Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №5-7 по курсу**

**«Операционные системы»**

Студент: Чапалда Мария Олеговна

Группа: М8О–201Б–22

Вариант: 29

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2023

**Постановка задачи**

## Цель работы

Приобретение практических навыков в:

* Управлении серверами сообщений (№5)
* Применение отложенных вычислений (№6)
* Интеграция программных систем друг с другом (№7)

## Задание

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать два 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. Список основных поддерживаемых команд:

* Создание нового вычислительного узла.
* Удаление существующего вычислительного узла.
* Исполнение команды на вычислительном узле.
* Проверка доступности вычислительного узла.

Вариант 29. Все вычислительные узлы хранятся в бинарном дереве поиска. Исполнение программы – подсчет суммы n чисел. Команда проверки – узел начинает сообщать раз в time миллисекунд о том, что он работоспособен.

**Общие сведения о программе**

Программа распределительного узла компилируется из файла main.c, программа вычислительного узла компилируется из файла node.c. Во время работы программы используется библиотека для работы с сервером сообщений ZeroMQ. Также используются следующие системные вызовы:

* **fork()** –– создает новый процесс, который является копией родительского процесса, за исключением разных process ID и parent process ID. В случае успеха fork() возвращает 0 для ребенка, число больше 0 для родителя – child ID, в случае ошибки возвращает -1.
* **exec()** — используется для выполнения другой программы. Эта другая программа, называемая процессом-потомком (child process), загружается поверх программы, содержащей вызов exec. Имя файла, содержащего процесс-потомок, задано с помощью первого аргумента.
* **zmq\_ctx\_new()** –– создает новый контекст ZMQ.
* **zmq\_connect()** — создает входящее соединение на сокет.
* **zmq\_disconnect()** — отсоединяет сокет от заданного endpoint.
* **zmq\_socket()** — создает ZMQ сокет.
* **zmq\_close()** — закрывает ZMQ сокет.
* **zmq\_ctx\_destroy()** –– уничтожает контекст ZMQ.

**Общий метод и алгоритм решения**

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы с ZMQ.

2. Проработать принцип общения между клиентскими узлами и между первым клиентом и сервером и алгоритм выполнения команд клиентами.

3. Реализовать необходимые функции-обертки над вызовами функций библиотеки ZMQ.

4. Написать программу сервера и клиента.

**Основные файлы программы**

====================== main.c ======================

#include <stdio.h>

#include <pthread.h>

#include "zmq\_tools.h"

int NODES[MAX\_NODES];

int tree[MAX\_TREE\_SIZE];

typedef struct {

const int\* count;

useconds\_t timeout;

} ping\_token;

void\* pinging(void\* arg);

void start\_ping\_all(ping\_token\* token);

void stop\_all\_nodes(char\* addr, message token, int count);

void send\_receive\_msg(message\* token);

void node\_append(int value);

int find\_creator(int id);

int split\_copy(const char\* text, char\* dest, int index);

bool node\_exist(int value, int count);

int main (int argc, char const \*argv[]) {

bool heartbit = false;

int nodes\_count = 0;

char query\_line [MAX\_LEN];

char query\_word[MAX\_LEN];

char query\_str\_int[MAX\_LEN];

char addr[MAX\_LEN] = SERVER\_SOCKET\_PATTERN;

printf("[%d] started\n", getpid());

message token = {create, 0, 0, ""};

ping\_token ping\_arg = {NULL, 1};

for (int i = 0; i < MAX\_TREE\_SIZE; ++i) {

tree[i] = -1;

}

while (fgets(query\_line, MAX\_LEN, stdin) != NULL) {

if (split\_copy(query\_line, query\_word, 0) == 0) {

printf("\tbad command\n");

continue;

}

if (split\_copy(query\_line, query\_str\_int, 1) == 0) {

printf("\tbad command\n");

continue;

}

long int query\_int = strtol(query\_str\_int, NULL, 10);

if(query\_int == LONG\_MAX || query\_int < 0) {

printf("\tbad command's arg: it's too large or negative\n");

continue;

}

if (strcmp(query\_word, "create") == 0) {

if (nodes\_count == MAX\_NODES) {

printf("\tyou cannot create more than %d nodes\n", MAX\_NODES);

continue;

}

if (query\_int > 9999) {

printf("\tbad command's arg: it's too large\n");

continue;

}

int id\_process = (int)query\_int + MIN\_ADDR;

if (node\_exist(id\_process, nodes\_count)) {

printf("\tthis node was created earlier\n");

continue;

}

int tree\_index = find\_creator(id\_process);

if (tree\_index != 0) {

int parent\_to\_create\_proc = tree[tree\_index];

if (ping\_process(parent\_to\_create\_proc)) {

token.cmd = create;

token.dest\_id = parent\_to\_create\_proc;

token.value = id\_process;

printf("\task %d to create a new node\n", parent\_to\_create\_proc - MIN\_ADDR);

send\_receive\_msg(&token);

node\_append(id\_process);

NODES[nodes\_count] = id\_process;

++nodes\_count;

} else {

printf("\tcannot create a new node: parent is not available\n");

continue;

}

} else {

printf("\tI'm creating %d\n", id\_process - MIN\_ADDR);

memset(query\_str\_int, 0, MAX\_LEN);

sprintf(query\_str\_int, "%d", id\_process);

char \*Child\_argv[] = {"node", query\_str\_int, NULL};

int pid = fork();

if (pid == -1) {

printf("\tfork error\n");

return 1;

}

if (pid == 0) {

execv("node", Child\_argv);

return 0;

}

NODES[nodes\_count] = id\_process;

node\_append(id\_process);

++nodes\_count;

}

} else if (strcmp(query\_word, "exec") == 0) {

if (query\_int > 9999) {

printf("\tbad command's arg: it's too large\n");

continue;

}

int id\_process = (int)query\_int + MIN\_ADDR;

if (node\_exist(id\_process, nodes\_count) && ping\_process(id\_process)) {

clear\_token(&token);

fgets(token.str, MAX\_LEN, stdin);

token.str[strlen(token.str) - 1] = '\0';

token.cmd = exec;

token.dest\_id = id\_process;

send\_receive\_msg(&token);

} else {

printf("\t[%d] node hasn't connection\n", id\_process - MIN\_ADDR);

}

} else if (strcmp(query\_word, "remove") == 0) {

if (query\_int > 9999) {

printf("\tbad command's arg: it's too large\n");

continue;

}

int id\_process = (int)query\_int + MIN\_ADDR;

if (node\_exist(id\_process, nodes\_count) && ping\_process(id\_process)) {

token.cmd = delete;

token.dest\_id = id\_process;

send\_receive\_msg(&token);

} else {

printf("\t[%d] node hasn't connection\n", id\_process - MIN\_ADDR);

}

} else if (strcmp(query\_word, "heartbit") == 0) {

if (heartbit == true) continue;

ping\_arg.count = &nodes\_count;

ping\_arg.timeout = (useconds\_t) query\_int;

start\_ping\_all(&ping\_arg);

heartbit = true;

} else {

printf("\tbad command. Please, try again\n");

}

memset(query\_line, 0, MAX\_LEN);

memset(query\_word, 0, MAX\_LEN);

memset(query\_str\_int, 0, MAX\_LEN);

}

stop\_all\_nodes(addr, token, nodes\_count);

return 0;

}

int split\_copy(const char\* text, char\* dest, int index) {

int i = 0;

for (int j = 0; j < index; ++j) {

while (text[i] != ' ' && text[i] != '\0' && text[i] != '\n') {

++i;

}

if (text[i] == ' ') {

++i;

}

}

int k = i;

while (text[i] != ' ' && text[i] != '\0' && text[i] != '\n') {

dest[i - k] = text[i];

++i;

}

dest[i - k] = '\0';

return i - k;

}

bool node\_exist(int value, int count) {

for (int i = 0; i < count; ++i) {

if (NODES[i] == value) return true;

}

return false;

}

void stop\_all\_nodes(char\* addr, message token, int count) {

void \*context = zmq\_ctx\_new();

void \*requester = create\_zmq\_socket(context, ZMQ\_REQ);

for (int i = 0; i < count; ++i) {

token.cmd = delete;

if (ping\_process(NODES[i])) {

reconnect\_zmq\_socket(requester, NODES[i], addr);

send\_msg\_wait(requester, &token);

receive\_msg\_wait(requester, &token);

}

}

close\_zmq\_socket(requester);

destroy\_zmq\_context(context);

}

void\* pinging(void\* arg) {

ping\_token token = \*((ping\_token\*) arg);

int nodes\_states[MAX\_NODES];

memset(nodes\_states, 0, sizeof(nodes\_states));

while (1) {

for (int i = 0; i < \*token.count; ++i) {

if (ping\_process(NODES[i]) == false) {

++nodes\_states[i];

if (nodes\_states[i] == 4) {

printf("Heartbit: node %d is unavailable\n", NODES[i] - MIN\_ADDR);

} else if (nodes\_states[i] > 4) {

nodes\_states[i] = 404;

}

} else {

nodes\_states[i] = 0;

}

}

usleep(token.timeout);

}

return NULL;

}

void start\_ping\_all(ping\_token\* token) {

pthread\_t th;

if (pthread\_create(&th, NULL, &pinging, token) != 0) {

printf("cannot create thread\n");

return;

}

}

void\* send\_receive\_msg\_thread(void\* arg) {

int id = ((message \*)arg)->dest\_id;

void \*context = zmq\_ctx\_new();

void \*requester = create\_zmq\_socket(context, ZMQ\_REQ);

char addr[MAX\_LEN];

create\_addr(addr, id, tcp\_serv);

connect\_zmq\_socket(requester, addr);

send\_msg\_wait(requester, (message\*) arg);

receive\_msg\_wait(requester, (message\*) arg);

if (((message \*)arg)->cmd != success){

printf("\t node cannot run your query\n");

}

close\_zmq\_socket(requester);

destroy\_zmq\_context(context);

return NULL;

}

void send\_receive\_msg(message \* token) {

pthread\_t th;

if (pthread\_create(&th, NULL, &send\_receive\_msg\_thread, token) != 0) {

printf("\tcannot create thread\n");

return;

}

}

void node\_append(int value) {

int i = 1;

while (i < MAX\_TREE\_SIZE) {

if (tree[i] == -1) {

tree[i] = value;

return;

} else {

if (value < tree[i]) {

i = i \* 2;

} else {

i = i \* 2 + 1;

}

}

}

printf("\t node cannot be added to the tree\n");

}

int find\_creator(int id) {

int i = 1;

while (i < MAX\_TREE\_SIZE) {

if (tree[i] == -1) {

return i/2;

} else {

if (id < tree[i]) {

i = i \* 2;

} else {

i = i \* 2 + 1;

}

}

}

return 0;

}

====================== node.c ======================

#include "zmq\_tools.h"

void execution(message\* token) {

char\* str = token->str;

char\* cur = str;

long long int sum = 0;

while (cur != str + strlen(str)) {

long int x = strtol(cur, &cur, 10);

if (x == LONG\_MAX || x == LONG\_MIN) {

return;

}

while (\*cur ==' ' || \*cur == '\n'){

cur++;

}

sum += x;

}

printf("\t[%d] sum = %lld\n", getpid(), sum);

token->cmd = success;

}

int main(int argc, char const \*argv[]) {

if (argc < 2) {

printf("\t[%d] argv error\n", getpid());

return 1;

}

void \*context = create\_zmq\_context();

void \*responder = create\_zmq\_socket(context, ZMQ\_REP);

char adr[MAX\_LEN] = TCP\_SOCKET\_PATTERN;

strcat(adr, argv[1]);

printf("\t[%d] has been created\n", getpid());

bind\_zmq\_socket(responder, adr);

while (1) {

message token;

receive\_msg\_wait(responder, &token);

int id\_process;

char query\_str\_int[MAX\_LEN];

switch (token.cmd) {

case delete:

token.cmd = success;

printf("\t[%d] has been destroyed\n", getpid());

send\_msg\_wait(responder, &token);

close\_zmq\_socket(responder);

destroy\_zmq\_context(context);

return 0;

case create:

id\_process = token.value;

memset(query\_str\_int, 0, MAX\_LEN);

sprintf(query\_str\_int, "%d", id\_process);

char \*Child\_argv[] = {"node", query\_str\_int, NULL};

int pid = fork();

if (pid == -1) {

return 1;

}

if (pid == 0) {

execv("node", Child\_argv);

return 0;

} else { // parent

token.cmd = success;

send\_msg\_wait(responder, &token);

}

break;

case exec:

execution(&token);

send\_msg\_wait(responder, &token);

break;

default:

token.cmd = success;

send\_msg\_wait(responder, &token);

close\_zmq\_socket(responder);

destroy\_zmq\_context(context);

return 0;

}

}

}

====================zmq\_tools.c =====================

# #include "zmq\_tools.h"

void create\_addr(char\* addr, int id, addr\_pattern pattern) {

char str[MAX\_LEN];

memset(str, 0, MAX\_LEN);

sprintf(str, "%d", id);

memset(addr, 0, MAX\_LEN);

switch (pattern) {

case tcp\_serv:

memcpy(addr, SERVER\_SOCKET\_PATTERN, sizeof(SERVER\_SOCKET\_PATTERN));

break;

case tcp\_node:

memcpy(addr, TCP\_SOCKET\_PATTERN, sizeof(TCP\_SOCKET\_PATTERN));

break;

case inproc:

memcpy(addr, PING\_SOCKET\_PATTERN, sizeof(PING\_SOCKET\_PATTERN));

break;

default:

fprintf(stderr, "[%d] ", getpid());

perror("ERROR create\_addr wrong argument: addr pattern");

exit(1);

}

strcat(addr, str);

}

void clear\_token(message\* msg) {

msg->cmd = delete;

msg->dest\_id = 0;

msg->value = 0;

memset(msg->str, 0, MAX\_LEN);

}

void\* create\_zmq\_context() {

void\* context = zmq\_ctx\_new();

if (context == NULL) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_ctx\_new ");

exit(ERR\_ZMQ\_CTX);

}

return context;

}

void bind\_zmq\_socket(void\* socket, char\* endpoint) {

if (zmq\_bind(socket, endpoint) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_bind ");

exit(ERR\_ZMQ\_BIND);

}

}

void disconnect\_zmq\_socket(void\* socket, char\* endpoint) {

if (zmq\_disconnect(socket, endpoint) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_disconnect ");

exit(ERR\_ZMQ\_DISCONNECT);

}

}

void connect\_zmq\_socket(void\* socket, char\* endpoint) {

if (zmq\_connect(socket, endpoint) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_connect ");

exit(ERR\_ZMQ\_CONNECT);

}

}

void\* create\_zmq\_socket(void\* context, const int type) {

void\* socket = zmq\_socket(context, type);

if (socket == NULL) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_socket ");

exit(ERR\_ZMQ\_SOCKET);

}

return socket;

}

void reconnect\_zmq\_socket(void\* socket, int to, char\* addr) {

if (addr[16] != '\0') {

disconnect\_zmq\_socket(socket, addr);

}

create\_addr(addr, to, tcp\_serv);

connect\_zmq\_socket(socket, addr);

}

void close\_zmq\_socket(void\* socket) {

if (zmq\_close(socket) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_close ");

exit(ERR\_ZMQ\_CLOSE);

}

}

void destroy\_zmq\_context(void\* context) {

if (zmq\_ctx\_destroy(context) != 0) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_ctx\_destroy ");

exit(ERR\_ZMQ\_CLOSE);

}

}

void receive\_msg\_wait(void\* socket, message\* token) {

zmq\_msg\_t reply;

zmq\_msg\_init(&reply);

if (zmq\_msg\_recv(&reply, socket, 0) == -1) {

zmq\_msg\_close(&reply);

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_recv ");

exit(ERR\_ZMQ\_MSG);

}

(\*token) = \* ((message\*) zmq\_msg\_data(&reply));

if (zmq\_msg\_close(&reply) == -1) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_close ");

exit(ERR\_ZMQ\_MSG);

}

}

void send\_msg\_wait(void\* socket, message\* token) {

zmq\_msg\_t msg;

zmq\_msg\_init(&msg);

if (zmq\_msg\_init\_size(&msg, sizeof(message)) == -1) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_init ");

exit(ERR\_ZMQ\_MSG);

}

if (zmq\_msg\_init\_data(&msg, token, sizeof(message), NULL, NULL) == -1) {

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_init ");

exit(ERR\_ZMQ\_MSG);

}

if (zmq\_msg\_send(&msg, socket, 0) == -1) {

zmq\_msg\_close(&msg);

fprintf(stderr, "[%d] ", getpid());

perror("ERROR zmq\_msg\_send ");

exit(ERR\_ZMQ\_MSG);

}

}

bool ping\_process(int id) {

char addr\_monitor[MAX\_LEN];

char addr\_connection[MAX\_LEN];

create\_addr(addr\_connection, id, tcp\_serv);

create\_addr(addr\_monitor, id, inproc);

void\* context = zmq\_ctx\_new();

void\* requester = zmq\_socket(context, ZMQ\_REQ);

zmq\_socket\_monitor(requester, addr\_monitor, ZMQ\_EVENT\_CONNECTED | ZMQ\_EVENT\_CONNECT\_RETRIED);

void\* socket = zmq\_socket(context, ZMQ\_PAIR);

zmq\_connect(socket, addr\_monitor);

zmq\_connect(requester, addr\_connection);

zmq\_msg\_t msg;

zmq\_msg\_init(&msg);

zmq\_msg\_recv(&msg, socket, 0);

uint8\_t\* data = (uint8\_t\*)zmq\_msg\_data(&msg);

uint16\_t event = \*(uint16\_t\*)(data);

zmq\_close(requester);

zmq\_close(socket);

zmq\_msg\_close(&msg);

zmq\_ctx\_destroy(context);

if (event == ZMQ\_EVENT\_CONNECT\_RETRIED) {

return false;

} else {

return true;

}

}

====================zmq\_tools.h =====================

#ifndef OS\_LAB6\_ZMQ\_TOOLS\_H

#define OS\_LAB6\_ZMQ\_TOOLS\_H

#include <stdbool.h>

#include <string.h>

#include <unistd.h>

#include <stdlib.h>

#include <errno.h>

#include <zmq.h>

#include <limits.h>

#define MAX\_LEN 128

#define MAX\_NODES 50

#define MAX\_TREE\_SIZE 512

#define ERR\_ZMQ\_CTX 100

#define ERR\_ZMQ\_SOCKET 101

#define ERR\_ZMQ\_BIND 102

#define ERR\_ZMQ\_CLOSE 103

#define ERR\_ZMQ\_CONNECT 104

#define ERR\_ZMQ\_DISCONNECT 105

#define ERR\_ZMQ\_MSG 106

#define SERVER\_SOCKET\_PATTERN "tcp://localhost:"

#define PING\_SOCKET\_PATTERN "inproc://ping"

#define TCP\_SOCKET\_PATTERN "tcp://\*:"

#define MIN\_ADDR 5555

typedef enum {

create,

delete,

exec,

success

} cmd\_type;

typedef enum {

tcp\_serv,

tcp\_node,

inproc

} addr\_pattern;

typedef struct {

cmd\_type cmd;

int dest\_id;

int value;

char str[MAX\_LEN];

} message;

void clear\_token(message\* msg);

void create\_addr(char\* addr, int id, addr\_pattern);

void bind\_zmq\_socket(void\* socket, char\* endpoint);

void\* create\_zmq\_context();

void\* create\_zmq\_socket(void\* context, int type);

void connect\_zmq\_socket(void\* socket, char\* endpoint);

void disconnect\_zmq\_socket(void\* socket, char\* endpoint);

void reconnect\_zmq\_socket(void\* socket, int to, char\* addr);

void close\_zmq\_socket(void\* socket);

void destroy\_zmq\_context(void\* context);

void receive\_msg\_wait(void\* socket, message\* token);

void send\_msg\_wait(void\* socket, message\* token);

bool ping\_process(int id);

#endif //OS\_LAB6\_ZMQ\_TOOLS\_H

======================== test1.txt ========================

create 10

create 10

create 11

create 15

create 9

remove 11

remove 11

exec 11

exec 15

1 1 1 1 1 50 1000000

exec 10

1000 10 1000000 1000000000000

======================== test2.txt ========================

create 5

create 6

create 4

create 1

heartbit 3000000

exec 10

remove 5

remove 6

heartbit 3000000

======================== test3.txt ========================

exec 3

create 3

create 5

create 10

remove 3

exec 10

1000000 102 1029310231 120193091

**Вывод**

Во время выполнения лабораторной работы я реализовала определенную систему по асинхронной обработке запросов. В программе используется протокол передачи данных через tcp, в котором общение между процессами происходит через определенные порты. Обмен происходит посредством функций библиотеки ZMQ.