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: Ханнанов Руслан Маратович
Группа: М8О-208Б-20
Вариант: 4
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021

Содержание

Репозиторий

https://github.com/Naksen/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 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 4. Команды:
create id [parent_id] – создать новый узел [id], родителем которого является узел [parent_id]. Если [parent_id] = - 1, то родительский узел – управляющий.
exec id n n1 n2… ni (набор чисел, требуется посчитать сумму)
ping id – проверка доступности узла
exit – выход из программы
kill id – удалить узел

Общие сведения о программе

Для выполнения данной лабораторной работы я предварительно реализовал 6 файлов с кодом:

topology.h – реализация топологии, в соответствии с вариантом – список из списков.

zmq_function.h - отдельный файл для функций zero-message queue, сделанный для удобства работы и во избежание загрязнения кода.

control.cpp - реализация программы клиента.

counting.cpp - реализация программы сервера.

Общий метод и алгоритм решения

Makefile:

```
all: control counting
control:
	g++ control.cpp -lzmq –o control -Wall -pedantic
counting:
	g++ control.cpp -lzmq –o counting -Wall -pedantic
clean:
	rm -rf control counting
```

В программе используется тип соединения Request-Response. Узлы передают информацию друг другу при помощи очереди сообщений. Все сообщений имеют следующий вид:
[id узла, которому предназначено сообщение] [команда] [аргументы]

Управляющий узел хранит структуру список списков, в которую записывает id существующих узлов. При помощи этой структуры он определяет, в какой список нужно направить сообщение.

Вычислительный узел, получи сообщение, сравнивает свой id и id из сообщения. Если они совпадают, то узел начинает обрабатывать запрос, в противном случае узел направляет это же сообщение своему ребенку и ждет от него ответа.

Для удобства функции отправки и получения сообщений, а также функции для подключения к сокетам вынесены в отдельный заголовочный файл, который подключается к программам узлов.

Исходный код

Topology.h

```
#include <list>
#include <stdexcept>

class topology {
private:
    std::list<std::list<int>> container;

public:
    void insert(int id, int parent_id) {
        if (parent_id == -1) {
            std::list<int> new_list;
            new_list.push_back(id);
            container.push_back(new_list);
        }
        else {
            int list_id = find(parent_id);
            if (list_id == -1) {
                throw std::runtime_error("Wrong parent id");
            }
            auto it1 = container.begin();
            std::advance(it1, list_id);
            for (auto it2 = it1->begin(); it2 != it1->end();
++it2) {
                if (*it2 == parent_id) {
                    it1->insert(++it2, id);
                    return;
```

```
                }
            }
        }
    }

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

    void erase(int id) {
        int list_id = find(id);
        if (list_id == -1) {
            throw std::runtime_error("Wrong id");
        }
        auto it1 = container.begin();
        std::advance(it1, list_id);
        for (auto it2 = it1->begin(); it2 != it1->end(); ++it2)
{
            if (*it2 == id) {
                it1->erase(it2, it1->end());
                if (it1->empty()) {
                    container.erase(it1);
                }
                return;
            }
        }
    }

    int get_first_id(int list_id) {
```

```
        auto it1 = container.begin();
        std::advance(it1, list_id);
        if (it1->begin() == it1->end()) {
            return -1;
        }
        return *(it1->begin());
    }

    friend std::ostream &operator<<(std::ostream &os, const
topology &topology) {
    for (auto &external : topology.container) {
      os << "{";
      for (auto &internal : external) {
        os << internal << " ";
      }
      os << "}" << std::endl;
    }
    return os;
    }
};
```

Zmq_functions.h

```
#include <zmq.hpp>
#include <iostream>

const int MAIN_PORT = 4040;

void send_message(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);
}

std::string receive_message(zmq::socket_t& socket) {
    zmq::message_t message;
    int chars_read;
    try {
```

```
            chars_read = (int)socket.recv(&message);
        }
        catch (...) {
            chars_read = 0;
        }
        if (chars_read == 0) {
            return "Error: node is unavailable [zmq_func]";
        }
        std::string
received_msg(static_cast<char*>(message.data()),
message.size());
        return received_msg;
}

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);
}

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);
}
```

control.cpp

```
#include <unistd.h>
#include <sstream>
#include <set>

#include "zmq_functions.h"
#include "topology.h"

int main() {
    topology network;
    std::vector<zmq::socket_t> branches;
    zmq::context_t context;

    std::string cmd;
    while (std::cin >> cmd) {

        if (cmd == "create") {
            int node_id, parent_id;
            std::cin >> node_id >> parent_id;

            if (network.find(node_id) != -1) {
                std::cout << "Error: already exists" <<
std::endl;
            }
            else if (parent_id == -1) {
                pid_t pid = fork();
                if (pid < 0) {
                    perror("Can't create new process");
                    return -1;
                }
                if (pid == 0) {
                    execl("./counting", "./counting",
std::to_string(node_id).c_str(), NULL);
                    perror("Can't execute new process");
                    return -2;
                }

                branches.emplace_back(context, ZMQ_REQ);
```

```
                branches[branches.size() -
1].setsockopt(ZMQ_SNDTIMEO, 5000);
                bind(branches[branches.size() - 1], node_id);
                send_message(branches[branches.size() - 1],
std::to_string(node_id) + "pid");

                std::string reply =
receive_message(branches[branches.size() - 1]);
                std::cout << reply << std::endl;
                network.insert(node_id, parent_id);
            }
            else if (network.find(parent_id) == -1) {
                std::cout << "Error: parent not found" <<
std::endl;
            }
            else {
                int branch = network.find(parent_id);
                send_message(branches[branch],
std::to_string(parent_id) + "create " +
std::to_string(node_id));

                std::string reply =
receive_message(branches[branch]);
                std::cout << reply << std::endl;
                network.insert(node_id, parent_id);
            }
        }
        else if (cmd == "exec") {
            int dest_id;
            int n;
            std::cin >> dest_id >> n;
            std::vector<int> v(n);
            std::string s;
            int branch = network.find(dest_id);
            for (int i = 0; i < n; ++i) {
                std::cin >> v[i];
                s = s + std::to_string(v[i]) + ' ';
            }
            if (branch == -1) {
```

```
                std::cout << "ERROR: incorrect node id" <<
std::endl;
            }
            else {
                send_message(branches[branch],
std::to_string(dest_id) + "exec " + std::to_string(n) + ' ' +
s);

                std::string reply =
receive_message(branches[branch]);
                std::cout << reply << std::endl;
            }

        }
        else if (cmd == "kill") {
            int id;
            std::cin >> id;
            int branch = network.find(id);
            if (branch == -1) {
                std::cout << "ERROR: incorrect node id" <<
std::endl;
            }
            else {
                bool is_first = (network.get_first_id(branch)
== id);
                send_message(branches[branch],
std::to_string(id) + " kill");

                std::string reply =
receive_message(branches[branch]);
                std::cout << reply << std::endl;
                network.erase(id);
                if (is_first) {
                    unbind(branches[branch], id);
                    branches.erase(branches.begin() + branch);
                }
            }
        }
        else if (cmd == "print") {
            std::cout << network;
```

```
        }
        else if (cmd == "pingall") {
            std::set<int> available_nodes;
            for (size_t i = 0; i < branches.size(); ++i) {
                int first_node_id = network.get_first_id(i);
                //std::cout << first_node_id << std::endl;
                send_message(branches[i],
std::to_string(first_node_id) + " pingall");

                std::string received_message =
receive_message(branches[i]);
                std::istringstream reply(received_message);
                int node;
                while(reply >> node) {
                    available_nodes.insert(node);
                }
            }
            std::cout << "OK: ";
            if (available_nodes.empty()) {
                std::cout << "no available nodes" << std::endl;
            }
            else {
                for (auto v : available_nodes) {
                    std::cout << v << " ";
                }
                std::cout << std::endl;
            }
        }
        else if (cmd == "ping") {
            int id;
            std::cin >> id;
            int branch = network.find(id);
            if (branch == -1) {
                std::cout << "Error: Not found" << std::endl;
                continue;
            }
            send_message(branches[branch], std::to_string(id) +
" ping");
```

```
            std::string reply =
receive_message(branches[branch]);
            std::cout << reply << std::endl;
        }
        else if (cmd == "exit") {
            for (size_t i = 0; i < branches.size(); ++i) {
                int first_node_id = network.get_first_id(i);
                send_message(branches[i],
std::to_string(first_node_id) + " kill");

                std::string reply =
receive_message(branches[i]);
                if (reply != "OK") {
                    std::cout << reply << std::endl;
                }
                else {
                    unbind(branches[i], first_node_id);
                }
            }
            exit(0);
        }
        else {
            std::cout << "Incorrect cmd" << std::endl;
        }
    }
}
```

Counting.cpp

```
#include <unordered_map>
#include <unistd.h>
#include <sstream>
#include <unordered_map>

#include "zmq_functions.h"

int main(int argc, char* argv[]) {
    if (argc != 2 && argc != 3) {
```

```
        throw std::runtime_error("Wrong args for counting
node");
    }
    int cur_id = std::atoi(argv[1]);
    int child_id = -1;
    if (argc == 3) {
        child_id = std::atoi(argv[2]);
    }

    //std::cout << child_id << std::endl;

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

    zmq::context_t context;
    zmq::socket_t parent_socket(context, ZMQ_REP);
    connect(parent_socket, cur_id);

    zmq::socket_t child_socket(context, ZMQ_REQ);
    child_socket.setsockopt(ZMQ_SNDTIMEO, 5000);
    if (child_id != -1) {
        bind(child_socket, child_id);
    }

    std::string message;
    while (true) {
        message = receive_message(parent_socket);
        std::istringstream request(message);
        int dest_id;
        request >> dest_id;

        std::string cmd;
        request >> cmd;

        if (dest_id == cur_id) {

            if (cmd == "pid") {
                send_message(parent_socket, "OK: " +
std::to_string(getpid()));
            }
```

```
            else if (cmd == "create") {
                int new_child_id;
                request >> new_child_id;

                if (child_id != -1) {
                    unbind(child_socket, child_id);
                }
                bind(child_socket, new_child_id);
                pid_t pid = fork();
                if (pid < 0) {
                    perror("Can't create new process");
                    return -1;
                }
                if (pid == 0) {
                    execl("./counting", "./counting",
std::to_string(new_child_id).c_str(),
std::to_string(child_id).c_str(), NULL);
                    perror("Can't execute new process");
                    return -2;
                }
                send_message(child_socket,
std::to_string(new_child_id) + "pid");
                child_id = new_child_id;
                send_message(parent_socket,
receive_message(child_socket));
            }

            else if (cmd == "exec") {
                int n;
                request >> n;
                int sum = 0;
                int x;
                for (int i = 0; i < n; ++i) {
                    request >> x;
                    sum += x;
                }
                send_message(parent_socket, "OK: " +
std::to_string(sum));
            }
```

```
            else if (cmd == "pingall") {
                std::string reply;
                if (child_id != -1) {
                    send_message(child_socket,
std::to_string(child_id) + " pingall");
                    std::string msg =
receive_message(child_socket);
                    reply += " " + msg;
                }
                send_message(parent_socket,
std::to_string(cur_id) + reply);
            }

            else if (cmd == "ping") {
                send_message(parent_socket, "OK 1");
            }

            else if (cmd == "kill") {
                //std::cout << child_id << std::endl;
                if (child_id != -1) {
                    send_message(child_socket,
std::to_string(child_id) + " kill");
                    std::string msg =
receive_message(child_socket);
                    if (msg == "OK") {
                        send_message(parent_socket, "OK");
                    }
                    unbind(child_socket, child_id);
                    disconnect(parent_socket, cur_id);
                    break;
                }
                send_message(parent_socket, "OK");
                disconnect(parent_socket, cur_id);
                break;
            }
        }
        else if (child_id != -1) {
            //std::cout << dest_id << " " << message <<
std::endl;
            send_message(child_socket, message);
```

```
            send_message(parent_socket,
receive_message(child_socket));
            if (child_id == dest_id && cmd == "kill") {
                std::cout << "SUCCESS" << child_id <<
std::endl;
                child_id = -1;
            }
        }
        else {
            send_message(parent_socket, "Error: node is
unavailable");
        }
    }
}
```

Демонстрация работы программы

naksan@LAPTOP-TL9L61MA:~/cprog/my_6-8_lab$ ./control

create 5 -1

OK: 31463

create 10 -1

OK: 31490

create 4 -1

OK: 31527

print

{5 }

{10 }

{4 }

ping 10

OK 1

create 3 4

OK: 31627

print

{5 }

{10 }

{4 3 }

exec 3 3 1 2 3

OK: 6

kill 3

OK

exit

Выводы

Данная лабораторная работа была направлена на изучение технологии очереди сообщений, на основе которой необходимо было построить сеть с заданной топологией.

Наряду с каналами и отображаемыми файлами, очереди сообщений являются достаточно удобным способом для взаимодействия между процессами. ZeroMQ предоставляет достаточно просто интерфейс для передачи сообщений, а также поддерживает все возможные типы соединений.

Полученные мной навыки работы с очередями сообщений можно использовать при проектировании мессенджеров, многопользовательских игр, да и вообще для любых мультипроцессорных программ.