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»**

Управлении серверами сообщений

Применение отложенных вычислений

Интеграция программных систем друг с другом

Студент: Киреев Александр Константинович

Группа: М8О–206Б–19

Вариант: 48

Преподаватель: Соколов Андрей Алексеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2020.

**Постановка задачи**

Цель работы

Целью является приобретение практических навыков в:

•  Управлении серверами сообщений (№6)

•  Применение отложенных вычислений (№7)

•  Интеграция программных систем друг с другом (№8)

Задание

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. 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 48:

Топология: узлы находятся в идеально сбалансированном бинарном дереве. Каждый следующий узел должен добавляться в самое наименьшее левое поддерево.

Тип команды для вычислительных узлов: поиск подстроки в строке.

Тип проверки доступности узлов: Heartbit. Каждый узел начинает сообщать раз в time миллисекунд о том, что он работоспособен. Если от узла нет сигнала в течении 4\*time миллисекунд, то должна выводится пользователю строка: «Heartbit: node id is unavailable now», где id – идентификатор недоступного вычислительного узла.

**Общие сведения о программе**

Программа представлена в виде двух директорий: avl и client\_and\_server. В перовой находится реализация avl дерева на C++ для построения и перестроения дерева процессов. Во второй папке находятся файлы, отвечающие за клиент-серверное сообщение. Программу собирать при помощи общего Makefile’a, запускать программу при помощи команд ./server и ./client. В программе используется очередь сообщений zmq. Так же используются заголовочные файлы: stdio.h, unistd.h, stdlib.h, stdbool.h, signal.h. В программе используются следующие системные вызовы:

1. **kill** ­­–– используется для отправки любого сигнала на любой процесс. В случае успеха возвращает 0. Иначе -1.
2. **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.
3. **execl –** загружает и запускает другую программу. Таким образом, новая программа полностью замещает текущий процесс. Новая программа начинает свое выполнение с функции main. Аргументы командной строки передаются в форме списка arg0, arg1.... argn, NULL.

В программе используются следующие функции из zeromq:

1. **zmq\_msg\_init** ­­–– создает сообщение для очереди сообщений.
2. **zmq\_msg\_close –** закрывает сообщение для очереди сообщений.
3. **zmq\_msg\_send –** отправить сообщение на сокет.
4. **zmq\_msg\_recv –** принять сообщение с сокета.
5. **zmq\_ctx\_nex –** создать новый контекст для ввода/вывода.
6. **zmq\_socket –** создать новый сокет.
7. **zmq\_bind –** связать сокет с именованным файлом.
8. **zmq\_connect –** присоединиться к сокету.
9. **zmq\_setsockopt –** установить опции сокета.
10. **zmq\_close –** закрыть сокет.

**Общий метод и алгоритм решения**.

Для реализации поставленной задачи необходимо:

1. Изучить zmq.
2. Продумать архитектуру для программы и межпроцессорного взаимодействия.
3. Продумать схему переподвешивания очередей сообщений при переподвешивании вершин дерева.
4. Написать avl дерево.
5. Написать общие функции для межпроцессного взаимодействия.
6. Реализовать клиент и сервер.
7. Отладить их работу для случаев, когда переподвешивание не происходит.
8. Интегрировать avl дерево в проект.
9. Написать тесты, протестировать и отладить конечный код.

**Основные файлы программы**

**avl.hpp:**

#pragma once

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

#include <memory>

#include <cstdint>

#include <zmq.h>

#include <unistd.h>

extern "C" {

#include "../client\_and\_server/zmq\_handle.h"

}

/\*\*

\* структура для процесса-вершины дерева с укаанным id

\*/

struct tree\_node {

int32\_t pid;

int32\_t balance;

std::weak\_ptr<tree\_node> parent;

std::shared\_ptr<tree\_node> left;

std::shared\_ptr<tree\_node> right;

tree\_node(): pid(-1), balance(0), left(nullptr), right(nullptr) {

parent.lock() = nullptr;

}

tree\_node(int32\_t pid): pid(pid), balance(0), left(nullptr), right(nullptr) {

parent.lock() = nullptr;

}

~tree\_node() = default;

};

class avl\_tree {

private:

// корень дерева

std::shared\_ptr<tree\_node> \_root;

// рекурсивный поиск по дереву, возвращает указатель на искомый узел или nullptr

std::shared\_ptr<tree\_node> \_find(int32\_t pid, std::shared\_ptr<tree\_node> node);

// рекурсивынй поиск с обновлением пути

bool \_search(int32\_t pid, int32\_t\* path, std::shared\_ptr<tree\_node> node, int32\_t\* height);

// рекурсивная функция для вставки

bool \_insert(int32\_t pid, std::shared\_ptr<tree\_node> node);

// левый поворот

void \_left\_rotate(std::shared\_ptr<tree\_node> node);

// правый поворот

void \_right\_rotate(std::shared\_ptr<tree\_node> node);

// перебалансировка вершины с балансом равным 2 по модулю

std::shared\_ptr<tree\_node> \_rebalance(std::shared\_ptr<tree\_node> node);

// удаление всего поддерева с корнем в вершине node

void \_delete\_sub\_tree(std::shared\_ptr<tree\_node> node);

// рекурсивно перестраиваем дерево после удаления целого поддерева

void \_rec\_reconstruct(std::shared\_ptr<tree\_node>& new\_root, std::shared\_ptr<tree\_node> node);

// функция, вызывающая \_rec\_reconstruct, она обновляет корень

void \_reconstruct();

// рекурсивное обновление баланса для вставки

void \_go\_up\_insert(std::shared\_ptr<tree\_node> node, std::shared\_ptr<tree\_node> prev);

// вывод дерева на экран рекурсивно

void \_print(const std::string& prefix, std::shared\_ptr<tree\_node> node, bool is\_left, int32\_t height);

// поиск и удаленеи вершины

void \_remove(std::shared\_ptr<tree\_node> node);

// рекурсивное обновление баланса для удаления

void \_go\_up\_remove(std::shared\_ptr<tree\_node> node, std::shared\_ptr<tree\_node> prev);

public:

/\*\*

\* конструктор по умолчанию

\*/

avl\_tree(): \_root(nullptr) {};

/\*\*

\* поиск в дереве по id процесса

\* возвращает вектор-путь от корневого процесса до искомого

\* если искомый процесс не найден, то возвращаемый вектор имеет длину 0

\*/

bool search(int32\_t pid, int32\_t\* path, int32\_t\* path\_len);

/\*\*

\* вставка в дерево по id процесса

\* в случае успеха вернет true

\* в случае неуспеха вернет false

\*/

bool insert(int32\_t pid);

/\*\*

\* печать дерева на экран

\*/

void print();

/\*\*

\* удаление всего поддерва, корнем которого является процесс с указанным id

\* в случае успеха дерево перебалансируется

\* в случае, если такого процесса нет, то метод вернет false

\*/

bool delete\_sub\_tree(int32\_t\* pids, int32\_t len);

/\*\*

\* удаление процесса с номером pid

\* в случае неуспеха вернет false

\*/

bool remove(int32\_t pid);

/\*\*

\* получить pid родителя

\* в случае неуспеха вернется -1

\*/

int32\_t get\_parent\_pid(int32\_t pid);

/\*\*

\* получить pid корня

\* в случае неудачи вернет -1

\*/

int32\_t get\_root\_pid();

/\*\*

\* деструктор по умолчанию

\*/

~avl\_tree() = default;

};

**avl.cpp:**

#include "avl.hpp"

void check\_and\_send\_rebind(std::shared\_ptr<tree\_node> node, std::shared\_ptr<tree\_node> new\_parent) {

if (!node.use\_count()) {

return;

}

if (!new\_parent.use\_count()) {

mm\_send\_rebind(node->pid, -1);

}

else {

mm\_send\_rebind(node->pid, new\_parent->pid);

}

}

void check\_and\_send\_rebind(std::shared\_ptr<tree\_node> node, std::weak\_ptr<tree\_node> new\_parent) {

if (!node.use\_count()) {

return;

}

if (!new\_parent.use\_count()) {

mm\_send\_rebind(node->pid, -1);

}

else {

mm\_send\_rebind(node->pid, new\_parent.lock()->pid);

}

}

// получить pid корня - начало

int32\_t avl\_tree::get\_root\_pid() {

if (\_root == nullptr) {

return -1;

}

return \_root->pid;

}

// получить pid корня - конец

// получить pid родителя - начало

int32\_t avl\_tree::get\_parent\_pid(int32\_t pid) {

std::shared\_ptr<tree\_node> node = \_find(pid, \_root);

if (node == nullptr || node == \_root) {

return -1;

}

return node->parent.lock()->pid;

}

// получить pid родителя - конец

// поиск по дереву - начало

bool avl\_tree::search(int32\_t pid, int32\_t\* path, int32\_t\* path\_len) {

\*path\_len = 0;

if (!\_search(pid, path, \_root, path\_len)) {

return false;

}

(\*path\_len)++;

return true;

}

bool avl\_tree::\_search(int32\_t pid, int32\_t\* path, std::shared\_ptr<tree\_node> node, int32\_t\* height) {

if (node == nullptr) {

\*height = 0;

return false;

}

else if (pid == node->pid) {

path[\*height] = node->pid;

return true;

}

else {

path[\*height] = node->pid;

std::shared\_ptr<tree\_node> to = pid < node->pid ? node->left : node->right;

(\*height)++;

return \_search(pid, path, to, height);

}

}

// поиск по дереву - конец

// повороты - начало

void avl\_tree::\_left\_rotate(std::shared\_ptr<tree\_node> node) {

int32\_t node\_b = node->balance;

std::shared\_ptr<tree\_node> right\_son = node->right; //запоминаем b

int32\_t rs\_b = right\_son->balance;

if (right\_son == nullptr) {

return;

}

node->right = right\_son->left; // перевешиваем a->b на a->q

if (right\_son->left != nullptr) {

check\_and\_send\_rebind(right\_son->left, node);

right\_son->left->parent = node; // связываем q->a

}

check\_and\_send\_rebind(right\_son, node->parent);

right\_son->parent = node->parent; // связываем b->p

if (node == \_root) {

\_root = right\_son;

}

else if (node == node->parent.lock()->left) {

node->parent.lock()->left = right\_son; // перевешиваем p->a на p->b

}

else {

node->parent.lock()->right = right\_son;

}

right\_son->left = node; // перевешиваем b->q на b->a

check\_and\_send\_rebind(node, right\_son);

node->parent = right\_son; // связываем a->b

node->balance++;

right\_son->balance++;

if (node\_b == -2 && rs\_b == -1) {

node->balance = 0;

right\_son->balance = 0;

}

else if (node\_b == -2 && rs\_b == 0) {

node->balance = -1;

right\_son->balance = 1;

}

}

void avl\_tree::\_right\_rotate(std::shared\_ptr<tree\_node> node) {

int32\_t node\_b = node->balance;

std::shared\_ptr<tree\_node> left\_son = node->left; // запоминаем a

int32\_t ls\_b = left\_son->balance;

if (left\_son == nullptr) {

return;

}

node->left = left\_son->right; // перевешиваем b->a на b->q

if (left\_son->right != nullptr) {

check\_and\_send\_rebind(left\_son->right, node);

left\_son->right->parent = node; // связываем q->b

}

check\_and\_send\_rebind(left\_son, node->parent);

left\_son->parent = node->parent; // связываем a->p

if (node == \_root) {

\_root = left\_son;

}

else if (node == node->parent.lock()->right) {

node->parent.lock()->right = left\_son; // перевешиваем p->b на p->a

}

else {

node->parent.lock()->left = left\_son;

}

left\_son->right = node; // перевешиваем a->q на a->b

check\_and\_send\_rebind(node, left\_son);

node->parent = left\_son; // связываем b->a

node->balance--;

left\_son->balance--;

if (node\_b == 2 && ls\_b == 1) {

node->balance = 0;

left\_son->balance = 0;

}

else if (node\_b == 2 && ls\_b == 0) {

node->balance = 1;

left\_son->balance = -1;

}

}

// повороты - конец

// перебалансировка - начало

std::shared\_ptr<tree\_node> avl\_tree::\_rebalance(std::shared\_ptr<tree\_node> node) {

if (node->balance == -2) {

if (node->right->balance > 0) {

\_right\_rotate(node->right);

}

\_left\_rotate(node);

}

else {

if (node->left->balance < 0)

\_left\_rotate(node->left);

\_right\_rotate(node);

}

return node->parent.lock();

}

// перебалансировка - конец

// вставка в дерево - начало

bool avl\_tree::insert(int32\_t pid) {

if (\_root == nullptr) {

// отдельный случай свзязывания с мастером

\_root = std::shared\_ptr<tree\_node>(new tree\_node(pid));

return true;

}

else {

int ret = \_insert(pid, \_root);

return ret;

}

}

bool avl\_tree::\_insert(int32\_t pid, std::shared\_ptr<tree\_node> node) {

if (pid == node->pid) {

return false;

}

else if (pid < node->pid) {

if (node->left == nullptr) {

node->left = std::shared\_ptr<tree\_node>(new tree\_node(pid));

node->left->parent = node; // связывание НОВОЙ ноды с родителем

\_go\_up\_insert(node, node->left);

return true;

}

else {

return \_insert(pid, node->left);

}

}

else {

if (node->right == nullptr) {

node->right = std::shared\_ptr<tree\_node>(new tree\_node(pid));

node->right->parent = node; // связывание НОВОЙ ноды с родителем

\_go\_up\_insert(node, node->right);

return true;

}

else {

return \_insert(pid, node->right);

}

}

}

void avl\_tree::\_go\_up\_insert(std::shared\_ptr<tree\_node> node, std::shared\_ptr<tree\_node> prev) {

if (prev == node->left) {

node->balance++;

}

else {

node->balance--;

}

if (node->balance == 0) {

return;

}

else if (std::abs(node->balance) == 1) {

if (node == \_root) {

return;

}

\_go\_up\_insert(node->parent.lock(), node);

}

else {

node = \_rebalance(node);

if (node->balance == 0 || node == \_root) {

return;

}

\_go\_up\_insert(node->parent.lock(), node);

}

}

// вставка в дерево - конец

// вывод дерева на экран - начало

void avl\_tree::print() {

\_print("", \_root, false, 1);

}

void avl\_tree::\_print(const std::string& prefix, std::shared\_ptr<tree\_node> node, bool is\_left, int32\_t height) {

if (node != nullptr) {

std::string new\_prefix = "";

for (int32\_t i = 0; i < height; ++i) {

new\_prefix += " ";

}

\_print(new\_prefix, node->left, true, height + 1);

std::cout << prefix;

if (height == 1) {

std::cout << "───";

}

else {

std::cout << (is\_left ? "┌──" : "└──");

}

std::cout << node->pid << "\n";

\_print(new\_prefix, node->right, false, height + 1);

}

}

// вывод дерева на экран - конец

// удаление поддерева - начало

bool avl\_tree::delete\_sub\_tree(int32\_t\* pids, int32\_t len) {

for (int32\_t i = 0; i < len; ++i) {

int32\_t pid = pids[i];

std::shared\_ptr<tree\_node> to\_delete = \_find(pid, \_root);

if (to\_delete == nullptr) {

continue;

}

if (to\_delete == \_root) {

\_root = nullptr;

}

\_delete\_sub\_tree(to\_delete);

}

if (\_root != nullptr) {

\_reconstruct();

}

return true;

}

std::shared\_ptr<tree\_node> avl\_tree::\_find(int32\_t pid, std::shared\_ptr<tree\_node> node) {

if (node == nullptr) {

return nullptr;

}

else if (pid == node->pid) {

return node;

}

else {

std::shared\_ptr<tree\_node> to = pid < node->pid ? node->left : node->right;

return \_find(pid, to);

}

}

void avl\_tree::\_delete\_sub\_tree(std::shared\_ptr<tree\_node> node) {

if (node == nullptr) {

return;

}

else {

if (node->parent.lock() != nullptr) {

if (node->parent.lock()->left == node) {

node->parent.lock()->left = nullptr;

}

else {

node->parent.lock()->right = nullptr;

}

node->parent.lock() = nullptr;

}

}

}

void avl\_tree::\_reconstruct() {

std::shared\_ptr<tree\_node> old\_root = \_root;

\_root = std::shared\_ptr<tree\_node>(new tree\_node(old\_root->pid));

\_rec\_reconstruct(\_root, old\_root->left);

\_rec\_reconstruct(\_root, old\_root->right);

}

void avl\_tree::\_rec\_reconstruct(std::shared\_ptr<tree\_node>& new\_root, std::shared\_ptr<tree\_node> node) {

if (node == nullptr) {

return;

}

\_insert(node->pid, new\_root);

\_rec\_reconstruct(new\_root, node->left);

\_rec\_reconstruct(new\_root, node->right);

}

// удаление поддерева - конец

// удаление вершины дерева - начало

bool avl\_tree::remove(int32\_t pid) {

std::shared\_ptr<tree\_node> node = \_root;

while ((node != nullptr) && (pid != node->pid)) {

std::shared\_ptr<tree\_node> to = (pid < node->pid) ? node->left : node->right;

node = to;

}

if (node == nullptr) {

return false;

}

\_remove(node);

return true;

}

void avl\_tree::\_remove(std::shared\_ptr<tree\_node> node) {

std::shared\_ptr<tree\_node> to\_delete = node;

int32\_t to\_delete\_balance = to\_delete->balance;

std::shared\_ptr<tree\_node> to\_replace;

std::shared\_ptr<tree\_node> to\_replace\_parent;

if (node->left == nullptr) {

to\_replace = node->right;

if (to\_replace != nullptr) {

check\_and\_send\_rebind(to\_replace, node->parent);

to\_replace->parent = node->parent; // (!!!!!!!) связываение to\_replace->node.parent

to\_replace\_parent = node->parent.lock();

}

else {

to\_replace\_parent = node->parent.lock();

if (node == \_root) {

to\_replace\_parent = nullptr;

\_root = nullptr;

}

}

if (\_root != nullptr) {

if (node != \_root) {

if (node->parent.lock()->left == node) {

node->parent.lock()->left = to\_replace;

}

else {

node->parent.lock()->right = to\_replace;

}

}

else {

\_root = to\_replace;

}

}

}

else if (node->right == nullptr) {

to\_replace = node->left;

check\_and\_send\_rebind(to\_replace, node->parent);

to\_replace->parent = node->parent; // (!!!!!!) связывание to\_replace->node.parent

to\_replace\_parent = node->parent.lock();

if (node != \_root) {

if (node->parent.lock()->left == node) {

node->parent.lock()->left = to\_replace;

}

else {

node->parent.lock()->right = to\_replace;

}

}

else {

\_root = to\_replace;

}

}

else { // если есть дети и слева и справа

std::shared\_ptr<tree\_node> min\_in\_right = node->right;

while (min\_in\_right->left != nullptr) {

min\_in\_right = min\_in\_right->left;

}

to\_delete = min\_in\_right;

to\_delete\_balance = to\_delete->balance;

to\_replace = to\_delete->right;

if (to\_delete->parent.lock() == node) {

if (to\_replace != nullptr) {

check\_and\_send\_rebind(to\_replace, to\_delete);

to\_replace->parent = to\_delete; // (!!!!!!!!!) связывание to\_replace->to\_delete

}

to\_replace\_parent = to\_delete;

}

else {

to\_delete->parent.lock()->left = to\_replace;

if (to\_replace != nullptr) {

check\_and\_send\_rebind(to\_replace, to\_delete->parent);

to\_replace->parent = to\_delete->parent; // (!!!!!!!!!) связывание to\_replace->to\_delete.parent

}

to\_replace\_parent = to\_delete->parent.lock();

to\_delete->right = node->right;

check\_and\_send\_rebind(to\_delete->right, to\_delete);

to\_delete->right->parent = to\_delete; // (!!!!!!!!!) связывание to\_delete.right->to\_delete

}

if (node != \_root) {

if (node->parent.lock()->left == node) {

node->parent.lock()->left = to\_delete;

}

else {

node->parent.lock()->right = to\_delete;

}

}

else {

\_root = to\_delete;

}

check\_and\_send\_rebind(to\_delete, node->parent);

to\_delete->parent = node->parent; // (!!!!!!!!!) связывание to\_delete -> node.parent

to\_delete->left = node->left;

check\_and\_send\_rebind(to\_delete->left, to\_delete);

to\_delete->left->parent = to\_delete; // (!!!!!!!!!) связывание to\_delete.left -> to\_delete

to\_delete->balance = node->balance;

}

if (to\_replace\_parent != nullptr) {

\_go\_up\_remove(to\_replace\_parent, to\_replace);

}

}

void avl\_tree::\_go\_up\_remove(std::shared\_ptr<tree\_node> node, std::shared\_ptr<tree\_node> prev) {

if (node->left == nullptr && node->right == nullptr) {

node->balance = 0;

}

else {

if (prev == node->left) {

node->balance--;

}

else {

node->balance++;

}

}

if (std::abs(node->balance) == 1) {

return;

}

else if (node->balance == 0) {

if (node == \_root) {

return;

}

\_go\_up\_remove(node->parent.lock(), node);

}

else {

node = \_rebalance(node);

if (node->balance == 0 || node == \_root) {

return;

}

\_go\_up\_remove(node->parent.lock(), node);

}

}

// удаление вершины дерева - конец

**zmq\_handle.h:**

#pragma once

#include <zmq.h>

#include <zmq\_utils.h>

#include <stdio.h>

#include <stdint.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include "defines.h"

typedef enum {

exec\_cmd,

hrbt\_cmd,

rebind\_cmd,

relax\_cmd

} cmd\_type;

typedef struct {

char text[MAX\_STRLEN];

char pattern[MAX\_STRLEN];

int32\_t text\_size;

int32\_t pattern\_size;

int32\_t to;

cmd\_type cmd;

int32\_t sleep\_time;

int32\_t change\_to;

} event;

typedef enum {

left,

right,

hrbt

} socket\_type;

void create\_message(zmq\_msg\_t\* msg, event\* e);

void init\_cmp\_name(int pid, char\* name, socket\_type node\_type);

void send\_to(void\* socket, event\* e);

void mm\_send\_relax();

void mm\_send\_rebind(int id, int target\_id);

void print\_err\_cmp(int32\_t pid);

void print\_err\_mas();

int32\_t init\_master\_socket(void\*\* context, void\*\* pub, void\*\* sub);

int32\_t init\_computing\_socket( void\*\* context, void\*\* left\_pub, void\*\* right\_pub,

void\*\* parent\_sub, void\*\* hrbt\_pub, char\* socket\_name\_l, char\* socket\_name\_r, char\* socket\_name\_h, char\* parent\_name, int32\_t pid);

int32\_t deinit\_master\_socket(void\* context, void\* pub, void\* sub);

int32\_t deinit\_computing\_socket(void\* context, void\* left\_pub, void\* right\_pub,

void\* parent\_sub, void\* hrbt\_pub, int32\_t pid);

**zmq\_handle.c:**

#include "zmq\_handle.h"

static const int32\_t TIME\_MS = 100;

void\* EXEC\_PUB;

void\* HEARTBIT\_SUB;

void

create\_message(zmq\_msg\_t\* msg, event\* e) {

zmq\_msg\_init\_size(msg, sizeof(event));

memcpy(zmq\_msg\_data(msg), e, sizeof(event));

}

void

init\_cmp\_name(int pid, char\* name, socket\_type node\_type) {

char pid\_str[MAX\_STRLEN];

sprintf(pid\_str, "%d", pid);

if (node\_type == left) {

strcpy(name, CMP\_SOCKET\_PATTERN\_L);

} else if (node\_type == right) {

strcpy(name, CMP\_SOCKET\_PATTERN\_R);

}

strcat(name, pid\_str);

}

void

send\_to(void\* socket, event\* e) {

zmq\_msg\_t message;

zmq\_msg\_init(&message);

create\_message(&message, e);

zmq\_msg\_send(&message, socket, 0);

zmq\_msg\_close(&message);

}

void

mm\_send\_rebind(int id, int target\_id) {

event sent\_cmd;

sent\_cmd.to = id;

sent\_cmd.cmd = rebind\_cmd;

sent\_cmd.change\_to = target\_id;

zmq\_msg\_t zmqmsg;

zmq\_msg\_init\_size(&zmqmsg, sizeof(event));

memcpy(zmq\_msg\_data(&zmqmsg), &sent\_cmd, sizeof(event));

int send = zmq\_msg\_send(&zmqmsg, EXEC\_PUB, 0);

zmq\_msg\_close(&zmqmsg);

}

void

mm\_send\_relax() {

event sent\_cmd;

sent\_cmd.cmd = relax\_cmd;

zmq\_msg\_t zmqmsg;

zmq\_msg\_init\_size(&zmqmsg, sizeof(event));

memcpy(zmq\_msg\_data(&zmqmsg), &sent\_cmd, sizeof(event));

int send = zmq\_msg\_send(&zmqmsg, EXEC\_PUB, 0);

zmq\_msg\_close(&zmqmsg);

}

void

print\_err\_cmp(int32\_t pid) {

char msg[MAX\_STRLEN];

sprintf(msg, "ERROR, PID %d", pid);

perror(msg);

}

void

print\_err\_mas() {

perror("ERROR, MASTER: ");

}

int32\_t

init\_master\_socket(void\*\* context, void\*\* pub, void\*\* sub) {

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

if (\*context == NULL) {

fprintf(stderr, "MASTER: Unable to create context.\n");

return MASTER\_CTX\_CREATE\_ERR;

}

void\* publisher = zmq\_socket(\*context, ZMQ\_PUB);

if (publisher == NULL) {

print\_err\_mas();

return MASTER\_PUB\_CREATE\_ERR;

}

int32\_t pc = zmq\_bind(publisher, MASTER\_SOCKET\_PUB);

if (pc != 0) {

print\_err\_mas();

return MASTER\_PUB\_BIND\_ERR;

}

\*pub = publisher;

void\* subscriber = zmq\_socket(\*context, ZMQ\_SUB);

if (subscriber == NULL) {

print\_err\_mas();

return MASTER\_SUB\_CREATE\_ERR;

}

zmq\_setsockopt(subscriber, ZMQ\_SUBSCRIBE, NULL, 0);

zmq\_setsockopt(subscriber, ZMQ\_CONNECT\_TIMEOUT, &TIME\_MS, sizeof(TIME\_MS));

pc = zmq\_bind(subscriber, MASTER\_SOCKET\_SUB);

if (pc != 0) {

print\_err\_mas();

return MASTER\_SUB\_BIND\_ERR;

}

\*sub = subscriber;

return OK;

}

int32\_t

init\_computing\_socket( void\*\* context, void\*\* left\_pub, void\*\* right\_pub, void\*\* parent\_sub,

void\*\* hrbt\_pub, char\* socket\_name\_l, char\* socket\_name\_r,

char\* socket\_name\_h, char\* parent\_name, int32\_t pid) {

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

if (\*context == NULL) {

fprintf(stderr, "PID %d: Unable to create context.\n", pid);

return CLIENT\_CTX\_CREATE\_ERR;

}

void\* left\_publisher = zmq\_socket(\*context, ZMQ\_PUB);

if (left\_publisher == NULL) {

print\_err\_cmp(pid);

return CLIENT\_PUB\_L\_CREATE\_ERR;

}

int32\_t pc = zmq\_bind(left\_publisher, socket\_name\_l);

if (pc != 0) {

print\_err\_cmp(pid);

return CLIENT\_PUB\_L\_BIND\_ERR;

}

\*left\_pub = left\_publisher;

void\* right\_publisher = zmq\_socket(\*context, ZMQ\_PUB);

if (right\_publisher == NULL) {

print\_err\_cmp(pid);

return CLIENT\_PUB\_R\_CREATE\_ERR;

}

pc = zmq\_bind(right\_publisher, socket\_name\_r);

if (pc != 0) {

print\_err\_cmp(pid);

return CLIENT\_PUB\_R\_BIND\_ERR;

}

\*right\_pub = right\_publisher;

void\* hrbt\_publisher = zmq\_socket(\*context, ZMQ\_PUB);

if (hrbt\_publisher == NULL) {

print\_err\_cmp(pid);

return CLIENT\_PUB\_H\_CREATE\_ERR;

}

pc = zmq\_connect(hrbt\_publisher, socket\_name\_h);

if (pc != 0) {

print\_err\_cmp(pid);

return CLIENT\_PUB\_H\_CON\_ERR;

}

\*hrbt\_pub = hrbt\_publisher;

void\* parent\_subscriber = zmq\_socket(\*context, ZMQ\_SUB);

if (parent\_subscriber == NULL) {

print\_err\_cmp(pid);

return CLIENT\_SUB\_P\_CREATE\_ERR;

}

zmq\_setsockopt(parent\_subscriber, ZMQ\_SUBSCRIBE, NULL, 0);

zmq\_setsockopt(parent\_subscriber, ZMQ\_CONNECT\_TIMEOUT, &TIME\_MS, sizeof(TIME\_MS));

pc = zmq\_connect(parent\_subscriber, parent\_name);

if (pc != 0) {

print\_err\_cmp(pid);

return CLIENT\_SUB\_P\_CON\_ERR;

}

\*parent\_sub = parent\_subscriber;

return OK;

}

int32\_t

deinit\_master\_socket(void\* context, void\* pub, void\* sub) {

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

print\_err\_mas();

return MASTER\_CLOSE\_PUB\_ERR;

}

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

print\_err\_mas();

return MASTER\_CLOSE\_SUB\_ERR;

}

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

print\_err\_mas();

return MASTER\_CLOSE\_CTX\_ERR;

}

return OK;

}

int32\_t

deinit\_computing\_socket(void\* context, void\* left\_pub, void\* right\_pub,

void\* parent\_sub, void\* hrbt\_pub, int32\_t pid) {

if (zmq\_close(left\_pub) != 0) {

print\_err\_cmp(pid);

return CLIENT\_CLOSE\_PUB\_L\_ERR;

}

if (zmq\_close(right\_pub) != 0) {

print\_err\_cmp(pid);

return CLIENT\_CLOSE\_PUB\_R\_ERR;

}

if (zmq\_close(hrbt\_pub) != 0) {

print\_err\_cmp(pid);

return CLIENT\_CLOSE\_PUB\_H\_ERR;

}

if (zmq\_close(parent\_sub) != 0) {

print\_err\_cmp(pid);

return CLIENT\_CLOSE\_SUB\_P\_ERR;

}

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

print\_err\_cmp(pid);

return CLIENT\_CLOSE\_CTX\_ERR;

}

return OK;

}

**client.c:**

#include <stdio.h>

#include <unistd.h>

#include <zmq.h>

#include <stdbool.h>

#include <stdint.h>

#include <string.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <limits.h>

#include "zmq\_handle.h"

#include "defines.h"

int32\_t CLIENT\_PID;

int32\_t PARENT\_PID;

int32\_t TMP\_PARENT\_PID;

static char SOCKET\_NAME\_L[MAX\_STRLEN];

static char SOCKET\_NAME\_R[MAX\_STRLEN];

static char SOCKET\_NAME\_H[MAX\_STRLEN];

static char PARENT\_NAME[MAX\_STRLEN];

static void\* CONTEXT;

static void\* LEFT\_SOCKET;

static void\* RIGHT\_SOCKET;

static void\* PARENT\_SOCKET;

static void\* HRBT\_SOCKET;

void

process\_sigterm(int32\_t sig) {

printf("Computing node %d finishing it's work...\n", CLIENT\_PID);

deinit\_computing\_socket(CONTEXT, LEFT\_SOCKET, RIGHT\_SOCKET,

PARENT\_SOCKET, HRBT\_SOCKET, CLIENT\_PID);

exit(EXIT\_SUCCESS);

}

void

mm\_pass\_rebind(int id, event\* cmd) {

if (CLIENT\_PID == id) {

TMP\_PARENT\_PID = cmd->change\_to;

} else {

event sent\_cmd;

sent\_cmd.cmd = rebind\_cmd;

sent\_cmd.to = id;

sent\_cmd.change\_to = cmd->change\_to;

zmq\_msg\_t zmqmsg;

zmq\_msg\_init\_size(&zmqmsg, sizeof(event));

memcpy(zmq\_msg\_data(&zmqmsg), &sent\_cmd, sizeof(event));

if (id < CLIENT\_PID) {

zmq\_msg\_send(&zmqmsg, LEFT\_SOCKET, 0);

} else {

zmq\_msg\_send(&zmqmsg, RIGHT\_SOCKET, 0);

}

zmq\_msg\_close(&zmqmsg);

}

}

void

mm\_pass\_relax() {

{

event sent\_cmd;

sent\_cmd.cmd = relax\_cmd;

zmq\_msg\_t zmqmsg;

zmq\_msg\_init\_size(&zmqmsg, sizeof(event));

memcpy(zmq\_msg\_data(&zmqmsg), &sent\_cmd, sizeof(event));

zmq\_msg\_send(&zmqmsg, LEFT\_SOCKET, 0);

zmq\_msg\_close(&zmqmsg);

}

{

event sent\_cmd;

sent\_cmd.cmd = relax\_cmd;

zmq\_msg\_t zmqmsg;

zmq\_msg\_init\_size(&zmqmsg, sizeof(event));

memcpy(zmq\_msg\_data(&zmqmsg), &sent\_cmd, sizeof(event));

zmq\_msg\_send(&zmqmsg, RIGHT\_SOCKET, 0);

zmq\_msg\_close(&zmqmsg);

}

{

if (PARENT\_PID != TMP\_PARENT\_PID) {

zmq\_disconnect(PARENT\_SOCKET, PARENT\_NAME);

if (CLIENT\_PID < TMP\_PARENT\_PID) {

if (TMP\_PARENT\_PID == -1) {

sprintf(PARENT\_NAME, MASTER\_SOCKET\_PUB);

} else {

sprintf(PARENT\_NAME, CMP\_SOCKET\_PATTERN\_L"%d", TMP\_PARENT\_PID);

}

} else {

if (TMP\_PARENT\_PID == -1) {

sprintf(PARENT\_NAME, MASTER\_SOCKET\_PUB);

} else {

sprintf(PARENT\_NAME, CMP\_SOCKET\_PATTERN\_R"%d", TMP\_PARENT\_PID);

}

}

zmq\_connect(PARENT\_SOCKET, PARENT\_NAME);

PARENT\_PID = TMP\_PARENT\_PID;

}

}

}

int32\_t

naive(char\* text, char\* pattern, int32\_t text\_size, int32\_t pattern\_size, int32\_t\* res) {

int32\_t i = 0;

int32\_t j = 0;

int32\_t matching\_c = 0;

while (i < text\_size) {

int32\_t h = i;

while (h < text\_size && j < pattern\_size && text[h] == pattern[j]) {

h++;

j++;

}

if (j == pattern\_size) {

res[matching\_c] = i;

matching\_c++;

}

j = 0;

i++;

}

return matching\_c;

}

void

compute(event\* e) {

int32\_t res[MAX\_STRLEN];

int32\_t len = naive(e->text, e->pattern, e->text\_size, e->pattern\_size, res);

printf("Result is:\n");

if (len == 0) {

printf("There are no occurrences.\n");

} else {

for (int32\_t i = 0; i < len; ++i) {

printf("%d;", res[i]);

}

printf("\n");

}

}

void

computing\_loop() {

while (true) {

zmq\_msg\_t message;

zmq\_msg\_init(&message);

zmq\_msg\_init\_size(&message, sizeof(event));

zmq\_msg\_recv(&message, PARENT\_SOCKET, 0);

event e;

memcpy(&e, zmq\_msg\_data(&message), sizeof(event));

zmq\_msg\_close(&message);

if (e.cmd == exec\_cmd) {

if (e.to != CLIENT\_PID) {

if (e.to > CLIENT\_PID) {

send\_to(RIGHT\_SOCKET, &e);

} else {

send\_to(LEFT\_SOCKET, &e);

}

} else {

compute(&e);

}

} else if (e.cmd == hrbt\_cmd) {

send\_to(LEFT\_SOCKET, &e);

send\_to(RIGHT\_SOCKET, &e);

for (int32\_t i = 0; i < 5; ++i) {

event e\_copy = e;

e\_copy.to = CLIENT\_PID;

send\_to(HRBT\_SOCKET, &e\_copy);

usleep(1e3 \* e\_copy.sleep\_time);

}

} else if (e.cmd == rebind\_cmd) {

mm\_pass\_rebind(e.to, &e);

} else {

mm\_pass\_relax();

}

}

}

int

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

if (signal(SIGTERM, process\_sigterm) == SIG\_ERR) {

perror("ERROR: ");

return CLIENT\_SIG\_ERR;

}

CLIENT\_PID = atoi(argv[1]);

strcpy(PARENT\_NAME,argv[2]);

PARENT\_PID = atoi(argv[3]);

TMP\_PARENT\_PID = atoi(argv[3]);

init\_cmp\_name(CLIENT\_PID, SOCKET\_NAME\_L, left);

init\_cmp\_name(CLIENT\_PID, SOCKET\_NAME\_R, right);

strcpy(SOCKET\_NAME\_H, MASTER\_SOCKET\_SUB);

if (init\_computing\_socket( &CONTEXT, &LEFT\_SOCKET, &RIGHT\_SOCKET, &PARENT\_SOCKET,

&HRBT\_SOCKET, SOCKET\_NAME\_L, SOCKET\_NAME\_R, SOCKET\_NAME\_H,

PARENT\_NAME, CLIENT\_PID) != 0) {

return CLIENT\_INIT\_ERR;

}

computing\_loop();

if (deinit\_computing\_socket(CONTEXT, LEFT\_SOCKET, RIGHT\_SOCKET,

PARENT\_SOCKET, HRBT\_SOCKET, CLIENT\_PID) != 0) {

return CLIENT\_DEINIT\_ERR;

}

return 0;

}

**server.c:**

#include <stdio.h>

#include <unistd.h>

#include <zmq.h>

#include <stdbool.h>

#include <stdint.h>

#include <string.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <limits.h>

#include "../avl/avl\_c\_externs.h"

#include "ui.h"

#include "zmq\_handle.h"

#include "defines.h"

static int32\_t REAL\_PID\_TABLE[MAX\_PID\_NUM];

static avl\_tree\* AVL\_TREE\_PTR;

static void\* CONTEXT;

extern void\* EXEC\_PUB;

extern void\* HEARTBIT\_SUB;

void

create\_event(event\* e, char\* text, char\* pattern, int32\_t pid, cmd\_type cmd, int32\_t sleep\_time) {

strcpy(e->text, text);

strcpy(e->pattern, pattern);

e->text\_size = 0;

for (int32\_t i = 0; text[i] != '\0'; i++) {

e->text\_size++;

}

e->pattern\_size = 0;

for (int32\_t i = 0; pattern[i] != '\0'; i++) {

e->pattern\_size++;

}

e->to = pid;

e->cmd = cmd;

e->sleep\_time = sleep\_time;

}

void

init\_table() {

for (int32\_t i = 0; i < MAX\_PID\_NUM; ++i) {

REAL\_PID\_TABLE[i] = VOID\_PID;

}

}

bool

remove\_node(int32\_t pid) {

if (pid < 0 || pid > MAX\_PID\_NUM - 1) {

printf("Invalid pid.\n");

return false;

}

int32\_t rpid = REAL\_PID\_TABLE[pid];

if (rpid == VOID\_PID) {

printf("There is no node with this pid.\n");

return false;

}

REAL\_PID\_TABLE[pid] = VOID\_PID;

remove\_from\_tree(AVL\_TREE\_PTR, pid);

sleep(1);

kill(rpid, SIGTERM);

return true;

}

void

terminate\_all\_nodes() {

for (int32\_t i = 0; i < MAX\_PID\_NUM; ++i) {

if (REAL\_PID\_TABLE[i] != VOID\_PID) {

remove\_node(i);

}

}

}

void

process\_signal(int32\_t sig) {

terminate\_all\_nodes();

deinit\_avl(AVL\_TREE\_PTR);

deinit\_master\_socket(CONTEXT, EXEC\_PUB, HEARTBIT\_SUB);

exit(EXIT\_SUCCESS);

}

bool

create\_node(int32\_t pid) {

if (pid < 0 || pid > MAX\_PID\_NUM - 1) {

printf("Invalid pid.\n");

return false;

}

if (REAL\_PID\_TABLE[pid] != VOID\_PID) {

printf("Node with this pid has already created.\n");

return false;

}

char parent\_name[MAX\_STRLEN];

int32\_t path\_len = MAX\_STRLEN;

int32\_t path[MAX\_STRLEN];

add\_to\_tree(AVL\_TREE\_PTR, pid);

get\_path(AVL\_TREE\_PTR, pid, &path\_len, path);

int32\_t parent\_pid = -1;

if (path\_len == 1) {

strcpy(parent\_name, MASTER\_SOCKET\_PUB);

} else {

parent\_pid = get\_parent\_id(AVL\_TREE\_PTR, pid);

if (pid < parent\_pid) {

init\_cmp\_name(parent\_pid, parent\_name, left);

} else {

init\_cmp\_name(parent\_pid, parent\_name, right);

}

}

int32\_t fv = fork();

if (fv < 0) {

remove\_from\_tree(AVL\_TREE\_PTR, pid);

printf("Unable to create node, fork err.\n");

return false;

} else if (fv == 0) {

char client\_id[MAX\_STRLEN];

sprintf(client\_id, "%d", pid);

char client\_name[MAX\_STRLEN];

sprintf(client\_name, "client\_%d", pid);

char parent\_id[MAX\_STRLEN];

sprintf(parent\_id, "%d", parent\_pid);

execl(CLIENT\_PROG\_NAME, client\_name, client\_id, parent\_name, parent\_id, NULL);

} else {

REAL\_PID\_TABLE[pid] = fv;

}

return true;

}

bool

send\_exec(char\* text, char\* pattern, int32\_t pid) {

if (pid < 0 || pid > MAX\_PID\_NUM - 1) {

printf("Invalid pid.\n");

return false;

}

if (REAL\_PID\_TABLE[pid] == VOID\_PID) {

printf("There is no node with this pid.\n");

return false;

}

event e;

create\_event(&e, text, pattern, pid, exec\_cmd, 0);

send\_to(EXEC\_PUB, &e);

return true;

}

void

send\_heartbit(int32\_t input\_time) {

event e;

create\_event(&e, "", "", -1, hrbt\_cmd, input\_time);

send\_to(EXEC\_PUB, &e);

}

void

kill\_child(int32\_t pid) {

int32\_t rpid = REAL\_PID\_TABLE[pid];

if (rpid == VOID\_PID) {

printf("There is no node with this pid.\n");

return;

}

REAL\_PID\_TABLE[pid] = VOID\_PID;

kill(rpid, SIGTERM);

}

void

heartbit(int32\_t input\_time) {

int32\_t time\_to\_wait = 4 \* input\_time;

zmq\_setsockopt(HEARTBIT\_SUB, ZMQ\_RCVTIMEO, &time\_to\_wait, sizeof(time\_to\_wait));

int32\_t heartbit\_table[MAX\_PID\_NUM];

for (int32\_t i = 0; i < MAX\_PID\_NUM; ++i) {

if (REAL\_PID\_TABLE[i] == VOID\_PID) {

heartbit\_table[i] = -1;

} else {

heartbit\_table[i] = 0;

}

}

send\_heartbit(input\_time);

while (true) {

zmq\_msg\_t message;

zmq\_msg\_init(&message);

zmq\_msg\_init\_size(&message, sizeof(event));

int32\_t rv = zmq\_msg\_recv(&message, HEARTBIT\_SUB, 0);

if (rv == -1 && errno == EAGAIN) {

break;

}

event\* e = (event\*)zmq\_msg\_data(&message);

heartbit\_table[e->to]++;

zmq\_msg\_close(&message);

}

int32\_t count\_of\_dead = 0;

int32\_t count\_of\_void = 0;

int32\_t dead\_table[MAX\_PID\_NUM];

for (int32\_t i = 0; i < MAX\_PID\_NUM; ++i) {

if (heartbit\_table[i] == 0) {

printf("OK: %d\n", i);

dead\_table[count\_of\_dead] = i;

count\_of\_dead++;

} else if (heartbit\_table[i] == -1) {

count\_of\_void++;

}

}

if (count\_of\_void == MAX\_PID\_NUM) {

printf("There are no alive nodes now.\n");

} else if (count\_of\_dead == 0) {

printf("All is alive.\n");

}

if (count\_of\_dead != 0) {

delete\_subtree(AVL\_TREE\_PTR, dead\_table, count\_of\_dead);

sleep(1);

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

kill\_child(dead\_table[i]);

}

}

}

void

print\_table() {

for (int32\_t i = 0; i < MAX\_PID\_NUM; ++i) {

if (REAL\_PID\_TABLE[i] != VOID\_PID) {

printf("Client id: %d, pid: %d\n", i, REAL\_PID\_TABLE[i]);

}

}

}

void

server\_loop() {

char cmd[MAX\_STRLEN];

int32\_t pid = VOID\_PID;

while (true) {

input\_val iv = input\_str(cmd);

if (iv == eof) {

printf("Server finishing it's work...\n");

break;

} else if (iv == bad) {

printf("Unknown command, try help.\n");

continue;

}

if (strcmp(cmd, "create") == 0) {

iv = input\_int(&pid);

if (iv == eof) {

printf("Server finishing it's work...\n");

break;

} else if (iv == bad) {

printf("Unknown arg, try help.\n");

continue;

}

if (!create\_node(pid)) {

printf("Unable to create node with pid %d\n", pid);

} else {

printf("OK: %d\n", REAL\_PID\_TABLE[pid]);

}

} else if (strcmp(cmd, "remove") == 0) {

iv = input\_int(&pid);

if (iv == eof) {

printf("Server finishing it's work...\n");

break;

} else if (iv == bad) {

printf("Unknown arg, try help.\n");

continue;

}

if (!remove\_node(pid)) {

printf("Unable to remove node with pid %d\n", pid);

} else {

printf("OK\n");

}

} else if (strcmp(cmd, "exec") == 0) {

char text[MAX\_STRLEN];

char pattern[MAX\_STRLEN];

iv = input\_int(&pid);

if (iv == eof) {

printf("Server finishing it's work...\n");

break;

} else if (iv == bad) {

printf("Unknown arg, try help.\n");

continue;

}

iv = input\_str(text);

if (iv == eof) {

printf("Server finishing it's work...\n");

break;

} else if (iv == bad) {

printf("Unknown arg, try help.\n");

continue;

}

iv = input\_str(pattern);

if (iv == eof) {

printf("Server finishing it's work...\n");

break;

} else if (iv == bad) {

printf("Unknown arg, try help.\n");

continue;

}

send\_exec(text, pattern, pid);

} else if (strcmp(cmd, "help") == 0) {

help();

} else if (strcmp(cmd, "heartbit") == 0) {

int32\_t input\_time;

iv = input\_int(&input\_time);

if (iv == eof) {

printf("Server finishing it's work...\n");

break;

} else if (iv == bad) {

printf("Unknown arg, try help.\n");

continue;

}

heartbit(input\_time);

} else if (strcmp(cmd, "print") == 0) {

print\_tree(AVL\_TREE\_PTR);

} else if (strcmp(cmd, "table") == 0) {

print\_table();

} else {

printf("Unknown command, try help.\n");

}

}

terminate\_all\_nodes();

}

int

main() {

if (signal(SIGSEGV, process\_signal) == SIG\_ERR) {

perror("ERROR: ");

return SERV\_SIG\_ERR;

}

if (signal(SIGINT, process\_signal) == SIG\_ERR) {

perror("ERROR: ");

return SERV\_SIG\_ERR;

}

init\_avl(&AVL\_TREE\_PTR);

init\_table();

if (init\_master\_socket(&CONTEXT, &EXEC\_PUB, &HEARTBIT\_SUB) != 0) {

return SERV\_INIT\_ERR;

}

help();

server\_loop();

deinit\_avl(AVL\_TREE\_PTR);

if (deinit\_master\_socket(CONTEXT, EXEC\_PUB, HEARTBIT\_SUB) != 0) {

return SERV\_DEINIT\_ERR;

}

return 0;

}

**Пример работы**

MacBook-Air-K:os\_lab\_0678 AK$ ./server

+--------------+-------------------------------------------+

| command | description |

|--------------+-------------------------------------------|

|--------------+-------------------------------------------|

| create <pid> | creates computing node |

| | (pids range from 0 to 63) |

|--------------+-------------------------------------------|

| remove <pid> | removes computing node |

| | (pids range from 0 to 63) |

|--------------+-------------------------------------------|

| exec <pid> | exec computing node |

| <text> | finds pattern in text |

| <pattern> | (pids range from 0 to 63) |

|--------------+-------------------------------------------|

| help | print usage |

| | (pids range from 0 to 63) |

|--------------+-------------------------------------------|

| print | print process avl tree |

|--------------+-------------------------------------------|

| heartbit | heartbit all computing nodes |

| <time> | master will wait signal for 4\*time ms |

|--------------+-------------------------------------------|

| table | print pid table |

+--------------+-------------------------------------------+

create 1

OK: 1760

create 2

OK: 1764

create 3

OK: 1768

create 4

OK: 1772

create 5

OK: 1777

create 6

OK: 1781

create 7

OK: 1785

print

┌──1

┌──2

└──3

───4

┌──5

└──6

└──7

heartbit 100

All is alive.

exec 1 aaba a

Result is:

0;1;3;

exec 7 aaba a

Result is:

0;1;3;

remove 4

OK

Computing node 4 finishing it's work...

print

┌──1

┌──2

└──3

───5

└──6

└──7

heartbit 100

All is alive.

exec 1 aaba a

Result is:

0;1;3;

exec 7 aaba a

Result is:

0;1;3;

Server finishing it's work...

Computing node 1 finishing it's work...

Computing node 2 finishing it's work...

Computing node 3 finishing it's work...

Computing node 5 finishing it's work...

Computing node 6 finishing it's work...

Computing node 7 finishing it's work...

**Вывод**

Выполняя данную лабораторную работу, я познакомился с очередью сообщений zeromq. Очереди сообщений очень полезны для межпроцессного и межмашинного сообщения, ведь они являются связующим звеном между различными компонентами в ваших приложениях и обеспечивают надежный и масштабируемый интерфейс взаимодействия с другими подключенными системами и устройствами. Навык использования данной технологии очень пригодится в будущем, так как сейчас много внимания уделяется масштабируемым сервисам с большим числом вычислительных блоков или процессов, которые позволяют параллельно и независимо друг от друга выполнять рутинную работу.

Также данная лабораторная работа показала мне важность проектирования архитектуры приложения с точки зрения масштабируемости и максимальной модульности ПО для более удобной интеграции одних блоков кода в другие.