Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №5-7 по курсу

«Операционные системы»

Группа: М8О-209Б-23

Студент: Калинин И.Н.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 18.12.24

Москва, 2024

Постановка задачи

Вариант 15.

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. 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.

Примечание: выполнение команд должно быть асинхронным. Т.е. пока выполняется команда на одном из вычислительных узлов, то можно отправить следующую команду на другой вычислительный узел.

Топология: узлы находятся в дереве общего вида.

Команда: локальный целочисленный словарь

Формат команды сохранения значения: exec id name value

id – целочисленный идентификатор вычислительного узла, на который отправляется команда

name – ключ, по которому будет сохранено значение (строка формата [A-Za-z0-9]+)

value – целочисленное значение

Формат команды загрузки значения: exec id name.

Проверка доступности: Формат команды: ping id

Команда проверяет доступность конкретного узла. Если узла нет, то необходимо выводить ошибку.

Общий метод и алгоритм решения

Для реализации системы очереди сообщений используем библиотеку ZeroMQ.

Использованные системные вызовы:

1. int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout); Ожидает готовности файловых дескрипторов.
2. pid_t fork(void); Создает новый процесс.
3. int execl(const char *path, const char *arg, ...); Заменяет текущий процесс новым процессом.
4. pid_t getpid(void); Возвращает идентификатор текущего процесса.
5. void zmq_msg_init_size(zmq_msg_t *msg, size_t size); Инициализирует сообщение ZeroMQ с указанным размером.
6. int zmq_msg_send(zmq_msg_t *msg, void *socket, int flags); Отправляет сообщение через сокет ZeroMQ.
7. void zmq_msg_init(zmq_msg_t *msg); Инициализирует сообщение ZeroMQ.
8. int zmq_msg_recv(zmq_msg_t *msg, void *socket, int flags); Получает сообщение через сокет ZeroMQ.
9. void *zmq_msg_data(zmq_msg_t *msg); Возвращает указатель на данные сообщения ZeroMQ.
10. void *zmq_ctx_new(void); Создает новый контекст ZeroMQ.
11. void *zmq_socket(void *context, int type); Создает новый сокет ZeroMQ.
12. int zmq_connect(void *socket, const char *addr); Подключает сокет ZeroMQ к указанному адресу.

13. int zmq_bind(void *socket, const char *addr); Привязывает сокет ZeroMQ к указанному адресу.

Для реализации потребуется написать два исполняемых файлов. Управляющий узел и вычислительный. Для удобства напишем отдельные функции отправки, принятия сообщения, создания дочернего процесса.

Очередь сообщений будем реализовывать с помощью библиотеки ZeroMQ. Сообщения будем передавать через сокеты. Контекст в ZeroMQ — это основной объект, который управляет всеми ресурсами, необходимыми для работы с сокетами. Контекст отвечает за управление потоками, соединениями и другими низкоуровневыми деталями. Сокеты в ZeroMQ — это абстракции, которые представляют конечные точки для отправки и получения сообщений. ZeroMQ предоставляет несколько типов сокетов, каждый из которых имеет свою семантику и предназначен для различных шаблонов взаимодействия. Мы будем использовать DEALER, так как он позволяет создать двустороннюю очередь сообщений в которую сможет писать и родитель и ребенок. Для того, чтобы система работала асинхронно необходимо использовать специальные флаги ZMQ_DONTWAIT. Этот флаг означает отправить или принять сообщения без ответа. Т.е. по принципу «отправил и забыл» может относиться к асинхронному обмену сообщениями. В этом случае отправитель не обязан ожидать подтверждения о получении и обработке сообщения от принимающей стороны.

Для того, чтобы стандартизировать данные, которые будет принимать и отправлять в очередь процессы создадим класс message, который будет хранить данные, которые мы будем отправлять.

Класс message будет хранить команду(None, Create, Ping, ExecAdd, ExecFnd, ExecErr) айди кому отправляем, числовая часть данных, строковую часть данных и время отправки сообщения, для определения недошедших сообщений.

Опишем функцию createNode, она создаёт новый процесс функцией fork и замещает память программой вычислительного узла после команды execl, так же ключом запуска передаётся id нового узла.

Опишем функции отрывки и принятия сообщения. Каждая нода ребенка будет хранить собственный адрес, контекс, сокет, айди, пид. Соответственно функция отправки будет брать сообщение и зная длину в байтах посылать не дожидаясь ответа в очередь сообщений по своему сокету. Аналогично функция приёмки сообщений в случае получения байтов, будет получать сообщение и возвращать его в основную программу. Иначе возвращать сообщение с типом None.

Для того, чтобы отправить команду, управляющему узлу придется отправить команду всем своим детям и ждать ответа от хоть кого-нибудь об успешном выполнении. Каждый следящий узел будет сравнивать id команды со своим id и если он не совпадает отправлять всем своим детям. Если id совпадают, то узел будет выполнять команду в зависимости от её значения, установленного условием задачи.

Для реализации проверки доступности узла у управляющего узла введем список отправленных сообщений и каждый такт цикла будем проверять все сообщения и сравнивать их время отправки с текущем временем. Если оно отличается больше чем на дельту, то необходимо вывести сообщение о недоступности узла. При успешном выполнении команды, то удаляем сообщения из списка оправленных.

Для реализации топологии дерева общего вида будем использовать его определение, в нем сказано, что дочерние вершины хранятся у каждого родителя в очереди вершин. Это удобно в случае, когда нам нужно разово пройтись по дочерним вершинам. В нашем случае эффективно по памяти использовать std::list, аналогично для всех коллекций, которые будем использовать в данной лабораторной работе.

Для словаря будем использовать std::map где ключом будет std::string, а значением int. Для проверки наличия ключа будем использовать std::find который будет проверять, есть ли ключ в словаре.

Для меньшего потребления ресурсов процессами можно после каждого такта добавить sleep на значение 100ms, так как в данном контексте нет необходимости, чтобы программа работала со скоростью миллионы итераций в секунду.

Код программы

lib.h

```
#include <iostream>
#include <list>
#include <unordered_set>
#include <chrono>
#include <ctime>
#include <string>
#include <cstring>
#include <unistd.h>
#include <sys/wait.h>
#include "zmq.h"
#include <sys/select.h>
#include <map>

bool inputAvailable();

std::time_t t_now();

enum com : char {
    None = 0,
    Create = 1,
    Ping = 2,
    ExecAdd = 3,
    ExecFnd = 4,
    ExecErr = 5
};

class message {
public:
    message() {}
    message(com command, int id, int num) : command(command), id(id), num(num),
sent_time(t_now()) {}
    message(com command, int id, int num, char s[]) : command(command), id(id), num(num),
sent_time(t_now()) {
        for (int i = 0; i < 30; ++i) {
            st[i] = s[i];
        }
    }

    bool operator==(const message &other) const {
        return command == other.command && id == other.id && num == other.num && sent_time
== other.sent_time;
    }
```

```
    com command;           // команда
    int id;                // айди
    int num;               // данные
    std::time_t sent_time; // время отправки
    char st[30];           // строка
};

class Node {
public:
    int id;
    pid_t pid;
    void *context;
    void *socket;
    std::string address;

    bool operator==(const Node &other) const {
        return id == other.id && pid == other.pid;
    }
};

Node createNode(int id, bool is_child);

Node createProcess(int id);

void send_mes(Node &node, message m);

message get_mes(Node &node);
```

lib.cpp

```
#include <lib.h>

bool inputAvailable() {
    struct timeval tv;
    fd_set fds;
    tv.tv_sec = 0;
    tv.tv_usec = 0;
    FD_ZERO(&fds);
    FD_SET(STDIN_FILENO, &fds);
    select(STDIN_FILENO + 1, &fds, NULL, NULL, &tv);
    return (FD_ISSET(STDIN_FILENO, &fds));
}

std::time_t t_now() {
    return std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
}

Node createNode(int id, bool is_child) {
    Node node;
    node.id = id;
    node.pid = getpid();
    node.context = zmq_ctx_new();
    node.socket = zmq_socket(node.context, ZMQ_DEALER);
```

```
    node.address = "tcp://127.0.0.1:" + std::to_string(5555 + id);

    if (is_child)
        zmq_connect(node.socket, (node.address).c_str());
    else
        zmq_bind(node.socket, (node.address).c_str());
    return node;
}

Node createProcess(int id) {
    pid_t pid = fork();
    if (pid == 0) {
        execl("./computing", "computing", std::to_string(id).c_str(), NULL);
        std::cerr << "execl failed" << std::endl;
        exit(1);
    }
    if (pid == -1) {
        std::cerr << "Fork failed" << std::endl;
        exit(1);
    }
    Node node = createNode(id, false);
    node.pid = pid;
    return node;
}

void send_mes(Node &node, message m) {
    zmq_msg_t request_message;
    zmq_msg_init_size(&request_message, sizeof(m));
    std::memcpy(zmq_msg_data(&request_message), &m, sizeof(m));
    zmq_msg_send(&request_message, node.socket, ZMQ_DONTWAIT);
}

message get_mes(Node &node) {
    zmq_msg_t request;
    zmq_msg_init(&request);
    auto result = zmq_msg_recv(&request, node.socket, ZMQ_DONTWAIT);
    if (result == -1) {
        return message(None, -1, -1);
    }
    message m;
    std::memcpy(&m, zmq_msg_data(&request), sizeof(message));
    return m;
}
```

control.cpp

```
#include <./lib.h>

int main() {
    std::unordered_set<int> all_id;
    all_id.insert(-1);
    std::list<message> saved_mes;
    std::list<Node> children;
```

```
    std::string command;
    while (true) {
        for (auto &i : children) {
            message m = get_mes(i);
            switch (m.command) {
            case Create:
                all_id.insert(m.id);
                std::cout << "Ok: " << m.num << std::endl;
                for (auto it = saved_mes.begin(); it != saved_mes.end(); ++it) {
                    if (it->command == Create and it->num == m.id) {
                        saved_mes.erase(it);
                        break;
                    }
                }
                break;
            case Ping:
                std::cout << "Ok: " << m.id << " is available" << std::endl;
                for (auto it = saved_mes.begin(); it != saved_mes.end(); ++it) {
                    if (it->command == Ping and it->id == m.id) {
                        saved_mes.erase(it);
                        break;
                    }
                }
                break;
            case ExecErr:
                std::cout << "Ok: " << m.id << " '" << m.st << "'not fount" << std::endl;
                for (auto it = saved_mes.begin(); it != saved_mes.end(); ++it) {
                    if (it->command == ExecFnd and it->id == m.id) {
                        saved_mes.erase(it);
                        break;
                    }
                }
                break;
            case ExecAdd:
                std::cout << "Ok: " << m.id << std::endl;
                for (auto it = saved_mes.begin(); it != saved_mes.end(); ++it) {
                    if (it->command == ExecAdd and it->id == m.id) {
                        saved_mes.erase(it);
                        break;
                    }
                }
                break;
            case ExecFnd:
                std::cout << "Ok: " << m.id << " '" << m.st << "' " << m.num << std::endl;
                for (auto it = saved_mes.begin(); it != saved_mes.end(); ++it) {
                    if (it->command == ExecFnd and it->id == m.id) {
                        saved_mes.erase(it);
                        break;
                    }
                }
                break;
            default:
                continue;
            }
```

```
        }
        for (auto it = saved_mes.begin(); it != saved_mes.end(); ++it) {
            if (std::difftime(t_now(), it->sent_time) > 5) {
                switch (it->command) {
                case Ping:
                    std::cout << "Error: Ok " << it->id << " is unavailable" << std::endl;
                    break;
                case Create:
                    std::cout << "Error: Parent  " << it->id << " is unavailable" <<
std::endl;
                    break;
                case ExecAdd:
                case ExecFnd:
                    std::cout << "Error: Node  " << it->id << " is unavailable" <<
std::endl;
                    break;
                default:
                    break;
                }
                saved_mes.erase(it);
                break;
            }
        }

        if (!inputAvailable()) {
            continue;
        }
        std::cin >> command;
        if (command == "create") {
            int parent_id, child_id;
            std::cin >> child_id >> parent_id;
            if (all_id.count(child_id)) {
                std::cout << "Error: Node with id " << child_id << " already exists" <<
std::endl;
            }
            else if (!all_id.count(parent_id)) {
                std::cout << "Error: Parent with id " << parent_id << " not found" <<
std::endl;
            }
            else if (parent_id == -1) {
                Node child = createProcess(child_id);
                children.push_back(child);
                all_id.insert(child_id);
                std::cout << "Ok: " << child.pid << std::endl;
            }
            else {
                message m(Create, parent_id, child_id);
                saved_mes.push_back(m);
                for (auto &i : children)
                    send_mes(i, m);
            }
        }
        else if (command == "exec") {
            char input[100];
```

```
            fgets(input, sizeof(input), stdin);

            int id, val;
            char key[30];

            if (sscanf(input, "%d %30s %d", &id, key, &val) == 3) {
                if (!all_id.count(id)) {
                    std::cout << "Error: Node with id " << id << " doesn't exist" <<
std::endl;
                    continue;
                }
                message m = {ExecAdd, id, val, key};
                saved_mes.push_back(m);
                for (auto &i : children) {
                    send_mes(i, m);
                }
            }
            else if (sscanf(input, "%d %30s", &id, key) == 2) {
                if (!all_id.count(id)) {
                    std::cout << "Error: Node with id " << id << " doesn't exist" <<
std::endl;
                    continue;
                }
                message m = {ExecFnd, id, -1, key};
                saved_mes.push_back(m);
                for (auto &i : children) {
                    send_mes(i, m);
                }
            }
        }
        else if (command == "ping") {
            int id;
            std::cin >> id;
            if (!all_id.count(id)) {
                std::cout << "Error: Node with id " << id << " doesn't exist" <<
std::endl;
            }
            else {
                message m(Ping, id, 0);
                saved_mes.push_back(m);
                for (auto &i : children) {
                    send_mes(i, m);
                }
            }
        }
        else if (command == "exit"){
            break;
        }
        else
            std::cout << "Error: Command doesn't exist!" << std::endl;
        usleep(100000);
    }
    return 0;
}
```

computing.cpp

```cpp
#include <lib.h>

int main(int argc, char *argv[]) {
    Node I = createNode(atoi(argv[1]), true);
    std::map<std::string, int> dict;

    std::list<Node> children;
    while (true) {
        for (auto &i : children) {
            message m = get_mes(i);
            if (m.command != None) {
                send_mes(I, m);
            }
        }

        message m = get_mes(I);
        switch (m.command) {
        case Create:
            if (m.id == I.id) {
                Node child = createProcess(m.num);
                children.push_back(child);
                send_mes(I, {Create, child.id, child.pid});
            }
            else {
                for (auto &i : children) {
                    send_mes(i, m);
                }
            }
            break;
        case Ping:
            if (m.id == I.id) {
                send_mes(I, m);
            }
            else {
                for (auto &i : children) {
                    send_mes(i, m);
                }
            }
            break;
        case ExecAdd:
            if (m.id == I.id) {
                dict[std::string(m.st)] = m.num;
                send_mes(I, m);
            }
            else {
                for (auto &i : children) {
                    send_mes(i, m);
                }
            }
            break;
        case ExecFnd:
            if (m.id == I.id) {
```

```
                if (dict.find(std::string(m.st)) != dict.end()) {
                    send_mes(I, {ExecFnd, I.id, dict[std::string(m.st)], m.st});
                }
                else {
                    send_mes(I, {ExecErr, I.id, -1, m.st});
                }
            }
            else {
                for (auto &i : children) {
                    send_mes(i, m);
                }
            }
            break;
        default:
            break;
        }
        usleep(100000);
    }
    return 0;
}
```

Протокол работы программы

Тестирование:

Тест 1.

create 1 -1
Ok: 19478
create 2 1
Ok: 19512
exec 2 malina 3
Ok: 2
exec 2 malina
Ok: 2 'malina' 3
ping 2
Ok: 2 is available
pig 1
Error: Command doesn't exist!
ping 1
Error: Command doesn't exist!
Ok: 1 is available

Тест 2.

create 1 -1
Ok: 22497
create 21 1
Ok: 22565
create 12 1
Ok: 22616
exec 21 kol 2
Ok: 21
exec 21 kol

Ok: 21 'kol' 2
ping 21
Ok: 21 is available
ping 12
Ok: 12 is available

Вывод

Очень понравилось выполнять данную лабораторную работу. Было крайне интересно изучить очереди сообщений и потратить большое число времени на отладку отправки и приему сообщений через неё. Очередь сообщений является эффективным инструментом для межпроцессорного взаимодействия, так как позволяет легко асинхронно обмениваться сообщениями и масштабировать систему, а так же синхронизировать процессы, выполняющие действия с разными скоростями, что очень важно в клиент-серверной архитектуре. ZeroMQ является удобной и эффективной библиотекой для создания пользовательских очередей сообщений.