Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет: “Информационные технологии и прикладная математика”
Кафедра: 806 “Вычислительная математика и программирование”

Лабораторная работа №6-8
по курсу “Операционные системы”

Студент	Полей-Добронравова А.В.
Группа	М8О-307Б-18
Вариант	46
Преподаватель	Миронов Е.С.
Дата	
Оценка	

Москва, 2020

1. 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.
Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

Создание нового вычислительного узла

Формат команды: create id [parent]
id – целочисленный идентификатор нового вычислительного узла
parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)
Формат вывода:
«Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла
«Error: Already exists» - вычислительный узел с таким идентификатором уже существует
«Error: Parent not found» - нет такого родительского узла с таким идентификатором
«Error: Parent is unavailable» - родительский узел существует, но по каким-то причинам с ним не удается связаться
«Error: [Custom error]» - любая другая обрабатываемая ошибка.

Удаление существующего вычислительного узла

Формат команды: remove id
id – целочисленный идентификатор удаляемого вычислительного узла

Формат вывода:

«Ok» - успешное удаление

«Error: Not found» - вычислительный узел с таким идентификатором не найден

«Error: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

«Error: [Custom error]» - любая другая обрабатываемая ошибка

Исполнение команды на вычислительном узле

Формат команды: exec id [params]

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

Формат вывода:

«Ok:id: [result]», где result – результат выполненной команды

«Error:id: Not found» - вычислительный узел с таким идентификатором не найден

«Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом

«Error:id: [Custom error]» - любая другая обрабатываемая ошибка

Топология:

Все вычислительные узлы находятся в дереве общего вида. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

Команда на вычислительном узле:

Набора команд 2 (локальный целочисленный словарь)

Формат команды сохранения значения: exec id name value

id – целочисленный идентификатор вычислительного узла, на который отправляется команда name – ключ, по которому будет сохранено значение (строка формата [A-Za-z0-9]+)

value – целочисленное значение

Формат команды загрузки значения: exec id name

Тип проверки доступности узлов:

Команда проверки 2

Формат команды: ping id

Команда проверяет доступность конкретного узла. Если узла нет, то необходимо выводить ошибку: «Error: Not found»

Пример:

> ping 10

Ok: 1 // узел 10 доступен

> ping 17

Ok: 0 // узел 17 недоступен

2. Решение задачи

Для решения задачи выбрана библиотека сервера сообщений ZeroMQ, которая разворачивает сервер сообщений на текущей машине.

Для получения и передачи сообщений между узлами создана библиотека “message.h”. В ней реализована структура данных msg, хранящая все необходимые поля для одного сообщения: команду action, id получателя, path путь по топологии для передачи сообщения, name имя в словаре для команды exec, value значение имени в словаре для команды exec. Кроме того реализованы две функции send - отправить; rec - получить, работающие именно с библиотекой zmq.

Для организации топологии дерева общего вида вычислительных узлов, связанных между собой через сервер сообщений, создана **динамическая библиотека “knot.h”**. В ней структура данных knot - узел, хранящая id узла, номер процесса его работы pid и необходимые для работы с zeroMQ параметры: Для реализации топологии дерева общего вида удобен паттерн соединения PUB-SUB (публикатор-подписчики). PUB опубликовывает

сообщения, а несколько подписчиков все их читают, обратной связи от SUB в чистом виде не предусмотрено. Авторы библиотеки сравнивают этот паттерн с радио или рассылками в интернете.

Непосредственное создание PUB на узле происходит после первой команды добавляющей к текущему узлу подузлы, с помощью функции knot_add(knot* k, int d). В ней проверка на наличие уже сокета у публикатора и последующее добавление подузла. knot_create(char * id, char * pport) запускается как конструктор в начале работы подузла с созданием подписчика SUB, подсоединенного с помощью connect к порту PUB.

Начало работы проекта для решения данной задачи начинается с программы server.c являющейся управляющим узлом. Далее при каждом создании подузла на текущем узле с помощью fork() запускается подпроцесс, который заменяется с помощью execl("client", str, str1, NULL); на программу client.c.

Библиотека для целочисленного словаря (команда для вычислительного узла exec) “vocabulary.h”.

На управляющем узле (server.c) есть структуры данных типа db, хранящие деревья общего вида. Одно дерево work хранит все доступные узлы, дерево lost хранит все недоступные узлы. Команды для проверки доступности узлов работают с этими структурами данных

Сборка проекта в терминале:

clang -Wall -c message.c -o message.o

gcc -c -o vocabulary.o vocabulary.c

clang -Wall -c -fPIC knot.c -o knot.o

clang -shared -Wall -o libknot.so knot.o vocabulary.o message.o -L/usr/local/opt/ -lzmq

gcc -c -o server.o server.c

cp libknot.so /usr/local/opt/

gcc -o client.o -c client.c

clang -Wall -o client client.o -L/usr/local/opt/ -lzmq -lknot message.o

clang -Wall -o server server.o -L/usr/local/opt/ -lzmq -lknot message.o

./server please.txt

Функции для работы с сервером сообщений ZeroMQ:

void *zmq_ctx_new ();

Функция zmq_ctx_new () создает новый контекст ØMQ. Функция zmq_ctx_new () должна возвращать непрозрачный дескриптор вновь созданного контекста в случае успеха. В противном случае он должен вернуть NULL и установить для errno одно из значений.

void *zmq_socket (void *_context_, int _type_);

Функция zmq_socket () должна создать сокет ØMQ в указанном контексте и вернуть непрозрачный дескриптор для вновь созданного сокета. Аргумент type указывает тип сокета, который определяет семантику связи через сокет. Созданный сокет изначально не связан и не связан с какими-либо конечными точками. Чтобы установить поток сообщений, сокет должен сначала быть 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).

int zmq_bind (void *_socket_, const char *_endpoint_);

Функция zmq_bind () связывает сокет с локальной конечной точкой, а затем принимает входящие соединения на этой конечной точке. Конечная точка - это строка, состоящая из транспорта: //, за которым следует адрес. Транспорт определяет используемый протокол. Адрес указывает адрес транспорта для привязки. Функция zmq_bind () возвращает ноль в случае успеха. В противном случае он возвращает -1 и устанавливает errno.

int zmq_connect (void *_socket_, const char *_endpoint_);

Функция zmq_connect () подключает сокет к конечной точке, а затем принимает входящие соединения на этой конечной точке. Функция zmq_connect () возвращает ноль в случае успеха. В противном случае он возвращает -1 и устанавливает для errno одно из значений.

int zmq_setsockopt (void *_socket_, int _option_name_, const void *_option_value_, size_t _option_len_);

Функция zmq_setsockopt () должна установить для опции, указанной аргументом option_name, значение, на которое указывает аргумент

option_value для сокета ØMQ, на который указывает аргумент сокета. Аргумент option_len - это размер значения параметра в байтах.

int zmq_close (void **socket*);

Функция zmq_close () должна уничтожить сокет, на который ссылается аргумент сокета. Любые незавершенные сообщения, физически полученные из сети, но еще не полученные приложением с помощью zmq_recv (), должны быть отброшены. Поведение для отбрасывания сообщений, отправленных приложением с помощью zmq_send (), но еще не переданных физически в сеть, зависит от значения параметра сокета ZMQ_LINGER для указанного сокета.

int zmq_ctx_destroy (void **context*);

Должна уничтожить контекст.

int zmq_msg_init_data (zmq_msg_t **msg*, void **data*, size_t *size*, zmq_free_fn **ffn*, void **hint*);

Функция zmq_msg_init_data () должна инициализировать объект сообщения, на который ссылается msg, для представления содержимого, на которое ссылается буфер, расположенный по адресу data, размером в байтах. Копирование данных не должно выполняться, и ØMQ становится владельцем предоставленного буфера. Если предоставлено, функция освобождения ffn должна вызываться, когда буфер данных больше не требуется ØMQ, с аргументами данных и подсказки, предоставленными zmq_msg_init_data(). Никогда не обращайтесь к членам zmq_msg_t напрямую, вместо этого всегда используйте семейство функций zmq_msg. Функция освобождения памяти ffn должна быть потокобезопасной, поскольку она будет вызываться из произвольного потока. Функции zmq_msg_init (), zmq_msg_init_data () и zmq_msg_init_size () являются взаимоисключающими. Никогда не инициализируйте один и тот же zmq_msg_t дважды.

int zmq_msg_send (zmq_msg_t **msg*, void **socket*, int *flags*);

Функция zmq_msg_send() идентична функции zmq_sendmsg(3), которая будет исключена в будущих версиях. zmq_msg_send() более совместим с другими функциями обработки сообщений. Функция zmq_msg_send() должна поставить в очередь сообщение, на которое ссылается аргумент msg, для отправки в сокет, на который ссылается аргумент сокета.

int zmq_msg_close (zmq_msg_t *msg);
Должна информировать инфраструктуру ØMQ 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 ØMQ 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.

int zmq_msg_recv (zmq_msg_t *msg, void *socket, int flags);

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 в указанном сокете нет доступных частей сообщения, функция zmq_msg_recv() должна блокироваться до тех пор, пока запрос не будет удовлетворен.

2. Демонстрация работы программы:

Программе на вход подаётся файл с командами please.txt:

create 1 -1

create 2 -1

exec 1 MyVar 5

create 8 1

ping 1

ping 30

ping 2

create 4 2

exec 1 MyVa

remove 2

ping 4

remove 1

ping 1

exec 1 MyVar

create 2 -1

create 7 2

exec 7 Lipetc 15

exec 2 Lipetc

exec 7 Lipetc

remove 1

Далее работа программы:

```
MacBook-Pro-Amelia:6lab amelia$ ./server please.txt

 read new command: create 1 -1
Tree:
-1
 1

 read new command: create 2 -1
Tree:
-1
 1
 2

 read new command: exec 1 MyVar 5
Ok:1:created:65886
Ok:2:created:65887
Ok:1

 read new command: create 8 1
Ok:8:created:65890
-1
 1
  8
 2
```

```
 read new command: ping 1
Ok: 1

 read new command: ping 30
Error: Not found

 read new command: ping 2
Ok: 1

 read new command: create 4 2
Ok:4:created:65891
-1
 1
  8
 2
  4

 read new command: exec 1 MyVa
Ok:1:'MyVa' not found

 read new command: remove 2
Ok:2:removed
-1
 1
  8

 read new command: ping 4
Ok: 0

 read new command: remove 1
Ok:1:removed
-1

 read new command: ping 1
Error: Not found

 read new command: exec 1 MyVar
Error: Not found

 read new command: create 2 -1
Tree:
```

```
-1
 2

 read new command: create 7 2
Ok:2:created:65892
Ok:7:created:65894
-1
 2
  7

 read new command: exec 7 Lipetc 15

 read new command: exec 2 Lipetc
Ok:7
Ok:2:'Lipetc' not found

 read new command: exec 7 Lipetc
No more commands: 0
Ok:7: 15
Ok:2:removed
```

После того, как не осталось команд, запускается самоуничтожение всех доступных узлов. Недоступные узлы можно удалить из терминала:

С помощью команды ps -Af находим процессы с портами 8000+

kill [номер процесса]

3. Листинг программы

message.h

```
#ifndef _MESSAGE_H_

#define _MESSAGE_H_

#include <inttypes.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>
```

```
#include </usr/local/opt/zeromq/include/zmq.h>
typedef char* ID;
typedef struct _msg {
    char* action;
    int id;
    char* path;
    ID name;
    int value;
} message;

void send(void* r, char* action, int id, char* pat, ID nam, int value);
message* rec(void* r);
#endif
```

message.c

```
#include "message.h"
void send(void* r, char* action, int id, char* pat, ID nam, int value) {
    zmq_msg_t actionMes;
    zmq_msg_t idMes;
    zmq_msg_t pathMes;
    zmq_msg_t nameMes;
    zmq_msg_t valueMes;

    char str1[10];
    sprintf(str1, "%d", id);
```

```
char str[10];
sprintf(str, "%d", value);

zmq_msg_init_data (&actionMes, action, sizeof(action), NULL, NULL);
        zmq_msg_init_data (&idMes, str1, sizeof(str1), NULL, NULL);
        zmq_msg_init_data (&pathMes, pat, sizeof(pat), NULL, NULL);
        zmq_msg_init_data (&nameMes, nam,sizeof(nam), NULL, NULL);
        zmq_msg_init_data (&valueMes, str,sizeof(str), NULL, NULL);

        //ZMQ_SNDMORE
        sleep(2);
        zmq_msg_send(&actionMes, r, ZMQ_SNDMORE);
        sleep(1);
        zmq_msg_send(&pathMes, r, ZMQ_SNDMORE);
        sleep(1);
        zmq_msg_send(&idMes, r, ZMQ_SNDMORE);
        sleep(1);
        zmq_msg_send(&nameMes, r, ZMQ_SNDMORE);
        sleep(1);
        zmq_msg_send(&valueMes, r, 0);
        sleep(1);

        zmq_msg_close(&actionMes);
        zmq_msg_close(&idMes);
        zmq_msg_close(&pathMes);
        zmq_msg_close(&nameMes);
```

```
        zmq_msg_close(&valueMes);
}

message* rec(void* r) {
        message* m = malloc(sizeof(message));
        zmq_msg_t actionMes;
    zmq_msg_t idMes;
    zmq_msg_t pathMes;
    zmq_msg_t nameMes;
    zmq_msg_t valueMes;

    zmq_msg_init(&actionMes);
    zmq_msg_init(&idMes);
    zmq_msg_init(&pathMes);
    zmq_msg_init(&nameMes);
    zmq_msg_init(&valueMes);

    zmq_msg_recv(&actionMes, r, 0);
    int length = zmq_msg_size(&actionMes);
        m->action = malloc(length);
        memcpy(m->action, zmq_msg_data(&actionMes), length);
    zmq_msg_recv(&pathMes, r, 0);
    length = zmq_msg_size(&pathMes);
    m->path = malloc(length);
    memcpy(m->path, zmq_msg_data(&pathMes), length);
    zmq_msg_recv(&idMes, r, 0);
```

```
length = zmq_msg_size(&idMes);
char* str = malloc(length);
memcpy(str, zmq_msg_data(&idMes), length);
int k = 0;
int i = 0;
while (str[i] != '\0') {
	k = k * 10 + str[i] - '0';
	i++;
}
m->id = k;
	zmq_msg_recv(&nameMes, r, 0);
length = zmq_msg_size(&nameMes);
	m->name = malloc(length);
	memcpy(m->name, zmq_msg_data(&nameMes), length);
zmq_msg_recv(&valueMes, r, 0);
length = zmq_msg_size(&valueMes);
str = malloc(length);
memcpy(str, zmq_msg_data(&valueMes), length);
k = 0;
i = 0;
while (str[i] != '\0') {
	k = k * 10 + str[i] - '0';
	i++;
}
m->value = k;
```

```
    zmq_msg_close(&actionMes);
        zmq_msg_close(&idMes);
        zmq_msg_close(&pathMes);
        zmq_msg_close(&nameMes);
        zmq_msg_close(&valueMes);
        return m;
}
```

knot.h

```
#ifndef KNOT_H
#define KNOT_H
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

#include "vocabulary.h"
#include "message.h"

int BEGIN_PORT = 8000;
```

```
char* BindURLPort(int port);
char* ConURLPort(int port);
int TakePort(void* socket);
typedef struct _n {
        int id;
    void* context_me;
    void* r_me;
    pid_t pid;
    voc* v;
    int port_fl;
    void* context_fl;
    void* r_fl;
} knot;

knot* knot_create(char* id, char* pport);
void knot_add(knot* k, int d);
void knot_destroy(knot* k);
void knot_destroy_brahch(knot* k);
#endif
```

knot.c

```
#include "knot.h"

char* BindURLPort(int port) {
        char* str = malloc(6);
        char* dest = malloc(16);
        sprintf(dest, "%s", "tcp://*:");
```

```
	sprintf(str, "%d", port);
	return strncat(dest, str, strlen(str));
}

char* ConURLPort(int port) {
	char* dest = malloc(24);
	char* str = malloc(6);
	sprintf(dest, "%s", "tcp://localhost:");
	sprintf(str, "%d", port);
	return strncat(dest, str, strlen(str));
}

int TakePort(void* socket)
{
	int port = BEGIN_PORT;
	while (zmq_bind(socket, BindURLPort(port)) != 0) {
		port++;
		}
	return port;
}
// создаю узел
knot* knot_create(char * id, char * pport) {
		int inPort = 0;
		int i = 0;
		while (i < strlen(pport)) {
			inPort = inPort * 10 + pport[i] - '0';
```

```
                i++;
        }
        knot* k = malloc(sizeof(knot));
        i = 0;
        int j = 0;
        while ((id[i] - '0') >= 0 && (id[i] - '0') < 10) {
                j = j * 10 + id[i] - '0';
                i++;
        }
        k->id = j;
        k->pid = getpid();
        k->context_me = zmq_ctx_new();
    k->r_me = zmq_socket(k->context_me, ZMQ_SUB);
    zmq_connect(k->r_me, ConURLPort(inPort));
    zmq_setsockopt(k->r_me, ZMQ_SUBSCRIBE, "", 0);
        k->v = voc_create();
    k->port_fl = 0;
    k->context_fl = NULL;
    k->r_fl = NULL;
    printf("Ok:%d:created\n",k->id);
    return k;
    }
void knot_add(knot* k, int d) {
    char* str1 = malloc(10);
    char* str = malloc(10);
        sprintf(str, "%d", d);
```

```
                if (k->port_fl == 0) {
                k->context_fl = zmq_ctx_new();
                k->r_fl = zmq_socket(k->context_fl, ZMQ_PUB);
                k->port_fl = TakePort(k->r_fl);
        }
        sprintf(str1, "%d", k->port_fl);
        sprintf(str, "%d", d );
        pid_t pid = fork();
        if (pid == 0) {
                execl("client", str, str1, NULL);
        }
        else {
                free(str);
                free(str1);
        }
}
void knot_destroy(knot* k) {
        zmq_close (k->r_me);
   zmq_ctx_destroy(k->context_me);
   free(k);
}

void knot_destroy_brahch(knot* k) {
//отправить сообщение об удалении сыновей
        if (k->r_fl != NULL) {
                send(k->r_fl, "remove", -1, " ", " ", -1);
```

```
        }
        //удалиться
        zmq_close (k->r_me);
    zmq_ctx_destroy(k->context_me);
    free(k);
}
```

client.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>

#include "knot.h"
#include "message.h"

int main(int argc, char * argv[])
{
        message* mes;
        int vr,u;
```

```
char* th = malloc(255);
int i;
knot* k = knot_create(argv[0], argv[1]);
while(true) {
	mes = rec(k->r_me);
	if (strcmp(mes->action,"exec") == 0) {
		if (mes->id == k->id) {
			printf("Ok:%d",k->id);
			if (mes->value != -29) { //добавить в словарь
				voc_add_w(k->v, mes->name, mes->value);
				printf("\n");
			}
			else { //найти в словаре
				word* w = voc_find(k->v, mes->name);
				if (w == NULL) {
					printf(":'%s' not found\n",mes->name);
				}
				else {
					printf(": %d\n", w->value);
				}
			}
		}
		else {
			vr = 0;
			u = 0;
			while (mes->path[u] != ' ' && mes->path[u] != '\0') {
```

```
                                vr = vr * 10 + mes->path[u] - '0';
                                u++;
                        }
                        if (vr == k->id) {
                                u++;
                                i = u;
                                vr = 0;
                                while(i < strlen(mes->path)) {
                                        th[vr] = mes->path[i];
                                        i++;
                                        vr++;
                                }
                                if (mes->value == -29) {
                                        mes->value = -1;
                                }
                                send(k->r_fl, "exec", mes->id, th, mes->name,
mes->value);
                                i = 0;
                                while(i < strlen(th)) {
                                        th[i] = '\0';
                                        i++;
                                }
                        }
                }
        }
        else if (strcmp(mes->action,"remove") == 0) {
                if (mes->id == k->id || mes->id == -29) {
```

```
            printf("Ok:%d:removed\n", k->id);
            knot_destroy(k);
            exit(0);
        }
        else if (k->port_fl != 0) {
            u = 0;
            vr = 0;
            while (mes->path[u] != ' ' && mes->path[u] != '\0') {
                vr = vr * 10 + mes->path[u] - '0';
                u++;
            }
            if (vr == k->id) {
                u++;
                i = u;
                vr = 0;
                while(i < strlen(mes->path)) {
                    th[vr] = mes->path[i];
                    i++;
                    vr++;
                }
                send(k->r_fl, "remove", mes->id, th, "", -1);
                i = 0;
                while(i < strlen(th)) {
                    th[i] = '\0';
                    i++;
                }
```

```
                }
        }
}
else if (strcmp(mes->action,"create") == 0) {
        vr = 0;
        u = 0;
        while (mes->path[u] != ' ' && mes->path[u] != '\0') {
                vr = vr * 10 + mes->path[u] - '0';
                u++;
        }
        if (vr == k->id) {
                if (strlen(mes->path) == u) {
                        knot_add(k, mes->id);
                }
                else if (k->port_fl != 0) {
                        u++;
                        i = u;
                        vr = 0;
                        while(i < strlen(mes->path)) {
                                th[vr] = mes->path[i];
                                i++;
                                vr++;
                        }
                        send(k->r_fl, "create", mes->id, th, "", -1);
                        i = 0;
                        while (i < strlen(th)) {
```

```
                                        th[i] = '\0';
                                        i++;
                                    }
                                }
                            }
                            free(mes);
                    }}
            return 0;
}
```

server.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
#include "knot.h"
volatile sig_atomic_t flag = 0;
void block_func(int sig)
{   int i;
   printf("Write what you want:\n0|Block/Unblock server\n1|Close server\nYour choice:");
```

```
    scanf("%d", &i);
    if(!i){
      if(flag){
        printf("Server blocked!\n");
        flag = 0;
      }else{
        printf("Server unblocked!\n");
        flag = 1;
      }
      signal(SIGINT, block_func);
      return;
    }else{
      printf("Close server!\n");
      exit(0);
    }
}
typedef struct _DB {
    int id;
    struct _DB* l;
    struct _DB* r;
} db;
db* db_create() {
            db* a = malloc(sizeof(db));
            if (!a) {
                    fprintf(stderr, "ERROR: no memory\n");
        exit(-1);
```

```
    }
    a->id = -1;
    a->r = a->l = NULL;
    return a;
}
char* db_find(db* t, int a, char* path) {
            int i;
            if (t == NULL) {
        return NULL;
    }
    else {
            if (t->id == a) {
            char vr[5];
            sprintf(vr,"%d",t->id);
            path = strcat(path,vr);
            path[strlen(path)] = '\0';
            return path;
        }
        else {
            char vr[5];
            sprintf(vr,"%d",t->id);
            char* c1 = malloc(sizeof(255));
            strcpy(c1,path);
            i = 0;
            int j = strlen(c1);
            while (i < strlen(vr)) {
```

```
                c1[j] = vr[i];
                i++;
                j++;
        }
        c1 = strcat(c1," ");
        char* c2 = malloc(sizeof(255));
        strcpy(c2,path);
        c1 = db_find(t->l, a, c1);
        c2 = db_find(t->r, a, c2);
        if (c1 == NULL) {
                free(c1);
                return c2;
        }
        else {
                free(c2);
                return c1;
        }}}}
db* db_find_k(db* t, int a) {
        if (t == NULL) {
    return NULL;
   }
   else {
    if (t->id == a) {
        return t;
    }
    else {
```

```
        db* b = t;
        while (b->r != NULL) {
            if (b->r->id == a) {
                return b->r;
            }
            else {
                b = b->r;
            }
        }
        db* c1 = db_find_k(t->l, a);
        db* c2 = db_find_k(t->r, a);
        if (c1 != NULL) {
                return c1;
        }
        else {
                return c2;
        }
    }
  }
}

void db_add(db* t, int id, int p) {
        db* b = db_find_k(t, p);
        db* a = malloc(sizeof(db));
        if (!a) {
                fprintf(stderr, "ERROR: no memory\n");
```

```
        exit(-1);
    }
    a->id = id;
    a->r = a->l = NULL;
    if (b->l == NULL) {
            b->l = a;
    }
    else {
            b = b->l;
            while (b->r != NULL) {
                    b = b->r;
            }
            b->r = a;
    }}
void del_br(db* t, db* lost) {
            db* r = lost;
            if (lost->r != NULL) {
                    while (r->r != NULL ) {
                            r = r->r;
                    }
                    r->r = t;
            }
            else {
                    lost->r = t;
            }
}
```

```
void db_del_k(db* t, int p, db* lost) {
        if (t != NULL) {
                db* b = t;
                db* y = NULL;
                while (b->r != NULL) {
                        if (b->r->id == p) {
                                y = b->r->r;
                                del_br(b->r->l, lost);
                                b->r = y;
                                break;
                        }
                        b = b->r;
                }
                if (t->l != NULL) {
                        if (t->l->id == p) {
                                y = b->l->r;
                                del_br(b->l->l, lost);
                                b->l = y;
                        }
                        else {
                                db_del_k(t->l, p, lost);}}}}


void print_tree (db* t, int i) {
   if (t != NULL) {
      for (int j = 0; j < i; j++) {
         printf(" ");
```

```
        }
        printf("%d\n", t->id);
        print_tree(t->l, i+1);
        print_tree(t->r, i);
    }}

int main(int argc, char * argv[]) //ввод имя файла с командами
{
        int fp, vr, u;
        int n,i,j, p, va;
        char* th = malloc(255);
        if ((fp = open(argv[1], O_RDONLY)) < 0) {
                printf("Cannot open file.\n");
                exit(1);
        }
        knot* k = malloc(sizeof(knot));
        db* lost = malloc(sizeof(knot));
        lost->id = -1;
        k->id = -1;
        char name[20];
        for (i = 0; i < 20; i++) {
                name[i] = '\0';
        }
        char* path = NULL;
        db* work = db_create();
        char info[20],c;
```

```
char* action = malloc(10);
while (i < 10) {
        action[i] = '\0';
        i++;
}
i = 0;
while(i < strlen(th)) {
        th[i] = '\0';
        i++;
}
while (1) {
        //signal(SIGINT, block_func);
        //if (!flag) {
                i = 0;
                do {
                        if((n = read(fp,&c, 1)) > 0) {
                        info[i] = c;
                        i++;
                        }
                        else {
                                printf("No more commands: %d\n",n);
                                knot_destroy_brahch(k);
      close(fp);
                                exit(0);
                        }
                } while (c != '\n' && c != '\0');
```

```
info[i - 1] = '\0';
i = 0;
while (info[i] != ' ' && i < strlen(info)) {
        action[i] = info[i];
        i++;
}
action[i] = '\0';
printf("\n read new command: %s\n", info);
i++;
// типы команд:
// create id [parent]
// remove id
// exec id name value
// heartbit time
if (strcmp(action,"exec") == 0) {
        j = 0;
        while (info[i] != ' ' && i < strlen(info)) {
                j = j * 10 + info[i] - '0';
                i++;
        }
        i++;
        path = db_find(work, j, "");
        if (path != NULL) {
                u = 0;
                while (path[u] != ' ') {
                        u++;
```

```
}
u++;
vr = 0;
while(u < strlen(path)) {
        th[vr] = path[u];
        u++;
        vr++;
}
u = 0;
while (info[i] != ' ' && i < strlen(info)) {
        name[u] = info[i];
        i++;
        u++;
}
name[u] = '\0';
p = 0;
i++;
if (i < strlen(info)) {
        while (info[i] != '\0') {
                p = p * 10 + info[i] - '0';
                i++;
        }
        send(k->r_fl, "exec", j, th, name, p);
}
else {
        send(k->r_fl, "exec", j, th, name, -1);
```

```
                }
        }
        else {
                printf("Error: Not found\n");
        }
        for (i = 0; i <= strlen(name); i++) {
                                name[i] = '\0';
        }
}
else if (strcmp(action,"remove") == 0) {
        j = 0;
        while (info[i] != '\0') {
                j = j * 10 + info[i] - '0';
                i++;
        }
        i++;
        if (j == -29) {
                printf("Ok:%d:removed\n", k->id);
                knot_destroy_brahch(k);
                close(fp);
                exit(0);
        }
        path = db_find(work, j,"");
                if (path != NULL) {
                        u = 0;
                        while (path[u] != ' ') {
```

```
                        u++;
                    }
                    i = u + 1;
                    while(i < strlen(path)) {
                        th[i - u - 1] = path[i];
                        i++;
                    }
                send(k->r_fl, "remove", j, th, "", -1);
                for (i = 0; i < strlen(th); i++) {
                        th[i] = '\0';
                    }
                db_del_k(work, j, lost);
                print_tree (work, 0);
            }
            else {
                printf("Error: Not found\n");
            }
}
else if (!strcmp(action,"ping")) {
            j = 0;
            while (info[i] != ' ' && info[i] != '\0') {
                j = j * 10 + info[i] - '0';
                i++;
            }
            if (db_find(work, j, "") != NULL) {
                printf("Ok: 1\n");
```

```
            }
            else if (db_find(lost, j, "") != NULL) {
                printf("Ok: 0\n");
            }
            else {
                printf("Error: Not found\n");
            }
        }
        else if (strcmp(action,"create") == 0) {
            j = 0;
            while (info[i] != ' ' && info[i] != '\0') {
                j = j * 10 + info[i] - '0';
                i++;
            }
            i++;
            if (db_find(work, j, "") != NULL) {
                printf("Error: Already exists\n");
            } else {
                p = 0;
                while (info[i] != '\0') {
                    p = p * 10 + info[i] - '0';
                    i++;
                }
                i++;
            if (p == -29) {
                db_add(work, j, -1);
```

```
                printf("Tree: \n");
                print_tree(work, 0);
                knot_add(k, j);
        }
        else {
                //нормально составить путь
                path = db_find(work, p,"");
                if (path != NULL) {
                        db_add(work, j, p);
                        u = 0;
                        while (path[u] != ' ') {
                                u++;
                        }
                        i = u + 1;
                        while(i < strlen(path)) {
                                th[i - u - 1] = path[i];
                                i++;
                        }
                        send(k->r_fl, "create", j, th, "", -1);
                        for (i = 0; i < strlen(th); i++) {
                                th[i] = '\0';
                        }
                        print_tree (work, 0);
                }
                else {
                        printf("Error: Parent not found\n");
```

```
                                        }
                                }
                        }
                        }
                i = 0;
                path = NULL;
                while (i < strlen(info)) {
                        info[i] = '\0';
                        i++;
                }
                i = 0;
                while (i < strlen(action)) {
                        action[i] = '\0';
                        i++;}}}
```

vocabulary.h

```
#ifndef _VOCABULARY_H_
#define _VOCABULARY_H_
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include <inttypes.h>
typedef char* ID;
typedef struct _word {
        ID name;
```

```
    int value;
} word; // слово словаря
typedef struct _v {
        word head;
        struct _v* next;
} voc;
voc* voc_create(void);
void voc_add_w(voc* v, ID n, int val);
void voc_print(voc* v);
void voc_destroy(voc* v);
int voc_get_val(voc* v);
word* voc_find(voc* v, ID n);
#endif
```

vocabulary.c

```
#include "vocabulary.h"

voc* voc_create(void) {
    voc* v = malloc(sizeof(voc));
    if (!v) {
        fprintf(stderr, "ERROR: no memory\n");
        exit(-1);
    }
    v->head.name = "start";
    v->head.value = 0;
    v->next = NULL;
    return v;
```

```
}

word* voc_find(voc* v, ID n) {
    while (v != NULL) {
    if (strcmp(v->head.name, n) == 0) {
            return &(v->head);
        }
        else {
            v = v->next;
        }
    }
    return NULL;
}

void voc_add_w(voc* v, ID n, int val) {
    if (strcmp(v->head.name, n) == 0) {
        v->head.value = val;
    }
    else if (v->next == NULL) {
            v->next = malloc(sizeof(voc));
        v->next->head.name = n;
        v->next->head.value = val;
        v->next->next = NULL;
    }
    else {
        voc_add_w(v->next, n, val);
```

```
    }
}

void voc_print(voc* v) {
        if (v != NULL) {
        printf("%s %d\n", v->head.name, v->head.value);
        voc_print(v->next);
        }
}

void voc_destroy(voc* v) {
    if (v->next != NULL) {
        voc_destroy(v->next);
    }
    free(v);
}

int voc_get_val(voc* v) {
    if (v != NULL) {
    return v->head.value;
    }
    else {
        return -1;
    }
}
```

4. Вывод

Наиболее важные аспекты работы с ZeroMQ:

Очереди

+ отправляемые в ZeroMQ сообщения попадают во внутреннюю очередь, что позволяет не дожидаться окончания отправки, а в случае исходящего соединения — не имеет значения, установлено оно или нет. Размер очереди может меняться.

+ существует также очередь на прием. В случае входящего соединения, вы получаете сообщения из общей очереди на прием для всех клиентов.

— нельзя управлять очередями — очищать, считать фактический размер, и т.д.

— в случае переполнения очереди, новые сообщения отбрасываются

Сообщения

+ В ZeroMQ работа не с потоком байт, а с отдельными сообщениями, длина которых известна.

+ Сообщение в ZeroMQ состоит из одного или нескольких т.н. «фреймов», что довольно удобно — можно по мере прохождения сообщения по узлам добавлять/удалять фреймы с метаинформацией, не трогая фрейма с данными. Такой подход, в частности, используется в сокете типа ZMQ_ROUTER — ZeroMQ при приеме сообщения автоматически добавляет первым фреймом идентификатор клиента, от которого оно получено.

+ Каждое сообщение атомарно, т.е. всегда будет получено или передано полностью, включая все фреймы.

Сервер сообщений это удобнейший способ организации асинхронной работы приложения. Существует фильтрация сообщений, очередь упорядоченно доносит до получателя информацию (команды).

У меня главная ошибка состояла в том, что я следила за тем, чтобы не поменять порт у сокета PUB, но случайно обновляла контекст. Кроме того, после проделанной работы, я понимаю, что удобно хранить “схематично” всю архитектуру подпроцессов, чтобы планировать работу, а не сразу искать внутри этой архитектуры.