Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторные работы 5-7
по курсу
«Операционные системы»
III Семестр

Студент: Шляхтуров А.В
Группа: М8О-201Б-22
Преподаватель: Миронов Е. С.
Вариант 4
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

Цель работы.

Целью является приобретение практических навыков в:

Управлении серверами сообщений

Применение отложенных вычислений

Интеграция программных систем друг с другом

Задание

Реализовать распределенную систему по асинхронной обработке запросов на языке C++ без использования библиотек STL. 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

Формат команды: create id [parent] id – целочисленный идентификатор нового вычислительного узла parent – целочисленный идентификатор родительского узла. Если топологией не предусмотрено введение данного параметра, то его необходимо игнорировать (если его ввели)

Формат вывода: «Ok: pid», где pid – идентификатор процесса для созданного вычислительного узла «Error: Already exists» - вычислительный узел с таким идентификатором уже существует «Error: [Custom error]» - любая другая обрабатываемая ошибка

• Исполнение команды на вычислительном узле

Формат команды: exec id [params] id – целочисленный идентификатор вычислительного узла, на который отправляется команда Формат вывода: «Ok:id: [result]», где result – результат выполненной команды «Error:id: Not found» - вычислительный узел с таким идентификатором не найден «Error:id: Node is unavailable» - по каким-то причинам не удается связаться с вычислительным узлом «Error:id: [Custom error]» - любая другая обрабатываемая ошибка

• Тип проверки доступности узлов

4 Вариант)

• Топология – первая. Все вычислительные узлы находятся в списке. Есть только один управляющий узел. Чтобы добавить новый вычислительный узел к управляющему, то необходимо выполнить команду: create id -1.

• Тип команд для вычислительных узлов – третий. Локальный таймер. o Формат команды сохранения значения: exec id subcommand o subcommand – одна из трех команд: start, stop, time.

start – запустить таймер

stop – остановить таймер

time – показать время локального таймера в миллисекундах

•Команда проверки – второй

Формат команды: ping id Команда проверяет доступность конкретного узла. Если узла нет, то необходимо выводить ошибку: «Error: Not found» Пример: > ping 10 Ok: 1 // узел 10 доступен > ping 17 Ok: 0 // узел 17 недоступен

Решение

Управляющий узел будет создавать worker’ов с помощью системного вызова fork. Далее, при поступлении определенной команды эта команда будет передаваться на вычислительные узлы посредством очередей сообщений.

В качестве очередей сообщений мной была выбрана ZeroMQ. Каждый worker содержал 3 потока: первый, что принимает сообщение, которое необходимо обработать; второй, который выполняет задание; третий – получает результат вычисления от нижестоящих worker’ов и отправляет его вышестоящим. Управляющий узел состоит из двух потоков: первый считывает команды из консоли и отправляет рабочим, второй получает ответ от рабочих я выводит его на экран.

Код

Definitions.hpp

```cpp
#pragma once

#define MIN_DYNAMIC_PORT 49152
#define MAX_PORT_LENGTH 5
#define MAX_NODE_ID_LENGTH 5
#define PID_FOR_ALREADY_EXIST_NODE -1
#define ALREADY_REPLACED 0

enum class OperationType {
    CREATE,
    EXEC,
    PING,
    QUIT,
    NOTHING,
};

enum class TimerSubrequest {
    START,
    STOP,
    TIME,
    NOTHING,
};
```

```
enum class ErrorTypes {
    STOP_BEFORE_START,
    NO_ERRORS,
};

/*
Структура комманды
Сначала хранится тип команды
Далее, id, в какому узлу надо применить эту комманду
И, только для комманды exec будет хранится тип подкомманды, в остальных NOTHING
Для `create id -1` -1 хранить не будем
*/
struct Request {
    OperationType operationType;
    ssize_t id;
    TimerSubrequest subrequest;
};

struct Reply {
    OperationType operationType;
    ssize_t result;
    ErrorTypes error;
    ssize_t id;
    TimerSubrequest subrequest;
};
```

Functions.hpp

```
#pragma once

#include <map>
#include <string>
#include <zmq_addon.hpp>
#include <zmq.hpp>
#include <cerrno>
#include <sys/wait.h>

#include "definitions.hpp"

std::string getAddres(size_t);
void validateOperationType(Request&, std::string&);
void validateId(Request&, ssize_t);
void validateSubrequest(Request&, std::string&);
Request readRequest();

void pushRequest(zmq::socket_t& socket, Request& request);
void pushReply(zmq::socket_t& socket, Reply& request);
Reply pullReply(zmq::socket_t& socket);
Request pullRequest(zmq::socket_t& socket);
```

```
std::pair<pid_t, std::pair<bool, size_t>> createNewNode(std::unordered_map<ssize_t,
std::pair<pid_t, size_t>>&, ssize_t, size_t&);
void updateNodeMap(std::unordered_map<ssize_t, std::pair<pid_t, size_t>>&,
std::map<size_t, std::vector<pid_t>>&, size_t&, Request&);

void killWorkers(std::unordered_map<ssize_t, std::pair<pid_t, size_t>>&);

bool isProcessExists(pid_t);
bool isNodeAvaliable(std::map<size_t, std::vector<pid_t>>&, pid_t);
```

functions.cpp

```
#include "functions.hpp"
#include <stdlib.h>
#include <iostream>
#include <sstream>
#include <unistd.h>

std::string getAddres(size_t port) {
    return std::string("tcp://127.0.0.1:") + std::to_string(port);
}

// MESSAGE VALIDATOR ------------------------------------------------------------
void validateOperationType(Request& request, std::string& operationType) {
    if (operationType == "quit") {
        request.operationType = OperationType::QUIT;
    } else if (operationType == "create") {
        request.operationType = OperationType::CREATE;
    } else if (operationType == "exec") {
        request.operationType = OperationType::EXEC;
    } else if (operationType == "ping") {
        request.operationType = OperationType::PING;
    } else {
        // throw std::invalid_argument("Invalid operation");
        std::cout << "Error: Invalid operation" << std::endl;
    }
}
void validateId(Request& request, ssize_t id) {
    if (id < 0) {
        // throw std::invalid_argument("Invalid worker node id");
        std::cout << "Error: " << id << " : Invalid worker node id" << std::endl;
    }
    request.id = id;
}
void validateSubrequest(Request& request, std::string& timerSubrequest) {
    using std::invalid_argument;
    if (request.operationType == OperationType::PING && !timerSubrequest.empty()) {
        // throw invalid_argument("Ping can't accept third argument");
        std::cout << "Error: Ping can't accept third argument" << std::endl;
    } else if (request.operationType == OperationType::CREATE && timerSubrequest !=
"-1") {
```

```
        // throw invalid_argument("Create can push only in end");
        std::cout << "Error: Create can push only in end" << std::endl;
    } else if (request.operationType == OperationType::PING || request.operation-
Type == OperationType::CREATE) {
        request.subrequest = TimerSubrequest::NOTHING;
    }

    if (request.operationType == OperationType::EXEC && timerSubrequest == "start")
{
        request.subrequest = TimerSubrequest::START;
    } else if (request.operationType == OperationType::EXEC && timerSubrequest ==
"stop") {
        request.subrequest = TimerSubrequest::STOP;
    } else if (request.operationType == OperationType::EXEC && timerSubrequest ==
"time") {
        request.subrequest = TimerSubrequest::TIME;
    } else if (request.operationType == OperationType::EXEC) {
        // throw invalid_argument("Invalid subrequest");
        std::cout << "Error: Invalid subrequest" << std::endl;
    }
}

Request readRequest() {
    using std::cout, std::cin, std::endl, std::string, std::stringstream, std::get-
line, std::invalid_argument;

    string operation;
    Request request;

    string operationType;
    size_t id = -1;
    string timerSubrequest;

    getline(cin, operation);

    stringstream ss(operation);

    ss >> operationType;
    validateOperationType(request, operationType);

    if (request.operationType == OperationType::QUIT) {
        if (ss >> operationType) {
            // throw invalid_argument("Invalid input");
            std::cout << "Error: Invalid input" << std::endl;
        }
        request.id = -1;
        request.subrequest = TimerSubrequest::NOTHING;
        return request;
    }

    ss >> id;
```

```
    validateId(request, id);
    ss >> timerSubrequest;
    validateSubrequest(request, timerSubrequest);

    // Пришла еще одно строка
    if ((ss >> timerSubrequest)) {
        // throw invalid_argument("Invalid input");
        std::cout << "Error: Invalid input" << std::endl;
    }

    return request;
}

// MESSAGE VALIDATOR ------------------------------------------------------------

// MESSAGE SENDER/PULLER ---------------------------------------------------------
----

void pushRequest(zmq::socket_t& socket, Request& request) {
    zmq::message_t message(&request, sizeof(Request));

    #pragma GCC diagnostic push
    #pragma GCC diagnostic ignored "-Wunused-result"
    socket.send(message, zmq::send_flags::none);
    #pragma GCC diagnostic pop
}

void pushReply(zmq::socket_t& socket, Reply& reply) {
    zmq::message_t message(&reply, sizeof(Reply));

    #pragma GCC diagnostic push
    #pragma GCC diagnostic ignored "-Wunused-result"
    socket.send(message, zmq::send_flags::none);
    #pragma GCC diagnostic pop
}

Reply pullReply(zmq::socket_t& socket) {
    zmq::message_t message(sizeof(Reply));

    auto reply = socket.recv(message, zmq::recv_flags::dontwait);
    if (reply.has_value()) {
        return *message.data<Reply>();
    }
    return Reply {OperationType::NOTHING, -1, ErrorTypes::NO_ERRORS, -1, TimerSub-
request::NOTHING};
}

Request pullRequest(zmq::socket_t& socket) {
    zmq::message_t message(sizeof(Request));

    #pragma GCC diagnostic push
```

```
    #pragma GCC diagnostic ignored "-Wunused-result"
    socket.recv(message, zmq::recv_flags::none);
    #pragma GCC diagnostic pop

    return *message.data<Request>();
}
// MESSAGE SENDER/PULLER ------------------------------------------------------
----

// Валидация порта и nodeId
std::pair<pid_t, std::pair<bool, size_t>> createNewNode(std::unordered_map<ssize_t,
std::pair<pid_t, size_t>>& nodes, ssize_t nodeId, size_t& currentPort) {

    bool replace = false;

    if (nodes.count(nodeId) > 0) {
        std::cout << "Error: Already exists" << std::endl;
        return std::pair<pid_t, std::pair<bool, size_t>> {PID_FOR_ALREADY_EX-
IST_NODE, {replace, 0}};
    }


    size_t currentPortCopy = currentPort;
    for (auto& node : nodes) {
        if (!isProcessExists(node.second.first) && node.second.second != AL-
READY_REPLACED) {
            currentPortCopy = node.second.second;
            node.second.second = ALREADY_REPLACED;
            replace = true;
            break;
        }
    }

    pid_t processId = fork();
    if (processId == -1) {
        perror("Fork error (server)");
        exit(1);
    }

    // Нормальный путь сделать
    if (processId == 0) {
        char srtNodeId[MAX_NODE_ID_LENGTH + 1];
        char strCurrentPort[MAX_PORT_LENGTH + 1];
        sprintf(srtNodeId, "%zd", nodeId);
        sprintf(strCurrentPort, "%zu", currentPortCopy);
        if (execl("./build/worker_exe", "./build/worker_exe", srtNodeId, strCur-
rentPort, NULL) == -1) {
            perror("Exec error (server)");
            exit(1);
        }
    }
```

```
    std::cout << "Ok: " << processId << std::endl;
    return std::pair<pid_t, std::pair<bool, size_t>> {processId, std::pair<bool,
size_t>{replace, currentPortCopy}};
}

bool isNodeAvaliable(std::map<size_t, std::vector<pid_t>>& nodeByPort, pid_t pro-
cessId) {
    for (auto& portInfo : nodeByPort) {
        bool wasGoodWorker = false;
        for (pid_t id : nodeByPort[portInfo.first]) {
            if (id == processId) {
                return isProcessExists(processId);
            } else if (!wasGoodWorker) {
                wasGoodWorker = isProcessExists(id);
            }
        }
        if (!wasGoodWorker) {
            return false;
        }
    }
    return true;
}

void updateNodeMap(std::unordered_map<ssize_t, std::pair<pid_t, size_t>>& nodes,
std::map<size_t, std::vector<pid_t>>& nodeByPort, size_t& currentPort, Request& re-
quest) {
    std::pair<pid_t, std::pair<bool, size_t>> newWorkerInfo = createNewNode(nodes,
request.id, currentPort);
    if (newWorkerInfo.first != PID_FOR_ALREADY_EXIST_NODE) {
        nodes[request.id] = std::pair<pid_t, size_t>{newWorkerInfo.first, cur-
rentPort};

        if (nodeByPort.count(newWorkerInfo.second.second) == 0) {
            std::vector<pid_t> vct;
            vct.push_back(newWorkerInfo.first);
            nodeByPort[newWorkerInfo.second.second] = vct;
        } else {
            nodeByPort[newWorkerInfo.second.second].push_back(newWorkerInfo.first);
        }
        if (!newWorkerInfo.second.first) {
            currentPort += 2;
        }
    }
}

void killWorkers(std::unordered_map<ssize_t, std::pair<pid_t, size_t>>& nodes) {
    for (auto worker : nodes) {
        kill(worker.second.first, SIGTERM);
    }
}
```

```
bool isProcessExists(pid_t pid) {
    int errnoBefore = errno;
    kill(pid, 0);
    int newErrno = errno;
    errno = errnoBefore;
    return newErrno != ESRCH;
}
```

Server.cpp

```
#include <iostream>
#include <thread>
#include <chrono>
#include <semaphore>


#include "definitions.hpp"
#include "functions.hpp"

zmq::context_t context1;
zmq::context_t context2;

zmq::socket_t pullReplySocket(context1, zmq::socket_type::pull);
zmq::socket_t pushRequestSocket(context2, zmq::socket_type::push);

bool waitForNewRequest = true;

std::binary_semaphore endSemaphore(0);

void getReply() {
    while (waitForNewRequest) {
        Reply data = pullReply(pullReplySocket);
        if (data.operationType != OperationType::NOTHING) {

            if (data.error == ErrorTypes::STOP_BEFORE_START) {
                std::cout << "Error: " << data.id << ": Stop Before Start" <<
std::endl;
            } else if (data.subrequest == TimerSubrequest::START) {
                std::cout << "Ok: " << data.id << std::endl;
            } else if (data.subrequest == TimerSubrequest::STOP) {
                std::cout << "Ok: " << data.id << std::endl;
            } else if (data.subrequest == TimerSubrequest::TIME) {
                std::cout << "Ok: " << data.id << ": " << data.result << std::endl;
            }

        }
    }
    endSemaphore.release();
}
```

```
void updateWorkersCount(std::unordered_map<ssize_t, std::pair<pid_t, size_t>>&
nodes, size_t& workersCount) {
    workersCount = 0;
    for (auto node : nodes) {
        workersCount += isProcessExists(node.second.first);
    }
}

int main() {
    size_t workerksCount = 0;
    /*
        Key: nodeId
        Value: <ProcessId, currentPort>
    */
    std::unordered_map<ssize_t, std::pair<pid_t, size_t>> nodes;
    std::map<size_t, std::vector<pid_t>> nodesByPort;

    size_t currentPort = MIN_DYNAMIC_PORT;
    Request request;

    pullReplySocket.bind(getAddres(currentPort + 0));
    pushRequestSocket.bind(getAddres(currentPort + 1));

    currentPort += 2;

    std::thread replyThread(getReply);

    while (waitForNewRequest) {
        request = readRequest();

        if (workerksCount && waitpid(-1, NULL, WNOHANG) == -1) {
            perror("Wait error (server)");
            exit(1);
        }
        updateWorkersCount(nodes, workerksCount);
        switch (request.operationType) {
        case OperationType::QUIT:
            waitForNewRequest = false;
            endSemaphore.acquire();
            break;
        case OperationType::EXEC:
            if (nodes.count(request.id) == 0) {
                std::cout << "Error: " << request.id << ": Not found" << std::endl;
            } else if (!isNodeAvaliable(nodesByPort, nodes[request.id].first)){
                std::cout << "Error: " << request.id << ": Node is unvaliable" <<
std::endl;
            } else {
                pushRequest(pushRequestSocket, request);
            }
            break;
```

```
        case OperationType::CREATE:
           ++workerksCount;
            updateNodeMap(nodes, nodesByPort, currentPort, request);
            break;
        case OperationType::PING:
            if (nodes.count(request.id) == 0) {
                std::cout << "Error: " << request.id << ": Not found" << std::endl;
            } else {
                if (isNodeAvaliable(nodesByPort, nodes[request.id].first)) {
                    std::cout << "Ok: 1" << std::endl;
                } else {
                    std::cout << "Ok: 0" << std::endl;
                }
            }
            break;
        case OperationType::NOTHING:
            break;
        }
    }

    replyThread.join();
    pullReplySocket.close();
    pushRequestSocket.close();
    killWorkers(nodes);

    return 0;
}
```

Worker.cpp

```
#include <iostream>
#include <zmq_addon.hpp>
#include <zmq.hpp>
#include <thread>
#include <chrono>
#include <sys/time.h>
#include <semaphore>
#include <mutex>

#include "definitions.hpp"
#include "functions.hpp"

zmq::context_t context1;
zmq::context_t context2;
zmq::context_t context3;
zmq::context_t context4;

zmq::socket_t pullRequestSocket(context1, zmq::socket_type::pull);
zmq::socket_t pushRequestSocket(context2, zmq::socket_type::push);
zmq::socket_t pullReplySocket(context3, zmq::socket_type::pull);
```

```
zmq::socket_t pushReplySocket(context4, zmq::socket_type::push);

bool processWorking = true;

std::binary_semaphore killSemaphore(0);
std::vector<Request> tasks;
std::mutex taskMutex;

ssize_t startTime;
ssize_t stopTime;
ssize_t currentTime = 0;
bool isTimerStarted = false;


void signal_handler(int signal) {
    if (signal == SIGTERM) {
        processWorking = false;
        killSemaphore.acquire(); // Чтобы мы сначала вышли из цикла, и не было
ошибок
        pushRequestSocket.close();
        pullReplySocket.close();
    }
    exit(0);
}

/*

pushRequest старшего
| |
| |, + нечетное число к MIN_DYNAMIC_PORT
| |
pullRequest       pushRequest
-----------> NODE ----------->
<----------- NODE <-----------
pushReply         pullReply
| |
| |, + четное число к MIN_DYNAMIC_PORT
| |
pullReply старшего

*/

void getReply() {
    while (processWorking) {
        zmq::message_t replyFromWorker(sizeof(size_t));
        Reply reply = pullReply(pullReplySocket);
        if (reply.operationType != OperationType::NOTHING) {
            pushReply(pushReplySocket, reply);
        }
    }
    killSemaphore.release();
```

```
}

ssize_t getMillseconds() {
    struct timeval currentTime;
    gettimeofday(&currentTime, NULL);
    return ((size_t)currentTime.tv_sec * (size_t)1e6 + (size_t)currentTime.tv_usec)
/ 1000;
}

void taskComplete() {
    while (true) {
        taskMutex.lock();
        if (tasks.empty()) {
            taskMutex.unlock();
            continue;
        }
        Request request = tasks.back();
        tasks.pop_back();
        Reply reply;
        reply.operationType = request.operationType;
        reply.error = ErrorTypes::NO_ERRORS;
        reply.subrequest = request.subrequest;
        reply.id = request.id;

        if (request.subrequest == TimerSubrequest::START) {
            isTimerStarted = true;
            startTime = getMillseconds();
            reply.result = startTime;
        } else if (request.subrequest == TimerSubrequest::STOP) {
            if (!isTimerStarted) {
                reply.error = ErrorTypes::STOP_BEFORE_START;
            } else {
                isTimerStarted = false;
                stopTime = getMillseconds();
                currentTime = stopTime - startTime;
            }
            reply.result = stopTime;
        } else if (request.subrequest == TimerSubrequest::TIME) {
            reply.result = currentTime;
            std::this_thread::sleep_for(std::chrono::seconds(15));
        }
        pushReply(pushReplySocket, reply);
        taskMutex.unlock();
    }
}

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

    if (argc != 3) {
        std::cerr << "Invalid worker exec (worker)" << std::endl;
        exit(1);
```

```
    }

    if (signal(SIGTERM, signal_handler) == SIG_ERR) {
        perror("Can't catch signal");
        exit(1);
    }

    ssize_t nodeId = (ssize_t)atoll(const_cast<char*>(argv[1]));
    size_t currentPort = (size_t)atoll(const_cast<char*>(argv[2]));

    pushReplySocket.connect(getAddres(currentPort - 2));
    pullRequestSocket.connect(getAddres(currentPort - 1));

    pushRequestSocket.bind(getAddres(currentPort + 1));
    pullReplySocket.bind(getAddres(currentPort + 0));

    std::thread replyThread(getReply);
    std::thread taskCompleteThread(taskComplete);

    while (true) {
        Request request = pullRequest(pullRequestSocket);
        if (request.id == nodeId) { // Ответ
            taskMutex.lock();
            tasks.push_back(request);
            taskMutex.unlock();
        } else {
            pushRequest(pushRequestSocket, request);
        }
    }

    return 0;
}
```

Пример работы

create 0 -1

Ok: 5384

create 1 -1

Ok: 5404

create 2 -1

Ok: 5422

ping 0

Ok: 1

ping 1

Ok: 1

ping 2

Ok: 1

exec 0 start

Ok: 0

exec 1 stop

Error: 1: Stop Before Start

exec 2 start

Ok: 2

quit

Выводы

Эта лабораторная работа является самый сложной из всех сделанных за этот курс. Сложность обусловлена тем, что здесь используется новые вещи, такие, как сокеты, очереди сообщений, а кроме того, используется почти всё, что было пройдено во время курса: создание процессов, потоки, синхронизация потоков. В лабораторной работе были использованы очереди сообщений используя технологию ZEROMQ. Мной была реализована асинхронная система по выполнению заданий. Т.е. если есть несколько рабочих, мы можем каждому из них поручить выполнять определенные задания, и они будут выполнять это одновременно, нам не нужно будет ждать ответа от одного из рабочих, чтобы дать задание другому. Так же, хотя в лабораторной работе сокеты и очереди сообщений использовались для взаимодействия процессов, находящихся на одном устройстве, приобретенные навыки позволят использовать сокеты для взаимодействия по сети.