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»

Студент: Смирнов А.В.
Группа: М8О-207Б-21
Вариант: 25
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2022

Содержание

Репозиторий

https://github.com/Liguha/OS

Постановка задачи

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. Управляющий узел отвечает за ввод команд от пользователя и отправку этих команд на вычислительные узлы. Список основных поддерживаемых команд: create, exec, remove, pingall.

Общие сведения о программе

Программа распределительного узла компилируется из файла distribution_node.cpp, программа вычислительного узла компилируется из файла computing_node.cpp. В программе используется библиотека для работы с потоками и мьютексами, а также сторонняя библиотека для работы с сервером сообщений ZeroMQ.

Общий метод и алгоритм решения

Распределительный узел хранит в себе двоичное дерево, элементами которого являются async_node – структуры, хранящие в себе id вычислительного узла, порт для связи с ним, поток отправляющий/принимающий запросы/ответы к узлу, очередь отправленных на узел запросов и мьютекс, обеспечивающий возможность работать с этой очередью в несколько потоков.

В функции main запускается бесконечный цикл обработки пользовательских запросов, при получении запроса он отправляется в очередь обработчика соответствующего узла, откуда позже будет извлечён потоком обработки и переслан требуемому узлу (обеспечение асинхронности). При удалении узла у записи, соответствующей удаляемому узлу устанавливается переменная активности в false, благодаря чему обработчик, отправив на узел запрос удаления и получив ответ, выходит из цикла и очищает память от уже ненужной записи. Подобные действия применяются и ко всем дочерним узлам удаляемого.

Pingall выполняет обход дерева в ширину, проверяя доступность каждого узла. Проверка

осуществляется попыткой подключиться к узлу: если монитор сокета получает сигнал ZMQ_EVENT_CONNECTED, то узел считается доступным, если же вместо этого приходит ZMQ_EVENT_CONNECT_RETRIED, то узел считается недоступным.

Исходный код

distribution_node.cpp

```
#include "zmq.h"
#include "string.h"
#include "unistd.h"
#include "stdlib.h"
#include "pthread.h"

#include <iostream>
#include <queue>
#include <vector>

#define CREATE 1
#define EXEC 0
#define REMOVE -1

#define CHECK_ERROR(expr, stream, act) \
    do \
    { \
        int res = (expr); \
        if (res != 0) \
        { \
            std::cerr << stream; \
            act; \
        } \
    } while (0)

#define CHECK_ZMQ(expr, stream, act) \
    do \
    { \
        int res = (expr); \
        if (res == -1) \
        { \
            std::cerr << stream; \
            act; \
        } \
    } while (0)

const int MIN_PORT = 1024;

using namespace std;

string protocol = "tcp://localhost:";

void* async_node_thd(void*);

struct async_node
{
    int id;
    string port;
    bool active;
    async_node* L;
    async_node* R;
    pthread_mutex_t mutex;
    pthread_t thd;
    queue <vector <int>> q;

    async_node(int i)
    {
        id = i;
```

```
        port = protocol + to_string(i);
        active = true;
        L = nullptr;
        R = nullptr;
        CHECK_ERROR(pthread_mutex_init(&mutex, NULL), "Error:" << i - MIN_PORT << ": Gateway mutex error\n",
return);
        CHECK_ERROR(pthread_create(&thd, NULL, async_node_thd, this), "Error:" << i - MIN_PORT << ": Gateway
thread error\n", return);
        CHECK_ERROR(pthread_detach(thd), "Error:" << i << ": Gateway thread error\n", return);
    }

    void make_query(vector <int> v)
    {
        CHECK_ERROR(pthread_mutex_lock(&mutex), "Error:" << id - MIN_PORT << ": Gateway mutex lock error\n",
active = false; return);
        q.push(v);
        CHECK_ERROR(pthread_mutex_unlock(&mutex), "Error:" << id - MIN_PORT << ": Gateway mutex unlock er-
ror\n", active = false);
    }

    ~async_node()
    {
        pthread_mutex_destroy(&mutex);
    }
};

async_node* find_node_exec(async_node* ptr, int id)
{
    if (ptr == nullptr)
        return nullptr;
    if (ptr->id > id)
        return find_node_exec(ptr->L, id);
    if (ptr->id < id)
        return find_node_exec(ptr->R, id);
    return ptr;
}

async_node* find_node_create(async_node* ptr, int id)
{
    if (ptr == nullptr)
        return nullptr;
    if (ptr->L == nullptr && ptr->id > id)
        return ptr;
    if (ptr->R == nullptr && ptr->id < id)
        return ptr;
    if (ptr->id > id)
        return find_node_create(ptr->L, id);
    if (ptr->id < id)
        return find_node_create(ptr->R, id);
    return nullptr;
}

bool destroy_node(async_node*& ptr, int id)
{
    if (ptr == nullptr)
        return false;
    if (ptr->id > id)
        return destroy_node(ptr->L, id);
    if (ptr->id < id)
        return destroy_node(ptr->R, id);
    ptr->active = false;
    ptr->make_query({REMOVE});
    if (ptr->L != nullptr)
        destroy_node(ptr->L, ptr->L->id);
    if (ptr->R != nullptr)
        destroy_node(ptr->R, ptr->R->id);
    ptr = nullptr;
    return true;
```

```
}

void* async_node_thd(void* ptr)
{
    async_node* node = (async_node*)ptr;
    void* context = zmq_ctx_new();
    void *req = zmq_socket(context, ZMQ_REQ);
    CHECK_ZMQ(zmq_connect(req, node->port.c_str()), "Error: Connection with" << node->id - MIN_PORT <<
"\n",);
    while (node->active)
    {
        if (node->q.empty())
            continue;
        CHECK_ERROR(pthread_mutex_lock(&node->mutex), "Error:" << node->id - MIN_PORT << ": Gateway mutex
lock error\n", node->active = false; break);
        vector <int> v = node->q.front();
        node->q.pop();
        CHECK_ERROR(pthread_mutex_unlock(&node->mutex), "Error:" << node->id - MIN_PORT << ": Gateway mutex
unlock error\n", node->active = false; break);

        switch (v[0])
        {
            case CREATE:
            {
                zmq_msg_t msg;
                CHECK_ZMQ(zmq_msg_init_size(&msg, 2 * sizeof(int)), "Error:" << node->id - MIN_PORT << ":
Message error\n", break);
                memcpy(zmq_msg_data(&msg), &v[0], 2 * sizeof(int));
                CHECK_ZMQ(zmq_msg_send(&msg, req, 0), "Error:" << node->id - MIN_PORT << ": Message er-
ror\n", break);
                int pid;
                CHECK_ZMQ(zmq_recv(req, &pid, sizeof(int), 0), "Error:" << node->id - MIN_PORT << ": Message
error\n", break);
                if (v[1] < node->id)
                    node->L = new async_node(v[1]);
                else
                    node->R = new async_node(v[1]);
                cout << "Ok: " << pid << '\n';
                zmq_msg_close(&msg);
                break;
            }

            case EXEC:
            {
                zmq_msg_t msg;
                int len = sizeof(int) * v.size();
                CHECK_ZMQ(zmq_msg_init_size(&msg, len), "Error:" << node->id - MIN_PORT << ": Message er-
ror\n", break);
                memcpy(zmq_msg_data(&msg), &v[0], len);
                CHECK_ZMQ(zmq_msg_send(&msg, req, 0), "Error:" << node->id - MIN_PORT << ": Message er-
ror\n", break);
                long long S;
                CHECK_ZMQ(zmq_recv(req, &S, sizeof(long long), 0), "Error:" << node->id - MIN_PORT << ":
Message error\n", break);
                cout << "Ok:" << node->id - MIN_PORT << ':' << S << '\n';
                zmq_msg_close(&msg);
                break;
            }

            case REMOVE:
            {
                CHECK_ZMQ(zmq_send(req, &v[0], sizeof(int), 0), "Error:" << node->id - MIN_PORT << ": Mes-
sage error\n", break);
                int ans;
                CHECK_ZMQ(zmq_recv(req, &ans, sizeof(int), 0), "Error:" << node->id - MIN_PORT << ": Message
error\n", break);
                break;
            }
```

```
        }
    }
    zmq_close(req);
    zmq_ctx_destroy(context);
    delete node;
    return NULL;
}

bool ping(int id)
{
    string port = protocol + to_string(id);
    string ping = "inproc://ping" + to_string(id);
    void* context = zmq_ctx_new();
    void *req = zmq_socket(context, ZMQ_REQ);

    zmq_socket_monitor(req, ping.c_str(), ZMQ_EVENT_CONNECTED | ZMQ_EVENT_CONNECT_RETRIED);
    void *soc = zmq_socket(context, ZMQ_PAIR);
    zmq_connect(soc, ping.c_str());
    zmq_connect(req, port.c_str());

    zmq_msg_t msg;
    zmq_msg_init(&msg);
    zmq_msg_recv(&msg, soc, 0);
    uint8_t* data = (uint8_t*)zmq_msg_data(&msg);
    uint16_t event = *(uint16_t*)(data);

    zmq_close(req);
    zmq_close(soc);
    zmq_msg_close(&msg);
    zmq_ctx_destroy(context);
    return event % 2;
}

async_node* tree = nullptr;

int main()
{
    while (true)
    {
        string command;
        cin >> command;
        if (command == "create")
        {
            int id;
            cin >> id;
            id += MIN_PORT;
            if (tree == nullptr)
            {
                string id_str = to_string(id);
                int pid = fork();
                if (pid == 0)
                    CHECK_ERROR(execl("server", id_str.c_str(), NULL), "Error:" << id - MIN_PORT << ": Cre-
ating error\n", break);
                cout << "Ok: " << pid << '\n';
                tree = new async_node(id);
            }
            else
            {
                async_node* node = find_node_create(tree, id);
                if (!ping(node->id))
                {
                    cerr << "Error:" << id - MIN_PORT << ": Parent is unavailable\n";
                    continue;
                }
                if (node != nullptr)
                    node->make_query({CREATE, id});
                else
                    cerr << "Error: Already exists\n";
```

```
            }
        }

        if (command == "exec")
        {
            int id, n;
            cin >> id >> n;
            id += MIN_PORT;
            vector <int> v(n + 2);
            v[0] = EXEC;
            v[1] = n;
            for (int i = 0; i < n; i++)
                cin >> v[i + 2];
            if (!ping(id))
            {
                cerr << "Error:" << id - MIN_PORT << ": Node is unavailable\n";
                continue;
            }
            async_node* node = find_node_exec(tree, id);
            if (node != nullptr)
                node->make_query(v);
            else
                cerr << "Error:" << id - MIN_PORT << ": Not found\n";
        }

        if (command == "remove")
        {
            int id;
            cin >> id;
            id += MIN_PORT;
            if (!ping(id))
            {
                cerr << "Error:" << id - MIN_PORT << ": Node is unavailable\n";
                continue;
            }
            bool state = destroy_node(tree, id);
            if (state)
                cout << "Ok\n";
            else
                cerr << "Error: Not found\n";
        }

        if (command == "unsafe_remove")
        {
            int id;
            cin >> id;
            id += MIN_PORT;
            bool state = destroy_node(tree, id);
            if (state)
                cout << "Ok\n";
            else
                cerr << "Error: Not found\n";
        }

        if (command == "pingall")
        {
            queue <async_node*> q;
            if (tree != nullptr)
                q.push(tree);
            vector <int> v;
            while (!q.empty())
            {
                async_node* ptr = q.front();
                q.pop();
                if (ptr->L != nullptr)
                    q.push(ptr->L);
                if (ptr->R != nullptr)
                    q.push(ptr->R);
```

```
                bool check = ping(ptr->id);
                if (!check)
                    v.push_back(ptr->id - MIN_PORT);
            }
            if (v.empty())
                v.push_back(-1);
            cout << "Ok: ";
            for (int i = 0; i < v.size(); i++)
            {
                if (i != 0)
                    cout << ';';
                cout << v[i];
            }
            cout << '\n';
        }
    }
}
```

computing_node.cpp

```
#include <zmq.h>
#include <unistd.h>
#include <string.h>

#include <iostream>
#include <vector>

#define CREATE 1
#define EXEC 0
#define REMOVE -1

#define CHECK_ERROR(expr) \
    do \
    { \
        int res = (expr); \
        if (res == -1) \
            return -1; \
    } while (0)

using namespace std;

int main(int argc, char* argv[])
{
    int id = atoi(argv[0]);
    string port = "tcp://*:" + to_string(id);
    void *context = zmq_ctx_new();
    void *responder = zmq_socket(context, ZMQ_REP);
    CHECK_ERROR(zmq_bind(responder, port.c_str()));

    while (true)
    {
        zmq_msg_t msg;
        CHECK_ERROR(zmq_msg_init(&msg));
        CHECK_ERROR(zmq_msg_recv(&msg, responder, 0));
        int* data = (int*)zmq_msg_data(&msg);
        int t = *data;
        switch (t)
        {
            case CREATE:
            {
                int n = *(++data);
                string id_str = to_string(n);
                int pid = fork();
                if (pid == -1)
```

```
                    return -1;
                if (pid == 0)
                    CHECK_ERROR(execl("server", id_str.c_str(), NULL));
                else
                    CHECK_ERROR(zmq_send(responder, &pid, sizeof(int), 0));
                break;
            }

            case EXEC:
            {
                int n = *(++data);
                vector <int> v(n);
                memcpy(&v[0], (++data), n * sizeof(int));
                long long S = 0;
                for (int i = 0; i < n; i++)
                    S += v[i];
                CHECK_ERROR(zmq_send(responder, &S, sizeof(long long), 0));
                break;
            }

            case REMOVE:
            {
                zmq_send(responder, &id, sizeof(int), 0);
                zmq_close(responder);
                zmq_ctx_destroy(context);
                return 0;
            }
        }
        CHECK_ERROR(zmq_msg_close(&msg));
    }
}
```

Демонстрация работы программы

```
liguha@Laptop:~/OS/LR6-8/build$ ./client
create 20
Ok: 2565
create 10
Ok: 2583
create 30
Ok: 2593
exec 20 1 2
Ok:20:2
exec 30 2 1 3
Ok:30:4
exec 10 3 0 0 7
Ok:10:7
pingall
Ok: -1
remove 10
Ok
exec 10 1 1
Error:10: Not found
exec 20 1 1
Ok:20:1
remove 20
```

```
Ok
exec 30 1 1
Error:30: Not found
exec 20 1 1
Error:20: Not found
create 20
Ok: 2759
create 10
Ok: 2767
create 30
Ok: 2775
pingall
Ok: 30 (процесс 2775 был предварительно убит)
pingall
Ok: 10;30 (дополнительно убит процесс 2767 для демонстрации pingall)
remove 20
Ok
pingall
Ok: -1
remove 20
Error: Not found
create 20
Ok: 2929
pingall
Ok: 20 (заранее был убит процесс 2929 для демонстрации ошибки при попытке exec)
exec 20 1 1
Error:20: Node is unavailable
```

Выводы

Составлена и отлажена программа на языке C++, осуществляющая отложенные вычисления на нескольких вычислительных узлах. Пользователь управляет программой через распределительный узел, который перенаправляет запросы в асинхронном режиме.