Московский Авиационный Институт
(Национальный Исследовательский Университет)

Факультет информационных технологий и
прикладной математики
Кафедра вычислительной математики и
программирования

Лабораторные работы №6-8 по курсу
"Операционные системы"

УПРАВЛЕНИЕ СЕРВЕРАМИ СООБЩЕНИЙ

Студент: Сеимов Максим Сергеевич
Группа: М8О-208Б-21
Вариант: 19
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

1 Постановка задачи

Цель работы

Целью лабораторных работ №6-8 является приобретение практических навыков в:

- Управлении серверами сообщений (№6)
- Применении отложенных вычислений (№7)
- Интеграции 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.

Вариант 19

Вариантом №19 определены следующие параметры системы:

- Топология системы - дерево общего вида
- Тип компанд для вычислительных узлов - локальный таймер
- Тип проверки доступности узлов - команда *pingall*

Общие сведения о программе

Программа представлена следующими исполняемыми файлами (компилируемыми с помощью утилиты **make**:

- **client** - клиентская программа, через которую пользователь отправляет системе команды (управляющий узел).
- **server** - программа для вычислительного узла; не предназначена для запуска, запускается из клиентской программы.

Исходные файлы:

Заголовочные файлы:

- **interface.h** - объявление общих функций для реализации пользовательского интерфейса программы (считывание и распознавание команды, генерация имени порта и т.д.);

- **tree.h** - объявление функций для работы с деревом общего вида (нужно для вывода текущего состояния системы и грамотной обработки операции удаления узла);
- **interprocess.h** - объявление функций для обеспечения межпроцессорного взаимодействия (отправка/получение сообщений, создание нового узла и т.д.), а также некоторых вспомогательных функций.

Файлы реализации:

- **interface.c**, **tree.c** и **interprocess.c** - реализация функций из вышеописанных заголовочных файлов;
- **client.c** - реализация клиентского модуля (собственно запускаемой программы);
- **server.c** - реализация вычислительного модуля.

В качестве сервера сообщений была выбрана библиотека *ZeroMQ* (`zmq.h`).

Основные файлы программы

tree.h:

```c
#ifndef __TREE_H__
#define __TREE_H__

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

#define INIT_CAPACITY 5

typedef unsigned short id;

typedef struct _node {
    id node_id;
    struct _node **children;
    unsigned children_count;
    unsigned children_capacity;
} Node;

typedef Node *Tree;

Node *create_node(id init_id);
int exists(Node *root, id target_id);
Node *add_node(Node *root, id parent_id, id init_id);
Node *remove_node(Node *root, id init_id);
void print_tree(const Node *root, unsigned depth);

#endif
```

tree.c:

```c
#include "tree.h"

Node *create_node(id init_id) {
    Node *new_node = (Node *)malloc(sizeof(Node));
    new_node->node_id = init_id;
    new_node->children_count = 0;
    new_node->children_capacity = INIT_CAPACITY;
    new_node->children = (Node **)calloc(sizeof(Node *), INIT_CAPACITY);
```

```
9       for (int i = 0; i < INIT_CAPACITY; ++i) {
10          new_node->children[i] = NULL;
11      }
12
13      return new_node;
14  }
15
16  int exists(Node *root, id target_id) {
17      if (root == NULL)
18          return 0;
19      if (root->node_id == target_id)
20          return 1;
21
22      int result = 0;
23      for (int i = 0; i < root->children_count; i++)
24          result |= exists(root->children[i], target_id);
25
26      return result;
27  }
28
29  Node *add_node(Node *root, id parent_id, id new_id) {
30      if (root->node_id == parent_id) {
31          if (root->children_count >= root->children_capacity) {
32              root->children_capacity *= 2;
33              root->children = (Node **)realloc(root->children, sizeof(Node *) * root->
                    children_capacity);
34          }
35          root->children[(root->children_count)++] = create_node(new_id);
36          return root;
37      }
38
39      if (root->children_count == 0)
40          return root;
41
42      for (int i = 0; i < root->children_count; ++i)
43          root->children[i] = add_node(root->children[i], parent_id, new_id);
44
45      return root;
46  }
47
48  Node *delete_tree(Node *root) {
49      if (root->children_count == 0) {
50          free(root->children);
51          free(root);
52          return NULL;
53      }
54
55      for (int i = 0; i < root->children_count; ++i)
56          root->children[i] = delete_tree(root->children[i]);
57
58      return NULL;
59  }
60
61  Node *remove_node(Node *root, id target_id) {
62      if (root->node_id == target_id)
63          return delete_tree(root);
64
65      if (root->children_count == 0)
66          return root;
67
```

```
    Node *result;

    for (int i = 0; i < root->children_count; i++) {
        result = root->children[i] = remove_node(root->children[i], target_id);
        if (result == NULL) {
            for (int j = i; j < root->children_count - 1; ++j) {
                root->children[j] = root->children[j + 1];
            }
            root->children[root->children_count - 1] = NULL;
            root->children_count--;
            break;
        }
    }
    return root;
}

void print_tree(const Node *root, unsigned depth) {
    if (root == NULL) {
        return;
    }
    for (int i = 0; i < depth; ++i)
        printf("\t");
    if (root->node_id == (id)-1)
        printf("-1\n");
    else
        printf("%d\n", root->node_id);
    for (int i = 0; i < root->children_count; ++i) {
        print_tree(root->children[i], depth + 1);
    }
}
```

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 5555
#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
```

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;
}

// void to_low(char *s, size_t n) {
// for (int i = 0; i < n; ++i) {
// if (s[i] >= 'A' && s[i] <= 'Z') {
// s[i] += ('a' - 'A');
// }
// }
// }

command_t get_command() {
    const char *input = read_word();
    // to_low(input, strlen(input));

    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;
```

```
41
42      if (strcmp("exec", input) == 0)
43          return EXEC;
44
45      if (strcmp("pingall", input) == 0)
46          return PINGALL;
47
48      else
49          return UNKNOWN;
50  }
51
52  subcommand_t get_subcommand() {
53      const char *input = read_word();
54      // to_low(input, strlen(input));
55
56      if (strcmp("time", input) == 0)
57          return TIME;
58
59      if (strcmp("start", input) == 0)
60          return START;
61
62      if (strcmp("stop", input) == 0)
63          return STOP;
64
65      else
66          return UNDEFINED;
67  }
68
69  void print_help() {
70      printf("List of available commands:\n\ncreate [id] [parent] - creates a new computing
            module\n");
71      printf("remove [id] - removes computing module along with its children\n");
72      printf("exec [id] - exec task command (finding all substring occurances\n");
73      printf("pingall - check if there are unavailable modules\n");
74      printf("exit - terminates program\n\n");
75  }
76
77  const char *int_to_string(unsigned a) {
78      int x = a, i = 0;
79      if (a == 0)
80          return "0";
81      while (x > 0) {
82          x /= 10;
83          i++;
84      }
85      char *result = (char *)calloc(sizeof(char), i + 1);
86      while (i >= 1) {
87          result[--i] = a % 10 + '0';
88          a /= 10;
89      }
90
91      return result;
92  }
93
94  const char *portname_client(unsigned short port) {
95      const char *port_string = int_to_string(port);
96      char *name = (char *)calloc(sizeof(char), strlen(CLIENT_ADRESS_PREFIX) + strlen(
            port_string) + 1);
97      strcpy(name, CLIENT_ADRESS_PREFIX);
98      strcpy(name + strlen(CLIENT_ADRESS_PREFIX) * sizeof(char), port_string);
```

```
99      return name;
100 }
101
102 const char *portname_server(unsigned short port) {
103     const char *port_string = int_to_string(port);
104     char *name = (char *)calloc(sizeof(char), strlen(SERVER_ADRESS_PREFIX) + strlen(
            port_string) + 1);
105     strcpy(name, SERVER_ADRESS_PREFIX);
106     strcpy(name + strlen(SERVER_ADRESS_PREFIX) * sizeof(char), port_string);
107     return name;
108 }
109
110 char *message_prefix(unsigned node_id, subcommand_t s) {
111     char *result = (char *)calloc(sizeof(char), STR_LEN);
112     result[0] = 'O';
113     result[1] = 'K';
114     result[2] = ' ';
115     result[3] = '[';
116     result[4] = '\0';
117     const char *id_str = int_to_string(node_id);
118     result = strcat(result, id_str);
119     result = strcat(result, "]: ");
120     switch (s) {
121     case START:
122         result = strcat(result, "started timer");
123         break;
124     case STOP:
125         result = strcat(result, "stopped timer");
126         break;
127     default:
128         break;
129     }
130     result = (char *)realloc(result, strlen(result) + 1);
131     return result;
132 }
```

interprocess.h:

```
1  #ifndef __INTERPROCESS_H__
2  #define __INTERPROCESS_H__
3
4  #include <assert.h>
5  #include <zmq.h>
6
7  #include "interface.h"
8  #include "tree.h"
9
10 #define SERVER_PATH "./server"
11
12 #define REQUEST_TIMEOUT 2000
13 #define ADDITIONAL_TIME 100
14
15 #define EMPTY_MSG ""
16
17 void send_exec(void *socket, subcommand_t subcommand, id node_id);
18 void send_create(void *socket, id init_id, id parent_id);
19 void send_exit(void *socket);
20 void send_pingall(void *socket, int layer);
21 void send_remove(void *socket, id remove_id);
22 char *get_reply(void *socket);
23 char *get_reply_pingall(void *socket);
```

```
void create_worker(id init_id);
int is_available_recv(void *socket);
int is_available_recv_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
```

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));
```

```
49      memcpy(zmq_msg_data(&command_msg), &c, sizeof(c));
50      zmq_msg_send(&command_msg, socket, ZMQ_SNDMORE);
51      zmq_msg_close(&command_msg);
52
53      zmq_msg_t id_msg;
54      zmq_msg_init_size(&id_msg, sizeof(remove_id));
55      memcpy(zmq_msg_data(&id_msg), &remove_id, sizeof(remove_id));
56      zmq_msg_send(&id_msg, socket, 0);
57      zmq_msg_close(&id_msg);
58  }
59
60  void send_exit(void *socket) {
61      command_t c = EXIT;
62      zmq_msg_t command_msg;
63      zmq_msg_init_size(&command_msg, sizeof(c));
64      memcpy(zmq_msg_data(&command_msg), &c, sizeof(c));
65      zmq_msg_send(&command_msg, socket, 0);
66      zmq_msg_close(&command_msg);
67  }
68
69  void send_pingall(void *socket, int layer) {
70      command_t c = PINGALL;
71      zmq_msg_t command_msg;
72      zmq_msg_init_size(&command_msg, sizeof(c));
73      memcpy(zmq_msg_data(&command_msg), &c, sizeof(c));
74      zmq_msg_send(&command_msg, socket, ZMQ_SNDMORE);
75      zmq_msg_close(&command_msg);
76
77      zmq_msg_t l_msg;
78      zmq_msg_init_size(&l_msg, sizeof(layer));
79      memcpy(zmq_msg_data(&l_msg), &layer, sizeof(layer));
80      zmq_msg_send(&l_msg, socket, 0);
81      zmq_msg_close(&l_msg);
82  }
83
84  char *get_reply(void *socket) {
85      zmq_msg_t reply;
86      zmq_msg_init(&reply);
87      zmq_msg_recv(&reply, socket, 0);
88      size_t result_size = zmq_msg_size(&reply);
89
90      char *result = (char *)calloc(sizeof(char), result_size + 1);
91      memcpy(result, zmq_msg_data(&reply), result_size);
92      zmq_msg_close(&reply);
93
94      return result;
95  }
96
97  char *get_reply_pingall(void *socket) {
98      int arg = 0;
99      char *result = NULL;
100     while (1) {
101         zmq_msg_t part;
102         int rec = zmq_msg_init(&part);
103         assert(rec == 0);
104         rec = zmq_msg_recv(&part, socket, 0);
105         assert(rec != -1);
106         switch (arg) {
107         case 0:
108             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) {
    const char *arg0 = SERVER_PATH;
    const char *arg1 = int_to_string(init_id);
    execl(SERVER_PATH, arg0, arg1, (char *)NULL);
}

int is_available_recv(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 is_available_recv_pingall(void *socket, int layer) {
    zmq_pollitem_t items[1] = {{socket, 0, ZMQ_POLLIN, 0}};
    int rc = zmq_poll(items, 1, REQUEST_TIMEOUT - layer * 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;
}

```

```c
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;
}
```

server.c:

```c
#include <sys/time.h>

#include "interprocess.h"

#define OFF 0
#define ON 1

#define INIT_CAPACITY 5

short timer = OFF;

unsigned children_count = 0;
unsigned capacity = INIT_CAPACITY;

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 **children_sockets = calloc(sizeof(void *), INIT_CAPACITY);
    id *children_ids = calloc(sizeof(id), INIT_CAPACITY);

    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 layer = 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(&layer, 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) {
            for (int i = 0; i < children_count; ++i) {
                send_exit(children_sockets[i]);
            }
            break;
```

```
100         }
101
102         int replied = 0;
103         int not_replied = 0;
104         char *reply = (char *)calloc(sizeof(char), STR_LEN);
105
106         if (current == CREATE) {
107             if (parent_id == self_id) {
108                 int fork_val = fork();
109                 if (fork_val == 0)
110                     create_worker(target_id);
111
112                 if (children_count >= capacity) {
113                     capacity *= 2;
114                     children_sockets = realloc(children_sockets, sizeof(void *) * capacity);
115                     children_ids = realloc(children_ids, sizeof(id) * capacity);
116                 }
117                 children_sockets[children_count] = zmq_socket(context, ZMQ_REQ);
118                 int opt_val = 0;
119                 int rc = zmq_setsockopt(children_sockets[children_count], ZMQ_LINGER, &
                        opt_val, sizeof(opt_val));
120                 assert(rc == 0);
121                 if (children_sockets[children_count] == NULL) {
122                     perror("socket");
123                     exit(EXIT_FAILURE);
124                 }
125
126                 rc = zmq_connect(children_sockets[children_count], portname_client(BASE_PORT
                        + target_id));
127                 assert(rc == 0);
128
129                 children_ids[children_count++] = target_id;
130
131                 sprintf(reply, "Created node %d with PID %d", target_id, fork_val);
132                 replied = 1;
133
134             } else {
135                 for (int i = 0; i < children_count; ++i) {
136                     send_create(children_sockets[i], target_id, parent_id);
137                     if (!is_available_recv(children_sockets[i])) {
138                         continue;
139                     }
140                     const char *reply_child = get_reply(children_sockets[i]);
141                     if (strcmp(EMPTY_MSG, reply_child) != 0) {
142                         sprintf(reply, "%s", reply_child);
143                         replied = 1;
144                         break;
145                     }
146                 }
147             }
148         }
149
150         else if (current == EXEC) {
151
152             if (target_id == self_id) {
153                 switch (current_sub) {
154                 case START:
155                     gettimeofday(&start, NULL);
156                     timer = ON;
157                     break;
```

```
158                 case STOP:
159                     if (timer == ON) {
160                         gettimeofday(&finish, NULL);
161                         timer = OFF;
162                         diff_sec = finish.tv_sec - start.tv_sec;
163                         diff_msec = (diff_sec * 1000) + (finish.tv_usec - start.tv_usec) /
                                1000;
164                     }
165                     break;
166                 case TIME:
167                     if (timer == ON) {
168                         gettimeofday(&finish, NULL);
169                         diff_sec = finish.tv_sec - start.tv_sec;
170                         diff_msec = (diff_sec * 1000) + (finish.tv_usec - start.tv_usec) /
                                1000;
171                     }
172                     break;
173                 default:
174                     break;
175                 }
176 
177                 char *result = message_prefix(self_id, current_sub);
178 
179                 if (current_sub == TIME) {
180                     const char *time_string = int_to_string((unsigned)diff_msec);
181                     result = strcat(result, time_string);
182                 }
183                 strcpy(reply, result);
184                 replied = 1;
185             } else {
186                 for (int i = 0; i < children_count; ++i) {
187                     send_exec(children_sockets[i], current_sub, target_id);
188                     if (!is_available_recv(children_sockets[i])) {
189                         continue;
190                     }
191                     char *reply_child = get_reply(children_sockets[i]);
192                     if (strcmp(EMPTY_MSG, reply_child) != 0) {
193                         sprintf(reply, "%s", reply_child);
194                         replied = 1;
195                         break;
196                     }
197                 }
198             }
199         }
200 
201         else if (current == REMOVE) {
202             int i = in_list(children_ids, target_id, children_count);
203             if (i != -1) {
204                 shift_id(children_ids, i, children_count);
205                 send_exit(children_sockets[i]);
206                 zmq_close(children_sockets[i]);
207                 shift_void(children_sockets, i, children_count);
208                 children_count--;
209                 sprintf(reply, "Successfully removed node %d from system", target_id);
210                 replied = 1;
211             } else {
212                 for (int i = 0; i < children_count; ++i) {
213                     send_remove(children_sockets[i], target_id);
214                     if (!is_available_recv(children_sockets[i])) {
215                         continue;
```

```
                }
                char *reply_child = get_reply(children_sockets[i]);
                if (strcmp(EMPTY_MSG, reply_child) != 0) {
                    sprintf(reply, "%s", reply_child);
                    replied = 1;
                    break;
                }
            }
        }
    }

    else if (current == PINGALL) {

        for (int i = 0; i < children_count; ++i) {
            send_pingall(children_sockets[i], layer + 1);
            if (!is_available_recv_pingall(children_sockets[i], layer)) {
                reply = strcat(reply, int_to_string(children_ids[i]));
                reply = strcat(reply, " ");
                not_replied++;
                continue;
            }
            char *reply_child = get_reply(children_sockets[i]);
            if (strcmp(EMPTY_MSG, reply_child) != 0) {
                reply = strcat(reply, reply_child);
                replied = 1;
                break;
            }
        }
    }

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

client.c:

```
#include "interprocess.h"

#define CLIENT_ID -1

unsigned children_count = 0;
unsigned capacity = INIT_CAPACITY;

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

```

```
    Tree system;
    system = create_node(CLIENT_ID);

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

    void **children_sockets = calloc(sizeof(void *), INIT_CAPACITY);
    id *children_ids = calloc(sizeof(id), INIT_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:
            print_tree(system, 0);
            break;
        case EXIT:
            for (int i = 0; i < children_count; ++i) {
                send_exit(children_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)) {
                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 (!exists(system, parent_id)) {
                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);
                printf("Created node %d with PID %d\n", init_id, fork_val);

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

```
69                  children_sockets = realloc(children_sockets, sizeof(void *) * capacity);
70                  children_ids = realloc(children_ids, sizeof(id) * capacity);
71              }
72              children_sockets[children_count] = zmq_socket(context, ZMQ_REQ);
73              int opt_val = 0;
74              int rc = zmq_setsockopt(children_sockets[children_count], ZMQ_LINGER, &
                    opt_val, sizeof(opt_val));
75              assert(rc == 0);
76              if (children_sockets[children_count] == NULL) {
77                  perror("socket");
78                  exit(EXIT_FAILURE);
79              }
80
81              rc = zmq_connect(children_sockets[children_count], portname_client(BASE_PORT
                    + init_id));
82              assert(rc == 0);
83
84              children_ids[children_count++] = init_id;
85              system = add_node(system, parent_id, init_id);
86              break;
87          } else {
88              int replied = 0;
89              for (int i = 0; i < children_count; ++i) {
90                  send_create(children_sockets[i], init_id, parent_id);
91                  if (!is_available_recv(children_sockets[i])) {
92                      continue;
93                  }
94                  const char *reply = get_reply(children_sockets[i]);
95                  if (strcmp(EMPTY_MSG, reply) != 0) {
96                      printf("%s\n", reply);
97                      replied = 1;
98                      break;
99                  }
100             }
101             if (replied == 0)
102                 printf("Node %d seems to be unavailable\n", parent_id);
103             else
104                 system = add_node(system, parent_id, init_id);
105         }
106         break;
107
108     case EXEC:;
109         const char *target_id_str = read_word();
110         int target_id = atoi(target_id_str);
111         if (target_id <= 0) {
112             printf("Error: invalid node id (id should be an integer greater than 0).\n");
113             break;
114         }
115         if (!exists(system, target_id)) {
116             printf("Error: this node does not exists.\n");
117             break;
118         }
119         subcommand_t current_sub = get_subcommand();
120         if (current_sub == UNDEFINED) {
121             printf("Invalid subcommand!\n");
122             break;
123         }
124
125         int replied = 0;
126         for (int i = 0; i < children_count; ++i) {
```

```
127                 send_exec(children_sockets[i], current_sub, target_id);
128                 if (!is_available_recv(children_sockets[i])) {
129                     continue;
130                 }
131                 char *reply = get_reply(children_sockets[i]);
132                 if (strcmp(EMPTY_MSG, reply) != 0) {
133                     printf("%s\n", reply);
134                     replied = 1;
135                     break;
136                 }
137             }
138             if (replied == 0)
139                 printf("Node %d seems to be unavailable\n", target_id);
140 
141             break;
142         case REMOVE:;
143             const char *remove_id_str = read_word();
144             int remove_id = atoi(remove_id_str);
145             if (remove_id <= 0) {
146                 printf("Error: invalid node id (id should be an integer greater than 0).\n");
147                 break;
148             }
149             if (!exists(system, remove_id)) {
150                 printf("Error: this node does not exist.\n");
151                 break;
152             }
153             int i = in_list(children_ids, remove_id, children_count);
154             if (i != -1) {
155                 shift_id(children_ids, i, children_count);
156                 send_exit(children_sockets[i]);
157                 zmq_close(children_sockets[i]);
158                 shift_void(children_sockets, i, children_count);
159                 children_count--;
160                 printf("Successfully removed node %d from system\n", remove_id);
161             } else {
162                 replied = 0;
163                 for (int i = 0; i < children_count; ++i) {
164                     send_remove(children_sockets[i], remove_id);
165                     if (!is_available_recv(children_sockets[i])) {
166                         continue;
167                     }
168                     char *reply = get_reply(children_sockets[i]);
169                     if (strcmp(EMPTY_MSG, reply) != 0) {
170                         printf("%s\n", reply);
171                         replied = 1;
172                         break;
173                     }
174                 }
175                 if (replied == 0)
176                     printf("Node %d seems to be unavailable, removed it from tree anyways\n",
                            remove_id);
177             }
178             system = remove_node(system, remove_id);
179             break;
180         case PINGALL:;
181             int not_replied = 0;
182             int layer = 0;
183             char *unavailable = (char *)calloc(sizeof(char), STR_LEN);
184             for (int i = 0; i < children_count; ++i) {
185                 send_pingall(children_sockets[i], layer + 1);
```

```
186                 if (!is_available_recv_pingall(children_sockets[i], layer)) {
187                     unavailable = strcat(unavailable, int_to_string(children_ids[i]));
188                     unavailable = strcat(unavailable, " ");
189                     not_replied++;
190                     continue;
191                 }
192                 char *reply_child = get_reply(children_sockets[i]);
193                 if (strcmp(EMPTY_MSG, reply_child) != 0) {
194                     unavailable = strcat(unavailable, reply_child);
195                     replied = 1;
196                     break;
197                 }
198             }
199             if (not_replied == 0 && replied == 0) {
200                 printf("Every process is available\n");
201             } else {
202                 fprintf(stderr, "%s\n", unavailable);
203             }
204             break;
205         default:
206             printf("Error: invalid command.\n");
207             break;
208         }
209         if (current == EXIT) {
210             break;
211         }
212     }
213
214     for (int i = 0; i < children_count; ++i) {
215         zmq_close(children_sockets[i]);
216     }
217     zmq_ctx_destroy(context);
218
219     return 0;
220 }
```

Пример работы

Лог сессии в клиентской программе (комментарии добавлены для пояснения):

```
> create 1 -1
Created node 1 with PID 7956
> create 2 -1
Created node 2 with PID 7961
> create 3 -1
Created node 3 with PID 7964
> create 5 1
Created node 5 with PID 7967
> create 4 1
Created node 4 with PID 7972
> create 6 5
Created node 6 with PID 7978
> print
```

```
-1
    1
        5
            6
        4
    2
    3
> exec 4 start
OK [4]: started timer
> exec 6 time
OK [6]: 0
> exec 6 stop
OK [6]: stopped timer
> exec 4 time
OK [4]: 16284
> exec 4 stop
OK [4]: stopped timer
> exec 4 time
OK [4]: 20853
> remove 3
Successfully removed node 3 from system
> print
-1
    1
        5
            6
        4
    2
> exec 3 time
Error: this node does not exists. /* Узел успешно удалён */
> Error: invalid command.
> pingall
        /* Пустой вывод pingall означает отсутствие недоступных узлов */

/* Здесь в другом терминале была вызвана команда kill 7967 (PID узла с ID = 5) */

> exec 6 time
Node 6 seems to be unavailable /* Сообщение напечатано после двух секунд ожидания
*/
> exec 5 time
Node 5 seems to be unavailable
> pingall
5         /* pingall печатает все корни недоступных поддеревьев; информацию о
недоступных потомках можно получить из визуального представления дерева (print)
*/
> remove 5
Successfully removed node 5 from system
> exec 1 time /* Родительский узел по-прежнему доступен */
OK [1]: 0
> print
```

```
-1
     1
          4
     2
> pingall
          /* Снова доступны все узлы */
> exit
```

Вывод

В ходе выполнения задания мной были изучены принципы работы очередей сообщений и получено понимание о том, что такое сокеты, порты, сообщения и т.д. Также я познакомился с API ZeroMQ и научился с помощью этой библиотеки писать приложения типа клиент-сервер.