, "OK: " + std::to_string(getpid()));
            } else if (command == "create") {

                int new_childId;
                request >> new_childId;
                if (childId != -1) {
```

```
48                      unbind(childSocket, childId);
49                  }
50                  bind(childSocket, new_childId);
51                  pid_t pid = fork();
52                  if (pid < 0) {
53                      std::cout << "Can't create new process" << std::endl;
54                      return -1;
55                  }
56                  if (pid == 0) {
57                      execl(path.c_str(), path.c_str(), std::to_string(new_childId).
      c_str(), std::to_string(childId).c_str(), NULL);
58                      std::cout << "Can't execute new process" << std::endl;
59                      return -2;
60                  }
61                  sendMessage(childSocket, std::to_string(new_childId) + " pid");
62                  childId = new_childId;
63                  sendMessage(parentSocket, receiveMessage(childSocket));
64
65              } else if (command == "check") {
66                  std::string key;
67                  request >> key;
68                  if (dictionary.find(key) != dictionary.end()) {
69                      sendMessage(parentSocket, "OK: " + std::to_string(curId) + ":
      " + std::to_string(dictionary[key]));
70                  } else {
71                      sendMessage(parentSocket, "OK: " + std::to_string(curId) + ":
      '" + key + "' not found");
72                  }
73              } else if (command == "add") {
74                  std::string key;
75                  int value;
76                  request >> key >> value;
77                  dictionary[key] = value;
78                  sendMessage(parentSocket, "OK: " + std::to_string(curId));
79              }  else if (command == "ping") {
80                  std::string reply;
81                  if (childId != -1) {
82                      sendMessage(childSocket, std::to_string(childId) + " ping");
83                      std::string msg = receiveMessage(childSocket);
84                      reply += " " + msg;
85                  }
86                  sendMessage(parentSocket, std::to_string(curId) + reply);
87              } else if (command == "kill") {
88                  if (childId != -1) {
89                      sendMessage(childSocket, std::to_string(childId) + " kill");
90                      std::string msg = receiveMessage(childSocket);
91                      if (msg == "OK") {
92                          sendMessage(parentSocket, "OK");
93                      }
94                      unbind(childSocket, childId);
95                      disconnect(parentSocket, curId);
96                      break;
97                  }
98                  sendMessage(parentSocket, "OK");
99                  disconnect(parentSocket, curId);
100                 break;
101             }
102         }
103          else if (childId != -1) {
104             sendMessage(childSocket, message);
105             sendMessage(parentSocket, receiveMessage(childSocket));
106             if (childId == destId && command == "kill") {
107                 childId = -1;
108             }
109         } else {
110             sendMessage(parentSocket, "Error: Node is unavailable");
111         }
112     }
113 }
```

control_node.cpp

```cpp
#include "topology.hpp"
#include "socket.hpp"

int main() {
    std::string path = getenv("PATH_TO_CLIENT");
    Topology list;
    std::vector<zmq::socket_t> branches;
    zmq::context_t context;

    std::string command;

    while (true) {
        std::cin >> command;
        if (command == "create") {
            int nodeId, parentId;
            std::cin >> nodeId >> parentId;
            if (list.find(nodeId) != -1) {
                std::cout << "Error: Already exists" << std::endl;
            } else if (parentId == -1) {
                pid_t pid = fork();
                if (pid < 0) {
                    std::cout << "Can't create new process" << std::endl;
                    return -1;
                } else if (pid == 0) {
                    execl(path.c_str(), path.c_str(), std::to_string(nodeId).c_str(), NULL);
                    std::cout << "Can't execute new process" << std::endl;
                    return -2;
                }
                branches.emplace_back(context, ZMQ_REQ);
                branches[branches.size() - 1].set(zmq::sockopt::sndtimeo, 5000);
                bind(branches[branches.size()-1], nodeId);

                sendMessage(branches[branches.size() - 1], std::to_string(nodeId) + " pid");

                std::string reply = receiveMessage(branches[branches.size() - 1]);
                std::cout << reply << std::endl;
                list.insert(nodeId, parentId);

            } else if (list.find(parentId) == -1) {

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

            } else {
                int branch = list.find(parentId);
                sendMessage(branches[branch], std::to_string(parentId) + "create " + std::to_string(nodeId));

                std::string reply = receiveMessage(branches[branch]);
                std::cout << reply << std::endl;
                list.insert(nodeId, parentId);
            }
        } else if (command == "exec") {
            std::string s;
            getline(std::cin, s);
            std::string execCommand;
            std::vector<std::string> tmp;
            std::string tmp1 = " ";
            for (size_t i = 1; i < s.size(); i++) {
                tmp1 += s[i];
                if (s[i] == ' ' || i == s.size() - 1) {
                    tmp.push_back(tmp1);
                    tmp1 = " ";
                }
            }
            if (tmp.size() == 2) {
```

```
65                  execCommand = "check";
66              } else {
67                  execCommand = "add";
68              }
69              int destId = stoi(tmp[0]);
70              int branch = list.find(destId);
71              if (branch == -1) {
72                  std::cout << "There is no such node id" << std::endl;
73              } else {
74                  if (execCommand == "check") {
75                      sendMessage(branches[branch], tmp[0] + "check" + tmp[1]);
76
77                  } else if (execCommand == "add") {
78                      std::string value;
79                      sendMessage(branches[branch], tmp[0] + "add" + tmp[1] + " " +
    tmp[2]);
80                  }
81                  std::string reply = receiveMessage(branches[branch]);
82                  std::cout << reply << std::endl;
83              }
84          } else if (command == "kill") {
85              int id;
86              std::cin >> id;
87              int branch = list.find(id);
88              if (branch == -1) {
89                  std::cout << " Error: incorrect node id" << std::endl;
90              } else {
91                  bool is_first = (list.getFirstId(branch) == id);
92                  sendMessage(branches[branch], std::to_string(id) + "kill");
93                  std::string reply = receiveMessage(branches[branch]);
94                  std::cout << reply << std::endl;
95                  list.erase(id);
96                  if (is_first) {
97                      unbind(branches[branch], id);
98                      branches.erase(branches.begin() + branch);
99                  }
100             }
101         } else if (command == "ping") {
102             int nodeId;
103             std::cin >> nodeId;
104             std::set<int> available_nodes;
105             if (list.find(nodeId) == -1) {
106                 std::cout << "Error: Not found" << std::endl;
107             } else {
108                 for (size_t i = 0; i < branches.size(); ++i) {
109                     int first_node_id = list.getFirstId(i);
110                     sendMessage(branches[i], std::to_string(first_node_id) + "
    ping");
111
112                     std::string received_message = receiveMessage(branches[i]);
113                     std::istringstream reply(received_message);
114                     int node;
115                     while(reply >> node) {
116                         available_nodes.insert(node);
117                     }
118                 }
119                 if (available_nodes.empty()) {
120                     std::cout << "OK: 0" << std::endl;
121                 } else {
122                     if(available_nodes.find(nodeId) != available_nodes.end()) {
123                         std::cout << "OK: 1" << std::endl;
124                     }
125                 }
126             }
127         } else if (command == "exit") {
128             for (size_t i = 0; i < branches.size(); ++i) {
129                 int firstNodeId = list.getFirstId(i);
130                 sendMessage(branches[i], std::to_string(firstNodeId) + " kill");
131                 std::string reply = receiveMessage(branches[i]);
```

```
                if (reply != "OK") {
                    std::cout << reply << std::endl;
                } else {
                    unbind(branches[i], firstNodeId);
                }
            }
            exit(0);
        } else {
            std::cout << "Not correct command" << std::endl;
        }
    }
}
```

lab5-7_test.cpp

```
#include <gtest/gtest.h>

#include "topology.hpp"
#include "socket.hpp"
#include <thread>

TEST(FifthSeventhLabTest, SocketTest) {
    zmq::context_t context;
    zmq::socket_t repSocket(context, ZMQ_REP);
    bind(repSocket, 3);

    std::thread serverThread([&repSocket]() {
        std::string receivedMessage = receiveMessage(repSocket);
        EXPECT_EQ(receivedMessage, "TestMSG");

        sendMessage(repSocket, "ReplyMSG");
    });

    zmq::socket_t reqSocket(context, ZMQ_REQ);
    connect(reqSocket, 3);

    sendMessage(reqSocket, "TestMSG");

    std::string replyMessage = receiveMessage(reqSocket);
    EXPECT_EQ(replyMessage, "ReplyMSG");

    disconnect(reqSocket, 3);
    unbind(repSocket, 3);
    serverThread.join();
}

TEST(FifthSeventhLabTest, TopologyTest) {
    Topology topology;

    topology.insert(1, -1);
    topology.insert(2, 1);
    topology.insert(3, 2);

    EXPECT_EQ(topology.find(1), 0);
    EXPECT_EQ(topology.find(2), 0);
    EXPECT_EQ(topology.find(3), 0);

    EXPECT_EQ(topology.getFirstId(0), 1);

    topology.erase(2);

    EXPECT_EQ(topology.find(2), -1);
}


int main(int argc, char *argv[]) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Демонстрация работы программы

Сама программа

```
arnemkova@LAPTOP-TA2RV74U:~/OS_labs/build$ ./lab5-7/client
create 1 -1
OK: 18007
create 2 1
OK: 18018
create 3 2
OK: 18030
exec 2 may
OK: 2: 'may' not found
exec 2 may 1234
OK: 2
exec 2 may
OK: 2: 1234
ping 1
OK: 1
ping 2
OK: 1
ping 3
OK: 1
kill 1
OK
exit
```

Тесты

```
arnemkova@LAPTOP-TA2RV74U:~/OS_labs/build$ ./tests/lab5-7_test
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from FifthSeventhLabTest
[ RUN      ] FifthSeventhLabTest.SocketTest
[       OK ] FifthSeventhLabTest.SocketTest (1 ms)
[ RUN      ] FifthSeventhLabTest.TopologyTest
[       OK ] FifthSeventhLabTest.TopologyTest (0 ms)
[----------] 2 tests from FifthSeventhLabTest (1 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (1 ms total)
[  PASSED  ] 2 tests.
```

Вывод

В ходе данной лабораторной работы я познакомилась с библиотекой ZeroMQ, которая позволяет создавать распределенные системы с использованием различных моделей взаимодействия между узлами. Я изучила работу с сокетами ZMQ_REP и ZMQ_REQ, которые позволяют установить соединение между управляющим и вычислительным узлами и организовать взаимодействие между ними через запросы и ответы. Также я узнала о принципах мультиплексирования ввода/вывода для эффективной работы с несколькими сокетами одновременно, а также применении асинхронности для параллельной обработки запросов без блокировки основного потока выполнения.