ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ОТЧЕТ
О ВЫПОЛНЕНИИ ЛАБОРАТОРНЫХ РАБОТ №№ 5-7
ПО ДИСЦИПЛИНЕ
«ОПЕРАЦИОННЫЕ СИСТЕМЫ»
ВАРИАНТ ЗАДАНИЙ №1

Выполнил студент группы М8О-208Б-23
Денисов.К.Д_____
подпись, дата
Проверил и принял
Живалев.Е.А_____
подпись, дата
с оценкой _____

Москва 2024

Задания

- *Топология 3.* Все вычислительные узлы хранятся в бинарном дереве поиска. [parent] — является необязательным параметром.
- *Набора 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
- *Команда проверки 3*
    - Формат команды: heartbit time
    - 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.

Текст программы

Текст программы в приложениях №№ 1, 2.

Вывод

В ходе выполнения лабораторной работы была написана программа на языке C для ОС Linux, реализующая поставленные задания. Я получил практические навыки использования очереди сообщений ZeroMQ, для обмена данными между разными программами. Лабораторная работа помогла мне лучше понять, принцип работы очередей сообщения с отправителями и подписчиками.

Приложение №1 – файл «controller.c»

```
#include <zmq.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <time.h>
#include <signal.h>
#include <stdbool.h>

int max_time = -1;

// Определение структуры Node
typedef struct Node
{
    int id;
    struct Node *left;
    struct Node *right;
} Node;

// Функция создания нового узла
Node *createNode(int id)
{
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->id = id;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

// Функция вставки узла в дерево
Node *insertNode(Node *root, int id)
{
    if (root == NULL)
    {
        return createNode(id);
    }
    if (id < root->id)
    {
        root->left = insertNode(root->left, id);
    }
    else if (id > root->id)
    {
        root->right = insertNode(root->right, id);
    }
    return root;
}

// Функция поиска узла в дереве
Node *findNode(Node *root, int id)
{
    if (root == NULL || root->id == id)
    {
        return root;
    }
```

```
    if (id < root->id)
    {
        return findNode(root->left, id);
    }
    return findNode(root->right, id);
}

// Функция для сердцебиения (heartbeat), отправляет сообщения на порт 5555
void *heartbit_thread(void *arg)
{
    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_PUB);
    zmq_bind(socket, "tcp://*:5555");

    int time = *(int *)arg;
    while (1)
    {
        zmq_send(socket, "HEARTBIT", 8, 0);
        usleep(time * 1000);
    }
    zmq_close(socket);
    zmq_ctx_destroy(context);
    return NULL;
}

// Структура для результата пинга
typedef struct PingResult
{
    int id;
    bool success;
} PingResult;

// Структура для элемента очереди
typedef struct QueueNode
{
    PingResult *data;
    struct QueueNode *next;
} QueueNode;

// Структура для очереди
typedef struct
{
    QueueNode *head;
    QueueNode *tail;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} Queue;

// Функция инициализации очереди
void queue_init(Queue *queue)
{
    queue->head = NULL;
    queue->tail = NULL;
    pthread_mutex_init(&queue->mutex, NULL);
    pthread_cond_init(&queue->cond, NULL);
}
```

```
// Функция добавления элемента в очередь
void enqueue(Queue *queue, PingResult *data)
{
    QueueNode *newNode = (QueueNode *)malloc(sizeof(QueueNode));
    newNode->data = data;
    newNode->next = NULL;

    pthread_mutex_lock(&queue->mutex);

    if (queue->tail == NULL)
    {
        queue->head = newNode;
        queue->tail = newNode;
    }
    else
    {
        queue->tail->next = newNode;
        queue->tail = newNode;
    }

    pthread_cond_signal(&queue->cond);
    pthread_mutex_unlock(&queue->mutex);
}

// Функция извлечения элемента из очереди
PingResult *dequeue(Queue *queue)
{
    pthread_mutex_lock(&queue->mutex);

    while (queue->head == NULL)
    {
        pthread_cond_wait(&queue->cond, &queue->mutex);
    }

    QueueNode *temp = queue->head;
    PingResult *result = temp->data;
    queue->head = queue->head->next;

    if (queue->head == NULL)
    {
        queue->tail = NULL;
    }

    pthread_mutex_unlock(&queue->mutex);
    free(temp);
    return result;
}

// Функция освобождения памяти, выделенной под очередь
void queue_destroy(Queue *queue)
{
    pthread_mutex_destroy(&queue->mutex);
    pthread_cond_destroy(&queue->cond);
}

// Функция отправки команды ping на узел (теперь для асинхронного использования)
void *ping_node(void *arg)
```

```
{
    int id = *(int *)arg;
    free(arg); // Освобождаем выделенную память

    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_REQ);
    char address[256];
    snprintf(address, sizeof(address), "tcp://localhost:5557%d", id);
    zmq_connect(socket, address);

    zmq_send(socket, "ping", strlen("ping"), 0);
    zmq_pollitem_t items[] = {{socket, 0, ZMQ_POLLIN, 0}};
    int rc = zmq_poll(items, 1, 4 * max_time * 1000); // Ждем max_time * 4 секунд

    PingResult *result = (PingResult *)malloc(sizeof(PingResult));
    result->id = id;
    result->success = false;

    if (rc > 0 && (items[0].revents & ZMQ_POLLIN))
    {
        char buffer[256];
        zmq_recv(socket, buffer, 256, 0);
        if (strncmp(buffer, "Pong", 4) == 0)
        {
            result->success = true;
        }
    }

    zmq_close(socket);
    zmq_ctx_destroy(context);

    // Отправляем результат в очередь
    extern Queue pingResultsQueue; // Используем extern
    enqueue(&pingResultsQueue, result);

    return NULL;
}

// Функция для рекурсивного обхода дерева и сбора всех ID узлов
void collectNodeIds(Node *root, int *ids, int *count)
{
    if (root != NULL)
    {
        ids[*count] = root->id;
        (*count)++;
        collectNodeIds(root->left, ids, count);
        collectNodeIds(root->right, ids, count);
    }
}

int pings = 0;
bool pingFail = false;

// Функция для обработки результатов пинга из очереди
void *process_ping_results(void *arg)
{
    extern Queue pingResultsQueue; // Используем extern
```

```
    while (true)
    {
        PingResult *result = dequeue(&pingResultsQueue);
        if (!result->success)
        {
            printf("Node %d: Timeout\n", result->id);
            pingFail = true;
        }
        ++pings;
        free(result);
    }
    return NULL;
}

void send_command_to_node(int id, const char *command)
{
    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_REQ);
    char address[256];
    snprintf(address, sizeof(address), "tcp://localhost:5557%d", id);
    zmq_connect(socket, address);

    zmq_send(socket, command, strlen(command), 0);
    char buffer[256];
    for (int i = 0; i < 256; ++i)
        buffer[i] = '\0';
    zmq_recv(socket, buffer, 256, 0);
    printf("Response from node %d: %s\n", id, buffer);

    zmq_close(socket);
    zmq_ctx_destroy(context);
}

Queue pingResultsQueue;
bool exit_flag = false;
pthread_mutex_t exit_mutex;
pthread_cond_t exit_cond;

int main()
{
    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_REP);
    zmq_bind(socket, "tcp://*:5556");

    Node *root = NULL;
    int heartbit_time = 2000;
    pthread_t heartbit_thread_id;
    pthread_create(&heartbit_thread_id, NULL, heartbit_thread, &heartbit_time);

    // Инициализация очереди и mutex для exit
    queue_init(&pingResultsQueue);
    pthread_mutex_init(&exit_mutex, NULL);
    pthread_cond_init(&exit_cond, NULL);

    // Запускаем поток для обработки результатов ping
    pthread_t results_thread_id;
    pthread_create(&results_thread_id, NULL, process_ping_results, NULL);
```

```
char buffer[256];
while (1)
{
    fgets(buffer, sizeof(buffer), stdin);
    buffer[strcspn(buffer, "\n")] = 0; // Убираем символ новой строки

    if (strncmp(buffer, "create", 6) == 0)
    {
        int id = atoi(buffer + 7);
        if (id <= 0)
        {
            printf("Error: Id must be > 0\n");
        }
        else if (findNode(root, id) != NULL)
        {
            printf("Error: Already exists\n");
        }
        else
        {
            root = insertNode(root, id);
            pid_t pid = fork();
            if (pid == 0)
            {
                // Дочерний процесс
                char id_str[16];
                snprintf(id_str, sizeof(id_str), "%d", id);
                execl("./computational", "computational", id_str, (char *)NULL);
                exit(EXIT_FAILURE);
            }
            else if (pid > 0)
            {
                // Родительский процесс
                char response[256];
                snprintf(response, sizeof(response), "OK: %d", pid);
                printf("%s\n", response);
            }
            else
            {
                perror("fork");
                exit(EXIT_FAILURE);
            }
        }
    }
    else if (strncmp(buffer, "exec", 4) == 0)
    {
        int id = atoi(buffer + 5);
        Node *node = findNode(root, id);
        if (node == NULL)
        {
            printf("Error: Not found\n");
        }
        else
        {
            char *command = strchr(buffer, ' ');
            if (command != NULL)
            {
```

```
            command++;
            while (*command == '0' || *command == '1' || *command == '2' || *command == '3' || *command
== '4' || *command == '5' || *command == '6' || *command == '7' || *command == '8' || *command == '9')
            {
               command++;
            }
            command++;
            send_command_to_node(id, command);
         }
         else
         {
            printf("Error: No command specified\n");
         }
      }
   }
   else if (strncmp(buffer, "heartbit", 8) == 0)
   {
      pings = 0;
      max_time = atoi(buffer + 9);
      int nodeIds[256];
      int count = 0;

      collectNodeIds(root, nodeIds, &count);

      if (count == 0)
      {
         printf("No nodes in the tree to heartbit.\n");
      }
      else
      {
         // Запускаем пинг для каждого узла асинхронно
         for (int i = 0; i < count; ++i)
         {
            pthread_t thread;
            int *id_copy = (int *)malloc(sizeof(int)); // Выделяем память для ID
            *id_copy = nodeIds[i];                     // Копируем ID
            if (pthread_create(&thread, NULL, ping_node, id_copy) != 0)
            {
               perror("pthread_create");
               free(id_copy);
               continue;
            }
         }
         while (pings != count)
         {
            sleep(1);
         }
         if (!pingFail)
            printf("OK\n");
         pingFail = false;
      }
   }
   else if (strncmp(buffer, "exit", 4) == 0)
   {

      pthread_mutex_lock(&exit_mutex);
      exit_flag = true;
```

```
            pthread_cond_signal(&pingResultsQueue.cond);
            pthread_mutex_unlock(&exit_mutex);

            system("pkill -f computational"); // kill other processes
            printf("Exiting...\n");
            break;
        }
    }
    zmq_close(socket);
    zmq_ctx_destroy(context);
    pthread_mutex_destroy(&exit_mutex);
    pthread_cond_destroy(&exit_cond);
    return 0;
}
```

Приложение №2 – файл «computational.c»

```
#include <zmq.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>

void *heartbit_thread(void *arg)
{
    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_SUB);
    zmq_connect(socket, "tcp://localhost:5555");
    zmq_setsockopt(socket, ZMQ_SUBSCRIBE, "", 0);

    int time = *(int *)arg;
    while (1)
    {
        char buffer[256];
        zmq_recv(socket, buffer, 256, 0);
        usleep(time * 1000);
    }
    zmq_close(socket);
    zmq_ctx_destroy(context);
    return NULL;
}

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s <id>\n", argv[0]);
        return 1;
    }

    int id = atoi(argv[1]);
    void *context = zmq_ctx_new();
    void *socket = zmq_socket(context, ZMQ_REP);
    char address[256];
```

```
snprintf(address, sizeof(address), "tcp://*:5557%d", id);
zmq_bind(socket, address);

time_t start_time = 0;
int timer_running = 0;

char buffer[256];
while (1)
{
   zmq_recv(socket, buffer, 256, 0);
   if (strncmp(buffer, "start", 5) == 0)
   {
      if (!timer_running)
      {
         start_time = time(NULL);
         timer_running = 1;
         zmq_send(socket, "Ok", 2, 0);
      }
      else
      {
         zmq_send(socket, "Error", 5, 0);
      }
   }
   else if (strncmp(buffer, "stop", 4) == 0)
   {
      if (timer_running)
      {
         timer_running = 0;
         zmq_send(socket, "Ok", 2, 0);
      }
      else
         zmq_send(socket, "Error", 5, 0);
   }
   else if (strncmp(buffer, "time", 4) == 0)
   {
      if (timer_running)
      {
         char response[256];
         snprintf(response, sizeof(response), "%ld", time(NULL) - start_time);
         zmq_send(socket, response, strlen(response), 0);
      }
      else
         zmq_send(socket, "Error", 5, 0);
   }
   else if (strncmp(buffer, "ping", 4) == 0)
   {
      if (timer_running)
         zmq_send(socket, "Error", 5, 0);
      else
      {
         zmq_send(socket, "Pong", 4, 0);
      }
   }
   else
      zmq_send(socket, "Error", 5, 0);
}
```

```
    zmq_close(socket);
    zmq_ctx_destroy(context);
    return 0;
}
```