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»

СЕРВЕРА СООБЩЕНИЙ

Студент: Харьков Павел Александрович
Группа: М8О–206Б–19
Вариант: 35
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2021.

Постановка задачи

Цель работы

Приобретение практических навыков в:

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

Задание

Реализовать распределенную систему по асинхронной обработке запросов. В данной распределенной системе должно существовать 2 вида узлов: «управляющий» и «вычислительный». Необходимо объединить данные узлы в соответствии с той топологией, которая определена вариантом. Связь между узлами необходимо осуществить при помощи технологии очередей сообщений. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом. При убийстве («kill -9») любого вычислительного узла система должна пытаться максимально сохранять свою работоспособность, а именно все дочерние узлы убитого узла могут стать недоступными, но родительские узлы должны сохранить свою работоспособность.

Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд:

- Создание нового вычислительного узла.
- Удаление существующего вычислительного узла.
- Исполнение команды на вычислительном узле.

Вариант 35:

Топология 3: все вычислительные узлы хранятся в бинарном дереве поиска.

Набор команд 4: поиск подстроки в строке.

Команда проверки 2: команда проверяет доступность конкретного узла.

Общие сведения о программе.

Для написания этой лабораторной работы я решил использовать сервер сообщений ZeroMQ.

Я решил сделать две программы: сервер, которая считывает команды и отправляет вычислительным узла, и клиент, отвечающая за работу вычислительных узлов. Программа сервер компилируется из файла

server.cpp, а клиент из файла client.cpp. Обе программы используют заголовочный файл zeromq.hpp и zeromq.cpp, в котором определены функции для работы сервера сообщений.

Также используется заголовочные файлы: iostream, string, stdio.h, signal.h, unistd.h, zmq.hpp. В программе используются следующие системные вызовы:

- fork — создает копию процесса.
- execl — заменяет текущий образ процесса новым образом процесса.
- kill — посылает сигнал по pid процесса.
- sleep – останавливает работу процесса на время.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы с ZeroMQ.
2. Написать инициализацию и завершение сервера и узлов.
3. Написать считывание команды из консоли и их отправку узлам.
4. Написать обработку команд узлами.
5. Отладить программу и протестировать на тестах.

Основные файлы программы.

server.cpp

```
#include <iostream>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include "zmq.hpp"

#include "zeromq.hpp"

using namespace std;

void* context;
void* pub;
void* sub;

char pub_endpoint[MAX_LEN] = "ipc:///tmp/zeromqlab/serv_pub";
char sub_endpoint[MAX_LEN];

bool has_client;
```

```
int client_id;
int server_pid;

void Init()
{
   context = CreateContext();

   pub = CreateSocket(context, ZMQ_PUB);
   ConnectSocket(pub, pub_endpoint);

   sub = CreateSocket(context, ZMQ_SUB);
   int timewait = 1000;
   zmq_setsockopt(sub, ZMQ_RCVTIMEO, &timewait, sizeof(timewait));
}

void Deinit() {
   DisconnectSocket(pub, pub_endpoint);
   CloseSocket(pub);

   if (has_client) {
      UnbindSocket(sub, sub_endpoint);
   }
   CloseSocket(sub);

   DestroyContext(context);
}

void TermAll() {
   if (has_client) {
      message msg(terminate_cl, -1);
      SendMsg(pub, msg);
   }
}

void sig_handler(int signal) {
   printf("[%d] Terminating server...\n", server_pid);
   TermAll();
   Deinit();
   exit(0);
}

int PingClient(int id) {
   message sendmsg(ping_cl, id);
   SendMsg(pub, sendmsg);
   message msg;
   if (!GetMsg(sub, msg)) {
      return 0;
   }
   if (msg.id == 0 && msg.cmd == ping_cl) {
      return msg.pid;
```

```
    }
    return 0;
}

void CreateClient(int id) {
    if (!has_client) {
        client_id = id;

        int fork_pid = fork();
        if (fork_pid < 0) {
            printf("[%d] Error: Unable to fork a child\n", server_pid);
            exit(-1);
        } else if (fork_pid == 0) {
            ExeclClient(id, 0, pub_endpoint);
        } else {
            SetEndpoint(sub_endpoint, id, cl_parent_pub);
            BindSocket(sub, sub_endpoint);
            sleep(1);
            int pinged_pid = PingClient(id);
            if (!pinged_pid) {
                printf("[%d] Error: client wasn't created\n", server_pid);
                UnbindSocket(sub, sub_endpoint);
            } else {
                printf("[%d] OK: %d\n", server_pid, pinged_pid);
                has_client = true;
            }
        }
    } else {
        if (PingClient(id)) {
            printf("[%d] Error: client already exist\n", server_pid);
        } else {
            message sendmsg(create_cl, id);
            SendMsg(pub, sendmsg);
            sleep(1);
            int pinged_pid = PingClient(id);
            if (!pinged_pid) {
                printf("[%d] Error: client wasn't created\n", server_pid);
            } else {
                printf("[%d] OK: %d\n", server_pid, pinged_pid);
            }
        }
    }
}

void RemoveClient(int id) {
    if (!PingClient(id)) {
        printf("[%d] Error: client not found\n", server_pid);
        return;
    }
```

```
    message sendmsg(remove_cl, id);
    SendMsg(pub, sendmsg);
    if (client_id == id) {
        DisconnectSocket(sub, sub_endpoint);
        has_client = false;
    }

    if (!PingClient(id)) {
        printf("[%d] OK\n", server_pid);
    } else {
        printf("[%d] Error: client wasn't removed\n", server_pid);
    }
}

void SearchPattern(int id) {
    if (!PingClient(id)) {
        printf("[%d] Error: client client not found\n", server_pid);
        return;
    }

    string text;
    string pattern;
    if(cin.peek() == '\n'){
        cin.ignore(1, '\n');
    }
    getline(cin, text);
    if(cin.peek() == '\n'){
        cin.ignore(1, '\n');
    }
    getline(cin, pattern);

    char textc[text.length()+1] = "";
    char patternc[pattern.length()+1] = "";

    strcpy(textc, text.c_str());
    strcpy(patternc, pattern.c_str());

    SendSearchMsg(pub, id, textc, patternc);

    message msg;
    if(!GetMsg(sub, msg)){
        printf("[%d] Error: client haven't responed\n", server_pid);
        return;
    }

    char ans[msg.textsz];

    if(!GetAnsExecMsg(sub, msg.textsz, ans)){
        printf("[%d] Error: client haven't responed\n", server_pid);
        return;
```

```
    }

    if(msg.textsz != 1)
        printf("[%d] OK:%d: %s\n", server_pid, id, ans);
    else
        printf("[%d] OK:%d: -1\n", server_pid, id);
}

void ClearInput()
{
    cin.ignore(64, '\n');
}

int main (int argc, char *argv[])  {
    if (signal(SIGINT, sig_handler) == SIG_ERR) {
        perror("ERROR signal ");
        return -1;
    }

    has_client = false;
    server_pid = getpid();
    printf("[%d] Starting server...\n", server_pid);
    Init();

    string cmd;
    int id;
    while(cin >> cmd >> id) {
        if (id < 1) {
            printf("[%d] Error: Invalid id\n", server_pid);
            continue;
        }
        if (cmd == "create") {
            CreateClient(id);
        }else if (cmd == "remove") {
            RemoveClient(id);
        } else if (cmd == "exec") {
            SearchPattern(id);
        } else if (cmd == "ping") {
            if (PingClient(id) == 0) {
                printf("[%d] Error: client not found\n", server_pid);
            } else {
                printf("[%d] OK: 1\n", server_pid);
            }
        } else  {
            printf("[%d] Error: Invalid command\n", server_pid);
            ClearInput();
        }
    }

    printf("[%d] Shutting down server...\n", server_pid);
```

```
    TermAll();
    Deinit();
    return 0;
}
```

client.cpp

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include "zmq.hpp"

#include "zeromq.hpp"

using namespace std;

void* context;
void* sub;
void* parent_pub;
void* left_pub;
void* right_pub;

char parent_sub_end[MAX_LEN];
char left_sub_end[MAX_LEN];
char right_sub_end[MAX_LEN];

char parent_pub_end[MAX_LEN];
char left_pub_end[MAX_LEN];
char right_pub_end[MAX_LEN];

int parent_id;
int client_pid;
int client_id;

bool has_left;
bool has_right;

void Init() {
    SetEndpoint(left_pub_end, client_id, cl_left_pub);
    SetEndpoint(right_pub_end, client_id, cl_right_pub);
    SetEndpoint(parent_pub_end, client_id, cl_parent_pub);

    context = CreateContext();

    sub = CreateSocket(context, ZMQ_SUB);
    BindSocket(sub, parent_sub_end);

    parent_pub = CreateSocket(context, ZMQ_PUB);
```

```
    ConnectSocket(parent_pub, parent_pub_end);

    right_pub = CreateSocket(context, ZMQ_PUB);
    ConnectSocket(right_pub, right_pub_end);

    left_pub = CreateSocket(context, ZMQ_PUB);
    ConnectSocket(left_pub, left_pub_end);
}

void Deinit() {
    UnbindSocket(sub, parent_sub_end);
    if (has_left) {
        UnbindSocket(sub, left_sub_end);
    }
    if (has_right) {
        UnbindSocket(sub, right_sub_end);
    }
    CloseSocket(sub);

    ConnectSocket(parent_pub, parent_pub_end);
    ConnectSocket(left_pub, left_pub_end);
    ConnectSocket(right_pub, right_pub_end);

    CloseSocket(parent_pub);
    CloseSocket(left_pub);
    CloseSocket(right_pub);

    DestroyContext(context);
}

void sig_handler(int signal) {
    Deinit();
    exit(0);
}

void CreateClient(int id) {
    int fork_pid = fork();
    if (fork_pid < 0) {
        printf("[%d] Error: Unable to fork a child\n", client_pid);
        exit(-1);
    } else if (fork_pid == 0) {
        if (id < client_id) {
            ExeclClient(id, client_id, left_pub_end);
        } else {
            ExeclClient(id, client_id, right_pub_end);
        }
    } else {
        if (id < client_id) {
            SetEndpoint(left_sub_end, id, cl_parent_pub);
            BindSocket(sub, left_sub_end);
```

```
            has_left = true;
        } else {
            SetEndpoint(right_sub_end, id, cl_parent_pub);
            BindSocket(sub, right_sub_end);
            has_right = true;
        }
    }
}

void SearchPattern(message& msg){
    char text[msg.textsz] = "";
    char pattern[msg.textsz] = "";

    GetSearchMsg(sub, msg.textsz, msg.textsz, text, pattern);
    if(msg.id != client_id){
        if (msg.id < client_id) {
            SendSearchMsg(left_pub, msg.id, text, pattern);
        } else {
            SendSearchMsg(right_pub, msg.id, text, pattern);
        }
        return;
    }

    string ans;
    for (int i = 0; i <= msg.textsz - msg.patternsz; i++)
    {
        int j = 0;
        for (j = 0; j < msg.patternsz - 1; j++){
            if (text[i + j] != pattern[j])
                break;
        }

        if (j == msg.patternsz - 1)
            ans += to_string(i) + "; ";
    }

    char ansc[ans.length()+1] = "";
    strcpy(ansc, ans.c_str());
    SendAnsExecMsg(parent_pub, ansc);
}

void RemoveSub(int id) {
    if (id < client_id) {
        has_left = false;
        UnbindSocket(sub, left_sub_end);
    } else {
        has_right = false;
        UnbindSocket(sub, right_sub_end);
    }
}
```

```
int main(int argc, char *argv[])  {
    if (signal(SIGTERM, sig_handler) == SIG_ERR) {
        printf("[%d] ", getpid());
        perror("Error signal ");
        return -1;
    }

    client_pid = getpid();
    client_id = atoi(argv[1]);
    parent_id = atoi(argv[2]);
    strcpy(parent_sub_end, argv[3]);

    Init();

    while(true) {
        message msg;
        GetMsg(sub, msg);
        if (msg.id == 0) {
            if(msg.cmd == exec){
                char ans[msg.textsz];
                GetAnsExecMsg(sub, msg.textsz, ans);
                SendAnsExecMsg(parent_pub, ans);
            }else{
                SendMsg(parent_pub, msg);
            }
        } else if (msg.id != client_id && msg.cmd != create_cl
                && msg.cmd != terminate_cl && msg.cmd != exec) {
            if (has_left && msg.id < client_id) {
                SendMsg(left_pub, msg);
            } else if(has_right && msg.id > client_id) {
                SendMsg(right_pub, msg);
            }
        } else if (msg.cmd == create_cl) {
            if (has_left && msg.id < client_id) {
                SendMsg(left_pub, msg);
            } else if (has_right && msg.id > client_id) {
                SendMsg(right_pub, msg);
            } else {
                CreateClient(msg.id);
            }
        } else if (msg.cmd == remove_cl) {
            if (parent_id != 0) {
                msg.cmd = remove_sub;
                msg.id = parent_id;
                msg.pid = client_id;
                SendMsg(parent_pub, msg);
            }
            msg.cmd = terminate_cl;
            SendMsg(left_pub, msg);
```

```
            SendMsg(right_pub, msg);
            raise(SIGTERM);
        } else if (msg.cmd == terminate_cl) {
            SendMsg(left_pub, msg);
            SendMsg(right_pub, msg);
            raise(SIGTERM);
        } else if (msg.cmd == ping_cl) {
            msg.id = 0;
            msg.pid = client_pid;
            SendMsg(parent_pub, msg);
        } else if (msg.cmd == remove_sub) {
            RemoveSub(msg.pid);
        } else if(msg.cmd == exec){
            SearchPattern(msg);
        }
    }

    Deinit();
    return 0;
}
```

zeromq.hpp

```
#pragma once

#include <iostream>
#include <string>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include "zmq.hpp"

#define MAX_LEN 64

enum command{
    create_cl,
    remove_cl,
    terminate_cl,
    exec,
    ping_cl,
    remove_sub
};

struct message
{
    command cmd;
    int id;
    int pid;
    int textsz;
    int patternsz;
```

```
    message(command cmd_, int id_){
        cmd = cmd_;
        id = id_;
    }
    message(){
        id = -1;
    }
};

enum endpoint{
    cl_left_pub,
    cl_right_pub,
    cl_parent_pub
};

void SendMsg(void* socket, message& msg);
bool GetMsg(void* socket, message& msg);
void SetEndpoint(char* endp, int id, endpoint type);
void ExeclClient(int id, int parent_id, char* sub_endpoint);

void SendSearchMsg(void* socket, int id, char* text, char* pattern);
bool GetSearchMsg(void* socket, int textsz, int patternsz, char* text, char* pattern);
void SendAnsExecMsg(void* socket, char* text);
bool GetAnsExecMsg(void* socket, int textsz, char* text);

void* CreateContext();
void DisconnectSocket(void* socket, char* endpoint);
void ConnectSocket(void* socket, char* endpoint);
void BindSocket(void* socket, char* endpoint);
void UnbindSocket(void* socket, char* endpoint);
void* CreateSocket(void* context, int type);
void CloseSocket(void* socket);
void DestroyContext(void* context);
```

zeromq.cpp

```
#include "zeromq.hpp"

using namespace std;

void SendMsg(void* socket, message& msg) {
    if (!zmq_send(socket, &msg, sizeof(message), 0)) {
        printf("[%d] ", getpid());
        perror("Error: SendMsg \n");
        exit(-1);
    }
}

bool GetMsg(void* socket, message& msg) {
    if (zmq_recv(socket, &msg, sizeof(message), 0) == -1) {
```

```
        return false;
    }
    return true;
}

void SetEndpoint(char* endp, int id, endpoint type) {
    string tmp;
    if (type == cl_left_pub) {
        tmp = "ipc:///tmp/zeromqlab/left_pub";
    } else if (type == cl_right_pub) {
        tmp = "ipc:///tmp/zeromqlab/right_pub";
    } else if (type == cl_parent_pub) {
        tmp = "ipc:///tmp/zeromqlab/par_pub";
    }
    strcpy(endp, (tmp+to_string(id)).c_str());
}

void ExeclClient(int id, int parent_id, char* sub_endpoint){
    char client[MAX_LEN] = "./client";
    execl(client, client, to_string(id).c_str(), to_string(parent_id).c_str(), sub_endpoint, NULL);
}

void SendSearchMsg(void* socket, int id, char* text, char* pattern){
    message msg(exec, id);
    msg.textsz = strlen(text)+1;
    msg.patternsz = strlen(pattern)+1;

    if (!zmq_send(socket, &msg, sizeof(message), ZMQ_SNDMORE)) {
        printf("[%d] ", getpid());
        perror("Error: SendSearchMsg \n");
        exit(-1);
    }

    if(!zmq_send(socket, text, msg.textsz * sizeof(char), ZMQ_SNDMORE)){
        printf("[%d] ", getpid());
        perror("Error: SendSearchMsg \n");
        exit(-1);
    }

    if(!zmq_send(socket, pattern, msg.patternsz * sizeof(char), 0)){
        printf("[%d] ", getpid());
        perror("Error: SendSearchMsg \n");
        exit(-1);
    }

}

bool GetSearchMsg(void* socket, int textsz, int patternsz, char* textstr, char* patternstr){
    if(zmq_recv(socket, textstr, textsz * sizeof(char), 0) == -1){
        return false;
```

```
    }
    if(zmq_recv(socket, patternstr, patternsz * sizeof(char), 0) == -1){
        return false;
    }

    return true;
}

void SendAnsExecMsg(void* socket, char* text){
    message msg(exec, 0);
    msg.textsz = strlen(text)+1;
    if (!zmq_send(socket, &msg, sizeof(message), ZMQ_SNDMORE)) {
        printf("[%d] ", getpid());
        perror("Error: SendAnsMsg \n");
        exit(-1);
    }
    if (!zmq_send(socket, text, msg.textsz, 0)) {
        printf("[%d] ", getpid());
        perror("Error: SendAnsMsg \n");
        exit(-1);
    }
}

bool GetAnsExecMsg(void* socket, int textsz, char* text){
    if(zmq_recv(socket, text, textsz * sizeof(char), 0) == -1){
        return false;
    }
    return true;
}

void* CreateContext() {
    void* context = zmq_ctx_new();
    if (context == NULL) {
        printf("[%d] ", getpid());
        perror("Error: CreateContext ");
        exit(-1);
    }
    return context;
}

void DisconnectSocket(void* socket, char* endpoint) {
    if (zmq_disconnect(socket, endpoint) != 0) {
        printf("[%d] ", getpid());
        perror("Error: DisconnectSocket ");
        exit(-1);
    }
}

void ConnectSocket(void* socket, char* endpoint) {
    if (zmq_connect(socket, endpoint) != 0) {
```

```
        printf("[%d] ", getpid());
        perror("Error: ConnectSocket ");
        exit(-1);
    }
}

void BindSocket(void* socket, char* endpoint) {
    if (zmq_bind(socket, endpoint) != 0) {
        printf("[%d] ", getpid());
        perror("Error: BindSocket ");
        exit(-1);
    }
}

void UnbindSocket(void* socket, char* endpoint) {
    if (zmq_unbind(socket, endpoint) != 0) {
        printf("[%d] ", getpid());
        perror("Error: UnbindSocket ");
        exit(-1);
    }
}

void* CreateSocket(void* context, int type) {
    void* socket = zmq_socket(context, type);
    if (socket == NULL) {
        printf("[%d] ", getpid());
        perror("Error: CreateSocket ");
        exit(-1);
    }
    if (type == ZMQ_SUB) {
        zmq_setsockopt(socket, ZMQ_SUBSCRIBE, 0, 0);
    }
    return socket;
}

void CloseSocket(void* socket) {
    if (zmq_close(socket) != 0) {
        printf("[%d] ", getpid());
        perror("Error: CloseSocket ");
        exit(-1);
    }
}

void DestroyContext(void* context) {
    if (zmq_ctx_destroy(context) != 0) {
        printf("[%d] ", getpid());
        perror("Error: DestroyContext ");
        exit(-1);
    }
}
```

Пример работы программы.

```
pablo$ make
g++ -c -o src/zeromq.o src/zeromq.cpp -lzmq
g++ -o server src/server.cpp src/zeromq.o -lzmq
g++ -o client src/client.cpp src/zeromq.o -lzmq
pablo$ cat tests/test1
create 10
ping 10
exec 10
helllo
l
create 5
exec 5
world
a
r
p
pablo$ ./server < tests/test1
[371] Starting server...
[371] OK: 374
[371] OK: 1
[371] OK:10: 2; 3; 4;
[371] OK: 377
[371] OK:5: -1
[371] OK
pablo$ ./server
[380] Starting server...
create 1
[380] OK: 383
create 10
[380] OK: 386
create 7
[380] OK: 389
ping 10
[380] OK: 1
remove 10
[380] OK
ping 10
[380] Error: client not found
ping 7
[380] Error: client not found
create 10
[380] OK: 392
exec 10
abababababa
aba
[380] OK:10: 0; 2; 4; 6;
^C[380] Terminating server...
```

Выводы.

Выполнив данную лабораторную работу, я научился использовать библиотеку ZeroMQ. Благодаря ей можно просто и быстро обмениваться сообщениями между разными процессами, так как она позволяет отправлять и принимать сообщения асинхронно – написание этого заняло бы у меня много времени. Я решил использовать паттерн publish-subscribe, так как он позволял проще отправлять своим подпроцессам сообщения и не делать ответ обязательным. Также я решил использовать протокол ipc, а не tcp, так как мне было проще создавать отдельные файлы, нежели каждый раз проверять свободность портов. И при создании ребенка я приостанавливал работу сервера перед проверкой, создался ли узел, так как он иногда не успевал подключаться к сокетам.