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: Гордовой Денис Сергеевич

Группа: М8О–201Б–19

Вариант: 9

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2021

**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.

Вариант 9: топология – 1, команда для вычислительных узлов - exec id (start, stop, time)

**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:

1. **kill** – убивает процесс с pid – первый аргумент и посылает сигнал – второй аргумент.
2. **socket.setsockopt** – устанавливает флаги для сокета.
3. **zmq::context\_t –** создает ZMQ контекст.
4. **zmq::socket\_t –** создает ZMQ сокет.
5. **zmq::message\_t** – создает ZMQ сообщение.
6. **socket.send** – отправляет ZMQ сообщение на socket.
7. **socket.bind** – принимает соединие к сокету.
8. **execv –** выполняет указанный файл.
9. **fork** – создает копию процесса.

**Общий метод и алгоритм решения**.

Создаем сервер – исполняющий узел, дальше делаем fork, в дочернем процессе при помощи execv запускаем client, а с родителя с сервера отсылаем сообщение, внутри клиента также создаются сокеты – левый и правый и на них отправляются сообщения с родителя, а родитель получает сообщения от детей и так по всему дереву. Исполняющий узел получает сообщение выполняет команду и отправляет ответ вверх по дереву до управляющего узла.

**Основные файлы программы**

**control.cpp:**

**#include <unistd.h>**

**#include <sstream>**

**#include <set>**

**#include "zmq\_functions.h"**

**#include "topology.h"**

**int main() {**

**topology network;**

**std::vector<zmq::socket\_t> branches;**

**zmq::context\_t context;**

**std::string cmd;**

**while (std::cin >> cmd) {**

**if (cmd == "create") {**

**int node\_id, parent\_id;**

**std::cin >> node\_id >> parent\_id;**

**if (network.find(node\_id) != -1) {**

**std::cout << "Error: already exists" << std::endl;**

**}**

**else if (parent\_id == -1) {**

**pid\_t pid = fork();**

**if (pid < 0) {**

**perror("Can't create new process");**

**return -1;**

**}**

**if (pid == 0) {**

**execl("./counting", "./counting", std::to\_string(node\_id).c\_str(), NULL);**

**}**

**branches.emplace\_back(context, ZMQ\_REQ);**

**branches[branches.size() - 1].setsockopt(ZMQ\_SNDTIMEO, 5000);**

**bind(branches[branches.size() - 1], node\_id);**

**send\_message(branches[branches.size() - 1], std::to\_string(node\_id) + "pid");**

**std::string reply = receive\_message(branches[branches.size() - 1]);**

**std::cout << reply << std::endl;**

**network.insert(node\_id, parent\_id);**

**}**

**else if (network.find(parent\_id) == -1) {**

**std::cout << "Error: parent not found" << std::endl;**

**}**

**else {**

**int branch = network.find(parent\_id);**

**send\_message(branches[branch], std::to\_string(parent\_id) + "create " + std::to\_string(node\_id));**

**std::string reply = receive\_message(branches[branch]);**

**std::cout << reply << std::endl;**

**network.insert(node\_id, parent\_id);**

**}**

**}**

**else if (cmd == "exec") {**

**int dest\_id;**

**std::string cmd;**

**std::cin >> dest\_id;**

**std::getline(std::cin, cmd);**

**int branch = network.find(dest\_id);**

**if (branch == -1) {**

**std::cout << "ERROR: incorrect node id" << std::endl;**

**}**

**else {**

**send\_message(branches[branch], std::to\_string(dest\_id) + "exec " + cmd);**

**std::string reply = receive\_message(branches[branch]);**

**std::cout << reply << std::endl;**

**}**

**}**

**else if (cmd == "kill") {**

**int id;**

**std::cin >> id;**

**int branch = network.find(id);**

**if (branch == -1) {**

**std::cout << "ERROR: incorrect node id" << std::endl;**

**}**

**else {**

**bool is\_first = (network.get\_first\_id(branch) == id);**

**send\_message(branches[branch], std::to\_string(id) + " kill");**

**std::string reply = receive\_message(branches[branch]);**

**std::cout << reply << std::endl;**

**network.erase(id);**

**if (is\_first) {**

**unbind(branches[branch], id);**

**branches.erase(branches.begin() + branch);**

**}**

**}**

**}**

**else if (cmd == "exit") {**

**for (size\_t i = 0; i < branches.size(); ++i) {**

**int first\_node\_id = network.get\_first\_id(i);**

**send\_message(branches[i], std::to\_string(first\_node\_id) + " kill");**

**std::string reply = receive\_message(branches[i]);**

**if (reply != "OK") {**

**std::cout << reply << std::endl;**

**}**

**else {**

**unbind(branches[i], first\_node\_id);**

**}**

**}**

**exit(0);**

**}**

**else {**

**std::cout << "Incorrect cmd" << std::endl;**

**}**

**}**

**}**

**counting.cpp:**

**#include <unordered\_map>**

**#include <unistd.h>**

**#include <sstream>**

**#include <unordered\_map>**

**#include "zmq\_functions.h"**

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

**if (argc != 2 && argc != 3) {**

**throw std::runtime\_error("Wrong args for counting node");**

**}**

**int cur\_id = std::atoi(argv[1]);**

**int child\_id = -1;**

**if (argc == 3) {**

**child\_id = std::atoi(argv[2]);**

**}**

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

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

**bool flag\_clock = false;**

**double time\_clock;**

**std::unordered\_map<std::string, int> dictionary;**

**zmq::context\_t context;**

**zmq::socket\_t parent\_socket(context, ZMQ\_REP);**

**connect(parent\_socket, cur\_id);**

**zmq::socket\_t child\_socket(context, ZMQ\_REQ);**

**child\_socket.setsockopt(ZMQ\_SNDTIMEO, 5000);**

**if (child\_id != -1) {**

**bind(child\_socket, child\_id);**

**}**

**std::string message;**

**while (true) {**

**message = receive\_message(parent\_socket);**

**std::istringstream request(message);**

**int dest\_id;**

**request >> dest\_id;**

**std::string cmd;**

**request >> cmd;**

**if (dest\_id == cur\_id) {**

**if (cmd == "pid") {**

**send\_message(parent\_socket, "OK: " + std::to\_string(getpid()));**

**}**

**else if (cmd == "create") {**

**int new\_child\_id;**

**request >> new\_child\_id;**

**if (child\_id != -1) {**

**unbind(child\_socket, child\_id);**

**}**

**bind(child\_socket, new\_child\_id);**

**pid\_t pid = fork();**

**if (pid < 0) {**

**perror("Can't create new process");**

**return -1;**

**}**

**if (pid == 0) {**

**execl("./counting", "./counting", std::to\_string(new\_child\_id).c\_str(), std::to\_string(child\_id).c\_str(), NULL);**

**perror("Can't execute new process");**

**return -2;**

**}**

**send\_message(child\_socket, std::to\_string(new\_child\_id) + "pid");**

**child\_id = new\_child\_id;**

**send\_message(parent\_socket, receive\_message(child\_socket));**

**}**

**else if (cmd == "exec") {**

**std::string result;**

**std::string cmd;**

**request >> cmd;**

**if (cmd == "start") {**

**start\_clock = std::chrono::high\_resolution\_clock::now();**

**flag\_clock = true;**

**}**

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

**if (flag\_clock) {**

**stop\_clock = std::chrono::high\_resolution\_clock::now();**

**time\_clock += std::chrono::duration\_cast<std::chrono::milliseconds>(stop\_clock - start\_clock).count();**

**flag\_clock = false;**

**}**

**}**

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

**result = std::to\_string(time\_clock);**

**}**

**send\_message(parent\_socket, "OK: " + std::to\_string(cur\_id) + ": " + result);**

**}**

**else if (cmd == "kill") {**

**if (child\_id != -1) {**

**send\_message(child\_socket, std::to\_string(child\_id) + " kill");**

**std::string msg = receive\_message(child\_socket);**

**if (msg == "OK") {**

**send\_message(parent\_socket, "OK");**

**}**

**unbind(child\_socket, child\_id);**

**disconnect(parent\_socket, cur\_id);**

**break;**

**}**

**send\_message(parent\_socket, "OK");**

**disconnect(parent\_socket, cur\_id);**

**break;**

**}**

**}**

**else if (child\_id != -1) {**

**send\_message(child\_socket, message);**

**send\_message(parent\_socket, receive\_message(child\_socket));**

**if (child\_id == dest\_id && cmd == "kill") {**

**child\_id = -1;**

**}**

**}**

**else {**

**send\_message(parent\_socket, "Error: node is unavailable");**

**}**

**}**

**}**

**topology.h:**

**#include <list>**

**#include <stdexcept>**

**class topology {**

**private:**

**std::list<std::list<int>> container;**

**public:**

**void insert(int id, int parent\_id) {**

**if (parent\_id == -1) {**

**std::list<int> new\_list;**

**new\_list.push\_back(id);**

**container.push\_back(new\_list);**

**}**

**else {**

**int list\_id = find(parent\_id);**

**if (list\_id == -1) {**

**throw std::runtime\_error("Wrong parent id");**

**}**

**auto it1 = container.begin();**

**std::advance(it1, list\_id);**

**for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {**

**if (\*it2 == parent\_id) {**

**it1->insert(++it2, id);**

**return;**

**}**

**}**

**}**

**}**

**int find(int id) {**

**int cur\_list\_id = 0;**

**for (auto it1 = container.begin(); it1 != container.end(); ++it1) {**

**for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {**

**if (\*it2 == id) {**

**return cur\_list\_id;**

**}**

**}**

**++cur\_list\_id;**

**}**

**return -1;**

**}**

**void erase(int id) {**

**int list\_id = find(id);**

**if (list\_id == -1) {**

**throw std::runtime\_error("Wrong id");**

**}**

**auto it1 = container.begin();**

**std::advance(it1, list\_id);**

**for (auto it2 = it1->begin(); it2 != it1->end(); ++it2) {**

**if (\*it2 == id) {**

**it1->erase(it2, it1->end());**

**if (it1->empty()) {**

**container.erase(it1);**

**}**

**return;**

**}**

**}**

**}**

**int get\_first\_id(int list\_id) {**

**auto it1 = container.begin();**

**std::advance(it1, list\_id);**

**if (it1->begin() == it1->end()) {**

**return -1;**

**}**

**return \*(it1->begin());**

**}**

**};**

**zmq\_functions.h:**

**#include <zmq.hpp>**

**#include <iostream>**

**const int MAIN\_PORT = 4040;**

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

**//std::cout << "Start sending " << msg << std::endl;**

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

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

**socket.send(message);**

**//std::cout << "Finished sending " << msg << std::endl;**

**}**

**std::string receive\_message(zmq::socket\_t& socket) {**

**//std::cout << "Start receiving" << std::endl;**

**zmq::message\_t message;**

**bool chars\_read;**

**try {**

**chars\_read = socket.recv(&message);**

**}**

**catch (...) {**

**chars\_read = false;**

**}**

**if (chars\_read == 0) {**

**return "Error: node is unavailable [zmq\_func]";**

**}**

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

**//std::cout << "Received " << received\_msg << std::endl;**

**return received\_msg;**

**}**

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

**std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);**

**socket.connect(address);**

**//std::cout << "Socket connected to " << address << std::endl;**

**}**

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

**std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);**

**socket.disconnect(address);**

**//std::cout << "Socket disconnected from " << address << std::endl;**

**}**

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

**std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);**

**socket.bind(address);**

**//std::cout << "Socket binded to " << address << std::endl;**

**}**

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

**std::string address = "tcp://127.0.0.1:" + std::to\_string(MAIN\_PORT + id);**

**socket.unbind(address);**

**//std::cout << "Socket unbinded from " << address << std::endl;**

**}**

**Пример работы**

**MacBook:src vladislove$ ./control**

**create 1 -1**

**OK: 3344**

**create 2 1**

**OK: 3345**

**create 3 2**

**OK: 3346**

**create 4 3**

**OK: 3347**

**heartbeat**

**OK: 1 2 3 4**

**kill 3**

**OK**

**heartbeat**

**OK: 1 2**

**exec 2 start**

**OK: 2:**

**create 10 -1**

**OK: 3353**

**heartbeat**

**OK: 1 2 10**

**xec 2 stop**

**OK: 2:**

**exec 2 time**

**OK: 2: 23212.0000**

**exit**

**Вывод**

Данная лабораторная работа была направлена на изучении технологии очереди сообщений, на основе которой необходимо было построить сеть с заданной топологией.

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