Московский авиационный институт
(Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Операционные системы»

Лабораторная работа № 6-8

Студент: Марочкин И.А.
Группа: М8О-206Б-19
Преподаватель: Соколов А.А.
Дата: 24.04.2021
Оценка:

Москва, 2020

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.

Вариант:

Топология 2: Все вычислительные узлы находятся в дереве общего вида. Есть только 1 управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

Набор команд 2(локальный целочисленный словарь): Формат сохранения значения: exec id name value.

Команда проверки 2: Формат команды: ping id.

Общие сведения о программе

Для работы с очередями используется ZMQ, программа собирается при помощи Makefile. Управляющий узел – server, вычислительные узлы – client. В программе используются следующие системные вызовы:

zmq_msg_init_size - выделяет и инициализирует память для сообщения указанного размера. Если сообщение короткое, память выделяется на стеке, в ином случае - в куче.

zmq_msg_data - возвращает указатель на то, что лежит в сообщении.

zmq_msg_send - ставит сообщение в очередь на отправку.

zmq_msg_close - удаляет сообщение и ранее выделенную память.

zmq_setsockopt - устанавливает параметр на сокет.
zmq_connect - подключается к сокету для публикации сообщений.
zmq_ctx_new - создает новый контекст.
zmq_socket - создает новый сокет.
zmq_bind - подключается к сокету для получения сообщений.

Листинг программы

commands.hpp

```
#pragma once

#include <algorithm>
#include <unistd.h>
#include <string>
#include <vector>
#include <cstdio>
#include <cstdlib>
#include <utility>
#include <memory.h>
#include <pthread.h>
#include <zmq.h>

#define MAJOR_SOCKET_RCVTIMEO 5 * 1000
#define MINOR_SOCKET_RCVTIMEO 5 * 1000

struct ClientInfo {
    int64_t id;
    void *majorSocket;
    void *minorSocket;

    ClientInfo(int64_t id, void *majorSocket, void *minorSocket) :
                id(id), majorSocket(majorSocket), minorSocket(minorSocket){}
};

enum Command {
    CreateCmd,
    RemoveCmd,
    ExecCmd,
    PingCmd,
    HeartbeatCmd,
};

enum Status {
    OkStatus,
    DoneStatus,
    ErrorStatus
};

enum ExecStatus {
    GetExec,
    SetExec,
    NotFoundExec
};

struct Message {
    int64_t clientId;
    int64_t parentId;
    std::string key;
    int64_t value;
    Command command;
    Status status;
```

```
    uint64_t taskId;
    time_t time;
    ExecStatus execStatus;

    Message() = default;

    Message(int64_t clientId, Command command, Status status) :
            clientId(clientId), command(command), status(status){}

    Message(int64_t clientId, int64_t parentId, Command command) :
            clientId(clientId), parentId(parentId), command(command){}

    Message(int64_t clientId, Command command, uint64_t taskId) :
            clientId(clientId), command(command), taskId(taskId){}

    Message(int64_t clientId, Command command, uint64_t taskId, time_t time) :
            clientId(clientId), command(command), taskId(taskId), time(time){}
};

int sendMessage(void *socket, Message message){
    zmq_msg_t sendingMessage;
    zmq_msg_init_size(&sendingMessage, sizeof(message));
    memcpy(zmq_msg_data(&sendingMessage), &message, sizeof(message));

    int status = zmq_msg_send(&sendingMessage, socket, ZMQ_NOBLOCK);
    zmq_msg_close(&sendingMessage);
    return status;
}

int receiveMessage(void *socket, Message *message){
    zmq_msg_t receivingMessage;
    zmq_msg_init(&receivingMessage);
    if (zmq_msg_recv(&receivingMessage, socket, 0) == -1) {
        if (errno == EAGAIN) {
            fprintf(stderr, "timeout error\n");
            return 0;
        }
        return -1;
    }

    auto *masterMessage = (Message *)zmq_msg_data(&receivingMessage);
    *message = *masterMessage;
    zmq_msg_close(&receivingMessage);

    return sizeof(*message);
}

void *clientMonitor(void *params);

extern int64_t id;
extern void *context;
extern std::vector<ClientInfo> clients;
extern std::vector<pthread_t *> threads;

int create(int64_t clientId, int64_t parentId){
    if (parentId == id) {
        pid_t pid = fork();
        if (pid == -1) {
            fprintf(stderr, "fork failed\n");
            return -1;
        } else if (pid == 0) {
            if (id == -1) {
                char arg1[20];
                snprintf(arg1, 20, "%lld", clientId);

                execl("client", "client", arg1, (char*)(NULL));
            } else {
```

```
                char arg1[20];
                char arg2[20];
                sprintf(arg1, "%lld", clientId);
                sprintf(arg2, "%lld", parentId);

                execl("client", "client", arg1, arg2, (char*)(NULL));
            }

            fprintf(stderr, "execl error\n");
            exit(1);
        }

        sleep(1);

        char buff[41];
        char buff2[42];

        if (id == -1) {
            sprintf(buff, "ipc://_%lld.ipc", clientId);
            sprintf(buff2, "ipc://_%lld_.ipc", clientId);
        } else {
            sprintf(buff, "ipc://%llu_%llu.ipc", parentId, clientId);
            sprintf(buff2, "ipc://%llu_%llu_.ipc", parentId, clientId);
        }

        void *majorSock = zmq_socket(context, ZMQ_REQ);
        void *minorSock = zmq_socket(context, ZMQ_REP);

        int time = MAJOR_SOCKET_RCVTIMEO;
        int time2 = MINOR_SOCKET_RCVTIMEO;

        zmq_setsockopt(majorSock, ZMQ_RCVTIMEO, &time, sizeof(time));
        zmq_setsockopt(minorSock, ZMQ_RCVTIMEO, &time2, sizeof(time2));


        for (int i = 0; i < 4; i++) {
            if (i == 3) {
                fprintf(stderr, "zmq_connect error\n");
                return -1;
            }

            if ((zmq_connect(majorSock, buff) == -1) || (zmq_connect(minorSock, buff2) ==
-1)) {
                sleep(2);
            }
            else {
                break;
            }
        }

        ClientInfo info(clientId, majorSock, minorSock);
        clients.push_back(info);

        auto *thread = new pthread_t;
        threads.push_back(thread);
        auto *newInfo = (ClientInfo *) malloc(sizeof(info));
        *newInfo = info;

        pthread_create(threads[threads.size() - 1], nullptr, clientMonitor, (void *)
newInfo);
        return pid;
    }
    else {
        for (const auto &client: clients) {
            Message message(clientId, parentId, CreateCmd);
            sendMessage(client.majorSocket, message);
```

```
            Message received_message;
            int status = receiveMessage(client.majorSocket, &received_message);
            if (status == -1) {
                fprintf(stderr, "receive message error\n");
            }
        }

        return 0;
    }
}

int remove(int64_t clientId, uint64_t taskId){
    if (clientId == id) {
        return clientId;
    }

    for (const auto &client: clients) {
        Message msg(clientId, RemoveCmd, taskId);
        sendMessage(client.majorSocket, msg);
        receiveMessage(client.majorSocket, &msg);
    }
    return 0;
}

int ping(int64_t clientId, uint64_t taskId){
    if (clientId == id) {
        return clientId;
    }

    Message msg(clientId, PingCmd, taskId);
    for (const auto &client: clients) {
        sendMessage(client.majorSocket, msg);
        receiveMessage(client.majorSocket, &msg);
    }

    return 0;
}

int exec(int64_t clientId, ExecStatus execStatus, std::string key, int64_t value, uint64_t
taskId){
    if (clientId == id) {
        return clientId;
    }

    Message msg;
    msg.command = ExecCmd;
    msg.clientId = clientId;
    msg.execStatus = execStatus;
    msg.key = std::move(key);
    msg.value = value;
    msg.taskId = taskId;

    for (const auto &client: clients) {
        sendMessage(client.majorSocket, msg);
        receiveMessage(client.majorSocket, &msg);
    }

    return 0;
}
```

client.cpp

```
#include <csignal>
#include <deque>
#include <ctime>
#include <map>

#include "commands.hpp"
```

```
#define MINOR_SOCKET_MONITOR_SLEEP_TIME 1

std::map<std::string, int64_t> dictionary;
int64_t id;

void *context;
void *majorSocket;
void *minorSocket;

pthread_t majorThread;
pthread_t minorThread;

std::vector<pthread_t *> threads;
std::vector<ClientInfo> clients;


bool flag = true;
Message messageToMaster;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void setMessage(Message message) {
    pthread_mutex_lock(&mutex);

    messageToMaster = std::move(message);
    flag = true;

    pthread_mutex_unlock(&mutex);
}

bool get_message(Message *message) {
    bool result;
    pthread_mutex_lock(&mutex);

    result = flag;

    if (flag) {
        *message = messageToMaster;
        flag = false;
    }

    pthread_mutex_unlock(&mutex);

    return result;
}


void *clientMonitor(void *params) {
    auto *client = (ClientInfo *)params;

    while (true) {
        Message msg;
        if (receiveMessage(client->minorSocket, &msg) == -1) {
            break;
        }

        if (sendMessage(client->minorSocket, msg) == -1) {
            break;
        }
        if (msg.command == HeartbeatCmd) {
            continue;
        }

        switch (msg.status) {
            case DoneStatus:
            case ErrorStatus: {
                setMessage(msg);
```

```
                break;
            }
            default:
                break;
        }

        if (msg.clientId == client->id && msg.command == RemoveCmd && msg.status ==
DoneStatus) {
            break;
        }
    }

    zmq_close(client->majorSocket);
    zmq_close(client->minorSocket);

    auto end = std::remove_if(clients.begin(), clients.end(), [&](auto clt){
        return clt.id == client->id;
    });
    clients.erase(end, clients.end());

    free(params);
    return nullptr;
}


void *major_socket_monitor(void *) {
    while (true) {
        Message message;
        receiveMessage(majorSocket, &message);

        message.status = OkStatus;
        switch (message.command) {
            case CreateCmd: {
                sendMessage(majorSocket, message);

                create(message.clientId, message.parentId);
                break;
            }
            case RemoveCmd:
                sendMessage(majorSocket, message);
                if (remove(message.clientId, message.taskId) == id) {
                    message.status = DoneStatus;
                    setMessage(message);
                    return nullptr;
                }
                break;
            case ExecCmd: {
                sendMessage(majorSocket, message);
                if (exec(message.clientId, message.execStatus, message.key, message.value,
message.taskId) == id) {
                    if (message.execStatus == GetExec) {
                        if (dictionary.find(message.key) == dictionary.end()) {
                            message.execStatus = NotFoundExec;
                        } else {
                            message.value = dictionary.at(message.key);
                        }
                    } else {
                        dictionary[message.key] = message.value;
                    }

                    message.status = DoneStatus;
                    setMessage(message);
                }
                break;
            }
            case PingCmd: {
                sendMessage(majorSocket, message);
```

```
                if (ping(message.clientId, message.taskId) == id) {
                    message.status = DoneStatus;
                    setMessage(message);
                }
                break;
            }
            default: {
                message.status = ErrorStatus;
                sendMessage(majorSocket, message);
                break;
            }
        }
    }
}


void *minor_socket_monitor(void *) {
    sleep(1);
    flag = true;
    messageToMaster.clientId = id;
    messageToMaster.value = getpid();
    messageToMaster.status = DoneStatus;

    while (true) {
        Message msg(id, HeartbeatCmd, OkStatus);

        get_message(&msg);
        if (sendMessage(minorSocket, msg) == -1) {
            break;
        }

        if (receiveMessage(minorSocket, &msg) == -1) {
            break;
        }

        if (msg.command == RemoveCmd && msg.status == DoneStatus && msg.clientId == id) {
            break;
        }

        sleep(MINOR_SOCKET_MONITOR_SLEEP_TIME);
    }

    pthread_kill(majorThread, SIGKILL);
    return nullptr;
}


int main(int argc, char *argv[]) {
    id = std::stoll(argv[1]);

    context = zmq_ctx_new();
    majorSocket = zmq_socket(context, ZMQ_REP);
    minorSocket = zmq_socket(context, ZMQ_REQ);

    if (argc == 2) {
        char buff[31];
        char buff2[32];

        sprintf(buff, "ipc://_%lld.ipc", id);
        sprintf(buff2, "ipc://_%lld_.ipc", id);

        zmq_bind(majorSocket, buff);

        int time = MINOR_SOCKET_RCVTIMEO;
        zmq_setsockopt(minorSocket, ZMQ_RCVTIMEO, &time, sizeof time);
        zmq_bind(minorSocket, buff2);
    } else if (argc == 3) {
```

```
        char buff[41];
        char buff2[42];

        sprintf(buff, "ipc://%llu_%llu.ipc", std::stoll(argv[2]), id);
        sprintf(buff2, "ipc://%llu_%llu_.ipc", std::stoll(argv[2]), id);

        zmq_bind(majorSocket, buff);

        int time = MINOR_SOCKET_RCVTIMEO;
        zmq_setsockopt(minorSocket, ZMQ_RCVTIMEO, &time, sizeof time);
        zmq_bind(minorSocket, buff2);
    } else {
        fprintf(stderr, "invalid arguments\n");
        exit(1);
    }

    pthread_create(&majorThread, nullptr, major_socket_monitor, nullptr);
    pthread_create(&minorThread, nullptr, minor_socket_monitor, nullptr);

    pthread_join(majorThread, nullptr);
    pthread_join(minorThread, nullptr);

    zmq_close(majorSocket);
    zmq_close(minorSocket);

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

server.cpp

```
#include <iostream>
#include <sstream>
#include <csignal>
#include <deque>
#include <ctime>
#include <set>

#include "commands.hpp"

#define MAXIMUM_COMMAND_EXECUTION_TIME 10

std::vector<pthread_t *> threads;
std::vector<ClientInfo> clients;
std::set<int64_t> ids;
std::deque<Message> tasks;

int64_t id{-1};
int64_t task_id_{0};

pthread_t takeUnfinishedTaskThread;
pthread_mutex_t taskIdMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t tasksMutex = PTHREAD_MUTEX_INITIALIZER;

void *context;

int64_t taskId() {
    pthread_mutex_lock(&taskIdMutex);

    auto result = task_id_++;

    pthread_mutex_unlock(&taskIdMutex);
    return result;
}

void *takeUnfinishedTask(void *) {
    while (true) {
        sleep(MAXIMUM_COMMAND_EXECUTION_TIME);
```

```
        if (pthread_mutex_lock(&tasksMutex)) {
            printf("lock error\n");
            break;
        }

        if (!tasks.empty() && (time(nullptr) - tasks.front().time) >=
MAXIMUM_COMMAND_EXECUTION_TIME) {
            auto task = tasks.front();
            switch (task.command) {
                case CreateCmd: {
                    printf("Error: Parent is unavailable\n");
                    break;
                }
                case RemoveCmd:
                case ExecCmd: {
                    printf("Error: Node is unavailable\n");
                    break;
                }
                case PingCmd: {
                    printf("Ok: 0\n");
                    break;
                }
                default: {
                    break;
                }
            }
            tasks.pop_front();
        }
        if (pthread_mutex_unlock(&tasksMutex)) {
            printf("unlock error\n");
            break;
        }
    }
    return nullptr;
}

void *clientMonitor(void *params) {
    auto *client = (ClientInfo *) params;

    while (true) {
        Message msg;
        if (receiveMessage(client->minorSocket, &msg) == -1) {
            break;
        }

        if (sendMessage(client->minorSocket, msg) == -1) {
            break;
        }

        pthread_mutex_lock(&tasksMutex);
        switch (msg.command) {
            case CreateCmd: {
                switch (msg.status) {
                    case DoneStatus: {
                        for (const auto& task: tasks) {
                            if (task.clientId == msg.clientId) {
                                printf("Ok: %lld\n", msg.value);
                                ids.insert(task.clientId);
                                break;
                            }
                        }
                        auto end = std::remove_if(tasks.begin(), tasks.end(), [&](auto
task) {
                            return task.clientId == msg.clientId;
                        });
                        tasks.erase(end, tasks.end());
                        break;
```

```
                    }
                    default:
                        break;
                }
                break;
            }
            case RemoveCmd:
            case ExecCmd:
            case PingCmd: {
                switch (msg.status) {
                    case DoneStatus: {
                        for (const auto& task: tasks) {
                            if (task.taskId == msg.taskId) {
                                if (msg.command == RemoveCmd) {
                                    printf("Ok\n");
                                } else if (msg.command == ExecCmd) {
                                    if (msg.execStatus == GetExec) {
                                        printf("Ok:%lld: %lld\n", msg.clientId, msg.value);
                                    } else if (msg.execStatus == SetExec) {
                                        printf("Ok:%lld\n", msg.clientId);
                                    } else {
                                        printf("Ok:%lld:%s not found\n", msg.clientId,
msg.key.c_str());
                                    }
                                } else {
                                    printf("Ok: 1\n");
                                }

                                if (msg.command == RemoveCmd) {
                                    ids.erase(task.clientId);
                                }
                                break;
                            }
                        }
                        auto end = std::remove_if(tasks.begin(), tasks.end(), [&](auto
task) {
                            return task.taskId == msg.taskId;
                        });
                        tasks.erase(end, tasks.end());

                        break;
                    }
                    default:
                        break;
                }
                break;
            }
            default:
                break;
        }
        pthread_mutex_unlock(&tasksMutex);

        if (msg.clientId == client->id && msg.command == RemoveCmd && msg.status ==
DoneStatus) {
            break;
        }
    }

    zmq_close(client->majorSocket);
    zmq_close(client->minorSocket);

    auto end = std::remove_if(clients.begin(), clients.end(), [&](auto clt) {
        return clt.id == client->id;
    });
    clients.erase(end, clients.end());

    free(params);
```

```
    return nullptr;
}


int main() {
    pthread_create(&takeUnfinishedTaskThread, nullptr, takeUnfinishedTask, nullptr);

    context = zmq_ctx_new();

    for (std::string line; std::getline(std::cin, line);) {
        std::stringstream ss(line);
        std::string command;
        if (!(ss >> command)) {
            std::cout << "Error: Invalid input" << std::endl;
            continue;
        }

        if (command == "create") {
            int64_t clientId;
            int64_t parentId;

            if (!(ss >> clientId) || !(ss >> parentId)) {
                std::cout << "Error: Invalid input" << std::endl;
                continue;
            }

            if (clientId < 0) {
                printf("Error: Invalid client id\n");
                continue;
            } else if (parentId < -1) {
                printf("Error: Invalid parent id\n");
                continue;
            }

            if (ids.find(clientId) != ids.end()) {
                printf("Error: Already exists\n");
                continue;
            }

            if (parentId != -1 && ids.find(parentId) == ids.end()) {
                printf("Error: Parent not found\n");
                continue;
            }

            Message msg;
            msg.clientId = clientId;
            msg.command = CreateCmd;
            msg.time = time(nullptr);

            pthread_mutex_lock(&tasksMutex);
            tasks.push_back(msg);
            pthread_mutex_unlock(&tasksMutex);

            int status = create(clientId, parentId);
            if (status == -1) {
                printf("Error: Can't create child client\n");
                tasks.pop_back();
            }
        }
        else if (command == "remove") {
            int64_t clientId;

            if (!(ss >> clientId)) {
                std::cout << "Error: Invalid input" << std::endl;
                continue;
            }
```

```
    if (clientId < 0) {
        printf("Error: Invalid client id\n");
        continue;
    }

    if (ids.find(clientId) == ids.end()) {
        printf("Error: Not found\n");
        continue;
    }

    auto tskId = taskId();
    Message msg;
    msg.clientId = clientId;
    msg.command = RemoveCmd;
    msg.taskId = tskId;
    msg.time = time(nullptr);

    pthread_mutex_lock(&tasksMutex);
    tasks.push_back(msg);
    pthread_mutex_unlock(&tasksMutex);

    remove(clientId, tskId);
}
else if (command == "exec") {
    int64_t clientId;
    std::string key;
    int64_t value;
    bool flag = true;

    if (!(ss >> clientId)) {
        std::cout << "Error: Invalid input" << std::endl;
        continue;
    }

    if (!(ss >> key)) {
        std::cout << "Error: Invalid key" << std::endl;
        continue;
    }

    if (!(ss >> value)) {
        flag = false;
    }

    if (clientId < 0) {
        printf("Error: Invalid client id\n");
        continue;
    }

    if (ids.find(clientId) == ids.end()) {
        printf("Error: Not found\n");
        continue;
    }

    Message msg(clientId, ExecCmd, taskId(), time(nullptr));
    if (flag) {
        msg.execStatus = SetExec;
    } else {
        msg.execStatus = GetExec;
    }

    pthread_mutex_lock(&tasksMutex);
    tasks.push_back(msg);
    pthread_mutex_unlock(&tasksMutex);

    exec(clientId, msg.execStatus, key, value, msg.taskId);
}
else if (command == "ping") {
```

```
            int64_t clientId;

            if (!(ss >> clientId)) {
                std::cout << "Error: Invalid input" << std::endl;
                continue;
            }

            if (clientId < 0) {
                printf("Error: Invalid client id\n");
                continue;
            }

            if (ids.find(clientId) == ids.end()) {
                printf("Error: Not found\n");
                continue;
            }

            Message msg(clientId, PingCmd, taskId(), time(nullptr));

            pthread_mutex_lock(&tasksMutex);
            tasks.push_back(msg);
            pthread_mutex_unlock(&tasksMutex);

            ping(clientId, msg.taskId);
        }
        else if (command == "exit") {
            break;
        }
        else {
            printf("Error: Invalid command\n");
        }
    }

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

Пример работы

```
create 8 -1
Ok: 2878
exec 8 one 1
Ok:8
exec one
Error: Invalid input
exec 8 one
Ok:8: 1
ping 8
Ok: 1
exec 8 two 2
Ok:8
exec 8 two
Ok:8: 2
remove 8
Ok
exit
Vanya:gwfdgdfg ivan$
```

Вывод

В C, как и большинстве ЯП есть такая структура, как сокеты, которая позволяет удобно организовывать построение и использование архитектуры клиент-сервер. Для общения в архитектуре клиент-сервер существуют очереди сообщений, при помощи них можно достаточно просто организовать обмен информацией, однако ZMQ имеет не самую лучшую документацию и в связке с fork может вызывать некоторые трудности. Такие структуры, как деревья хорошо подходят для хранения информации о клиентах и сервере.