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

**Лабораторная работа №6 по курсу**
**«Операционные системы»**

# Управление серверами сообщений

Студент: Косогоров Владислав Валерьевич
Группа: М80 – 206Б-18
Вариант: 31
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2019

## Постановка задачи

Реализовать распределенную систему по обработке запросов. В данной 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 сервера сообщений zmq. Также в данной системе необходимо предусмотреть проверку доступности узлов в соответствии с вариантом.

**Вариант задания:** 31. Топология — бинарное дерево. Тип вычислительной команды — локальный целочисленный словарь. Тип проверки узлов на доступность — pingall.

## Общие сведения о программе

Программа состоит из двух файлов, которые компилируются в исполнительные файлы(которые представляют управляющий и вычислительные узлы), а так же из статической библиотеки, которая подключается к вышеуказанным файлам. Общение между процессами происходит с помощью библиотеки zmq.

## Общий метод и алгоритм решения

- Управляющий узел принимает команды, обрабатывает их и пересылает дочерним узлам или выводит сообщение об ошибке.
- Дочерние узлы проверяют, может ли быть команда выполнена в данном узле, если нет, то команда пересылается в один из дочерних узлов, из которого возвращается некоторое сообщение об успехе или об ошибке, которое потом пересылается обратно по дереву.
- Для корректной проверки на доступность узлов, используется дерево, эмулирующее поведение узлов в данной топологии (например, при удалении узла, удаляются все его потомки).
- Если узел недоступен, то по истечении времени будет выведено сообщение о недоступности узла и оно будет передано управляющему узлу.
- При удалении узла, все его потомки рекурсивно уничтожаются.

## Код программы

**server_functions.h**

```
#pragma once
#include <string>
#include "unistd.h"
#include "zmq.hpp"

bool send_message(zmq::socket_t& socket, const std::string& message_string);
std::string recieve_message(zmq::socket_t& socket);
std::string get_port_name(int port);
int bind_socket(zmq::socket_t& socket);
void create_node(int id, int port);
```

**server_functions.cpp**

```
#include "server_functions.h"

bool send_message(zmq::socket_t& socket, const std::string& message_string) {
    zmq::message_t message(message_string.size());
    memcpy(message.data(), message_string.c_str(), message_string.size());
    return socket.send(message);
}

std::string recieve_message(zmq::socket_t& socket) {
    zmq::message_t message;
    bool ok;
    try {
        ok = socket.recv(&message);
    } catch (...) {
```

```
        ok = false;
    }
    std::string recieved_message(static_cast<char*>(message.data()),
message.size());
    if (recieved_message.empty() || !ok) {
        return "Error: Node is not available";
    }
    return recieved_message;
}

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

int bind_socket(zmq::socket_t& socket) {
    int port = 30000;
    while (true) {
        try {
            socket.bind(get_port_name(port));
            break;
        } catch(...) {
            port++;
        }
    }
    return port;
}

void create_node(int id, int port) {
    char* arg1 = strdup((std::to_string(id)).c_str());
```

```
    char* arg2 = strdup((std::to_string(port)).c_str());
    char* args[] = {"./child_node", arg1, arg2, NULL};
    execv("./child_node", args);
}
```

**child_node.cpp**

```
#include <iostream>
#include "zmq.hpp"
#include <string>
#include <sstream>
#include <exception>
#include <signal.h>
#include <unordered_map>
#include <iterator>
#include "server_functions.h"

int main(int argc, char** argv) { //аргументы - айди и номер порта, к которому нужно подключиться
    int id = std::stoi(argv[1]);
    int parent_port = std::stoi(argv[2]);
    zmq::context_t context(3);
    zmq::socket_t parent_socket(context, ZMQ_REP);

    parent_socket.connect(get_port_name(parent_port));

    int left_pid = 0;
    int right_pid = 0;
    int left_id = 0;
    int right_id = 0;
```

```
zmq::socket_t left_socket(context, ZMQ_REQ);
zmq::socket_t right_socket(context, ZMQ_REQ);
int linger = 0;
left_socket.setsockopt(ZMQ_SNDTIMEO, 2000);
left_socket.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
right_socket.setsockopt(ZMQ_SNDTIMEO, 2000);
right_socket.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));

int left_port = bind_socket(left_socket);
int right_port = bind_socket(right_socket);

while (true) {
    std::string request_string;
    request_string = recieve_message(parent_socket);
    std::istringstream command_stream(request_string);
    std::string command;
    command_stream >> command;

    if (command == "id") {
        std::string parent_string = "Ok:" + std::to_string(id);
        send_message(parent_socket, parent_string);
    } else if (command == "pid") {
        std::string parent_string = "Ok:" + std::to_string(getpid());
        send_message(parent_socket, parent_string);
    } else if (command == "create") {
        int id_to_create;
        command_stream >> id_to_create;

        if (id_to_create == id) {
```

```
    std::string message_string = "Error: Already exists";
    send_message(parent_socket, message_string);
  } else if (id_to_create < id) {
    if (left_pid == 0) {
      left_pid = fork();
      if (left_pid == -1) {
        send_message(parent_socket, "Error: Cannot fork");
        left_pid = 0;
      } else if (left_pid == 0) {
        create_node(id_to_create,left_port);
      } else {
        left_id = id_to_create;
        send_message(left_socket, "pid");
        send_message(parent_socket, recieve_message(left_socket));
      }
    } else {
      send_message(left_socket, request_string);
      send_message(parent_socket, recieve_message(left_socket));
    }
  } else {
    if (right_pid == 0) {
      right_pid = fork();
      if (right_pid == -1) {
        send_message(parent_socket, "Error: Cannot fork");
        right_pid = 0;
      } else if (right_pid == 0) {
        create_node(id_to_create,right_port);
      } else {
```

```
                right_id = id_to_create;
                send_message(right_socket, "pid");
                send_message(parent_socket, recieve_message(right_socket));
            }
        } else {
            send_message(right_socket, request_string);
            send_message(parent_socket, recieve_message(right_socket));
        }
    }

} else if (command == "remove") {
    int id_to_delete;
    command_stream >> id_to_delete;
    if (id_to_delete < id) {
        if (left_id == 0) {
            send_message(parent_socket, "Error: Not found");
        } else if (left_id == id_to_delete) {
            send_message(left_socket, "kill_children");
            recieve_message(left_socket);
            kill(left_pid,SIGTERM);
            kill(left_pid,SIGKILL);
            left_id = 0;
            left_pid = 0;
            send_message(parent_socket, "Ok");
        } else {
            send_message(left_socket, request_string);
            send_message(parent_socket, recieve_message(left_socket));
        }
    } else {
```

```
            if (right_id == 0) {
                send_message(parent_socket, "Error: Not found");
            } else if (right_id == id_to_delete) {
                send_message(right_socket, "kill_children");
                recieve_message(right_socket);
                kill(right_pid,SIGTERM);
                kill(right_pid,SIGKILL);
                right_id = 0;
                right_pid = 0;
                send_message(parent_socket, "Ok");
            } else {
                send_message(right_socket, request_string);
                send_message(parent_socket, recieve_message(right_socket));
            }
        }

    } else if (command == "exec") {
        int exec_id;
        command_stream >> exec_id;
        if (exec_id == id) {
            std::string recieve_message = "Node is available";
            send_message(parent_socket, recieve_message);
        } else if (exec_id < id) {
                if (left_pid == 0) {
                    std::string recieve_message = "Error:" + std::to_string(exec_id) +
": Not found";
                    send_message(parent_socket, recieve_message);
                } else {
                    send_message(left_socket, request_string);
```

```
                    send_message(parent_socket, recieve_message(left_socket));
                }
            } else {
                if (right_pid == 0) {
                    std::string recieve_message = "Error:" + std::to_string(exec_id) + ": Not found";
                    send_message(parent_socket, recieve_message);
                } else {
                    send_message(right_socket, request_string);
                    send_message(parent_socket, recieve_message(right_socket));
                }
            }

        } else if (command == "pingall") {
            std::ostringstream res;
            std::string left_res;
            std::string right_res;
            if (left_pid != 0) {
                send_message(left_socket, "pingall");
                left_res = recieve_message(left_socket);
            }
            if (right_pid != 0) {
                send_message(right_socket, "pingall");
                right_res = recieve_message(right_socket);
            }
            if (!left_res.empty() && left_res.substr(std::min<int>(left_res.size(),5)) != "Error") {
                res << left_res;
            }
```

```
            if (!right_res.empty() && right_res.substr(std::min<int>(right_res.size(),5))
!= "Error") {
                res << right_res;
            }
            send_message(parent_socket, res.str());

        } else if (command == "kill_children") {
            if (left_pid == 0 && right_pid == 0) {
                send_message(parent_socket, "Ok");
            } else {
                if (left_pid != 0) {
                    send_message(left_socket, "kill_children");
                    recieve_message(left_socket);
                    kill(left_pid,SIGTERM);
                    kill(left_pid,SIGKILL);
                }
                if (right_pid != 0) {
                    send_message(right_socket, "kill_children");
                    recieve_message(right_socket);
                    kill(right_pid,SIGTERM);
                    kill(right_pid,SIGKILL);
                }
                send_message(parent_socket, "Ok");
            }
        }
        if (parent_port == 0) {
            break;
        }
    }
```

```
}
```

**main_node.cpp**

```
#include <iostream>
#include "zmq.hpp"
#include <string>
#include <vector>
#include <signal.h>
#include <sstream>
#include <set>
#include <algorithm>
#include <unordered_map>
#include "server_functions.h"

class IdTree {
public:
    IdTree() = default;

    ~IdTree() {
        delete_node(head_);
    }

    bool contains(int id) {
        TreeNode* temp = head_;
        while(temp != nullptr) {
            if (temp->id_ == id) {
                break;
            }
            if (id > temp->id_) {
                temp = temp->right;
```

```
        }
        if (id < temp->id_) {
            temp = temp->left;
        }
    }
    return temp != nullptr;
}

void insert(int id) {
    if (head_ == nullptr) {
        head_ = new TreeNode(id);
        return;
    }
    TreeNode* temp = head_;
    while(temp != nullptr) {
        if (id == temp->id_) {
            break;
        }
        if (id < temp->id_) {
            if (temp->left == nullptr) {
                temp->left = new TreeNode(id);
                break;
            }
            temp = temp->left;
        }
        if (id > temp->id_) {
            if (temp->right == nullptr) {
                temp->right = new TreeNode(id);
                break;
```

```
            }
            temp = temp->right;
        }
    }
}

void erase(int id) {
    TreeNode* prev_id = nullptr;
    TreeNode* temp = head_;
    while (temp != nullptr) {
        if (id == temp->id_) {
            if (prev_id == nullptr) {
                head_ = nullptr;
            } else {
                if (prev_id->left == temp) {
                    prev_id->left = nullptr;
                } else {
                    prev_id->right = nullptr;
                }
            }
            delete_node(temp);

        } else if (id < temp->id_) {
            prev_id = temp;
            temp = temp->left;
        } else if (id > temp->id_) {
            prev_id = temp;
            temp = temp->right;
        }
```

```
        }
    }

    void add_to_dictionary(int id, std::string name, int value) {
        TreeNode* neededNode = search(head_, id);
        neededNode->dictionary[name] = value;
    }

    void get_from_dictionary(int id, std::string name) {
        TreeNode* neededNode = search(head_, id);
        if (neededNode->dictionary.find(name) == neededNode->dictionary.end()) {
            std::cout << "'" << name << "' not found" << std::endl;
        } else {
            std::cout << neededNode->dictionary[name] << std::endl;
        }
    }

    std::vector<int> get_nodes() const {
        std::vector<int> result;
        get_nodes(head_, result);
        return result;
    }

private:
    struct TreeNode {
        TreeNode(int id) : id_(id) {}
        int id_;
        TreeNode* left = nullptr;
        TreeNode* right = nullptr;
```

```
    std::unordered_map<std::string, int> dictionary;
};

TreeNode* search(TreeNode* root, int id) {
    if (root == nullptr || root->id_ == id) {
        return root;
    }

    if (root->id_ < id) {
        return search(root->right, id);
    }

    return search(root->left, id);
}

void get_nodes(TreeNode* node, std::vector<int>& v) const {
    if (node == nullptr) {
        return;
    }
    get_nodes(node->left,v);
    v.push_back(node->id_);
    get_nodes(node->right, v);
}

void delete_node(TreeNode* node) {
    if (node == nullptr) {
        return;
    }
    delete_node(node->right);
```

```
        delete_node(node->left);
        delete node;
    }

    TreeNode* head_ = nullptr;
};

int main() {
    std::string command;
    IdTree ids;
    size_t child_pid = 0;
    int child_id = 0;
    zmq::context_t context(1);
    zmq::socket_t main_socket(context, ZMQ_REQ);
    int linger = 0;
    main_socket.setsockopt(ZMQ_SNDTIMEO, 2000);
    main_socket.setsockopt(ZMQ_LINGER, &linger, sizeof(linger));
    int port = bind_socket(main_socket);

    while (true) {
        std::cin >> command;
        if (command == "create") {
            size_t node_id;
            std::string result;
            std::cin >> node_id;
            if (child_pid == 0) {
                child_pid = fork();
                if (child_pid == -1) {
                    std::cout << "Unable to create the first node\n";
```

```
            child_pid = 0;
            exit(1);
        } else if (child_pid == 0) {
            create_node(node_id, port);
        } else {
            child_id = node_id;
            send_message(main_socket,"pid");
            result = recieve_message(main_socket);
        }
    } else {
        std::ostringstream msg_stream;
        msg_stream << "create " << node_id;
        send_message(main_socket, msg_stream.str());
        result = recieve_message(main_socket);
    }

    if (result.substr(0,2) == "Ok") {
        ids.insert(node_id);
    }
    std::cout << result << "\n";

} else if (command == "remove") {
    if (child_pid == 0) {
        std::cout << "Error:Not found\n";
        continue;
    }
    size_t node_id;
    std::cin >> node_id;
    if (node_id == child_id) {
```

```
            kill(child_pid, SIGTERM);
            kill(child_pid, SIGKILL);
            child_id = 0;
            child_pid = 0;
            std::cout << "Ok\n";
            ids.erase(node_id);
            continue;
        }
        std::string message_string = "remove " + std::to_string(node_id);
        send_message(main_socket, message_string);

        std::string recieved_message = recieve_message(main_socket);
        if (recieved_message.substr(0, std::min<int>(recieved_message.size(), 2))
== "Ok") {
            ids.erase(node_id);
        }
        std::cout << recieved_message << "\n";

    } else if (command == "exec") {
        int id;
        std::cin >> id;

        char nameAndValueArr[256];
        std::cin.getline(nameAndValueArr, 256);
        std::string nameAndValue = nameAndValueArr;

        std::string message_string = "exec " + std::to_string(id);
        send_message(main_socket, message_string);
```

```
    std::string recieved_message = recieve_message(main_socket);
    if (recieved_message == "Node is available") {
        std::string name;
        int value;
        std::stringstream ss(nameAndValue);
        bool searchNeeded = true;
        for (int i = 1; i < 256; ++i) {
            if (nameAndValueArr[i] == ' ') {
                ss >> name;
                ss >> value;
                ids.add_to_dictionary(id, name, value);
                std::cout << "Ok:" << id << std::endl;
                searchNeeded = false;
                break;
            }
        }
        if (searchNeeded) {
            ss >> name;
            std::cout << "Ok:" << id << ": ";
            ids.get_from_dictionary(id, name);
        }
    } else {
        std::cout << recieved_message << std::endl;
    }

} else if (command == "pingall") {
    send_message(main_socket,"pingall");
    std::string recieved = recieve_message(main_socket);
    std::istringstream is;
```

```
            if (recieved.substr(0,std::min<int>(recieved.size(), 5)) == "Error") {
                is = std::istringstream("");
            } else {
                is = std::istringstream(recieved);
            }

            std::set<int> recieved_ids;
            int rec_id;
            while (is >> rec_id) {
                recieved_ids.insert(rec_id);
            }
            std::vector from_tree = ids.get_nodes();
            auto part_it = std::partition(from_tree.begin(), from_tree.end(),
[&recieved_ids] (int a) {
                return recieved_ids.count(a) == 0;
            });
            if (part_it == from_tree.begin()) {
                std::cout << "Ok: -1\n";
            } else {
                std::cout << "Ok:";
                for (auto it = from_tree.begin(); it != part_it; ++it) {
                    std::cout << " " << *it;
                }
                std::cout << "\n";
            }

        } else if (command == "exit") {
            break;
        }
```

```
    }
    return 0;
}
```

**Демонстрация работы программы**

```
 …/lab6  ./terminal 15 0
create 1
Ok:25839
create 2
Ok:25848
create 3
Ok:25853
create 5000
Ok:25858
pingall
Ok: 1 2 3 5000
remove 2
Ok
pingall
Ok: 1
create 3
Ok:25868
create 5000
Ok:25877
pingall
Ok: 1 3 5000
exec 1 test 123
Ok:1
exec 1 test
Ok:1: 123
```

exec 3 testtest123 12345

Ok:3

exec 3 testtest123

Ok:3: 12345

exec 1 TEST

Ok:1: 'TEST' not found

exit

## Вывод

В результате данной лабораторной работы я научился работать с технологией очереди сообщений, создавать программы, создающие и связывающие процессы в определенные топологии. Также я познакомился с принципами работы стандартного контрейнера unordered_map.