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

Лабораторная работа №6 по курсу

«Операционные системы»

Управление серверами сообщений

Студент: Паленов Павел Сергеевич
Группа: М8О –301Б-18
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва
2021

Содержание

1. Постановка задачи
2. Общие сведения о программе
3. Общий метод и алгоритм решения
4. Листинг программы
5. Демонстрация работы программы
6. Вывод

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.
Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы.

Задание:

Тип топологии: 1.

Тип вычислительных команд: 1.

тип команд доступности узлов 2.

Общие сведения о программе

Программа состоит из двух основных файлов и библиотеки, реализующей взаимодействия с узлами. Помимо этого используется библиотека zmq, которая реализует очередь сообщений.

1) client.cpp - программа, которая является управляющим узлом

2) server.cpp – программа, которая производит вычисления

3) zmq_requests.h – небольшая библиотека для принятия/отправки сообщений.

Очередь сообщений - компонент, используемый для межпроцессного или межпотокового взаимодействия внутри одного процесса. Для обмена сообщениями используется очередь. Очереди сообщений предоставляют асинхронный протокол передачи данных, означая, что отправитель и получатель сообщения не обязаны взаимодействовать с очередью сообщений одновременно. Размещённые в очереди сообщения хранятся до тех пор, пока получатель не примет их.

Очереди сообщений имеют неявные или явные ограничения на размер данных, которые могут передаваться в одном сообщении, и количество сообщений, которые могут оставаться в очереди.

Многие реализации очередей сообщений функционируют внутренне: внутри операционной системы или внутри приложения. Такие очереди существуют только для целей этой системы.

Другие реализации позволяют передавать сообщения между различными компьютерными системами, потенциально подключая несколько приложений и несколько операционных систем. Эти системы очередей сообщений обычно обеспечивают расширенную функциональность для обеспечения устойчивости, чтобы гарантировать, что сообщения не будут «потеряны» в случае сбоя системы.

ZMQ – это высокопроизводительная библиотека асинхронных сообщений, предназначенная для использования в распределенных или параллельных приложениях. Он обеспечивает очередь сообщений, но в отличие от промежуточного программного обеспечения, ориентированного на сообщения, система ZMQ может работать без выделенного посредника сообщений.

Сокеты— название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Основные функции и вызовы:

1) **void * zmq_ctx_new()** - создать новый контекст ZMQ. Функция *zmq_ctx_new ()* должна вернуть дескриптор вновь созданного контекста в случае успеха. В противном случае он должен вернуть NULL и установить для *errno* одно из значений.

2) **void * zmq_socket (void *** ***context*****, int** ***type*****)** - Функция должна создать сокет ZMQ в указанном контексте и вернуть непрозрачный дескриптор вновь созданному сокету. Аргумент type указывает тип сокета, который определяет семантику связи через сокет.

Вновь созданный сокет изначально не связан и не связан ни с какими конечными точками. Чтобы установить поток сообщений, сокет должен быть сначала подключен по крайней мере к одной конечной точке с помощью zmq_connect (3), или по крайней мере одна конечная точка должна быть создана для приема входящих соединений с помощью zmq_bind (3).

Сокет типа ZMQ_REQ используется клиентом для отправки запросов и получения ответов от службы.

ZMQ_REP используется службой для получения запросов и отправки ответов клиенту.

3) **int execv(const char *_path_, char *const _argv_[])** - функция заменяет текущий образ процесса новым.**execv()**предоставляет новой программе список аргументов в виде массива указателей на строки, заканчивающиеся null. Первый аргумент, по соглашению, должен указать на имя, ассоциированное с файлом, который необходимо запустить. Массив указателей должен заканчиваться указателем null.

4) **int zmq_msg_init_size (zmq_msg_t * _msg_, size_t _size_) -** функция должна выделить любые ресурсы, необходимые для хранения сообщения размером *size* в байтах, и инициализировать объект сообщения, на который ссылается *msg,* для представления вновь выделенного сообщения. Реализация должна выбирать, хранить ли содержимое сообщения в стеке (маленькие сообщения) или в куче (большие сообщения). По причинам производительности *zmq_msg_init_size ()* не должен очищать данные сообщения.

5) **int zmq_msg_recv (zmq_msg_t * _msg_, void * _socket_, int _flags_) -** функция должна получить часть сообщения из сокета, на который ссылается аргумент сокета*,* и сохранить ее в сообщении, на которое ссылается аргумент *msg*. Любой контент, ранее сохраненный в *msg,* должен быть должным образом освобожден. Если есть нет частей сообщений доступны на указанном *сокете zmq_msg_recv ()* функция должна блокироваться, пока запрос не может быть удовлетворен. Флаги аргумент представляет собой комбинацию флагов, определенных ниже: **ZMQ_DONTWAIT -** Указывает, что операция должна выполняться в неблокирующем режиме. Если в указанном сокете нет доступных сообщений, функция *zmq_msg_recv ()* завершится ошибкой.

6) **int zmq_send (void * _socket_, void * _buf_, size_t _len_, int _flags_) -** функция должна поставить в очередь сообщение, созданное из буфера, на который ссылаются аргументы *buf* и *len*. *flags* - аргумент представляет собой комбинацию следующих флагов:

ZMQ_DONTWAIT - для типов сокетов (DEALER, PUSH), которые блокируются, когда нет доступных одноранговых узлов (или все одноранговые узлы имеют максимальную отметку), указывает, что операция должна выполняться в неблокирующем режиме. Если сообщение не может быть поставлено в очередь на сокете, функция *zmq_send ()* завершится ошибкой.

ZMQ_SNDMORE - указывает, что отправляемое сообщение является сообщением, состоящим из нескольких частей, и что последующие части сообщения должны следовать.

7) **void * memcpy(void *dest, const void *source, size_t count)** - функция копирует *count* символов из массива, на который указывает *source*, в массив, на

который указывает *dest*. Если массивы перекрываются, поведение *memcpy*() не определено.

8) **int zmq_msg_close (zmq_msg_t * *msg*) -** функция должна информировать инфраструктуру ZMQ о том, что любые ресурсы, связанные с объектом сообщения, на который ссылается *msg*, больше не требуются и могут быть освобождены. Фактическое высвобождение ресурсов, связанных с объектом сообщения, должно быть отложено ZMQ до тех пор, пока все пользователи сообщения или базового буфера данных не укажут, что он больше не требуется. Приложения должны гарантировать, что *zmq_msg_close ()* вызывается, когда сообщение больше не требуется, в противном случае могут возникнуть утечки памяти. Это не требуется после успешного выполнения *zmq_msg_send ()*.

9) **void* zmq_msg_data (zmq_msg_t * *msg*) -** функция должна возвращать указатель на содержимое сообщения объекта сообщения, на который ссылается *msg*.

10) **size_t zmq_msg_size (zmq_msg_t * *msg*) –** функция должна возвращать размер в байтах содержимого объекта сообщения, на который ссылается *msg*.

Общий метод и алгоритм решения

Имеется управляющий узел и выполняющие узлы

1. Управляющий узел принимает команды, обрабатывает их и пересылает их дочерним узлам. В случае ошибок, управляющий узел выводит сообщение об этом.
2. Из дочерних узлов возвращается некоторое сообщение (об успехе либо ошибки выполнения переданной команды).
3. Если узел недоступен, то по истечении попытки подключиться к узлу будет выдано сообщение о недоступности узла.

Листинг программы

client.cpp

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

#include <iostream>
#include <exception>
#include <unordered_map>
#include <vector>
#include <string>
#include <cstring>

#include "zmq.h"

#include "zmq_requests.h"

using namespace std;

void PrintMenu(ostream& os = cout) {
        os << "List of commands: " << endl
           << "1) create id [parent] - create new server node" << endl
           << "2) remove id - remove existing server node" << endl
           << "3) exec id n k1...kn - evaluate sum of n numbers" << endl
           << "4) ping id - check if server still available" << endl
           << "5) exit - exit program" << endl
           << "6) print - print menu" << endl;
}

string CreateNewServerNode(const unordered_map<int, void*>& socket, const int id, const int
parent,
                unordered_map<int, pid_t>& server_nodes_pids) {
        string result = "";
        if (parent == -1) {
                if (server_nodes_pids.find(id) != server_nodes_pids.end()) {
                        result = string("Error: Already exists");
                } else {
```

```
                        cout << "Enter the port to connect: ";
                        int port = 0;
                        cin >> port;
                        string address = "tcp://localhost:" + to_string(port);
                        cout << "Enter the executable file (\"./smth\"): ";
                        string path = "";
                        cin >> path;
                        pid_t pid = fork();
                        if (pid == -1) {
                                result = "Error: Unable to create new server node";
                        } else if (pid == 0) { // ребенок
                                CreateServerNode(path, id, port);
                        } else { // родитель
                                if (static_cast<int>(zmq_connect(socket.at(id), ad-
dress.c_str())) == 0) {
                                        server_nodes_pids[id] = pid;
                                        SendMessage(socket.at(id), "Ping");
                                        result = ReceiveMessage(socket.at(id));
                                        result += ": " + to_string(pid);
                                } else {
                                        result = "Can't connect to the server with id " +
to_string(id);
                                        result += " and port " + to_string(port) + ". Please
try again";
                                }
                        }
                }
        } else {
                result = string("Error: Parent not found");
        }
        return result;
}


string RemoveServerNode(const pid_t pid) {
        string result = "";
        int is_killed = kill(pid, SIGKILL); // второй аргумент для немедленного завершения
процесса
        is_killed == 0 ? result = "Ok" : result = "Can't kill the process";
        return result;
```

```
}

string Exec(const unordered_map<int, void*>& socket, const int id, vector<int> v) {
        string result = "";
        if (socket.find(id) != socket.end()) {
                string to_calc = "Exec ";
                for (const auto& item : v) {
                        to_calc += to_string(item);
                        to_calc += " ";
                }
                SendMessage(socket.at(id), to_calc);
                result = ReceiveMessage(socket.at(id));
        } else {
                result = "Error: Not found";
        }
        return result;
}

string PingServerNode(const unordered_map<int, void*>& socket, const int id,
                unordered_map<int, pid_t>& server_nodes_pids) {
        string result = "";
        if (server_nodes_pids.find(id) != server_nodes_pids.end()) {
                try {
                        SendMessage(socket.at(id), "Ping");
                        result = ReceiveMessage(socket.at(id));
                        result += ": 1";
                } catch(...) {
                        result += ": 0";
                }
        } else {
                result = "Error: Not found";
        }
        return result;
}

int main () {
        unordered_map<int, pid_t> server_nodes_pids;
```

```
	void* context = zmq_ctx_new();
	cout << "Client starting..." << endl;
	PrintMenu();

	unordered_map<int, void*> request;

	while (true) {
		string command;
		cin >> command;
		if (command == "create") {
			int id = 0, parent = 0;
			cin >> id >> parent;
			if (server_nodes_pids.find(id) == server_nodes_pids.end()) {
				request[id] = zmq_socket(context, ZMQ_REQ);
			}
			cout << CreateNewServerNode(request, id, parent, server_nodes_pids) <<
endl;
		} else if (command == "remove") {
			int id = 0;
			cin >> id;
			auto it = server_nodes_pids.find(id);
			if (it != server_nodes_pids.end()) {
				cout << RemoveServerNode(it->second) << endl;
				server_nodes_pids.erase(it);
				request.erase(id);
			} else {
				cout << "Error: Not found" << endl;
			}
		} else if (command == "exec") {
			int id = 0, k = 0;
			cin >> id >> k;
			string ping = PingServerNode(request, id, server_nodes_pids);
			if (ping == "Ok: 1") {
				vector<int> numbers;
				numbers.reserve(k);
				for (int i = 0; i < k; ++i) {
					int n = 0;
					cin >> n;
```

```
                                        numbers.push_back(n);
                                }
                                cout << Exec(request, id, move(numbers)) << endl;
                        } else {
                                cout << "Error: Server is unavailiable" << endl;
                        }
                } else if (command == "ping") {
                        int id = 0;
                        cin >> id;
                        cout << PingServerNode(request, id, server_nodes_pids) << endl;
                } else if (command == "exit") {
                        break;
                } else if (command == "print") {
                        PrintMenu();
                } else {
                        cout << "Wrong input" << endl;
                }
        }

        for (const auto socket : request) {
                zmq_close(socket.second);
        }
        zmq_ctx_destroy(context);

        return 0;
}
```

server.cpp

```
#include <unistd.h>
#include <sys/types.h>

#include <iostream>
#include <string>
#include <cstring>

#include "zmq.h"
```

```
#include "zmq_requests.h"

using namespace std;

int ParseRequest(const string& s) {
        int result = 0;
        string current_number;
        for (const auto item : s) {
                if (item != ' ' && isdigit(item)) {
                        current_number.push_back(item);
                } else if (item == ' ' && !current_number.empty()) {
                        result += stoi(current_number);
                        current_number.clear();
                }
        }
        return result;
}

int main(int argc, char const *argv[]) {
        int id = stoi(argv[1]);
        int port = stoi(argv[2]);
        string address = "tcp://*:" + to_string(port);

        void* context = zmq_ctx_new();
        void* respond = zmq_socket(context, ZMQ_REP);

        zmq_bind(respond, address.c_str());

        string request = "";
        request = ReceiveMessage(respond);
        while (true) {
                sleep(1);
                if (request == "Ping") {
                        SendMessage(respond, "Ok");
                } else if (request.substr(0, 4) == "Exec") {
                        int sum = ParseRequest(request);
                        string result = "";
                        result = "Ok: " + to_string(id) + ": " + to_string(sum);
```

```
                        SendMessage(respond, result);
                }
                request = ReceiveMessage(respond);
        }
        zmq_close(respond);
        zmq_ctx_destroy(context);

        return 0;
}
```

zmq_requests.h

```
#include <unistd.h>
#include <cstring>

#include <iostream>
#include <exception>
#include <string>

#include "zmq.h"

void CreateServerNode(std::string path, const int id, const int port) {
        char* id_ = strdup(std::to_string(id).c_str());
        char* port_ = strdup(std::to_string(port).c_str());
        char* args[] = {strdup(path.c_str()), id_, port_, NULL};
        execv(strdup(path.c_str()), args);
}

void SendMessage(void* request, const std::string message) {
        zmq_msg_t req;
        zmq_msg_init_size(&req, message.size());
        memcpy(zmq_msg_data(&req), message.c_str(), message.size());
        zmq_msg_send(&req, request, 0);
        zmq_msg_close(&req);
}

std::string ReceiveMessage(void* request) {
        zmq_msg_t reply;
```

```
        zmq_msg_init(&reply);
        zmq_msg_recv(&reply, request, 0);
        std::string result(static_cast<char*>(zmq_msg_data(&reply)), zmq_msg_size(&reply));
        zmq_msg_close(&reply);
        return result;
}
```

Демонстрация работы программы

```
justp@Pavel:~/lab6$ ./client
Client starting...
List of commands:
1) create id [parent] - create new server node
2) remove id - remove existing server node
3) exec id n k1...kn - evaluate sum of n numbers
4) ping id - check if server still available
5) exit - exit program
6) print - print menu
create 1 -1
Enter the port to connect: 8080
Enter the executable file: ("./smth")./server
Ok: 1782
exec 1 5 1 2 3 4 5
Ok: 1: 15
ping 1
Ok: 1
create 2 -1
Enter the port to connect: 9090
Enter the executable file: ("./smth")./server
Ok: 1785
remove 0
```

```
Error: Not found

remove 1

Ok

exec 2 6 4 5 6 7 8 9

Ok: 2: 39

exit
```

Вывод

Очередь сообщений очень полезная технология для различных ситуаций. Они являются важным компонентом в приложениях и архитектурах программного обеспечения за счет масштабируемости (хорошо распределяют процессы обработки информации), асинхронности и удобного интерфейса обмена данными между процессами.