Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №5-7 по курсу**

**«Операционные системы»**

Студент: Бурдинский Владислав Дмитриевич

Группа: М8О–203Б–23

Вариант: 34

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2024.

**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.

**Код программы**

**client.cpp**

#include "node.h"

#include "net\_func.h"

#include "set"

#include <signal.h>

#include <chrono>

#include <thread>

#include <mutex>

#include <atomic>

#include <map>

static std::atomic<bool> heartbeat\_active(false);

static std::atomic<int> heartbeat\_interval\_ms(0);

static std::thread heartbeat\_thread;

static bool heartbeat\_thread\_started = false;

static std::mutex nodes\_mutex;

static std::map<int, std::chrono::time\_point<std::chrono::steady\_clock>> last\_success\_ping;

void heartbeat\_thread\_func(Node\* me, std::set<int>\* all\_nodes) {

// Поток отправляет пинги всем известным узлам каждые heartbeat\_interval\_ms миллисекунд

// и проверяет, есть ли узлы, не ответившие уже > 4 раза дольше интервала.

while (heartbeat\_active.load()) {

int interval = heartbeat\_interval\_ms.load();

if (interval <= 0) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

continue;

}

{

std::lock\_guard<std::mutex> lock(nodes\_mutex);

for (auto node\_id : \*all\_nodes) {

if (node\_id == -1) {

// Корневой узел - скип

continue;

}

// Пингуем узел

std::string ans;

if (me->children.find(node\_id) != me->children.end()) {

ans = me->Ping\_child(node\_id);

} else {

std::string str = "ping " + std::to\_string(node\_id);

ans = me->Send(str, node\_id);

if (ans == "Error: not find") {

ans = "Ok: 0";

}

}

if (ans == "Ok: 1") {

// Узел доступен, обновляем время последнего успешного ответа

last\_success\_ping[node\_id] = std::chrono::steady\_clock::now();

} else {

// Узел недоступен или не найден. Если его нет в last\_success\_ping, занесем текущее время,

// чтобы отсчитать таймер недоступности.

if (last\_success\_ping.find(node\_id) == last\_success\_ping.end()) {

last\_success\_ping[node\_id] = std::chrono::steady\_clock::now();

} else {

// Проверим, сколько времени прошло с последнего успешного пинга

auto now = std::chrono::steady\_clock::now();

auto diff = std::chrono::duration\_cast<std::chrono::milliseconds>(now - last\_success\_ping[node\_id]).count();

// Если прошло более чем 4 \* interval, сообщаем о недоступности

if (diff > 4 \* interval) {

std::cout << "Heartbit: node " << node\_id << " is unavailable now" << std::endl;

// Чтобы сообщение не повторялось бесконечно, обновим время так, чтобы снова ждать

last\_success\_ping[node\_id] = std::chrono::steady\_clock::now();

}

}

}

}

}

std::this\_thread::sleep\_for(std::chrono::milliseconds(heartbeat\_interval\_ms.load()));

}

}

int main() {

std::set<int> all\_nodes;

std::string prog\_path = "./worker";

Node me(-1);

all\_nodes.insert(-1);

last\_success\_ping[-1] = std::chrono::steady\_clock::now();

std::string command;

while (std::cin >> command) {

if (command == "create") {

int id\_child, id\_parent;

std::cin >> id\_child >> id\_parent;

if (all\_nodes.find(id\_child) != all\_nodes.end()) {

std::cout << "Error: Already exists" << std::endl;

} else if (all\_nodes.find(id\_parent) == all\_nodes.end()) {

std::cout << "Error: Parent not found" << std::endl;

} else if (id\_parent == me.id) {

std::string ans = me.Create\_child(id\_child, prog\_path);

std::cout << ans << std::endl;

all\_nodes.insert(id\_child);

} else {

std::string str = "create " + std::to\_string(id\_child);

std::string ans = me.Send(str, id\_parent);

std::cout << ans << std::endl;

all\_nodes.insert(id\_child);

}

} else if (command == "ping") {

int id\_child;

std::cin >> id\_child;

if (all\_nodes.find(id\_child) == all\_nodes.end()) {

std::cout << "Error: Not found" << std::endl;

} else if (me.children.find(id\_child) != me.children.end()) {

std::string ans = me.Ping\_child(id\_child);

std::cout << ans << std::endl;

} else {

std::string str = "ping " + std::to\_string(id\_child);

std::string ans = me.Send(str, id\_child);

if (ans == "Error: not find") {

ans = "Ok: 0";

}

std::cout << ans << std::endl;

}

} else if (command == "exec") {

int id;

std::string cmd;

std::cin >> id >> cmd;

std::string msg = "exec " + cmd;

if (all\_nodes.find(id) == all\_nodes.end()) {

std::cout << "Error: Not found" << std::endl;

} else {

std::string ans = me.Send(msg, id);

std::cout << ans << std::endl;

}

} else if (command == "remove") {

int id;

std::cin >> id;

std::string msg = "remove";

if (all\_nodes.find(id) == all\_nodes.end()) {

std::cout << "Error: Not found" << std::endl;

} else {

std::string ans = me.Send(msg, id);

if (ans != "Error: not find") {

std::istringstream ids(ans);

int tmp;

while (ids >> tmp) {

all\_nodes.erase(tmp);

}

ans = "Ok";

if (me.children.find(id) != me.children.end()) {

my\_net::unbind(me.children[id], me.children\_port[id]);

me.children[id]->close();

me.children.erase(id);

me.children\_port.erase(id);

}

}

std::cout << ans << std::endl;

}

}else if (command == "heartbit") {

int time\_ms;

std::cin >> time\_ms;

if (time\_ms <= 0) {

std::cout << "Error: invalid time" << std::endl;

continue;

}

heartbeat\_interval\_ms.store(time\_ms);

if (!heartbeat\_thread\_started) {

heartbeat\_active.store(true);

heartbeat\_thread = std::thread(heartbeat\_thread\_func, &me, &all\_nodes);

heartbeat\_thread.detach();

heartbeat\_thread\_started = true;

}

std::cout << "Ok" << std::endl;

}

}

heartbeat\_active.store(false);

me.Remove();

return 0;

}

**net\_func.h**

#pragma once

#include <iostream>

#include <zmq.hpp>

#include <sstream>

#include <string>

namespace my\_net {

#define MY\_PORT 4040

#define MY\_IP "tcp://127.0.0.1:"

int bind(zmq::socket\_t \*socket, int id) {

int port = MY\_PORT + id;

while (true) {

std::string adress = MY\_IP + std::to\_string(port);

try {

socket->bind(adress);

break;

} catch (...) {

port++;

}

}

return port;

}

void connect(zmq::socket\_t \*socket, int port) {

std::string adress = MY\_IP + std::to\_string(port);

socket->connect(adress);

}

void unbind(zmq::socket\_t \*socket, int port) {

std::string adress = MY\_IP + std::to\_string(port);

socket->unbind(adress);

}

void disconnect(zmq::socket\_t \*socket, int port) {

std::string adress = MY\_IP + std::to\_string(port);

socket->disconnect(adress);

}

void send\_message(zmq::socket\_t \*socket, const std::string msg) {

zmq::message\_t message(msg.size());

memcpy(message.data(), msg.c\_str(), msg.size());

try {

socket->send(message);

} catch (...) {}

}

std::string reseave(zmq::socket\_t \*socket) {

zmq::message\_t message;

bool success = true;

try {

socket->recv(&message, 0);

} catch (...) {

success = false;

}

if (!success || message.size() == 0) {

throw -1;

}

std::string str(static\_cast<char \*>(message.data()), message.size());

return str;

}

}

node.h

#include <iostream>

#include "net\_func.h"

#include <sstream>

#include <unordered\_map>

#include "unistd.h"

class Node {

private:

zmq::context\_t context;

public:

std::unordered\_map<int, zmq::socket\_t \*> children;

std::unordered\_map<int, int> children\_port;

zmq::socket\_t parent;

int parent\_port;

int id;

Node(int \_id, int \_parent\_port = -1) : parent(context, ZMQ\_REP),

parent\_port(\_parent\_port),

id(\_id) {

if (\_id != -1) {

my\_net::connect(&parent, \_parent\_port);

}

}

std::string Ping\_child(int \_id) {

std::string ans = "Ok: 0";

if (\_id == id) {

ans = "Ok: 1";

return ans;

} else if (children.find(\_id) != children.end()) {

std::string msg = "ping " + std::to\_string(\_id);

my\_net::send\_message(children[\_id], msg);

try {

msg = my\_net::reseave(children[\_id]);

if (msg == "Ok: 1")

ans = msg;

} catch (int) {}

return ans;

} else {

return ans;

}

}

std::string Create\_child(int child\_id, std::string program\_path) {

std::string program\_name = program\_path.substr(program\_path.find\_last\_of("/") + 1);

children[child\_id] = new zmq::socket\_t(context, ZMQ\_REQ);

int new\_port = my\_net::bind(children[child\_id], child\_id);

children\_port[child\_id] = new\_port;

int pid = fork();

if (pid == 0) {

execl(program\_path.c\_str(), program\_name.c\_str(), std::to\_string(child\_id).c\_str(),

std::to\_string(new\_port).c\_str(), (char \*) NULL);

} else {

std::string child\_pid;

try {

children[child\_id]->setsockopt(ZMQ\_SNDTIMEO, 3000);

my\_net::send\_message(children[child\_id], "pid");

child\_pid = my\_net::reseave(children[child\_id]);

} catch (int) {

child\_pid = "Error: can't connect to child";

}

return "Ok: " + child\_pid;

}

}

std::string Pid() {

return std::to\_string(getpid());

}

std::string Send(std::string str, int \_id) {

if (children.size() == 0) {

return "Error: now find";

} else if (children.find(\_id) != children.end()) {

if (Ping\_child(\_id) == "Ok: 1") {

my\_net::send\_message(children[\_id], str);

std::string ans;

try {

ans = my\_net::reseave(children[\_id]);

} catch (int) {

ans = "Error: now find";

}

return ans;

}

} else {

std::string ans = "Error: not find";

for (auto &child: children) {

if (Ping\_child(child.first) == "Ok: 1") {

std::string msg = "send " + std::to\_string(\_id) + " " + str;

my\_net::send\_message(children[child.first], msg);

try {

msg = my\_net::reseave(children[child.first]);

} catch (int) {

msg = "Error: not find";

}

if (msg != "Error: not find") {

ans = msg;

}

}

}

return ans;

}

return "Error: not find";

}

std::string Remove() {

std::string ans;

if (children.size() > 0) {

for (auto &child: children) {

if (Ping\_child(child.first) == "Ok: 1") {

std::string msg = "remove";

my\_net::send\_message(children[child.first], msg);

try {

msg = my\_net::reseave(children[child.first]);

if (ans.size() > 0)

ans = ans + " " + msg;

else

ans = msg;

} catch (int) {}

}

my\_net::unbind(children[child.first], children\_port[child.first]);

children[child.first]->close();

}

children.clear();

children\_port.clear();

}

return ans;

}

};

**worker.cpp**

#include "node.h"

#include "net\_func.h"

#include <fstream>

#include <vector>

#include <signal.h>

#include <chrono>

int my\_id = 0;

static bool timer\_running = false;

static std::chrono::time\_point<std::chrono::steady\_clock> start\_time;

static std::chrono::duration<double> elapsed(0.0);

void Log(std::string str) {

std::string f = std::to\_string(my\_id) + ".txt";

std::ofstream fout(f, std::ios\_base::app);

fout << str;

fout.close();

}

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

if (argc != 3) {

return -1;

}

Node me(atoi(argv[1]), atoi(argv[2]));

my\_id = me.id;

std::string prog\_path = "./worker";

while (1) {

std::string message;

std::string command = " ";

message = my\_net::reseave(&(me.parent));

std::istringstream request(message);

request >> command;

if (command == "create") {

int id\_child;

request >> id\_child;

std::string ans = me.Create\_child(id\_child, prog\_path);

my\_net::send\_message(&me.parent, ans);

} else if (command == "pid") {

std::string ans = me.Pid();

my\_net::send\_message(&me.parent, ans);

} else if (command == "ping") {

int id\_child;

request >> id\_child;

std::string ans = me.Ping\_child(id\_child);

my\_net::send\_message(&me.parent, ans);

} else if (command == "send") {

int id;

request >> id;

std::string str;

getline(request, str);

str.erase(0, 1);

std::string ans;

ans = me.Send(str, id);

my\_net::send\_message(&me.parent, ans);

} else if (command == "exec") {

std::string cmd;

request >> cmd;

std::string ans;

if (cmd == "start") {

if (!timer\_running) {

timer\_running = true;

elapsed = std::chrono::duration<double>(0);

start\_time = std::chrono::steady\_clock::now();

ans = "Ok:" + std::to\_string(me.id) + ":timer started";

} else {

ans = "Ok:" + std::to\_string(me.id) + ":timer already running";

}

} else if (cmd == "time") {

if (timer\_running) {

auto now = std::chrono::steady\_clock::now();

auto diff = std::chrono::duration\_cast<std::chrono::milliseconds>(now - start\_time).count();

ans = "Ok:" + std::to\_string(me.id) + ":elapsed " + std::to\_string(diff) + " ms";

} else {

ans = "Ok:" + std::to\_string(me.id) + ":timer not running";

}

} else if (cmd == "stop") {

if (timer\_running) {

auto now = std::chrono::steady\_clock::now();

auto diff = std::chrono::duration\_cast<std::chrono::milliseconds>(now - start\_time).count();

timer\_running = false;

ans = "Ok:" + std::to\_string(me.id) + ":timer stopped at " + std::to\_string(diff) + " ms";

} else {

ans = "Ok:" + std::to\_string(me.id) + ":timer not running";

}

} else {

ans = "Error: invalid exec command";

}

my\_net::send\_message(&me.parent, ans);

} else if (command == "remove") {

std::cout << "[WORKER] Removing node " << my\_id << "..." << std::endl;

std::string ans = me.Remove();

ans = std::to\_string(me.id) + " " + ans;

my\_net::send\_message(&me.parent, ans);

my\_net::disconnect(&me.parent, me.parent\_port);

me.parent.close();

std::cout << "[WORKER] Node " << my\_id << " removed successfully." << std::endl;

break;

}

}

sleep(1);

return 0;

}

CMakeLists.txt

# Указываем имя проекта

project(ZMQProject)

# Указываем стандарт C++

set(CMAKE\_CXX\_STANDARD 17)

set(CMAKE\_CXX\_STANDARD\_REQUIRED True)

# Параметры компиляции

include\_directories(/opt/homebrew/include)

include\_directories(${CMAKE\_SOURCE\_DIR}) # Добавляем текущую директорию для заголовков

link\_directories(/opt/homebrew/lib)

# Основные цели: клиент и воркер

add\_executable(client client.cpp)

add\_executable(worker worker.cpp)

# Линковка ZeroMQ

target\_link\_libraries(client zmq)

target\_link\_libraries(worker zmq)

# Подключаем Google Test

find\_package(GTest REQUIRED) # Найти установленный Google Test

find\_package(Threads REQUIRED) # Подключаем поддержку потоков

include\_directories(${GTEST\_INCLUDE\_DIRS})

# Создаём цель для тестов

add\_executable(tests

tests/test\_main.cpp # Файл с тестами

node.h # Заголовочные файлы

net\_func.h

)

# Линкуем тесты с Google Test, ZeroMQ и потоками

target\_link\_libraries(tests PRIVATE ${GTEST\_LIBRARIES} zmq Threads::Threads)

# Включаем тестирование

enable\_testing()

add\_test(NAME ZMQTests COMMAND tests)

# Команда для очистки

add\_custom\_target(clean COMMAND ${CMAKE\_COMMAND} -E remove -f client worker tests)

test\_main.cpp

#include <gtest/gtest.h>

#include <chrono>

#include <thread>

#include <unordered\_map>

#include <unordered\_set>

#include <iostream>

// ------------------------------

// Класс TimerManager

// ------------------------------

class TimerManager {

public:

void Start() {

running = true;

start\_time = std::chrono::steady\_clock::now();

}

long ElapsedMs() const {

if (!running) return 0;

auto now = std::chrono::steady\_clock::now();

return std::chrono::duration\_cast<std::chrono::milliseconds>(now - start\_time).count();

}

void Stop() {

running = false;

}

private:

bool running = false;

std::chrono::time\_point<std::chrono::steady\_clock> start\_time;

};

// ------------------------------

// Класс TreeManager

// Управляет "узлами" в дереве:

// - Узел имеет id

// - Есть корневой узел (-1)

// - Можно создавать узлы с заданным родителем

// - Можно удалять узлы и их потомков

// - Можно проверять, доступен ли узел

// ------------------------------

class TreeManager {

public:

TreeManager() {

// Можно сразу создать корневой узел, если нужно

CreateNode(-1, -1);

}

bool CreateNode(int child\_id, int parent\_id) {

// Если узел уже существует, ошибка

if (nodes.find(child\_id) != nodes.end()) return false;

// Если родитель не существует и это не корень

if (parent\_id != -1 && nodes.find(parent\_id) == nodes.end()) return false;

NodeInfo info;

info.id = child\_id;

info.parent = parent\_id;

nodes[child\_id] = info;

// Добавляем child\_id в список детей родителя

if (parent\_id != -1) {

nodes[parent\_id].children.insert(child\_id);

}

return true;

}

bool RemoveNode(int id) {

// Если узел не существует

if (nodes.find(id) == nodes.end()) return false;

// Рекурсивно удаляем всех потомков

RemoveSubtree(id);

return true;

}

bool IsNodeAccessible(int id) const {

return nodes.find(id) != nodes.end();

}

private:

struct NodeInfo {

int id;

int parent;

std::unordered\_set<int> children;

};

std::unordered\_map<int, NodeInfo> nodes;

void RemoveSubtree(int id) {

// Сначала удаляем всех детей

for (auto c : nodes[id].children) {

RemoveSubtree(c);

}

// Удаляем связь родителя

int parent\_id = nodes[id].parent;

if (parent\_id != -1) {

nodes[parent\_id].children.erase(id);

}

// Теперь удаляем сам узел

nodes.erase(id);

}

};

TEST(TimerManagerTest, StartStopCheck) {

TimerManager tm;

tm.Start();

std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

long elapsed = tm.ElapsedMs();

// Проверяем, что прошло хотя бы 50 мс

EXPECT\_GE(elapsed, 50);

tm.Stop();

// После остановки таймер должен возвращать 0

EXPECT\_EQ(tm.ElapsedMs(), 0);

}

TEST(TimerManagerTest, NoStartCheck) {

TimerManager tm;

// Без Start elapsed всегда 0

EXPECT\_EQ(tm.ElapsedMs(), 0);

tm.Stop();

EXPECT\_EQ(tm.ElapsedMs(), 0);

}

TEST(TreeManagerTest, CreateRemoveCheck) {

TreeManager tm;

// По умолчанию корневой узел (-1) существует

EXPECT\_TRUE(tm.IsNodeAccessible(-1));

// Создаём узел 10 от -1

EXPECT\_TRUE(tm.CreateNode(10, -1));

EXPECT\_TRUE(tm.IsNodeAccessible(10));

// Создаём узел 11 от 10

EXPECT\_TRUE(tm.CreateNode(11, 10));

EXPECT\_TRUE(tm.IsNodeAccessible(11));

// Удаляем узел 10, должно исчезнуть и 11

EXPECT\_TRUE(tm.RemoveNode(10));

EXPECT\_FALSE(tm.IsNodeAccessible(10));

EXPECT\_FALSE(tm.IsNodeAccessible(11));

}

TEST(TreeManagerTest, FailCases) {

TreeManager tm;

// Нельзя создать узел с уже существующим ID

EXPECT\_FALSE(tm.CreateNode(-1, -1));

// Нельзя создать узел с несуществующим родителем (кроме корня)

EXPECT\_FALSE(tm.CreateNode(100, 9999));

// Удаление несуществующего узла

EXPECT\_FALSE(tm.RemoveNode(9999));

}

// Главная функция запуска тестов

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

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

**Пример работы**

(base) vladislavburdinskij@MacBook-Pro-Vladislav build % ./client

create 10 -1

Ok: 38216

ping 10

Ok: 1

exec 10 start

Ok:10:timer started

exec 10 time

Ok:10:elapsed 2936 ms

exec 10 stop

Ok:10:timer stopped at 5531 ms

^C

(base) vladislavburdinskij@MacBook-Pro-Vladislav build % **Вывод**

В данной лабораторной я узнал, как работают очереди сообщений и смог построить систему узлов, которая с помощью ZMQ передает данные от родителя к ребенку и может передавать команды для таймера на любом из узлов.