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: Брюханов З. Д.
Преподаватель: Миронов Е. С.
Группа: М8О-207Б-22
Дата: 09.12.2023
Оценка:
Подпись:

Москва 2023 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

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 Пример:

> create 10 5

Ok: 3128

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

Пример:

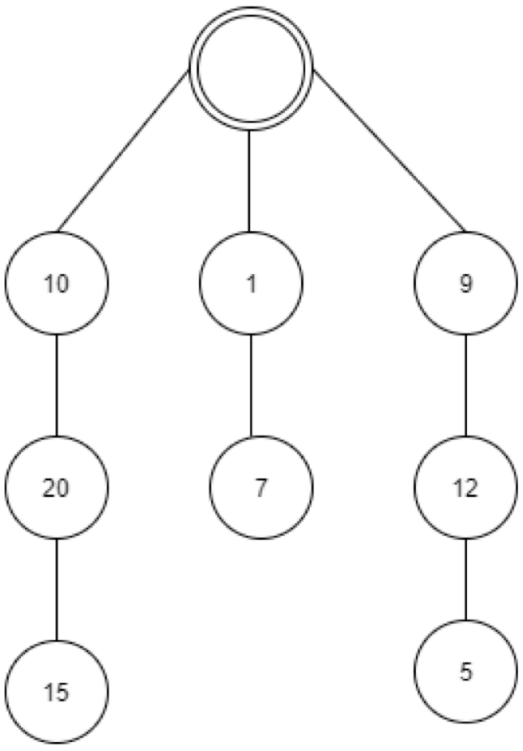
Можно найти в описании конкретной команды, определенной вариантом задания.

Примечание: выполнение команд должно быть асинхронным. Т.е. пока выполняется команда на одном из вычислительных узлов, то можно отправить следующую команду на другой вычислительный узел.

Вариант 17:

Топология 1.

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.



Набор команд 4 (поиск подстроки в строке).

Формат команды:

> exec id

> text_string

> pattern_string

[result] – номера позиций, где найден образец, разделенный точкой с запятой

text_string — текст, в котором искать образец. Алфавит: [A-Za-z0-9]. Максимальная длина строки 108 символов

pattern_string — образец

Пример:

> exec 10

> abracadabra

> abra
Ok:10:0;7
> exec 10
> abracadabra
> mmm
Ok:10: -1
Примечания: Выбор алгоритма поиска не важен

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

Общий алгоритм решения

В данной лабораторной есть две программы: родительская и дочерняя. Родительская программа обрабатывает данные с консоли и выводит все ответы пользователям. В зависимости от команды она отправляет различные сообщения через очередь сообщений **zeromq** своим дочерним программам. Информацию о всех вычислительных программах, привязанных к управляющей программе я храню в **vector,** и, когда мне нужно достучаться до определенной программы, я для каждого списка дочерних программ прокидываю сообщения с определенным текстом сообщения. Дочерняя программа получает родительские сообщения, и в зависимости от их содержания выполняет определенные действия. Наиболее интересным из всего является реализация функции exec. Нахождение подстроки в строке реализовано с помощью стандартной функции **find**. Так как поиск может быть достаточно долгим мы не должны блокировать процесс на все время поиска, поэтому сам поиск запускается в отдельном потоке, который с помощью функции **detach** отвязывается от основного потока (я не жду когда он выполнится). Это позволяет процессу дальше продолжать обрабатывать все сообщения и при необходимости прокидывать их дочерним вычислительным узлам. В функции нахождении подстроки в строке я усыпляю поток на 10 секунд для того, чтобы было нагляднее видно, что работа программы не нарушается и все команды продолжают корректно обрабатываться.

Реализация

child.cpp

```
#include "zmq.hpp"
#include <sstream>
#include <string>
#include <iostream>
#include <zconf.h>
#include <vector>
#include <signal.h>
#include <fstream>
#include <algorithm>
#include <thread>


using namespace std;

string adr, adrChild;
zmq::context_t context(1);
zmq::socket_t mainSocket(context, ZMQ_REQ);
```

```
zmq::context_t contextChild(1);
zmq::socket_t childSocket(contextChild, ZMQ_REP);
int idThisNode, childNodeId;

void sendMessage(const string& messageString, zmq::socket_t& socket) {
   zmq::message_t messageBack(messageString.size());
   memcpy(messageBack.data(), messageString.c_str(), messageString.size());

   if (!socket.send(messageBack)) {
         cerr << "Error: can't send message from node with pid " << getpid() <<
endl;
   }
}

void calculationSeparateThread(string receivedMessage, string idProcString, int
idProc) {
   cout << "Function started in thread: " << std::this_thread::get_id() << endl;
   sleep(10);

   int flag = 0;
   string text, pattern, returnMessage;
   vector<int> answer;

   for (int i = 6 + idProcString.size(); i < receivedMessage.size(); ++i) {
      if (receivedMessage[i] == ' ') {
         ++flag;
      } else if ((receivedMessage[i] != ' ') && (flag == 0)) {
         text += receivedMessage[i];
      } else if ((receivedMessage[i] != ' ') && (flag == 1)) {
         pattern += receivedMessage[i];
      }
   }

   if (text.size() >= pattern.size()) {
      int start = 0;
      while (text.find(pattern, start) != -1) {
         start = text.find(pattern, start);
         answer.push_back(start);
         ++start;
      }
   }

   if (answer.empty()) {
      returnMessage = "-1";
   } else {
```

```
            returnMessage = to_string(answer[0]);
            for (int i = 1; i < answer.size(); ++i) {
                returnMessage = returnMessage + ";" + to_string(answer[i]);
            }
        }

        cout << endl << "OK:" << to_string(idProc) + ":" + returnMessage << endl;
            cout << "Function completed in thread: " << std::this_thread::get_id() <<
endl;
    }

    void funcCreate(string receivedMessage) {
        bool isSpace = false;
        int idNewProc, parentIdNewProc;
        string idNewProcString, parentIdNewProcString;

        for (int i = 7; i < receivedMessage.size(); ++i) {
            if (receivedMessage[i] == ' ') {
                isSpace = true;
            } else if (receivedMessage[i] != ' ' && !isSpace) {
                idNewProcString += receivedMessage[i];
            } else if (receivedMessage[i] != ' ' && isSpace) {
                parentIdNewProcString += receivedMessage[i];
            }
        }

        idNewProc = stoi(idNewProcString);
        parentIdNewProc = stoi(parentIdNewProcString);

        if (idNewProc == idThisNode) {
            sendMessage("Error: Already exists", mainSocket);
        } else {

            if (childNodeId == 0 && parentIdNewProc == idThisNode) {

                childNodeId = idNewProc;
                childSocket.bind(adrChild + to_string(childNodeId));
                adrChild += to_string(childNodeId);

                char* adrChildTmp = new char[adrChild.size() + 1];
                memcpy(adrChildTmp, adrChild.c_str(), adrChild.size() + 1);
                char* childIdTmp = new char[to_string(childNodeId).size() + 1];
                                    memcpy(childIdTmp,  to_string(childNodeId).c_str(),
to_string(childNodeId).size() + 1);
                char* args[] = {"./child", adrChildTmp, childIdTmp, NULL};
```

```
                int procesId = fork();

                if (procesId == 0) {
                    execv("./child", args);
                } else if (procesId < 0) {
                    cerr << "Error in forking in node with pid: " << getpid() << endl;
                } else {
                    zmq::message_t messageFromNode;

                    if (!childSocket.recv(messageFromNode)) {
                        cerr << "Error: can't receive message from child node in node with
pid:" << getpid()
                             << endl;
                    }

                    if (!mainSocket.send(messageFromNode)) {
                        cerr << "Error: can't send message to main node from node with
pid:" << getpid() << endl;
                    }
                }

                delete[] adrChildTmp;
                delete[] childIdTmp;

            } else if (childNodeId == 0 && parentIdNewProc != idThisNode) {
                sendMessage("Error: there is no such parent", mainSocket);
            } else if (childNodeId != 0 && parentIdNewProc == idThisNode) {
                sendMessage("Error: this parent already has a child", mainSocket);
            } else {
                sendMessage(receivedMessage, childSocket);
                zmq::message_t message;
                if (!childSocket.recv(message)) {
                    cerr << "Error: can't receive message from child node in node with
pid: " << getpid() << endl;
                }
                if (!mainSocket.send(message)) {
                    cerr << "Error: can't send message to main node from node with pid: "
<< getpid() << endl;
                }
            }
        }
    }

    void funcExec(string receivedMessage) {
```

```
        int idProc;
        string idProcString;

        for (int i = 5; i < receivedMessage.size(); ++i) {
            if (receivedMessage[i] != ' ') {
                idProcString += receivedMessage[i];
            } else {
                break;
            }
        }

        idProc = stoi(idProcString);

        if (idProc == idThisNode) {

                    thread workThread(calculationSeparateThread, receivedMessage,
idProcString, idProc);

            workThread.detach();

                string returnMessage = "The child process performs calculations and
outputs them when it finishes calculations";
            sendMessage(returnMessage, mainSocket);

        } else {

            if (childNodeId == 0) {
                sendMessage("Error: id: Not found", mainSocket);
            } else {
                zmq::message_t message(receivedMessage.size());
                                    memcpy(message.data(), receivedMessage.c_str(),
receivedMessage.size());

                if (!childSocket.send(message)) {
                    cerr << "Error: can't send message to child node from node with pid: "
<< getpid() << endl;
                }
                if (!childSocket.recv(message)) {
                        cerr << "Error: can't receive message from child node in node with
pid: " << getpid() << endl;
                }
                if (!mainSocket.send(message)) {
                    cerr << "Error: can't send message to main node from node with pid: "
<< getpid() << endl;
                }
```

```
            }
        }
    }

    void funcPing(string receivedMessage) {
        int idProc;
        string idProcString;

        for (int i = 5; i < receivedMessage.size(); ++i) {
            if (receivedMessage[i] != ' ') {
                idProcString += receivedMessage[i];
            } else {
                break;
            }
        }

        idProc = stoi(idProcString);

        if (idProc == idThisNode) {
            sendMessage("OK: 1", mainSocket);
        } else {
            if (childNodeId == 0) {
                sendMessage("OK: 0", mainSocket);
            } else {
                zmq::message_t message(receivedMessage.size());
                                        memcpy(message.data(),  receivedMessage.c_str(),
receivedMessage.size());
                childSocket.send(message);
                childSocket.recv(message);
                mainSocket.send(message);
            }
        }
    }

    void funcKill(string receivedMessage) {
        int idProcToKill;
        string idProcToKillString;

        for (int i = 5; i < receivedMessage.size(); ++i) {
            if (receivedMessage[i] != ' ') {
                idProcToKillString += receivedMessage[i];
            } else {
                break;
            }
        }
```

```
        idProcToKill = stoi(idProcToKillString);

        if (childNodeId == 0) {
            sendMessage("Error: there isn`t node with this id child", mainSocket);
        } else {
            if (childNodeId == idProcToKill) {
                sendMessage("OK: " + to_string(childNodeId), mainSocket);
                sendMessage("DIE", childSocket);
                childSocket.unbind(adrChild);
                adrChild = "tcp://127.0.0.1:300";
                childNodeId = 0;
            } else {
                zmq::message_t message(receivedMessage.size());
                                        memcpy(message.data(),  receivedMessage.c_str(),
receivedMessage.size());
                childSocket.send(message);
                childSocket.recv(message);
                mainSocket.send(message);
            }
        }
    }

    int main(int argc, char* argv[]) {
        adr = argv[1];
        mainSocket.connect(argv[1]);

        sendMessage("OK: " + to_string(getpid()), mainSocket);
        idThisNode = stoi(argv[2]);
        childNodeId = 0;
        adrChild = "tcp://127.0.0.1:300";

        while (true) {

            zmq::message_t messageMain;
            mainSocket.recv(messageMain);
                        string  receivedMessage(static_cast<char*>(messageMain.data()),
messageMain.size());
            string command;

            for (char element: receivedMessage) {
                if (element != ' ') {
                    command += element;
                } else {
                    break;
```

```
            }
        }

        if (command == "exec") {
            funcExec(receivedMessage);
        } else if (command == "create") {
            funcCreate(receivedMessage);
        } else if (command == "ping") {
            funcPing(receivedMessage);
        } else if (command == "kill") {
            funcKill(receivedMessage);
        } else if (command == "DIE") {
            if (childNodeId != 0) {
                sendMessage("DIE", childSocket);
                childSocket.unbind(adrChild);
            }
            mainSocket.unbind(adr);
            return 0;
        }
    }
}
```

---

parent.cpp

```
#include "zmq.hpp"
#include <sstream>
#include <string>
#include <iostream>
#include <zconf.h>
#include <vector>
#include <signal.h>
#include <sstream>
#include <set>
#include <algorithm>

using namespace std;

zmq::context_t context(1);
string adr = "tcp://127.0.0.1:300";
string command;
vector<int> childesId;
vector<unique_ptr<zmq::socket_t>> sockets;

void createChildFromMainNode(int childId) {
    auto socket = std::make_unique<zmq::socket_t>(context, ZMQ_REP);
```

```
    socket->bind(adr + to_string(childId));
    string new_adr = adr + to_string(childId);

    char* adr_ = new char[new_adr.size() + 1];
    memcpy(adr_, new_adr.c_str(), new_adr.size() + 1);

    char* id_ = new char[to_string(childId).size() + 1];
    memcpy(id_, to_string(childId).c_str(), to_string(childId).size() + 1);

    char* args[] = {"./child", adr_, id_, NULL};

    int processId = fork();
    if (processId < 0) {
        cerr << "Unable to create first worker node" << endl;
        childId = 0;
        exit(1);
    } else if (processId == 0) {
        execv("./child", args);
    }

    childesId.push_back(childId);
    sockets.push_back(std::move(socket));

    zmq::message_t message;
    sockets[sockets.size() - 1]->recv(message);


    string receiveMessage(static_cast<char*>(message.data()), message.size());
    cout << receiveMessage << endl;

    delete[] adr_;
    delete[] id_;
}

void funcCreate() {
    int childId, parentId;
    cin >> childId >> parentId;

    if (childesId.empty()) {

        if (parentId != -1) {
            cerr << "There is no such parent node" << endl;
            return;
        }
```

```
            createChildFromMainNode(childId);

        } else {
            if (parentId == -1) {

                bool wasChild = false;
                    for (int indexInChildes = 0; indexInChildes < childesId.size(); +
+indexInChildes) {
                    if (childesId[indexInChildes] == childId) {
                        cout << "This id has already been created" << endl;
                        wasChild = true;
                        break;
                    }
                }
                if (wasChild) {
                    return;
                }

                createChildFromMainNode(childId);

            } else {
                    string messageString = command + " " + to_string(childId) + " " +
to_string(parentId);

                    for (int indexOfSockets{0}; indexOfSockets < sockets.size(); +
+indexOfSockets) {

                    zmq::message_t message(messageString.size());
                    memcpy(message.data(), messageString.c_str(), messageString.size());

                    sockets[indexOfSockets]->send(message);
                    sockets[indexOfSockets]->recv(message);
                            string receiveMessage(static_cast<char*>(message.data()),
message.size());

                    if (receiveMessage[0] == 'O' && receiveMessage[1] == 'K') {
                        cout << receiveMessage << endl;
                        break;
                    } else if (receiveMessage == "Error: Already exists") {
                        cout << receiveMessage << endl;
                        break;
                    } else if (receiveMessage == "Error: this parent already has a child") {
                        cout << receiveMessage << endl;
                        break;
```

```
                } else if (receiveMessage == "Error: there is no such parent" &&
                        indexOfSockets == sockets.size() - 1) {
                    cout << receiveMessage << endl;
                    break;
                }
            }
        }
    }

    void funcExec() {
        int id;
        string text, pattern;
        cin >> id;
        cin >> text >> pattern;

        string messageString = command + " " + to_string(id) + " " + text + " " +
pattern;

            for (int indexOfSockets{0}; indexOfSockets < sockets.size(); +
+indexOfSockets) {

            zmq::message_t message(messageString.size());
            memcpy(message.data(), messageString.c_str(), messageString.size());

            sockets[indexOfSockets]->send(message);
            sockets[indexOfSockets]->recv(message);
            string receiveMessage(static_cast<char*>(message.data()), message.size());

                if (receiveMessage[0] == 'T' && receiveMessage[1] == 'h' &&
receiveMessage[2] == 'e') {
                cout << receiveMessage << endl;
                break;
            } else if (receiveMessage == "Error: id: Not found" &&
                    indexOfSockets == sockets.size() - 1) {
                cout << receiveMessage << endl;
                break;
            }

        }

    }

    void funcPing() {
        int id;
```

```
        cin >> id;

        if (childesId.empty()) {
            cout << "OK: 0" << endl;
        } else {

            string messageString = command + " " + to_string(id);

                    for (int indexOfSockets{0}; indexOfSockets < sockets.size(); +
+indexOfSockets) {

                zmq::message_t message(messageString.size());
                memcpy(message.data(), messageString.c_str(), messageString.size());

                sockets[indexOfSockets]->send(message);
                sockets[indexOfSockets]->recv(message);
                            string receiveMessage(static_cast<char*>(message.data()),
message.size());

                if (receiveMessage == "OK: 1") {
                    cout << receiveMessage << endl;
                    break;
                } else if (receiveMessage == "OK: 0" &&
                        indexOfSockets == sockets.size() - 1) {
                    cout << receiveMessage << endl;
                    break;
                }

            }
        }
    }

    void funcKill() {
        int id;
        cin >> id;

        if (childesId.empty()) {
            cout << "Error: there isn't nodes" << endl;
        } else {

                    for (int indexOfSockets{0}; indexOfSockets < sockets.size(); +
+indexOfSockets) {

                if (childesId[indexOfSockets] == id) {
                    string killMessage = "DIE";
```

```
                zmq::message_t message(killMessage.size());
                memcpy(message.data(), killMessage.c_str(), killMessage.size());
                sockets[indexOfSockets]->send(message);

                                        sockets[indexOfSockets]->unbind(adr +
to_string(childesId[indexOfSockets]));

                childesId.erase(childesId.begin() + indexOfSockets);
                sockets.erase(sockets.begin() + indexOfSockets);

                cout << "Node deleted successfully" << endl;

                break;
            } else {
                string killMessage = command + " " + to_string(id);
                zmq::message_t message(killMessage.size());
                memcpy(message.data(), killMessage.c_str(), killMessage.size());

                sockets[indexOfSockets]->send(message);
                sockets[indexOfSockets]->recv(message);
                        string receiveMessage(static_cast<char*>(message.data()),
message.size());

                if (receiveMessage[0] == 'O' && receiveMessage[1] == 'K') {
                    cout << receiveMessage << endl;
                    break;
                } else if (receiveMessage == "Error: there isn`t node with this id" &&
                        indexOfSockets == sockets.size() - 1) {
                    cout << receiveMessage << endl;
                    break;
                }
            }
        }
    }

    void funcExit() {
                for (int indexOfSockets{0}; indexOfSockets < sockets.size(); +
+indexOfSockets) {

            if (childesId[indexOfSockets]) {
                string killMessage = "DIE";
                zmq::message_t message(killMessage.size());
                memcpy(message.data(), killMessage.c_str(), killMessage.size());
```

```
            sockets[indexOfSockets]->send(message);
        }
        sockets[indexOfSockets]->close();

    }

    cout << "All node was deleted" << endl;
    context.close();
    exit(0);
}

int main() {

    while (true) {

        cout << "command:";
        cin >> command;

        if (command == "create") {
            funcCreate();
        } else if (command == "exec") {
            funcExec();
        } else if (command == "ping") {
            funcPing();
        } else if (command == "kill") {
            funcKill();
        } else if (command == "exit") {
            funcExit();
        } else {
            cout << "Error: incorrect command" << endl;
        }

    }
}
```

Пример работы

Test 1.

```
root@882dead06576:/tmp/laba05-07# ./main
command:create 5 -1
OK: 12
command:create 4 5
```

OK: 17
command:create 3 4
OK: 22
command:create 2 3
OK: 27
command:create 6 -1
OK: 32
command:create 7 6
OK: 37
command:create 8 7
OK: 42
command:ping 8
OK: 1
command:ping 7
OK: 1
command:ping 7
OK: 1
command:kill 7
OK: 7
command:ping 7
OK: 0
command:ping 8
OK: 0
command:ping 6
OK: 1
command:exec 4
asdasdasdasascsacsacsac s
Function started in thread: 281473004138144
The child process performs calculations and outputs them when it finishes calculations
command:ping 2
OK: 1
command:ping 3
OK: 1
command:ping 4
OK: 1
command:
OK:4:1;4;7;10;12;14;17;20
Function completed in thread: 281473004138144
ping 5
OK: 1
command:kill 5
Node deleted successfully
command:ping 5
OK: 0

```
command:ping 4
OK: 0
command:ping 3
OK: 0
command:ping 2
OK: 0
command:create 5 -1
OK: 48
command:exit
All node was deleted
```

Вывод

В ходе данной работы я познакомился с очередями сообщений - еще одним способом обмениваться данными между процессами. Я использовал библиотеку **zeromq** для реализации данной лабораторной. Я столкнулся с определенными трудностями в момент ее установки, из-за чего мне пришлось запускать образ Ubuntu через docker. На мой взгляд это самая интересная и полезная из всех лабораторных работ. В ней я использовал большое количество технологий и знаний из предыдущих лабораторных работ.