

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»**

Студент: Ивченко Анна Владимировна
Группа: М8О-208Б-20
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/Anetta123/OS/tree/main/lab6-8/src>

Постановка задачи

Реализовать распределенную систему по обработке запросов. В данной 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 сервера сообщений zmq. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом.

Вариант задания:

36. Топология — бинарное дерево. Тип вычислительной команды — сумма n чисел. Тип проверки узлов на доступность — пинг всех узлов.

Общие сведения о программе:

Программа состоит из двух файлов, которые компилируются в исполнительные файлы(которые представляют управляющий и вычислительные узлы). Общение между процессами происходит с помощью библиотеки zmq.

Общий метод и алгоритм решения:

- Управляющий узел принимает команды, обрабатывает их и пересылает дочерним узлам(или выводит сообщение об ошибке).
- Дочерние узлы проверяют, может ли быть команда выполнена в данном узле, если нет, то команда пересылается в один из дочерних узлов, из которого возвращается некоторое сообщение(об успехе или об ошибке), которое потом пересылается обратно по дереву.
- Для корректной проверки на доступность узлов, используется дерево, эмулирующее поведение узлов в данной топологии(например, при удалении узла, удаляются все его потомки).
- Если узел недоступен, то по истечении таймаута будет сгенерировано сообщение о недоступности узла и оно будет передано вверх по дереву, к управляющему узлу.

При удалении узла, все его потомки рекурсивно уничтожаются.

Исходный код:

Main_prog.cpp

```
#include "zmq.hpp"
```

```

#include <sstream>
#include <string>
#include <iostream>
#include <zconf.h>
#include <vector>
#include <signal.h>
#include <sstream>
#include <set>
#include <algorithm>
// g++ main_prog.cpp -lzmq -o main_prog -w
using namespace std;
int main(){
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REP);
    string adr = "tcp://127.0.0.1:500";
    string command;
    int child_id = 0;
    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;
                while(1){
                    try {
                        main_socket.bind(adr + to_string(++id_tmp));
                        break;
                    }
                    catch(...) { }
                }
                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 << "Unable to create first worker node\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);

    // catch message from new node
    main_socket.recv(&message);
    string recieved_message(static_cast<char*>(message.data()), message.size());
    cout << recieved_message << "\n";
}
} else if(command == "exec"){
    int id, value;
    string name;
    cin >> id >> name;
    string s;
    char q;
    while((q = getchar()) != '\n'){
        s += q;
    }
    if(s == ""){
        string message_string = command + " " + to_string(id) + " " + name;
        zmq::message_t message(message_string.size());
        memcpy(message.data(), message_string.c_str(), message_string.size());
        main_socket.send(message);
    }
}

```

```

// return value from map
main_socket.recv(&message);
string recieved_message(static_cast<char*>(message.data()), message.size());
cout << recieved_message << "\n";
} else {
    value = stoi(s);
    string message_string = command + " " + to_string(id) + " " + name + " " + to_string(value);
    zmq::message_t message(message_string.size());
    memcpy(message.data(), message_string.c_str(), message_string.size());
    main_socket.send(message);
    // add new element to map
    main_socket.recv(&message);
    string recieved_message(static_cast<char*>(message.data()), message.size());
    cout << recieved_message << "\n";
}
} else if(command == "ping"){
    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);
    // receive answer from child
    main_socket.recv(&message);
    string recieved_message(static_cast<char*>(message.data()), message.size());
    cout << recieved_message << "\n";

} else if(command == "kill"){
    int id;
    cin >> id;
    if(child_id == 0){
        cout << "Error: there isn't nodes\n";
    } else if(child_id == id){
        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";
        cout << "Tree deleted successfully\n";
    }
}

```

```

        return 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";
    }
    main_socket.close();
    context.close();
    break;
} else {
    cout << "Error: incorrect command\n";
}
}
}
}

```

Child_node2.cpp

```

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

```

```

// g++ child_node_2.cpp -lzmq -o child_node -w

```

```
using namespace std;
```

```
void send_message(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))
    {
        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]); // id of this node
    map<string, int> m;
    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://127.0.0.1:500";
    zmq::socket_t right_socket(context_r, ZMQ_REP);
    string adr_right = "tcp://127.0.0.1:500";
    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"){
    int id_proc; // id of node for adding
    string id_proc_, value_;
    string key;
    int value;
    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){ // id == proc_id
        for(int i = 6 + id_proc_.size(); i < recieved_message.size(); ++i){
            if(recieved_message[i] != ' '){
                key += recieved_message[i];
            } else {
                break;
            }
        }
        for(int i = 7 + id_proc_.size() + key.size(); i < recieved_message.size(); ++i){
            if(recieved_message[i] != ' ' || recieved_message[i] != '\n'){
                value_ += recieved_message[i];
            } else {
                break;
            }
        }
        if(value_ == ""){
            if(m.count(key)){
                int value_map = m[key];
                send_message("OK:" + id_proc_ + ":" + to_string(m[key]), main_socket);
            } else {
                cout << key;
                send_message("OK:" + id_proc_ + ":" + key + "' not found", main_socket);
            }
        } else {
            m[key] = stoi(value_);
            send_message("OK:" + id_proc_, main_socket);
        }
    } else {
        if(id > id_proc){

```

```

if(left_id == 0){ // if node not exists
    string message_string = "Error:id: Not found";
    send_message("Error:id: Not found", main_socket);
} else {
    zmq::message_t message(ricieved_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 {
    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(ricieved_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 {
            // catch message from new node
            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 == "ping") {

    int id_proc; // id of node for creating
    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){
        send_message("OK: 1", main_socket);
    }
}

```

```

} else if(id_proc < id) {
    if(left_id == 0){
        send_message("OK: 0", main_socket);
    } else {
        left_socket.send(message_main);
        zmq::message_t answ;
        left_socket.recv(&answ);
        main_socket.send(answ);
    }
} else if(id_proc > id) {
    if(right_id == 0){
        send_message("OK: 0", main_socket);
    } else {
        right_socket.send(message_main);
        zmq::message_t answ;
        right_socket.recv(&answ);
        main_socket.send(answ);
    }
}
} else if(command == "kill") {

    int id_proc; // id of node for killing
    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://127.0.0.1:500";
                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://127.0.0.1:500";
            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://127.0.0.1:500";
        left_id = 0;
    }
    if (right_id){
        send_message("DIE", right_socket);
        right_socket.unbind(adr_right);
        adr_right = "tcp://127.0.0.1:500";
        right_id = 0;
    }
    main_socket.unbind(adr);
    return 0;
}
}
}

```

Демонстрация работы программы

Please, enter command

create 9

OK: 3408

Please, enter command

create 7

OK: 3415

Please, enter command

create 1

OK: 3422

Please, enter command

create 12

OK: 3429

Please, enter command

create 10

OK: 3436

Please, enter command

create 20

OK: 3443

Please, enter command

exec 10 MyVar

OK:10: 'MyVar' not found

Please, enter command

exec 10 MyVar 5

OK:10

Please, enter command

exec 12 MyVar

OK:12: 'MyVar' not found

Please, enter command

exec 10 MyVar

OK:10:5

Please, enter command

exec 10 MyVar

OK:10:5

Please, enter command

ping 10

OK: 1

Please, enter command

ping 17

OK: 0

Please, enter command

exit

Tree was deleted

Выводы

Данная лабораторная работа оказалась очень сложной, но очень интересной. В ней сразу применяем знания, полученные в ходе выполнения предыдущих лабораторных работ, так как здесь и многопоточность, и межпроцессорное взаимодействие, и синхронизация потоков. А помимо всего этого также разобрались с дополнительной библиотекой (zmq).