Московский авиационный институт (национальный исследовательский университет)

Институт информационных технологий и прикладной математики

«Кафедра вычислительной математики и программирования»

Лабораторная работа по предмету "Операционные системы" №5-7

Студент: Бурунов М.А.

Преподаватель: Миронов Е.С.

Группа: М8О-206Б-22

Дата: 01.03.2024

Москва 2024 г.

Оглавление

Цель работы

Целью является приобретение практических навыков в:

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

Постановка задачи

Общий алгоритм решения

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. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)

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

Примечания: создание нового управляющего узла осуществляется пользователем программы при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы.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

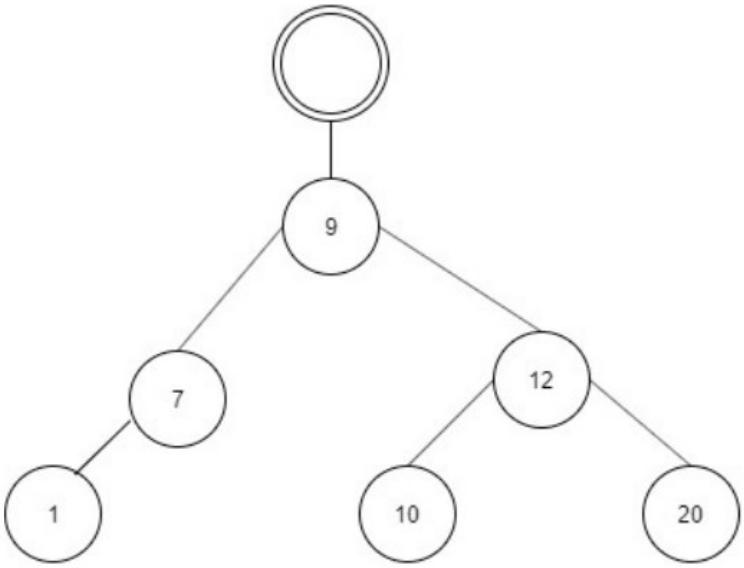
«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

«Error:id: [Custom error]» - любая другая обрабатываемая ошибка

Вариант 47:

Топология 3

Все вычислительные узлы хранятся в бинарном дереве поиска. [parent] — является необязательным параметром.



Набор команд 1 (подсчет суммы n чисел)

Формат команды: exec id n k1 … kn

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

n – количество складываемых чисел (от 1 до 108)

k1 … kn – складываемые числа

Пример:

> exec 10 3 1 2 3

Ok:10: 6

Команда проверки 1

Формат команды: pingall

Вывод всех недоступных узлов вывести разделенные через точку запятую.

Пример:

> pingall

Ok: -1 // Все узлы доступны

> pingall

Ok: 7;10;15 // узлы 7, 10, 15 — недоступны

Реализация

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 PORTBASE = 5050;
Tree T;
std::string request;
int childProcessId = 0;
int childId = 0;

zmq::context_t context(1); // 1 - один I/O поток
zmq::socket_t mainSocket(context, ZMQ_REQ); // Request-reply

bool sendMessage(zmq::socket_t &socket, const std::string &message) {
    zmq::message_t zmqMessage(message.c_str(), message.size());
    zmq::send_result_t result = socket.send(zmqMessage, zmq::send_flags::none);

    return result.has_value(); // Результат отправки (T or F)
}

std::string recieveMessage(zmq::socket_t &socket) {
    zmq::message_t message;
    bool ok = false;
    try { // В случае успеха вернём сообщение
        zmq::recv_result_t result = socket.recv(message, zmq::recv_flags::none);
        ok = result.has_value();
    }
    catch (...) {
        ok = false;
    }

    std::string receivedMessage(static_cast<char *>(message.data()), message.size());
    if (receivedMessage.empty() || !ok)
```

```
        return "Error: Node is not available";

    return receivedMessage;
}

void createNode(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);
}

// Преобразуем номер порта в строку нужного формата для ZMQ
std::string getPortName(const int port) {
    return "tcp://127.0.0.1:" + std::to_string(port);
}

bool isNumber(std::string value) {
    try {
        int check = std::stoi(value);
        return true;
    }
    catch (std::exception &e) {
        std::cout << "Error: " << e.what() << "\n";
        return false;
    }
}

void funcCreate() {
    size_t nodeId = 0;
    std::string lineInput = "";
    std::string result = "";
    std::cin >> lineInput;
    if (!isNumber(lineInput))
        return;

    nodeId = stoi(lineInput);
    if (childProcessId == 0) {
        mainSocket.bind(getPortName(PORTBASE + nodeId)); // Привязываем сокет к
порту
        childProcessId = fork();
        if (childProcessId == -1) {
            std::cout << "Unable to create first worker node\n";
            childProcessId = 0;
```

```
            exit(1);
        }
        else if (childProcessId == 0)
            createNode(nodeId, PORTBASE + nodeId);

        else {
            childId = nodeId;
            sendMessage(mainSocket, "pid");
            result = recieveMessage(mainSocket);
        }
    }
    else { // Если дочерний процеес существует, отправляем сообщение о создании нового узла
        std::string message_send = "create " + std::to_string(nodeId);
        sendMessage(mainSocket, message_send);
        result = recieveMessage(mainSocket);
    }
    if (result.substr(0, 2) == "Ok")
        T.push(nodeId);

    std::cout << result << "\n";
}

void funcKill() {
    int nodeId = 0;
    std::string lineInput = "";
    std::cin >> lineInput;
    if (!isNumber(lineInput))
        return;

    nodeId = stoi(lineInput);
    if (childProcessId == 0) {
        std::cout << "Error: Not found\n";
        return;
    }
    if (nodeId == childId) { // Если дочерний процесс соответсвует узлу
        kill(childProcessId, SIGTERM);
        kill(childProcessId, SIGKILL);
        childId = 0;
        childProcessId = 0;
        T.kill(nodeId);
        std::cout << "Ok\n";
        return;
    }
```

```
    // Если не соответсвтует
    std::string message = "kill " + std::to_string(nodeId);
    sendMessage(mainSocket, message);
    std::string recieved_message = recieveMessage(mainSocket);
    if (recieved_message.substr(0, std::min<int>(recieved_message.size(), 2)) == "Ok")
        T.kill(nodeId);

    std::cout << recieved_message << "\n";
}

void funcExec() {
    std::string stringID = "";
    std::string amountNumbers = "";
    std::string number = "";

    int id = 0;
    std::cin >> stringID >> amountNumbers;
    if (!isNumber(stringID))
        return;

    if (!isNumber(amountNumbers))
        return;

    std::vector<std::string> inputNumbers;
    for (size_t i = 0; i < stoi(amountNumbers); i++) {
        std::cin >> number;
        inputNumbers.push_back(number);
    }

    id = stoi(stringID);

    std::string messageLine = "exec " + std::to_string(id) + " " + amountNumbers;
    for (size_t i = 0; i < stoi(amountNumbers); i++)
        messageLine += (" " + (inputNumbers[i]));

    sendMessage(mainSocket, messageLine);
    std::string result = recieveMessage(mainSocket);
    std::cout << result << "\n";
}

int main() {
    std::cout << "requests:\n";
    std::cout << "create id\n";
    std::cout << "exec id amount_num num1 num2...num_n\n";
    std::cout << "kill id\n";
```

```
    std::cout << "pingall\n";
    std::cout << "exit\n" << std::endl;

    while (1) {
        std::cin >> request;
        if (request == "create")
            funcCreate();

        else if (request == "kill")
            funcKill();

        else if (request == "exec")
            funcExec();

        else if (request == "pingall") {
            sendMessage(mainSocket, "pingall");
            std::string recieved = recieveMessage(mainSocket);
            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 recievedId;
            while (is >> recievedId)
                recieved_T.insert(recievedId);

            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 (request == "exit") {
            int n = system("kill all client");
```

```
            break;
        }
    }

    return 0;
}
```

---

client.cpp

```
#include <iostream>
#include <unistd.h>
#include <string>
#include <sstream>
#include <exception>
#include <signal.h>
#include "zmq.hpp"

const int PORTBASE  = 5050;

bool sendMessage(zmq::socket_t& socket, const std::string& message) {
    zmq::message_t zmqMessage(message.c_str(), message.size());
    zmq::send_result_t result = socket.send(zmqMessage, zmq::send_flags::none);
    return result.has_value();
}

std::string recieveMessage(zmq::socket_t &socket) {
    zmq::message_t message;
    bool ok = false;
    try {
        zmq::recv_result_t result = socket.recv(message, zmq::recv_flags::none);
        ok = result.has_value();
    }
    catch (...) {
        ok = false;
    }

    std::string received_message(static_cast<char*>(message.data()), message.size());
    if (received_message.empty() || !ok)
        return "Error: Node is not available";

    return received_message;
}

void createNode(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 getPortName(const int port) {
    return "tcp://127.0.0.1:" + std::to_string(port);
}

void create(zmq::socket_t& parentsocket, zmq::socket_t& socket, int& create_id, int& id,
int& pid) {
    if (pid == -1) {
        sendMessage(parentsocket, "Error: Cannot fork");
        pid = 0;
    }
    else if (pid == 0)
        createNode(create_id,PORTBASE + create_id);

    else {
        id = create_id;
        sendMessage(socket, "pid");
        sendMessage(parentsocket, recieveMessage(socket));
    }
}

void kill(zmq::socket_t& parentsocket, zmq::socket_t& socket,  int& delete_id, int& id,
int& pid, std::string& request_string) {
    if (id == 0)
        sendMessage(parentsocket, "Error: Not found");

    else if (id == delete_id) {
        sendMessage(socket, "kill_children");
        recieveMessage(socket);
        kill(pid,SIGTERM);
        kill(pid,SIGKILL);
        id = 0;
        pid = 0;
        sendMessage(parentsocket, "Ok");
    }
    else {
        sendMessage(socket, request_string);
        sendMessage(parentsocket, recieveMessage(socket));
    }
```

```
}

void rl_exec(zmq::socket_t& parentsocket, zmq::socket_t& socket,  int& id, int& pid,
std::string& request_string) {
    if (pid == 0) {
        std::string recieveMessage = "Error:" + std::to_string(id);
        recieveMessage += ": Not found";
        sendMessage(parentsocket, recieveMessage);
    }
    else {
        sendMessage(socket, request_string);
        sendMessage(parentsocket, recieveMessage(socket));
    }
}

void exec(std::istringstream& request_stream, zmq::socket_t& parentsocket,
zmq::socket_t& left_socket,
            zmq::socket_t& right_socket, int& left_pid, int& right_pid, int& id, std::string&
request_string) {
    std::string size_str;
    std::string number;
    int exec_id;
    request_stream >> exec_id;
    if (exec_id == id) {
        request_stream >> size_str;
        int size=stoi(size_str);
        int sum=0;
        for (size_t i = 0; i < size; i++) {
            request_stream >> number;
            sum+=stoi(number);
        }

        std::string recieveMessage = std::to_string(sum);
        sendMessage(parentsocket, recieveMessage);
    }
    else if (exec_id < id) {
        rl_exec(parentsocket, left_socket, exec_id,
                left_pid, request_string);
    }
    else {
        rl_exec(parentsocket, right_socket, exec_id,
                right_pid, request_string);
    }
}
```

```
void pingall(zmq::socket_t& parentsocket, 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) {
        sendMessage(left_socket, "pingall");
        left_res = recieveMessage(left_socket);
    }
    if (right_pid != 0) {
        sendMessage(right_socket, "pingall");
        right_res = recieveMessage(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;
    }
    sendMessage(parentsocket, res.str());
}

void kill_children(zmq::socket_t& parentsocket, zmq::socket_t& left_socket,
zmq::socket_t& right_socket, int& left_pid, int& right_pid) {
    if (left_pid == 0 && right_pid == 0)
        sendMessage(parentsocket, "Ok");

    else {
        if (left_pid != 0) {
            sendMessage(left_socket, "kill_children");
            recieveMessage(left_socket);
            kill(left_pid,SIGTERM);
            kill(left_pid,SIGKILL);
        }
        if (right_pid != 0) {
            sendMessage(right_socket, "kill_children");
            recieveMessage(right_socket);
            kill(right_pid,SIGTERM);
            kill(right_pid,SIGKILL);
```

```
        }
        sendMessage(parentsocket, "Ok");
    }
}

int main(int argc, char** argv) {
    int id = std::stoi(argv[1]);
    int parent_port = std::stoi(argv[2]);
    zmq::context_t context(3);
    zmq::socket_t parentsocket(context, ZMQ_REP);
    parentsocket.connect(getPortName(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(true) {
        std::string request_string = recieveMessage(parentsocket);
        std::istringstream request_stream(request_string);
        std::string request;
        request_stream >> request;
        if (request == "id") {
            std::string parent_string = "Ok:" + std::to_string(id);
            sendMessage(parentsocket, parent_string);
        }
        else if (request == "pid") {
            std::string parent_string = "Ok:" + std::to_string(getpid());
            sendMessage(parentsocket, parent_string);
        }
        else if (request == "create") {
            int create_id;
            request_stream >> create_id;
            if (create_id == id) {
                std::string message_string = "Error: Already exists";
                sendMessage(parentsocket, message_string);
            }
            else if (create_id < id) {
                if (left_pid == 0) {
                    left_socket.bind(getPortName(PORTBASE + create_id));
                    left_pid = fork();
                    create(parentsocket, left_socket, create_id, left_id, left_pid);
                }
```

```
            else {
                sendMessage(left_socket, request_string);
                sendMessage(parentsocket, recieveMessage(left_socket));
            }
        }
        else {
            if (right_pid == 0) {
                right_socket.bind(getPortName(PORTBASE + create_id));
                right_pid = fork();
                create(parentsocket, right_socket, create_id, right_id, right_pid);
            }
            else {
                sendMessage(right_socket, request_string);
                sendMessage(parentsocket, recieveMessage(right_socket));
            }
        }
    }
    else if (request == "kill") {
        int delete_id;
        request_stream >> delete_id;
        if (delete_id < id)
            kill(parentsocket, left_socket, delete_id, left_id, left_pid, request_string);

        else
            kill(parentsocket, right_socket, delete_id, right_id, right_pid, request_string);
    }
    else if (request == "exec")
        exec(request_stream, parentsocket, left_socket, right_socket, left_pid, right_pid,
id, request_string);

    else if (request == "pingall")
        pingall(parentsocket, id, left_socket, right_socket, left_pid, right_pid);

    else if (request == "kill_children")
        kill_children(parentsocket, left_socket, right_socket, left_pid, right_pid);

    if (parent_port == 0)
        break;
  }

  return 0;
}
```

---

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;
}
```

Пример работы

./main

requests:

create id

exec id amount_num num1 num2...num_n

kill id

pingall

exit

create 1

Ok:329420

create 2

Ok:329446

create 3

Ok:329482

pingall

Ok:-1

kill 2

Ok

pingall

Ok:3

exec 1 4 10 20 30 40

100

exit

sh: 1: kill: Illegal number: all

Вывод

В процессе выполнения данной работы я ознакомился с понятием очередей сообщений, которые представляют собой дополнительный способ обмена данными между процессами. Для реализации данной лабораторной работы я применил библиотеку zeromq. В этой лабораторной работе мы использовали знания из прошлых лабораторных работ, а также применяли новые знания, из-за чего она является, на мой взгляд, самой сложной, но вместе с этим интересной лабораторной.