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: Знай Артемий Олегович
Группа: М8О-201Б-21
Вариант: 3
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/znako/OS_LABS/

Постановка задачи

Цель работы

Целью является приобретение практических навыков в:

- Управлении серверами сообщений (№6)
- Применение отложенных вычислений (№7)
- Интеграция программных систем друг с другом (№8)

Задание

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.

Вариант 3: топология – 1, команда для вычислительных узлов - exec id n k0..kn, проверка доступности узлов – heartbeat time

Общие сведения о программе

Для работы с очередями используется ZMQ, программа собирается при помощи Makefile. Управляющий узел – server, вычислительные узлы – client.

В программе используются следующие системные вызовы:

1. **kill** – убивает процесс с pid – первый аргумент и посылает сигнал – второй аргумент.
2. **socket.setsockopt** – устанавливает флаги для сокета.
3. **zmq::context_t –** создает ZMQ контекст.
4. **zmq::socket_t –** создает ZMQ сокет.
5. **zmq::message_t** – создает ZMQ сообщение.
6. **socket.send** – отправляет ZMQ сообщение на socket.
7. **socket.bind** – принимает соединие к сокету.

8. **execv –** выполняет указанный файл.
9. **fork** – создает копию процесса.

Общий метод и алгоритм решения.

Создаем сервер – исполняющий узел, дальше делаем fork, в дочернем процессе при помощи execv запускаем client, а с родителя с сервера отсылаем сообщение, внутри клиента также создаются сокеты – левый и правый и на них отправляются сообщения с родителя, а родитель получает сообщения от детей и так по всему дереву. Исполняющий узел получает сообщение выполняет команду и отправляет ответ вверх по дереву до управляющего узла.

Основные файлы программы

control.cpp:

```
#include <unistd.h>
#include <sstream>
#include <set>
#include <thread>
#include "zmq_functions.h"
#include "topology.h"

void Heartbeat(int time, std::vector<zmq::socket_t>& branches, topology& network){
    while(1)
    {
        std::set<int> availableNodes;

        for (size_t i = 0; i < branches.size(); ++i) {
            int firstNodeId = network.get_first_id(i);
            send_message(branches[i], std::to_string(firstNodeId) + " heartbeat");

            std::string receivedMessage = receive_message(branches[i]);
            std::istringstream reply(receivedMessage);
            int node;
            while(reply >> node) {
                availableNodes.insert(node);
            }
        }
        std::cout << "OK: ";
        if (availableNodes.empty()) {
            std::cout << "No available nodes" << std::endl;
        }
```

```
        else {
            for (auto v : availableNodes) {
                std::cout << v << " ";
            }
            std::cout << std::endl;
        }
        sleep(time);
    }
}

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 nodeId, parentId;
            std::cin >> nodeId >> parentId;

            if (network.find(nodeId) != -1) {
                std::cout << "Error: already exists" << std::endl;
            }
            else if (parentId == -1)
            {
                pid_t pid = fork();
                if (pid < 0) {
                    perror("Can't create new process");
                    return -1;
                }
                if (pid == 0) {
                    if (execl("./counting", "./counting",
std::to_string(nodeId).c_str(), NULL) < 0) {
                        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], nodeId);
                send_message(branches[branches.size() - 1], std::to_string(nodeId) +
"pid");

                std::string reply = receive_message(branches[branches.size() - 1]);
```

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

                std::string reply = receive_message(branches[branch]);
                std::cout << reply << std::endl;
                network.insert(nodeId, parentId);
            }
        }
        else if (cmd == "exec") {
            int destId;
            std::string numbers;
            std::cin >> destId;
            std::getline(std::cin, numbers);
            int branch = network.find(destId);
            if (branch == -1) {
                std::cout << "ERROR: incorrect node id" << std::endl;
            }
            else {
                send_message(branches[branch], std::to_string(destId) + "exec " +
numbers);
                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 isFirst = (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 (isFirst) {
```

```
                    unbind(branches[branch], id);
                    branches.erase(branches.begin() + branch);
                }
            }
        }
        else if (cmd == "heartbeat") {
            int time;
            std::cin >> time;

            std::thread thr(Heartbeat, time, std::ref(branches), std::ref(network));
            thr.detach();
        }
        else if (cmd == "exit") {
            for (size_t i = 0; i < branches.size(); ++i) {
                int firstNodeId = network.get_first_id(i);
                send_message(branches[i], std::to_string(firstNodeId) + " kill");

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

counting.cpp:

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

#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 curId = std::atoi(argv[1]);
    int childId = -1;
    if (argc == 3) {
        childId = std::atoi(argv[2]);
```

```
    }

    zmq::context_t context;
    zmq::socket_t parentSocket(context, ZMQ_REP);
    connect(parentSocket, curId);

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

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

        std::string cmd;
        request >> cmd;

        if (destId == curId) {

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

            else if (cmd == "create") {
                int newChildId;
                request >> newChildId;
                if (childId != -1) {
                    unbind(childSocket, childId);
                }
                bind(childSocket, newChildId);
                pid_t pid = fork();
                if (pid < 0) {
                    perror("Can't create new process");
                    return -1;
                }
                if (pid == 0) {
                    execl("./counting", "./counting",
std::to_string(newChildId).c_str(), std::to_string(childId).c_str(), NULL);
                    perror("Can't execute new process");
                    return -2;
                }
                send_message(childSocket, std::to_string(newChildId) + "pid");
```

```
                childId = newChildId;
                send_message(parentSocket, receive_message(childSocket));
            }
            else if (cmd == "exec") {
                long unsigned int sum = 0;
                std::string number;
                int count;
                request >> count;
                if ( count < 1){
                send_message(parentSocket, "Error: wrong count of numbers");
                }
                else
                {
                while (request >> number) {
                    sum += std::stoi(number);
                }
                send_message(parentSocket, "OK: " + std::to_string(curId) + ": " +
std::to_string(sum));
                }
            }

            else if (cmd == "heartbeat") {
                std::string reply;
                if (childId != -1) {
                send_message(childSocket, std::to_string(childId) + " heartbeat");
                std::string msg = receive_message(childSocket);
                reply += " " + msg;
                }
                send_message(parentSocket, std::to_string(curId) + reply);
            }
            else if (cmd == "kill") {
                if (childId != -1) {
                send_message(childSocket, std::to_string(childId) + " kill");
                std::string msg = receive_message(childSocket);
                if (msg == "OK") {
                    send_message(parentSocket, "OK");
                }
                unbind(childSocket, childId);
                disconnect(parentSocket, curId);
                break;
                }
                send_message(parentSocket, "OK");
                disconnect(parentSocket, curId);
                break;
            }
        }
        else if (childId != -1) {
```

```
            send_message(childSocket, message);
            send_message(parentSocket, receive_message(childSocket));
            if (childId == destId && cmd == "kill") {
                childId = -1;
            }
        }
        else {
            send_message(parentSocket, "Error: node is unavailable");
        }
    }
}
```

topology.h:

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

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

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

    int find(int id) {
        int curListId = 0;
        for (auto it1 = container.begin(); it1 != container.end(); ++it1) {
            for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {
                if (*it2 == id) {
```

```
                    return curListId;
                }
            }
            ++curListId;
        }
        return -1;
    }

    void erase(int id) {
        int listId = find(id);
        if (listId == -1) {
            throw std::runtime_error("Wrong id");
        }
        auto it1 = container.begin();
        std::advance(it1, listId);
        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 listId) {
        auto it1 = container.begin();
        std::advance(it1, listId);
        if (it1->begin() == it1->end()) {
            return -1;
        }
        return *(it1->begin());
    }
};
```

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, zmq :: send_flags ::none);
```

```
}

std::string receive_message(zmq::socket_t& socket) {
    zmq::message_t message;
    bool charsRead;
    try {
        charsRead = socket.recv(&message);
    }
    catch (...) {
        charsRead = false;
    }
    if (charsRead == 0) {
        return "Error: node is unavailable [zmq_func]";
    }
    std::string receivedMsg(static_cast<char*>(message.data()), message.size());

    return receivedMsg;
}

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);
}
```

Пример работы

```
znako@znako-VirtualBox:~/UtoW/OS_LABS/lab678/src$ ./control

create 1 -1

OK: 3481

create 2 1

OK: 3486

create 3 -1

OK: 3489

create 4 3

OK: 3492

kill 3

OK

exec 2 3 12 23 45

OK: 2: 80

heartbeat 3

OK: 1 2

OK: 1 2

OK: 1 2

OK: 1 2

exit
```

Вывод

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

В C, как и большинстве ЯП есть такая структура, как сокеты, которые позволяют удобно организовывать построение и использование архитектуры клиент-сервер. Для общение в архитектуре клиент-сервер существуют очереди сообщений, при помощи них можно достаточно не сложно организовать обмен информацией, однако ZMQ – имеет не самую лучшую документацию и в связке с fork и т.п. может вызывать некоторые трудности. Такие структуры, как деревья хорошо подходят для хранения информации о клиентах и сервере.