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»

ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ

Студент: Злобина Валерия Вадимовна
Группа: М8О–208Б–21
Вариант: 20
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022.

Цель работы

Целью является приобретение практических навыков в:

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

Задание

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить
свою работоспособность. 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)
Пример:
> `create 10 5`
Ok: 3128
Примечания: создание нового управляющего узла осуществляется пользователем программы
при помощи запуска исполняемого файла. Id и pid — это разные идентификаторы.

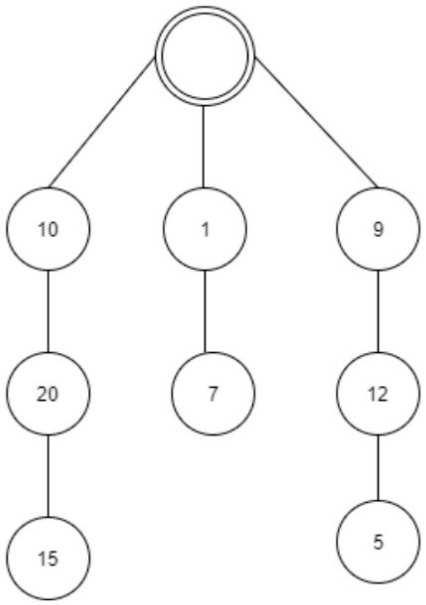
Удаление существующего вычислительного узла

Формат команды: remove id
id – целочисленный идентификатор удаляемого вычислительного узла
Пример:
> `remove 10`
Ok

Исполнение команды на вычислительном узле

Формат команды: exec id [params]
id – целочисленный идентификатор вычислительного узла, на который отправляется команда

Вариант
Топология 1



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

Набор команд 3 (локальный таймер)

Формат команды сохранения значения:
`exec id subcommand`
`subcommand` – одна из трех команд: `start, stop, time.`
`start` – запустить таймер
`stop` – остановить таймер
`time` – показать время локального таймера в миллисекундах

Общие сведения о программе

Программа состоит из 8 файлов:

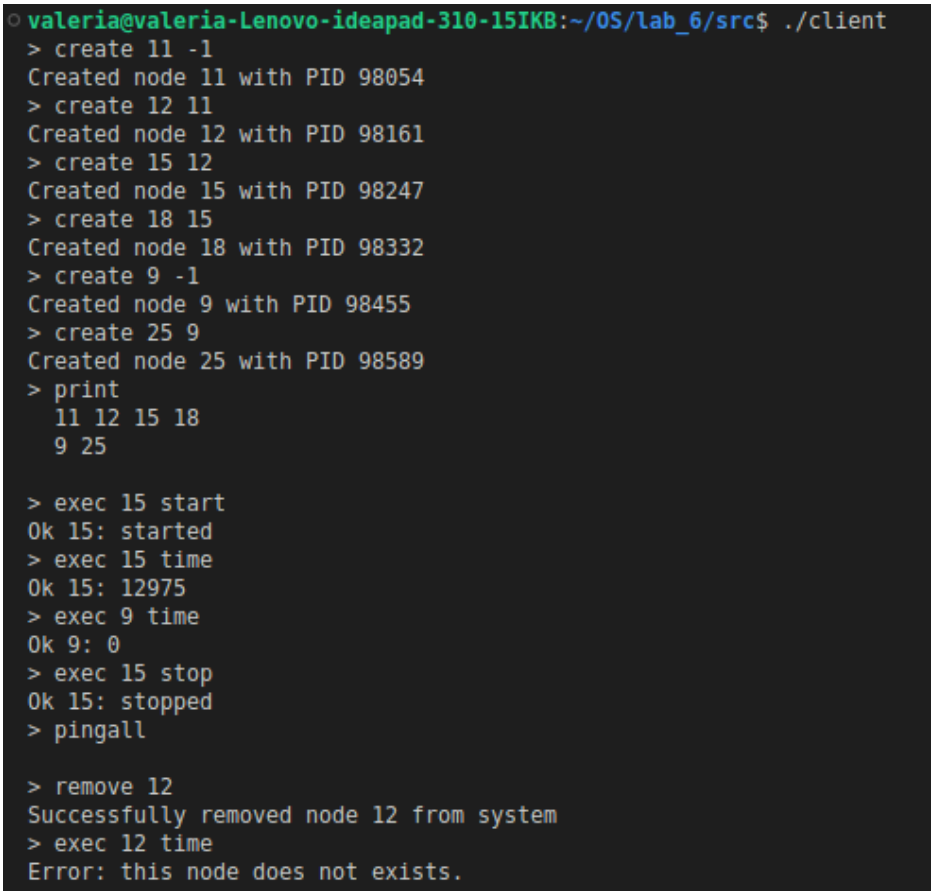
- server.c - получает команды от пользователя и отправляет их в вычислительный узел
- client.c - получает эти команды и выполняет их
- list.c — реализация структуры списка
- interface.c, interface.h - реализация интерфейса взаимодействия с командами
- interprocess.c. interprocess.h - реализация функций межпроцессорного взаимодействия

Компиляция выполняется с помощью Makefile

Общий метод и алгоритм решения:

- create id [parent id] — вставка вычислительного узла в один из списков
- exec id subcommand — отправка подкоманды вычислительному узлу
- remove id — удаление вычислительного узла и всех его дочерних узлов из списка
- pingall — проверка доступности всех узлов

Пример запуска программы:



Исходный код:

client.c

```
#include "interprocess.h"
#define INITIAL_CAPACITY 5

#define CLIENT_ID -1

unsigned child_count = 0;
unsigned capacity = INITIAL_CAPACITY;

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

    List *system = (List *)calloc(sizeof(List), INITIAL_CAPACITY);
```

```
    for (int i = 0; i < INITIAL_CAPACITY; ++i)
        system[i] = create_node(CLIENT_ID);

    void *context = zmq_ctx_new();
    if (context == NULL) {
        perror("context");
        exit(EXIT_FAILURE);
    }

    void **child_sockets = calloc(sizeof(void *), INITIAL_CAPACITY);
    id *child_ids = calloc(sizeof(id), INITIAL_CAPACITY);
    const char *arrow = "> ";
    size_t arrow_len = strlen(arrow);

    while (1) {
        if (write(1, arrow, arrow_len) <= 0) {
            perror("write");
            exit(EXIT_FAILURE);
        }
        command_t current = get_command();
        switch (current) {
        case PRINT:
            for (int i = 0; i <= child_count; ++i) {
                print_list(system[i]);
            }
            break;
        case EXIT:
            for (int i = 0; i < child_count; ++i) {
                send_exit(child_sockets[i]);
            }
            break;
        case CREATE:;
            const char *init_id_str = read_word();
            const char *parent_id_str = read_word();
            int init_id = atoi(init_id_str);
            if (init_id <= 0) {
                printf("Error: invalid node id (id should be an integer greater than 0).\n");
                break;
            }
            if (exists(system, init_id, child_count)) {
                printf("Error: already exists.\n");
                break;
            }
            int parent_id = atoi(parent_id_str);
            if (parent_id <= 0 && parent_id != CLIENT_ID) {
                  printf("Error: invalid parent id (parent should be an integer greater than 0 or %d for
root).\n", CLIENT_ID);
                break;
            }
            if ((parent_id != CLIENT_ID) && (!exists(system, parent_id, child_count))) {
                printf("Error: there is no nodes with id = %d.\n", parent_id);
                break;
            }

            if (parent_id == CLIENT_ID) {
                int fork_val = fork();
                if (fork_val == 0)
                    create_worker(init_id, 0);
                printf("Created node %d with PID %d\n", init_id, fork_val);

                if (child_count >= capacity) {
                    capacity *= 2;
```

```
                child_sockets = realloc(child_sockets, sizeof(void *) * capacity);
                child_ids = realloc(child_ids, sizeof(id) * capacity);
                for (int i = child_count; i < capacity; ++i)
                    system[i] = create_node(CLIENT_ID);
            }
            child_sockets[child_count] = zmq_socket(context, ZMQ_REQ);
            int opt_val = 0;
                    int rc = zmq_setsockopt(child_sockets[child_count], ZMQ_LINGER, &opt_val,
sizeof(opt_val));
            assert(rc == 0);
            if (child_sockets[child_count] == NULL) {
                perror("socket");
                exit(EXIT_FAILURE);
            }

            rc = zmq_connect(child_sockets[child_count], portname_client(BASE_PORT + init_id));
            assert(rc == 0);

            child_ids[child_count] = init_id;
            system[child_count] = add_node(system[child_count], parent_id, init_id);
            child_count++;
            break;
        } else {
            int replied = 0;
            for (int i = 0; i < child_count; ++i) {
                send_create(child_sockets[i], init_id, parent_id);
                if (!available_recive(child_sockets[i])) {
                    continue;
                }
                const char *reply = get_reply(child_sockets[i]);
                if (strcmp(EMPTY_MSG, reply) != 0) {
                    printf("%s\n", reply);
                    replied = 1;
                    break;
                }
            }
            if (replied == 0)
                printf("Node %d seems to be unavailable\n", parent_id);
            else
                for (int i = 0; i < child_count; ++i)
                    system[i] = add_node(system[i], parent_id, init_id);
        }
        break;

    case EXEC:;
        const char *target_id_str = read_word();
        int target_id = atoi(target_id_str);
        if (target_id <= 0) {
            printf("Error: invalid node id (id should be an integer greater than 0).\n");
            break;
        }
        if (!exists(system, target_id, child_count)) {
            printf("Error: this node does not exists.\n");
            break;
        }
        subcommand_t current_sub = get_subcommand();
        if (current_sub == UNDEFINED) {
            printf("Invalid subcommand!\n");
            break;
        }

        int replied = 0;
```

```
            for (int i = 0; i < child_count; ++i) {
                send_exec(child_sockets[i], current_sub, target_id);
                if (!available_recive(child_sockets[i])) {
                    continue;
                }
                char *reply = get_reply(child_sockets[i]);
                if (strcmp(EMPTY_MSG, reply) != 0) {
                    printf("%s\n", reply);
                    replied = 1;
                    break;
                }
            }
            if (replied == 0)
                printf("Node %d seems to be unavailable\n", target_id);

            break;
        case REMOVE:;
            const char *remove_id_str = read_word();
            int remove_id = atoi(remove_id_str);
            if (remove_id <= 0) {
                printf("Error: invalid node id (id should be an integer greater than 0).\n");
                break;
            }
            if (!exists(system, remove_id, child_count)) {
                printf("Error: this node does not exist.\n");
                break;
            }
            int i = in_list(child_ids, remove_id, child_count);
            if (i != -1) {
                shift_id(child_ids, i, child_count);
                send_exit(child_sockets[i]);
                zmq_close(child_sockets[i]);
                shift_void(child_sockets, i, child_count);
                child_count--;
                printf("Successfully removed node %d from system\n", remove_id);
            } else {
                replied = 0;
                for (int i = 0; i < child_count; ++i) {
                    send_remove(child_sockets[i], remove_id);
                    if (!available_recive(child_sockets[i])) {
                        continue;
                    }
                    char *reply = get_reply(child_sockets[i]);
                    if (strcmp(EMPTY_MSG, reply) != 0) {
                        printf("%s\n", reply);
                        replied = 1;
                        break;
                    }
                }
                if (replied == 0)
                        printf("Node %d seems to be unavailable, removed it from tree anyways\n",
remove_id);
            }
            for (int i = 0; i < child_count; ++i)
                system[i] = remove_node(system[i], remove_id);
            break;
        case PINGALL:;
            int not_replied = 0;
            int count = 0;
            char *unavailable = (char *)calloc(sizeof(char), STR_LEN);
            for (int i = 0; i < child_count; ++i) {
                send_pingall(child_sockets[i], count + 1);
```

```
            if (!available_recive_pingall(child_sockets[i], count)) {
                unavailable = strcat(unavailable, int_to_string(child_ids[i]));
                unavailable = strcat(unavailable, " ");
                not_replied++;
                continue;
            }
            char *reply_child = get_reply(child_sockets[i]);
            if (strcmp(EMPTY_MSG, reply_child) != 0) {
                unavailable = strcat(unavailable, reply_child);
                replied = 1;
                break;
            }
        }
        if (not_replied == 0 && replied == 0) {
            printf("Every process is available\n");
        } else {
            fprintf(stderr, "%s\n", unavailable);
        }
        break;
    default:
        printf("Error: invalid command.\n");
        break;
    }
    if (current == EXIT) {
        break;
    }
}

for (int i = 0; i < child_count; ++i) {
    zmq_close(child_sockets[i]);
}
zmq_ctx_destroy(context);

return 0;
}
```

server.c

```
#include <sys/time.h>
#include "interprocess.h"

#define OFF 0
#define ON 1

#define INITIAL_CAPACITY 5

short timer = OFF;

unsigned has_child = 0;

int main(int argc, const char **argv) {
    void *context = zmq_ctx_new();
    if (context == NULL) {
        perror("context");
        exit(EXIT_FAILURE);
    }
    void *parent = zmq_socket(context, ZMQ_REP);
    if (parent == NULL) {
        perror("socket");
        exit(EXIT_FAILURE);
    }
```

```
int self_id = atoi(argv[1]);
int rc = zmq_bind(parent, portname_server(BASE_PORT + self_id));
assert(rc == 0);

void *child_socket;
id child_id;

if (atoi(argv[2]) != 0) {
    child_socket = zmq_socket(context, ZMQ_REQ);
    has_child = 1;
    int opt_val = 0;
    int rc = zmq_setsockopt(child_socket, ZMQ_LINGER, &opt_val, sizeof(opt_val));
    assert(rc == 0);
    if (child_socket == NULL) {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    rc = zmq_connect(child_socket, portname_client(atoi(argv[2])));
    assert(rc == 0);

    child_id = atoi(argv[2]) - BASE_PORT;
}

double diff_sec = 0, diff_msec = 0;

struct timeval start, finish;

while (1) {
    command_t current = UNKNOWN;
    subcommand_t current_sub = UNDEFINED;
    int target_id = 0;
    int parent_id = 0;
    int arg = 0;
    int count = 0;

    while (1) {
        zmq_msg_t part;
        int rec = zmq_msg_init(&part);
        assert(rec == 0);
        rec = zmq_msg_recv(&part, parent, 0);
        assert(rec != -1);

        switch (arg) {
        case 0:
            memcpy(&current, zmq_msg_data(&part), zmq_msg_size(&part));
            break;
        case 1:
            switch (current) {
            case EXEC:
                memcpy(&current_sub, zmq_msg_data(&part), zmq_msg_size(&part));
                break;
            case CREATE:
                memcpy(&target_id, zmq_msg_data(&part), zmq_msg_size(&part));
                break;
            case REMOVE:
                memcpy(&target_id, zmq_msg_data(&part), zmq_msg_size(&part));
                break;
            case PINGALL:
                memcpy(&count, zmq_msg_data(&part), zmq_msg_size(&part));
                break;
            default:
```

```
            break;
        }
        break;
    case 2:
        switch (current) {
        case EXEC:
            memcpy(&target_id, zmq_msg_data(&part), zmq_msg_size(&part));
            break;
        case CREATE:
            memcpy(&parent_id, zmq_msg_data(&part), zmq_msg_size(&part));
            break;
        default:
            break;
        }
        break;
    default:
        printf("UNEXPECTED\n");
        exit(EXIT_FAILURE);
    }
    zmq_msg_close(&part);
    ++arg;
    if (!zmq_msg_more(&part))
        break;
}
if (current == EXIT) {
    if (has_child)
        send_exit(child_socket);
    break;
}

int replied = 0;
int not_replied = 0;
char *reply = (char *)calloc(sizeof(char), STR_LEN);

if (current == CREATE) {
    if (parent_id == self_id) {
        if (has_child == 0) {
            int fork_val = fork();
            if (fork_val == 0)
                create_worker(target_id, 0);

            child_socket = zmq_socket(context, ZMQ_REQ);
            has_child = 1;
            int opt_val = 0;
            int rc = zmq_setsockopt(child_socket, ZMQ_LINGER, &opt_val, sizeof(opt_val));
            assert(rc == 0);
            if (child_socket == NULL) {
                perror("socket");
                exit(EXIT_FAILURE);
            }

            rc = zmq_connect(child_socket, portname_client(BASE_PORT + target_id));
            assert(rc == 0);

            child_id = target_id;

            sprintf(reply, "Created node %d with PID %d", target_id, fork_val);
            replied = 1;
        } else {
            zmq_close(child_socket);
            int fork_val = fork();
            if (fork_val == 0)
```

```
            create_worker(target_id, BASE_PORT + child_id);

            child_socket = zmq_socket(context, ZMQ_REQ);
            has_child = 1;
            int opt_val = 0;
            int rc = zmq_setsockopt(child_socket, ZMQ_LINGER, &opt_val, sizeof(opt_val));
            assert(rc == 0);
            if (child_socket == NULL) {
                perror("socket");
                exit(EXIT_FAILURE);
            }

            rc = zmq_connect(child_socket, portname_client(BASE_PORT + target_id));
            assert(rc == 0);

            child_id = target_id;

            sprintf(reply, "Created node %d with PID %d", target_id, fork_val);
            replied = 1;
        }

    } else {
        if (has_child) {
            send_create(child_socket, target_id, parent_id);
            if (available_recive(child_socket)) {
                const char *reply_child = get_reply(child_socket);
                if (strcmp(EMPTY_MSG, reply_child) != 0) {
                    sprintf(reply, "%s", reply_child);
                    replied = 1;
                }
            }
        }
    }
}

else if (current == EXEC) {

    if (target_id == self_id) {
        switch (current_sub) {
        case START:
            gettimeofday(&start, NULL);
            timer = ON;
            break;
        case STOP:
            if (timer == ON) {
                gettimeofday(&finish, NULL);
                timer = OFF;
                diff_sec = finish.tv_sec - start.tv_sec;
                diff_msec = (diff_sec * 1000) + (finish.tv_usec - start.tv_usec) / 1000;
            }
            break;
        case TIME:
            if (timer == ON) {
                gettimeofday(&finish, NULL);
                diff_sec = finish.tv_sec - start.tv_sec;
                diff_msec = (diff_sec * 1000) + (finish.tv_usec - start.tv_usec) / 1000;
            }
            break;
        default:
            break;
        }
```

```
        char *result = message_prefix(self_id, current_sub);

        if (current_sub == TIME) {
            const char *time_string = int_to_string((unsigned)diff_msec);
            result = strcat(result, time_string);
        }
        strcpy(reply, result);
        replied = 1;
    } else {
        if (has_child) {
            send_exec(child_socket, current_sub, target_id);
            if (available_recive(child_socket)) {
                char *reply_child = get_reply(child_socket);
                if (strcmp(EMPTY_MSG, reply_child) != 0) {
                    sprintf(reply, "%s", reply_child);
                    replied = 1;
                }
            }
        }
    }
}

else if (current == REMOVE) {
    if (target_id == child_id) {
        send_exit(child_socket);
        zmq_close(child_socket);
        has_child = 0;
        sprintf(reply, "Successfully removed node %d from system", target_id);
        replied = 1;
    } else {
        if (has_child) {
            send_remove(child_socket, target_id);
            if (available_recive(child_socket)) {
                char *reply_child = get_reply(child_socket);
                if (strcmp(EMPTY_MSG, reply_child) != 0) {
                    sprintf(reply, "%s", reply_child);
                    replied = 1;
                }
            }
        }
    }
}

else if (current == PINGALL) {

    if (has_child) {
        send_pingall(child_socket, count + 1);
        if (!available_recive_pingall(child_socket, count)) {
            reply = strcat(reply, int_to_string(child_id));
            reply = strcat(reply, " ");
            not_replied++;
        } else {
            char *reply_child = get_reply(child_socket);
            if (strcmp(EMPTY_MSG, reply_child) != 0) {
                reply = strcat(reply, reply_child);
                replied = 1;
            }
        }
    }
}

if (replied == 0 && (current != PINGALL || (current == PINGALL && not_replied == 0)))
```

```
            strcpy(reply, EMPTY_MSG);
        size_t rep_len = strlen(reply) + 1;
        zmq_msg_t create_response;
        int rec = zmq_msg_init(&create_response);
        assert(rec != -1);
        zmq_msg_init_size(&create_response, rep_len);
        memcpy(zmq_msg_data(&create_response), reply, rep_len);
        zmq_msg_send(&create_response, parent, 0);
        zmq_msg_close(&create_response);
        free(reply);
    }

    zmq_close(parent);
    zmq_ctx_destroy(context);

    return 0;
}
```

list.c

```
#include "list.h"

Node *create_node(id init_id) {
    Node *new_node = (Node *)malloc(sizeof(Node));
    new_node->node_id = init_id;
    new_node->has_child = 0;
    new_node->child = (Node *)NULL;

    return new_node;
}

int exists_list(Node *root, id target_id) {
    if (root == NULL)
        return 0;
    if (root->node_id == target_id)
        return 1;

    int result = 0;
    result |= exists_list(root->child, target_id);
    return result;
}

int exists(List *root, id target_id, int count) {
    int result = 0;
    for (int i = 0; i < count; ++i) {
        result |= exists_list(root[i], target_id);
    }
    return result;
}

Node *add_node(Node *root, id parent_id, id new_id) {
    if (root->node_id == parent_id) {
        if (root->has_child) {
            Node *tmp = root->child;
            root->child = create_node(new_id);
            (root->child)->child = tmp;
            return root;
        }
        root->child = create_node(new_id);
        root->has_child = 1;
        return root;
    }
```

```
    if (root->has_child == 0)
        return root;

    root->child = add_node(root->child, parent_id, new_id);

    return root;
}

Node *delete_list(Node *root) {
    if (root->has_child == 0) {
        free(root);
        return NULL;
    }

    root->child = delete_list(root->child);
    free(root);

    return NULL;
}

Node *remove_node(Node *root, id target_id) {
    if (root->node_id == target_id)
        return delete_list(root);

    if (root->has_child == 0)
        return root;

    root->child = remove_node(root->child, target_id);
    return root;
}

void print_list_rec(const Node *root) {
    if (root == NULL) {
        return;
    }
    if (root->node_id != (id)-1)
        printf("%d ", root->node_id);
    print_list_rec(root->child);
}

void print_list(const Node *root) {
    printf("  ");
    print_list_rec(root);
    printf("\n");
}
```

interface.c

```
#include "interface.h"

const char *read_word() {
    char *result = (char *)calloc(sizeof(char), STR_LEN);
    char current;
    int i = 0;
    TRY_READ(&current);
    while (current != ' ') {
        if (current == '\n' || current == '\0')
            break;
        result[i++] = current;
        TRY_READ(&current);
```

```
    }
    result = (char *)realloc(result, sizeof(char) * (strlen(result) + 1));
    return result;
}

command_t get_command() {
    const char *input = read_word();

    if (strcmp("exit", input) == 0)
        return EXIT;

    if (strcmp("print", input) == 0)
        return PRINT;

    if (strcmp("create", input) == 0)
        return CREATE;

    if (strcmp("remove", input) == 0)
        return REMOVE;

    if (strcmp("exec", input) == 0)
        return EXEC;

    if (strcmp("pingall", input) == 0)
        return PINGALL;

    else
        return UNKNOWN;
}

subcommand_t get_subcommand() {
    const char *input = read_word();

    if (strcmp("time", input) == 0)
        return TIME;

    if (strcmp("start", input) == 0)
        return START;

    if (strcmp("stop", input) == 0)
        return STOP;

    else
        return UNDEFINED;
}

const char *int_to_string(unsigned a) {
    int x = a, i = 0;
    if (a == 0)
        return "0";
    while (x > 0) {
        x /= 10;
        i++;
    }
    char *result = (char *)calloc(sizeof(char), i + 1);
    while (i >= 1) {
        result[--i] = a % 10 + '0';
        a /= 10;
    }

    return result;
}
```

```
const char *portname_client(unsigned short port) {
    const char *port_string = int_to_string(port);
            char    *name    =    (char    *)calloc(sizeof(char),    strlen(CLIENT_ADRESS_PREFIX)    +
strlen(port_string) + 1);
    strcpy(name, CLIENT_ADRESS_PREFIX);
    strcpy(name + strlen(CLIENT_ADRESS_PREFIX) * sizeof(char), port_string);
    return name;
}

const char *portname_server(unsigned short port) {
    const char *port_string = int_to_string(port);
            char    *name    =    (char    *)calloc(sizeof(char),    strlen(SERVER_ADRESS_PREFIX)    +
strlen(port_string) + 1);
    strcpy(name, SERVER_ADRESS_PREFIX);
    strcpy(name + strlen(SERVER_ADRESS_PREFIX) * sizeof(char), port_string);
    return name;
}

char *message_prefix(unsigned node_id, subcommand_t s) {
    char *result = (char *)calloc(sizeof(char), STR_LEN);
    result[0] = 'O';
    result[1] = 'k';
    result[2] = ' ';
    result[3] = '\0';
    const char *id_str = int_to_string(node_id);
    result = strcat(result, id_str);
    result = strcat(result, ": ");
    switch (s) {
    case START:
        result = strcat(result, "started");
        break;
    case STOP:
        result = strcat(result, "stopped");
        break;
    default:
        break;
    }
    result = (char *)realloc(result, strlen(result) + 1);
    return result;
}
```

interface.h

```
#ifndef __INTERFACE_H__
#define __INTERFACE_H__

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define CLIENT_ADRESS_PREFIX "tcp://localhost:"
#define SERVER_ADRESS_PREFIX "tcp://*:"

#define BASE_PORT 7500
#define STR_LEN 64

#define TRY_READ(C)            \
    if (read(0, (C), 1) < 0) { \
        perror("read");        \
        exit(EXIT_FAILURE);    \
```

```
    }

typedef enum { EXIT = 0,
               CREATE,
               REMOVE,
               EXEC,
               PINGALL,
               PRINT,
               UNKNOWN } command_t;

typedef enum { START = 0,
               STOP,
               TIME,
               UNDEFINED } subcommand_t;

const char *read_word();
command_t get_command();
subcommand_t get_subcommand();

void print_help();

const char *int_to_string(unsigned a);
const char *portname_client(unsigned short port);
const char *portname_server(unsigned short port);

char *message_prefix(unsigned node_id, subcommand_t sub);

#endif
```

interprocess.c

```
#include "interprocess.h"

void send_exec(void *socket, subcommand_t subcommand, id node_id) {
    command_t c = EXEC;
    zmq_msg_t command_msg;
    zmq_msg_init_size(&command_msg, sizeof(c));
    memcpy(zmq_msg_data(&command_msg), &c, sizeof(c));
    zmq_msg_send(&command_msg, socket, ZMQ_SNDMORE);
    zmq_msg_close(&command_msg);

    zmq_msg_t subcommand_msg;
    zmq_msg_init_size(&subcommand_msg, sizeof(subcommand_t));
    memcpy(zmq_msg_data(&subcommand_msg), &subcommand, sizeof(subcommand_t));
    zmq_msg_send(&subcommand_msg, socket, ZMQ_SNDMORE);
    zmq_msg_close(&subcommand_msg);

    zmq_msg_t id_msg;
    zmq_msg_init_size(&id_msg, sizeof(node_id));
    memcpy(zmq_msg_data(&id_msg), &node_id, sizeof(node_id));
    zmq_msg_send(&id_msg, socket, 0);
    zmq_msg_close(&id_msg);
}

void send_create(void *socket, id init_id, id parent_id) {
    command_t c = CREATE;
    zmq_msg_t command_msg;
    zmq_msg_init_size(&command_msg, sizeof(c));
    memcpy(zmq_msg_data(&command_msg), &c, sizeof(c));
    zmq_msg_send(&command_msg, socket, ZMQ_SNDMORE);
    zmq_msg_close(&command_msg);
```

```
    zmq_msg_t id_msg;
    zmq_msg_init_size(&id_msg, sizeof(init_id));
    memcpy(zmq_msg_data(&id_msg), &init_id, sizeof(init_id));
    zmq_msg_send(&id_msg, socket, ZMQ_SNDMORE);
    zmq_msg_close(&id_msg);

    zmq_msg_t parent_id_msg;
    zmq_msg_init_size(&parent_id_msg, sizeof(parent_id));
    memcpy(zmq_msg_data(&parent_id_msg), &parent_id, sizeof(parent_id));
    zmq_msg_send(&parent_id_msg, socket, 0);
    zmq_msg_close(&parent_id_msg);
}

void send_remove(void *socket, id remove_id) {
    command_t c = REMOVE;
    zmq_msg_t command_msg;
    zmq_msg_init_size(&command_msg, sizeof(c));
    memcpy(zmq_msg_data(&command_msg), &c, sizeof(c));
    zmq_msg_send(&command_msg, socket, ZMQ_SNDMORE);
    zmq_msg_close(&command_msg);

    zmq_msg_t id_msg;
    zmq_msg_init_size(&id_msg, sizeof(remove_id));
    memcpy(zmq_msg_data(&id_msg), &remove_id, sizeof(remove_id));
    zmq_msg_send(&id_msg, socket, 0);
    zmq_msg_close(&id_msg);
}

void send_exit(void *socket) {
    command_t c = EXIT;
    zmq_msg_t command_msg;
    zmq_msg_init_size(&command_msg, sizeof(c));
    memcpy(zmq_msg_data(&command_msg), &c, sizeof(c));
    zmq_msg_send(&command_msg, socket, 0);
    zmq_msg_close(&command_msg);
}

void send_pingall(void *socket, int count) {
    command_t c = PINGALL;
    zmq_msg_t command_msg;
    zmq_msg_init_size(&command_msg, sizeof(c));
    memcpy(zmq_msg_data(&command_msg), &c, sizeof(c));
    zmq_msg_send(&command_msg, socket, ZMQ_SNDMORE);
    zmq_msg_close(&command_msg);

    zmq_msg_t l_msg;
    zmq_msg_init_size(&l_msg, sizeof(count));
    memcpy(zmq_msg_data(&l_msg), &count, sizeof(count));
    zmq_msg_send(&l_msg, socket, 0);
    zmq_msg_close(&l_msg);
}

char *get_reply(void *socket) {
    zmq_msg_t reply;
    zmq_msg_init(&reply);
    zmq_msg_recv(&reply, socket, 0);
    size_t result_size = zmq_msg_size(&reply);

    char *result = (char *)calloc(sizeof(char), result_size + 1);
    memcpy(result, zmq_msg_data(&reply), result_size);
    zmq_msg_close(&reply);
```

```
    return result;
}

char *get_reply_pingall(void *socket) {
    int arg = 0;
    char *result = NULL;
    while (1) {
        zmq_msg_t part;
        int rec = zmq_msg_init(&part);
        assert(rec == 0);
        rec = zmq_msg_recv(&part, socket, 0);
        assert(rec != -1);
        switch (arg) {
        case 0:
            break;
        case 1:;
            size_t result_size = zmq_msg_size(&part);
            result = (char *)calloc(sizeof(char), result_size + 1);
            memcpy(result, zmq_msg_data(&part), result_size);
            break;
        }
        zmq_msg_close(&part);
        ++arg;
        if (!zmq_msg_more(&part))
            break;
    }

    return result;
}

void create_worker(id init_id, int child_port) {
    const char *arg0 = SERVER_PATH;
    const char *arg1 = int_to_string(init_id);
    const char *arg2 = int_to_string(child_port);
    execl(SERVER_PATH, arg0, arg1, arg2, (char *)NULL);
}

int available_recive(void *socket) {
    zmq_pollitem_t items[1] = {{socket, 0, ZMQ_POLLIN, 0}};
    int rc = zmq_poll(items, 1, REQUEST_TIMEOUT);
    assert(rc != -1);
    if (items[0].revents & ZMQ_POLLIN)
        return 1;
    return 0;
}

int available_recive_pingall(void *socket, int count) {
    zmq_pollitem_t items[1] = {{socket, 0, ZMQ_POLLIN, 0}};
    int rc = zmq_poll(items, 1, REQUEST_TIMEOUT - count * ADDITIONAL_TIME);
    assert(rc != -1);
    if (items[0].revents & ZMQ_POLLIN)
        return 1;
    return 0;
}

int is_available_send(void *socket) {
    zmq_pollitem_t items[1] = {{socket, 0, ZMQ_POLLOUT, 0}};
    int rc = zmq_poll(items, 1, REQUEST_TIMEOUT);
    assert(rc != -1);
    if (items[0].revents & ZMQ_POLLOUT)
        return 1;
    return 0;
```

```
}

void shift_void(void **array, int pos, int capacity) {
    if (pos == capacity - 1) {
        array[pos] = NULL;
        return;
    }
    for (int i = pos; i < capacity - 1; ++i) {
        array[i] = array[i + 1];
    }
    array[capacity - 1] = NULL;
    return;
}

void shift_id(id *array, int pos, int capacity) {
    if (pos == capacity - 1) {
        array[pos] = 0;
        return;
    }
    for (int i = pos; i < capacity - 1; ++i) {
        array[i] = array[i + 1];
    }
    array[capacity - 1] = 0;
    return;
}

int in_list(id *array, id target, int capacity) {
    for (int i = 0; i < capacity; ++i) {
        if (array[i] == target)
            return i;
    }
    return -1;
}
```

interprocess.h

```
#ifndef __INTERPROCESS_H__
#define __INTERPROCESS_H__

#include <assert.h>
#include <zmq.h>

#include "interface.h"
#include "list.h"

#define SERVER_PATH "./server"

#define REQUEST_TIMEOUT 2000
#define ADDITIONAL_TIME 100

#define EMPTY_MSG ""

void send_exec(void *socket, subcommand_t subcommand, id node_id);
void send_create(void *socket, id init_id, id parent_id);
void send_exit(void *socket);
void send_pingall(void *socket, int count);
void send_remove(void *socket, id remove_id);
char *get_reply(void *socket);
char *get_reply_pingall(void *socket);
void create_worker(id init_id, int port);
int available_recive(void *socket);
```

```
int available_recive_pingall(void *socket, int additional);
int is_available_send(void *socket);

void shift_void(void **array, int pos, int capacity);
void shift_id(id *array, int pos, int capacity);
int in_list(id *array, id target, int capacity);

#endif
```

Makefile

```
CC = gcc
LD = gcc
LDFLAGS = -lzmq
CFLAGS = -Wall -pedantic -std=c99 -c

SRC_CLIENT = client.c
OBJ_CLIENT = $(SRC_CLIENT: .c = .o)

SRC_SERVER = server.c
OBJ_SERVER = $(SRC_SERVER: .c = .o)

SRC_OTH = list.c interface.c interprocess.c
OBJ_OTH = $(SRC_OTH: .c = .o)


all: object
	$(LD) $(OBJ_SERVER) $(OBJ_OTH) -o server $(LDFLAGS)
	$(LD) $(OBJ_CLIENT) $(OBJ_OTH) -o client $(LDFLAGS)

object: $(SRC_CLIENT) $(SRC_SERVER) $(SRC_OTH)
	$(CC) $(SRC_CLIENT) $(CFLAGS)
	$(CC) $(SRC_OTH) $(CFLAGS)
	$(CC) $(SRC_SERVER) $(CFLAGS)

remove: clean
	rm -rf server client

clean:
	rm -rf *.o
```

Вывод

В ходе выполнения данной лабораторной работы, были освоены основы библиотеки ZMQ, а также я познакомилась с очередями сообщений.