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: Ползикова Алина Владимировна
Группа: М8О–208Б–21
Вариант: 27
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Постановка задачи

Цель работы

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 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 27. 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

Пример работы

```
LAPTOP-9UGJH447:~/OS/lab6_var27$ ./main < test1.txt
started
I'm creating 10
this node was created earlier
[20303] has been created
cannot create a new node: parent is not available
ask 10 to create a new node
ask 10 to create a new node
[11] node hasn't connection
[11] node hasn't connection
[11] node hasn't connection
[15] node hasn't connection
bad command. Please, try again
[10] node hasn't connection
bad command. Please, try again
bad command
bad command
[20315] has been created
[20315] has been destroyed
LAPTOP-9UGJH447:~/OS/lab6_var27$ ./main < test2.txt
started
I'm creating 5
cannot create a new node: parent is not available
cannot create a new node: parent is not available
[20334] has been created
ask 5 to create a new node
[10] node hasn't connection
[6] node hasn't connection
bad command
bad command
[20334] has been destroyed
LAPTOP-9UGJH447:~/OS/lab6_var27$ ./main < test3.txt
started
[3] node hasn't connection
I'm creating 3
cannot create a new node: parent is not available
cannot create a new node: parent is not available
[3] node hasn't connection
[20363] has been created
[10] node hasn't connection
bad command. Please, try again
bad command
bad command
[20363] has been destroyed
LAPTOP-9UGJH447:~/OS/lab6_var27$
```

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.