

Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

**Лабораторная работа №6-8 по курсу
«Операционные системы»**

**Тема работы
“Очереди сообщений”**

Студент: Борисов Ян Артурович
Группа: М8О-208Б-20
Вариант: 30
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

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/Yannikupy/OS>

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.

Вариант 30. Команды:

create id

exec id text pattern (поиск подстроки)

pingall

Общие сведения о программе

Для выполнения данной лабораторной работы я предварительно реализовал 2 файла с кодом:

child_node.cpp (реализация дочерних процессов)

main_prog.cpp (реализация основного процесса)

Общий метод и алгоритм решения

В makefile у нас две команды:

all:

```
g++ -fsanitize=address main_prog.cpp -lzmq -o main_prog -w
```

```
g++ -fsanitize=address child_node.cpp -lzmq -o child_node -w
```

По сути, две работающие программы. В начале запускается main_prog, который уже в себе вызывает child_prog от конкретных значений child_id, parent_port и parent_id. После того, как main_prog получает сообщение, он отправляет его на child_node, который обрабатывает это сообщение.

Исходный код

main_prog.cpp

```
#include <zmq.hpp>
#include <unistd.h>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REP);
    string adr = "tcp://127.0.0.1:3000";
    string command;
    int child_id = 0;
    cout << "Commands:\ncreate id\nexec id text pattern\npingall\nkill\nid\nexit\n";
    while (1)
    {
        cout << "Please, enter command\n";
        cin >> command;
        if (command == "create")
        {
            if (child_id == 0)
            {
                int id;
                cin >> id;
                int id_tmp = id - 1;
                main_socket.bind(adr + to_string(++id_tmp));
                string new_adr = adr + to_string(id_tmp);
                char *adr_ = new char[new_adr.size() + 1];
                memcpy(adr_, new_adr.c_str(), new_adr.size() + 1);
                char *id_ = new char[to_string(id).size() + 1];
                memcpy(id_, to_string(id).c_str(),
                    to_string(id).size() + 1);
                char *args[] = { "./child_node", adr_, id_, NULL };
            }
        }
    }
}
```

```

        int id2 = fork();
        if (id2 == -1)
        {
            std::cout << "ERROR: CALCULATING NODE WAS NOT
CREATED\n";
            id = 0;
            exit(1);
        }
        else if (id2 == 0)
        {
            execv("./child_node", args);
        }
        else
        {
            child_id = id;
        }
        zmq::message_t message;
        main_socket.recv(&message);
        string recieved_message(static_cast<char
*>(message.data()), message.size());
        cout << recieved_message << "\n";
        delete[] adr_;
        delete[] id_;
    }
    else
    {
        int id;
        cin >> id;
        string message_string = command + " " + to_string(id);
        zmq::message_t message(message_string.size());
        memcpy(message.data(), message_string.c_str(),
message_string.size());
        main_socket.send(message);
        main_socket.recv(&message);
        string recieved_message(static_cast<char
*>(message.data()), message.size());
        cout << recieved_message << "\n";
    }
}
else if (command == "exec")
{
    int id;
    string text, pattern, s;
    cin >> id >> text >> pattern;
    s = text + "$" + pattern;
    string message_string = command + " " + to_string(id) + "
" + s;

    zmq::message_t message(message_string.size());
    memcpy(message.data(), message_string.c_str(),
message_string.size());
    main_socket.send(message);
    main_socket.recv(&message);
    string recieved_message(static_cast<char
*>(message.data()), message.size());
    cout << recieved_message << "\n";
}
else if (command == "pingall")

```

```

{
    if (child_id == 0)
    {
        cout << "No childs to ping!\n";
    }
    else
    {
        string message_string = command;
        zmq::message_t message(message_string.size());
        memcpy(message.data(), message_string.c_str(),
message_string.size());
        // receive answer from child
        //for (int j = 0; j < beat_amount; j++)
        //{
            main_socket.send(message);
            main_socket.recv(&message);
            string recieved_message(static_cast<char
*>(message.data()), message.size());
            if (recieved_message == "OK")
            {
                cout << "OK: -1\n";
            }
            else
            {
                cout << "OK:" << recieved_message;
            }
        }
    }
    else if (command == "kill")
    {
        int id;
        cin >> id;
        if (child_id == 0)
        {
            cout << "Error: there aren't any nodes\n";
        }
        else if (child_id == id)
        {
            string kill_message = "DIE";
            zmq::message_t message(kill_message.size());
            memcpy(message.data(), kill_message.c_str(),
kill_message.size());
            main_socket.send(message);
            cout << "Tree was deleted\n";
            child_id = 0;
        }
        else
        {
            string kill_message = command + " " + to_string(id);
            zmq::message_t message(kill_message.size());
            memcpy(message.data(), kill_message.c_str(),
kill_message.size());
            main_socket.send(message);
            main_socket.recv(&message);
            string received_message(static_cast<char
*>(message.data()), message.size());
            cout << received_message << "\n";
        }
    }
}

```

```

    }
}
else if (command == "exit")
{
    if (child_id)
    {
        string kill_message = "DIE";
        zmq::message_t message(kill_message.size());
        memcpy(message.data(), kill_message.c_str(),
kill_message.size());
        main_socket.send(message);
        cout << "Tree was deleted\n";
        child_id = 0;
    }
    main_socket.close();
    context.close();
    return 0;
}
else
{
    cout << "Error: incorrect command\n";
}
}
}

```

child_node.cpp

```

#include <zmq.hpp>
#include <iostream>
#include <unistd.h>
using namespace std;

void send_message(std::string message_string, zmq::socket_t &socket)
{
    zmq::message_t message_back(message_string.size());
    memcpy(message_back.data(), message_string.c_str(),
message_string.size());
    if (!socket.send(message_back))
    {
        std::cout << "Error: can't send message from node with pid "
<< getpid() << "\n";
    }
}

int main(int argc, char *argv[])
{
    string adr = argv[1];
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REQ);
    main_socket.connect(argv[1]);
    send_message("OK: " + to_string(getpid()), main_socket);
    int id = stoi(argv[2]);
    int left_id = 0;
    int right_id = 0;
    zmq::context_t context_l(1);
    zmq::context_t context_r(1);
    zmq::socket_t left_socket(context_l, ZMQ_REP);
    string adr_left = "tcp://127.0.0.1:300";
}

```

```

zmq::socket_t right_socket(context_r, ZMQ_REP);
string adr_right = "tcp://127.0.0.1:300";
while (1)
{
    zmq::message_t message_main;
    main_socket.recv(&message_main);
    string recieved_message(static_cast<char
*>(message_main.data()), message_main.size());
    string command;
    for (int i = 0; i < recieved_message.size(); ++i)
    {
        if (recieved_message[i] != ' ')
        {
            command += recieved_message[i];
        }
        else
        {
            break;
        }
    }
    if (command == "exec")
    { //-----EXEC-----
        int id_proc;
        string id_proc_, value_;
        string key;
        int pos;
        for (int i = 5; i < recieved_message.size(); ++i)
        {
            if (recieved_message[i] != ' ')
            {
                id_proc_ += recieved_message[i];
            }
            else
            {
                pos = i;
                break;
            }
        }
        id_proc = stoi(id_proc_);
        if (id_proc == id)
        { // id == proc_id, this node must execute
            int j = pos;
            string text, pattern;
            string res = "OK:" + to_string(id) + ":";
            while (recieved_message[j] != '$')
            {
                text.push_back(recieved_message[j]);
                ++j;
            }
            ++j;
            while (j != recieved_message.size())
            {
                pattern.push_back(recieved_message[j]); //tttt t
                ++j;
            }
            int flag = 0;
            if (pattern.size() <= text.size())

```



```

        {
            //cout << text << " " << pattern << "\n";
            for (int i = 0; i <= text.size() - pattern.size();
++i)
            {
                //cout << "TEST";
                string pat;
                for (int z = 0; z < pattern.size(); ++z)
                {
                    if (text[i + z] == pattern[z])
                        pat.push_back(text[i + z]);
                }
                if (pat == pattern)
                {
                    res = res + to_string(i) + ";";
                    flag = 1;
                }

                pat.clear();
            }
        }
        if (flag == 0)
        {
            res = res + "-1";
        }
        zmq::message_t message(res.size());
        memcpy(message.data(), res.c_str(), res.size());
        if (!main_socket.send(message))
        {
            cout << "Error: can't send message to main node
from node with pid: " << getpid() << "\n";
        }
    }
    else
    { // id != prod_id
        if (id > id_proc)
        { //go to left
            if (left_id == 0)
            { // if node not exists
                string message_string = "Error:id: Not found";
                zmq::message_t message(message_string.size());
                memcpy(message.data(), message_string.c_str(),
message_string.size());
                if (!main_socket.send(message))
                {
                    cout << "Error: can't send message to main
node from node with pid: " << getpid() << "\n";
                }
            }
            else
            {
                zmq::message_t
message(recieved_message.size());
                memcpy(message.data(),
recieved_message.c_str(), recieved_message.size());
                if (!left_socket.send(message))
                {

```

```

        cout << "Error: can't send message to left
node from node with pid: " << getpid() << "\n";
    }
    // catch and send to parent
    if (!left_socket.recv(&message))
    {
        cout << "Error: can't receive message from
left node in node with pid: " << getpid() << "\n";
    }
    if (!main_socket.send(message))
    {
        cout << "Error: can't send message to main
node from node with pid: " << getpid() << "\n";
    }
}
}
else
{ // go to right
    if (right_id == 0)
    { // if node not exists
        string message_string = "Error:id: Not found";
        zmq::message_t message(message_string.size());
        memcpy(message.data(), message_string.c_str(),
message_string.size());
        if (!main_socket.send(message))
        {
            cout << "Error: can't send message to main
node from node with pid: " << getpid() << "\n";
        }
    }
    else
    {
        zmq::message_t
message(recieved_message.size());
        memcpy(message.data(),
recieved_message.c_str(), recieved_message.size());
        if (!right_socket.send(message))
        {
            cout << "Error: can't send message to
right node from node with pid: " << getpid() << "\n";
        }
        // catch and send to parent
        if (!right_socket.recv(&message))
        {
            cout << "Error: can't receive message from
left node in node with pid: " << getpid() << "\n";
        }
        if (!main_socket.send(message))
        {
            cout << "Error: can't send message to main
node from node with pid: " << getpid() << "\n";
        }
    }
}
}
}
else if (command == "create")

```

```

{
    int id_proc; // id of node for creating
    string id_proc_;
    for (int i = 7; i < recieved_message.size(); ++i)
    {
        if (recieved_message[i] != ' ')
        {
            id_proc_ += recieved_message[i];
        }
        else
        {
            break;
        }
    }
    id_proc = stoi(id_proc_);
    if (id_proc == id)
    {
        send_message("Error: Already exists", main_socket);
    }
    else if (id_proc > id)
    {
        if (right_id == 0)
        { // there is not right node
            right_id = id_proc;
            int right_id_tmp = right_id - 1;
            while (1)
            {
                try
                {
                    right_socket.bind(adr_right +
to_string(++right_id_tmp));
                    break;
                }
                catch (...)
                {
                }
            }
            adr_right += to_string(right_id_tmp);
            char *adr_right_ = new char[adr_right.size() + 1];
            memcpy(adr_right_, adr_right.c_str(),
adr_right.size() + 1);
            char *right_id_ = new
char[to_string(right_id).size() + 1];
            memcpy(right_id_, to_string(right_id).c_str(),
to_string(right_id).size() + 1);
            char *args[] = {"./child_node", adr_right_,
right_id_, NULL};
            int f = fork();
            if (f == 0)
            {
                execv("./child_node", args);
            }
            else if (f == -1)
            {
                cout << "Error in forking in node with pid: "
<< getpid() << "\n";
            }
        }
    }
}

```

```

else
{
    zmq::message_t message_from_node;
    if (!right_socket.recv(&message_from_node))
    {
        cout << "Error: can't receive message from
right node in node with pid:" << getpid() << "\n";
    }
    string
recieved_message_from_node(static_cast<char
*>(message_from_node.data()), message_from_node.size());
    if (!main_socket.send(message_from_node))
    {
        cout << "Error: can't send message to main
node from node with pid:" << getpid() << "\n";
    }
    delete[] adr_right_;
    delete[] right_id_;
}
else
{ // send task to right node
    send_message(recieved_message, right_socket);
    // catch and send to parent
    zmq::message_t message;
    if (!right_socket.recv(&message))
    {
        cout << "Error: can't receive message from
left node in node with pid: " << getpid() << "\n";
    }
    if (!main_socket.send(message))
    {
        cout << "Error: can't send message to main
node from node with pid: " << getpid() << "\n";
    }
}
}
else
{
    if (left_id == 0)
    { // there is not left node
        left_id = id_proc;
        int left_id_tmp = left_id - 1;
        while (1)
        {
            try
            {
                left_socket.bind(adr_left +
to_string(++left_id_tmp));
                break;
            }
            catch (...)
            {
            }
        }
        adr_left += to_string(left_id_tmp);
        char *adr_left_ = new char[adr_left.size() + 1];

```

```

        memcpy(adr_left_, adr_left.c_str(),
adr_left.size() + 1);
        char *left_id_ = new
char[to_string(left_id).size() + 1];
        memcpy(left_id_, to_string(left_id).c_str(),
to_string(left_id).size() + 1);
        char *args[] = {"../child_node", adr_left_,
left_id_, NULL};

        int f = fork();
        if (f == 0)
        {
            execv("./child_node", args);
        }
        else if (f == -1)
        {
            cout << "Error in forking in node with pid: "
<< getpid() << "\n";
        }
        else
        {
            // catch message from new node
            zmq::message_t message_from_node;
            if (!left_socket.recv(&message_from_node))
            {
                cout << "Error: can't receive message from
left node in node with pid:" << getpid() << "\n";
            }
            string
recieved_message_from_node(static_cast<char
*>(message_from_node.data()), message_from_node.size());
            // send message to main node
            if (!main_socket.send(message_from_node))
            {
                cout << "Error: can't send message to main
node from node with pid:" << getpid() << "\n";
            }
        }
        delete[] adr_left_;
        delete[] left_id_;
    }
    else
    { // send task to left node
        send_message(recieved_message, left_socket);
        // catch and send to parent
        zmq::message_t message;
        if (!left_socket.recv(&message))
        {
            cout << "Error: can't receive message from
left node in node with pid: " << getpid() << "\n";
        }
        if (!main_socket.send(message))
        {
            cout << "Error: can't send message to main
node from node with pid: " << getpid() << "\n";
        }
    }
}
}

```

```

    }
    else if (command == "pingall")
    {
        if (left_id != 0)
            send_message(recieved_message, left_socket);
        if (right_id != 0)
            send_message(recieved_message, right_socket);
        string l = "OK";
        string r = "OK";
        if (left_id != 0)
        {
            zmq::message_t message_left;
            if (left_socket.recv(&message_left))
            {
                string left(static_cast<char
*>(message_left.data()), message_left.size());
                l = left;
            }
            else
            {
                l = to_string(left_id);
            }
        }
        if (right_id != 0)
        {
            zmq::message_t message_right;
            if (right_socket.recv(&message_right))
            {
                string right(static_cast<char
*>(message_right.data()), message_right.size());
                r = right;
            }
            else
            {
                r = to_string(right_id);
            }
        }
        if (l == r && l == "OK")
        {
            send_message("OK", main_socket);
        }
        else
        {
            if (l != "OK" && r != "OK")
            {
                send_message(l + "; " + r + "; ", main_socket);
            }
            else if (l != "OK")
            {
                send_message(l + "; ", main_socket);
            }
            else
            {
                send_message(r + "; ", main_socket);
            }
        }
    }
}

```

```

else if (command == "kill")
{
    int id_proc;
    string id_proc_;
    for (int i = 5; i < recieved_message.size(); ++i)
    {
        if (recieved_message[i] != ' ')
        {
            id_proc_ += recieved_message[i];
        }
        else
        {
            break;
        }
    }
    id_proc = stoi(id_proc_);
    if (id_proc > id)
    {
        if (right_id == 0)
        {
            send_message("Error: there isn't node with this
id", main_socket);
        }
        else
        {
            if (right_id == id_proc)
            {
                send_message("Ok: " + to_string(right_id),
main_socket);

                send_message("DIE", right_socket);
                right_socket.unbind(adr_right);
                adr_right = "tcp://127.0.0.1:300";
                right_id = 0;
            }
            else
            {
                right_socket.send(message_main);
                zmq::message_t message;
                right_socket.recv(&message);
                main_socket.send(message);
            }
        }
    }
    else if (id_proc < id)
    {
        if (left_id == 0)
        {
            send_message("Error: there isn't node with this
id", main_socket);
        }
        else
        {
            if (left_id == id_proc)
            {
                send_message("OK: " + to_string(left_id),
main_socket);

                send_message("DIE", left_socket);

```

```

        left_socket.unbind(adr_left);
        adr_left = "tcp://127.0.0.1:300";
        left_id = 0;
    }
    else
    {
        left_socket.send(message_main);
        zmq::message_t message;
        left_socket.recv(&message);
        main_socket.send(message);
    }
}
}
else if (command == "DIE")
{
    if (left_id)
    {
        send_message("DIE", left_socket);
        left_socket.unbind(adr_left);
        adr_left = "tcp://127.0.0.1:300";
        left_id = 0;
    }
    if (right_id)
    {
        send_message("DIE", right_socket);
        right_socket.unbind(adr_right);
        adr_right = "tcp://127.0.0.1:300";
        right_id = 0;
    }
    main_socket.unbind(adr);
    return 0;
}
}
}

```


Демонстрация работы программы

```
g++ -fsanitize=address main_prog.cpp -lzmq -o main_prog -w
```

```
g++ -fsanitize=address child_node.cpp -lzmq -o child_node -w
```

```
yanniku@DESKTOP-US1A6DR:/mnt/c/Users/boris/CLionProjects/OCs/OS/os_lab6-8/src$  
./main_prog
```

Commands:

create id

exec id text pattern

pingall

kill id

exit

Please, enter command

create 2

OK: 2373

Please, enter command

create 5

OK: 2380

Please, enter command

create 1

OK: 2387

Please, enter command

exec 1 manofman man

OK:1:1;6;

Please, enter command

pingall

OK: -1

Please, enter command

kill 1

OK: 1

Please, enter command

pingall

OK: -1
Please, enter command
create 6
OK: 2395
Please, enter command
pingall
OK: -1
Please, enter command
kill 5
Ok: 5
Please, enter command
pingall
OK: -1
Please, enter command
kill 2
Tree was deleted
Please, enter command
pingall
No childs to ping!
Please, enter command
exit

Выводы

Данная лабораторная работа была очень сложной для меня. Во время ее выполнения я полностью осознал концепцию очередей сообщений на основе zero message queue, научился читать документацию.