Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работ №6-8 по курсу

«Операционные системы»

Группа: М80 – 207Б-18

Студент: Цапков Александр Максимович

Преподаватель: Миронов Евгений Сергеевич

Оценка: _____

Дата: _____

Москва, 2019.

Содержание

Постановка задачи.

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.

Вариант: 2-4-2

Общие сведения о программе

Программа состоит из двух исполняемых файлов: одного для управляющего узла, другой для рабочих узлов. Оба они используют написанные мною статические библиотеки Message.hpp и Nodes.hpp. Связь процессов осуществляется при помощи очередей сообщений ZerroMQ (библиотека на C++).

Общий метод и алгоритм решения.

Топология моего варианта подразумевает список списков, то есть к управляющему узлу может быть подключено любое количество узлов, но у этих рабочих узлов может быть только 2 связи— родительская и дочерняя (как у двусвязного списка). Для общения процессов я использовал ZMQ. Её функционал очень понятен и удобен в использовании, но для еще большего удобства я написал собственные классы для обмена сообщениями на базе функций ZMQ.

В классе message я реализовал 3 поля: одно для типа передаваемого сообщения (через ENUM), второе для id, а третье data, которая может зависеть от типа сообщения. В этом же классе реализован методы send и recv, для передачи и приема сообщений. Класс Node нужен для хранения информации о смежных узлах. В нем есть поля для pid, id, сокета, контекста и порта. Именно объекты этого класса я передаются методам send и recv класса message для передачи сообщения.

Для оптимизации работы с узлами и их поиска в управляющем процессе я воспользовался map’ом. Он ассоциирует id узлов с номером списка в котором он находится. Это так же помогает в проверке того, имеет ли дерево узел с таким id или нет. Парсер сообщений находится только в управляющем узле, а рабочим узлам уже посылается стандартизированное сообщение класса message.

Рабочие узлы в моей программе должны работать как таймеры. При возвращении времени управляющему узлу, я также пользуюсь классом message и кладу время в поле data.

Основные файлы программы.

Файл Message.hpp

```
#ifndef MESSAGE_HPP
#define MESSAGE_HPP

#include <zmq.hpp>
#include <string>
#include <ctime>
#include <vector>
#include <unistd.h>
#include <iostream>
#include "Nodes.hpp"

class message {
   public:
      enum {
         ERR, CREATE,
         REMOVE, PING, REPLY, DEF,
         TAKEPORT, REPAR, EXEC, START,
         STOP, TIME, TERM, KILLNPASS
      };

      int type;
      int id;
      int data;

      message() {}
      message(Node& from) {
         recv(from);
      }
      message(int type, int id, int data) : type(type), id(id), data(data) {}
      void send(Node& to) {
         //puts("a");
         zmq::message_t typeMes(&type, sizeof(int));
         zmq::message_t idMes(&id, sizeof(int));
         zmq::message_t dataMes(&data, sizeof(int));
         //puts("b");
         to.Sock().send(typeMes, zmq::send_flags::sndmore);
         to.Sock().send(idMes, zmq::send_flags::sndmore);
         to.Sock().send(dataMes, zmq::send_flags::none);
         //puts("c");
      }
      void sendDW(Node& to) {
         //puts("a");
         zmq::message_t typeMes(&type, sizeof(int));
         zmq::message_t idMes(&id, sizeof(int));
         zmq::message_t dataMes(&data, sizeof(int));
         //puts("b");
         to.Sock().send(typeMes, zmq::send_flags::sndmore | zmq::send_flags::dontwait);
         to.Sock().send(idMes, zmq::send_flags::sndmore | zmq::send_flags::dontwait);
         to.Sock().send(dataMes, zmq::send_flags::dontwait);
```

```
        //std::this_thread::sleep_for(std::chrono::milliseconds(10));
        //puts("c");
    }
    void recv(Node& from) {
        //puts("aa");
        zmq::message_t typeMes;
        zmq::message_t idMes;
        zmq::message_t dataMes;
        from.Sock().recv(typeMes, zmq::recv_flags::none);
        from.Sock().recv(idMes, zmq::recv_flags::none);
        from.Sock().recv(dataMes, zmq::recv_flags::none);
        type = *((int*)(typeMes.data()));
        id = *((int*)(idMes.data()));
        data = *((int*)(dataMes.data()));
        //puts("cc");
    }
    int recvCheck(Node& from) {
        zmq::message_t typeMes;
        zmq::message_t idMes;
        zmq::message_t dataMes;
        auto start = clock();
        while (true) {
            if (from.Sock().recv(typeMes, zmq::recv_flags::dontwait) &&
                from.Sock().recv(idMes, zmq::recv_flags::dontwait)   &&
                from.Sock().recv(dataMes, zmq::recv_flags::dontwait)   )
            {
                type = *((int*)(typeMes.data()));
                id = *((int*)(idMes.data()));
                data = *((int*)(dataMes.data()));
                return 1;
            }
            if (clock() - start > (CLOCKS_PER_SEC/2))
                return 0;
        }
    }

};

#endif
```

Файл Nodes.hpp

```
#ifndef NODES_HPP
#define NODES_HPP

#include <zmq.hpp>
#include <string>
#include <vector>
#include <unistd.h>
#include <iostream>
#include <thread>
//#include <map>
```

```
int BEGIN_PORT = 30000;
const char* BIND_URL = "tcp://*:";
const char* CON_URL = "tcp://localhost:";

std::string BindURLPort(int port) {
   return BIND_URL + std::to_string(port);
}

std::string ConURLPort(int port) {
   return CON_URL + std::to_string(port);
}

int TakePort(zmq::socket_t& sock)
{
   int port = BEGIN_PORT;
   while (true) {
      try {
         sock.bind(BindURLPort(port));
         break;
      } catch (const zmq::error_t& err) {
         ++port;
      }
   }
   return port;
}

class Node {
   public:
      enum {
         ERR, CHILD,
         PARENT, NEW, ADD
      };
      Node() : id(-2), ctx(1), sock(ctx, ZMQ_PAIR) {}
      Node(int type, int id, int inPort = -1) : ctx(1), sock(ctx, ZMQ_PAIR), id(id) {
         switch(type) {
            case CHILD:
               port = inPort;
               sock.bind(BindURLPort(port));
               break;
            case PARENT:
               port = inPort;
               sock.connect(ConURLPort(port));
               //std::this_thread::sleep_for(std::chrono::milliseconds(100));
               break;
            case NEW:
               port = TakePort(sock);
               //std::this_thread::sleep_for(std::chrono::milliseconds(10));
               pid = fork();
               if (pid == 0) {
                  using std::to_string;
                  execl("WNode", to_string(id).c_str(), to_string(port).c_str(), (char*)NULL);
               }
```

```
            break;
        }
    }
    void New(int inId) {
        id = inId;
        port = TakePort(sock);
        pid = fork();
        if (pid == 0) {
            using std::to_string;
            execl("WNode", to_string(id).c_str(), to_string(port).c_str(), (char*)NULL);
        }
    }
    void TakePortSetId(int Id) {
        id = Id;
        port = TakePort(sock);
        //std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    // void ReBind() {
    //     sock.unbind(BindURLPort(port));
    //     sock.bind(BindURLPort(port));
    // }
    void connectTo(int inPort) {
        sock.disconnect(ConURLPort(port));
        port = inPort;
        sock.connect(ConURLPort(port));
    }
    void disConnect() {
        sock.disconnect(ConURLPort(port));
    }
    void unBind() {
        char* adr;
        size_t len;
        sock.getsockopt(ZMQ_LAST_ENDPOINT, adr, &len);
        sock.unbind(adr);
    }

    pid_t Pid() {
        return pid;
    }
    int& Id() {
        return id;
    }
    int Port() {
        return port;
    }
    zmq::socket_t& Sock() {
        return sock;
    }

private:
    int id;
    int port;
    pid_t pid;
```

```
        zmq::context_t ctx;
        zmq::socket_t sock;
};

#endif
```

Файл main.cpp

```
#include <iostream>
#include <zmq.hpp>
#include <string>
#include <vector>
#include <map>
#include "Nodes.hpp"
#include "Message.hpp"

enum {
   ERR, CREATE,
   REMOVE, EXEC,
   PING, EXIT, START,
   STOP, TIME
};


int main(void) {
   std::vector<Node*> lists;
   std::map<int, int> NodesAdr;
   NodesAdr[-1] = -1;

   //for text menu
   std::string comId;
   std::map <std::string, int> command;
   command["create"] = CREATE;     command["remove"] = REMOVE;
   command["exec"] = EXEC;         command["ping"] = PING;
   command["q"] = EXIT;            command["start"] = START;
   command["stop"] = STOP;         command["time"] = TIME;
   //
   //for parsing
   int id, parent, listID;
   //
   message to;
   message from;
   //message parsing
   int exit = 0;
   puts("Start");
   while(!exit) {
      std::cin >> comId;
      switch (command[comId]) {
         case CREATE:
            int id, parent;
            std::cin >> id >> parent;
            if (NodesAdr.find(parent) == NodesAdr.end()) {
```

```
            std::cout << "Error: Parent not found\n";
            break;
        }
        if (NodesAdr.find(id) != NodesAdr.end()) {
            std::cout << "Error: Already exists\n";
            break;
        }
        if (parent == -1) {
            lists.push_back(new Node(Node::NEW, id));
            pid_t added = lists[lists.size() - 1]->Pid();
            NodesAdr[id] = lists.size() - 1;
            std::cout << "Ok: " << added << '\n';
        } else {
            listID = NodesAdr[parent];
            to.type = message::CREATE;
            to.id = id;
            to.data = parent;
            to.sendDW(*lists[listID]);
            if(!from.recvCheck(*lists[listID])) {
                std::cout << "Error: Parent is unavailable\n";
                break;
            }
            if (from.type != message::ERR) {
                std::cout << "Ok: " << from.data << '\n';
                NodesAdr[id] = NodesAdr[parent];
            } else {
                std::cout << "Error: Parent is unavailable\n";
            }
        }
        break;
    case REMOVE:
        std::cin >> id;
        if (NodesAdr.find(id) == NodesAdr.end() || id == -1) {
            std::cout << "Error:id: Not found\n";
            break;
        }
        listID = NodesAdr[id];
        if (lists[listID]->Id() == id){
            to.type = message::KILLNPASS;
            //Node* tmp = new Node(Node::NEW, 0);
            Node* tmp = new Node();
            tmp->TakePortSetId(0);
            to.data = tmp->Port();
            to.send(*lists[listID]);
            if(!from.recvCheck(*lists[listID])) {
                std::cout << "Error: Node is unavailable\n";
                to.type = message::TERM;
                to.send(*tmp);
                delete tmp;
                break;
            }
            if (from.type != message::ERR) {
                tmp->Id() = from.id;
```

```
            std::swap(tmp, lists[listID]);
            delete tmp;
            if (lists[listID]->Id() == -2) {
                delete lists[listID];
                lists[listID] = nullptr;
            }
            std::cout << "Ok\n";
            NodesAdr.erase(id);
        } else {
            std::cout << "Error: Node is unavailable\n";
            to.type = message::TERM;
            to.send(*tmp);
            delete tmp;
        }
        break;
    } else {
        to.type = message::REMOVE;
        to.id = id;
        to.send(*lists[listID]);
        if(!from.recvCheck(*lists[listID])) {
            std::cout << "Error: Node is unavailable\n";
            break;
        }
        if (from.type != message::ERR) {
            std::cout << "Ok\n";
            NodesAdr.erase(id);
        } else {
            std::cout << "Error: Node is unavailable\n";
        }
    }
    break;
case EXEC:
    std::cin >> id;
    std::cin >> comId;
    if (NodesAdr.find(id) == NodesAdr.end()) {
        std::cout << "Error:id: Not found\n";
        break;
    }
    if (command[comId] ==  START ) {
        to.data = message::START;
    } else if (command[comId] ==  STOP) {
        to.data = message::STOP;
    } else if (command[comId] ==  TIME ) {
        to.data = message::TIME;
    } else {
        std::cout << "Error:id: Wrong param\n";
        break;
    }
    listID = NodesAdr[id];
    to.type = message::EXEC;
    to.id = id;
    to.send(*lists[listID]);
    if(!from.recvCheck(*lists[listID])) {
```

```
                std::cout << "Error: Node is unavailable\n";
                break;
            }
            if (from.type == message::TIME) {
                std::cout << "Ok:" << id << ": " <<from.data << '\n';
            } else if (from.type == message::ERR) {
                std::cout << "Error: Node is unavailable\n";
            } else {
                std::cout << "Ok:" << id << '\n';
            }
            break;
        case PING:
            std::cin >> id;
            if (NodesAdr.find(id) == NodesAdr.end() || id == -1) {
                std::cout << "Error: Not found\n";
                break;
            }
            listID = NodesAdr[id];
            to.type = message::PING;
            to.id = id;
            to.sendDW(*lists[listID]);
            if(!from.recvCheck(*lists[listID])) {
                std::cout << "Ok: 0\n";
            } else {
                std::cout << "Ok: 1\n";
            }
            break;
        case EXIT:
            exit = 1;
            break;
        case ERR:
            std::cout << "Wrong Command\n";
            break;
    }
  }

  for (int i = 0; i < lists.size(); i++) {
    to.type = message::TERM;
    if (lists[i] != nullptr)
        to.sendDW(*lists[i]);
    delete lists[i];
  }
  puts("Stop");
  return 0;
}
```

Файл work.cpp

```
//#include <iostream>
#include <zmq.hpp>
#include <string>
#include <vector>
```

```
#include <chrono>
//#include <map>
#include "Nodes.hpp"
#include "Message.hpp"



int main(int argc, char** argv) {
    if (argc != 2)
        return 0;

    int Id = atoi(argv[0]);
    Node* Parent = new Node(Node::PARENT, 0, atoi(argv[1]));
    Node* Child = new Node;
    Node* tmp;

    message to;
    message from;

    //for exec
    auto start = std::chrono::system_clock::now();
    auto stop = std::chrono::system_clock::now();
    int timerStat = 0;
    //

    while(true) {
        //std::cout << Id << "wainting\n";
        from.recv(*Parent);
        switch (from.type) {
            case message::CREATE:
                if (Id == from.data) {
                    if (Child->Id() == -2) {
                        Child->New(from.id);
                        from.type = message::REPLY;
                        from.data = Child->Pid();
                        from.send(*Parent);
                    } else {
                        tmp = new Node(Node::NEW, from.id);
                        to.type = message::TAKEPORT;
                        to.id = Child->Id();
                        to.send(*tmp);
                        from.recv(*tmp);
                        // if(!from.recvCheck(*tmp) {
                        //     std::cout << "Error: Node is unavailable\n";
                        //     break;
                        // }
                        if (from.type != message::ERR) {
                            to.type = message::REPAR;
                            to.data = from.data;
                            to.send(*Child);
                            std::swap(tmp, Child);
                            delete tmp;
                            from.type = message::REPLY;
```

```
            from.data = Child->Pid();
            from.send(*Parent);
          } else {
            std::cout << "err in WNode Create\n";
          }

        }
      } else {
        from.send(*Child);
        if(!to.recvCheck(*Child)) {
          to.type = message::ERR;
          to.send(*Parent);
          break;
        } else {
          to.send(*Parent);
        }
      }
      break;
    case message::REMOVE:
      if (Child->Id() == from.id){
        //tmp = new Node(Node::NEW, 0);
        tmp = new Node();
        tmp->TakePortSetId(0);
        to.type = message::KILLNPASS;
        to.data = tmp->Port();
        to.send(*Child);
        //to.recv(*Child);
        if(!to.recvCheck(*Child)) {
          to.type = message::ERR;
          to.send(*Parent);
          to.type = message::TERM;
          to.send(*tmp);
          delete tmp;
          break;
        }
        if (to.type != message::ERR) {
          tmp->Id() = to.id;
          std::swap(tmp, Child);
          //to.type = message::TERM;
          //to.send(*tmp);
          delete tmp;
        } else {
          std::cout << "Error: while removing\n";
        }
        from.send(*Parent);
        break;
      } else {
        from.send(*Child);
        if(!to.recvCheck(*Child)) {
          to.type = message::ERR;
          to.send(*Parent);
          break;
        } else {
```

```
            to.send(*Parent);
        }
        break;
    }
case message::KILLNPASS:
    if (Child->Id() != -2) {
        to.type = message::REPAR;
        to.data = from.data;
        to.send(*Child);
    }
    to.type = message::REPLY;
    to.id = Child->Id();
    to.send(*Parent);
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    return 0;
    break;
case message::TAKEPORT:
    //puts("Taking");
    Child->TakePortSetId(from.id);
    from.type = message::REPLY;
    from.data = Child->Port();
    from.send(*Parent);
    break;
case message::REPAR:
    tmp = new Node(Node::PARENT, 0, from.data);
    std::swap(tmp, Parent);

    delete tmp;
    break;
case message::TERM:
    if (Child->Id() != -2) {
        from.send(*Child);
    }
    return 0;
    break;
case message::EXEC:
    if (Id == from.id) {
        switch (from.data) {
            case message::START:
                timerStat = 1;
                start = std::chrono::system_clock::now();
                stop = std::chrono::system_clock::now();
                from.type = message::REPLY;
                from.send(*Parent);
                break;
            case message::STOP:
                if (timerStat == 1) {
                    stop = std::chrono::system_clock::now();
                    timerStat = 0;
                }
                from.type = message::REPLY;
                from.send(*Parent);
                break;
```

```
                case message::TIME:
                    from.type = message::TIME;
                    if (timerStat == 1) {
                        stop = std::chrono::system_clock::now();
                    }
                    auto msS =
std::chrono::time_point_cast<std::chrono::milliseconds>(start).time_since_epoch().count();
                    auto msF =
std::chrono::time_point_cast<std::chrono::milliseconds>(stop).time_since_epoch().count();
                    from.data = msF - msS;
                    from.send(*Parent);
                    break;
                }
            } else {
                from.send(*Child);
                if(!to.recvCheck(*Child)) {
                    to.type = message::ERR;
                    to.send(*Parent);
                    break;
                } else {
                    to.send(*Parent);
                }
                break;
            }
            break;
        case message::PING:
            if (Id == from.id) {
                to.type = message::REPLY;
                to.send(*Parent);
            } else {
                from.send(*Child);
                if(!to.recvCheck(*Child)) {
                    to.type = message::ERR;
                    to.send(*Parent);
                    break;
                } else {
                    to.send(*Parent);
                }
            }
            break;
    }
  }
  return 0;
}
```

Пример работы программы

```
alex@Alexs-MacBook-Pro src % make
g++ -std=c++11 -o WNode work.cpp -lzmq
g++ -std=c++11 -o main main.cpp -lzmq
alex@Alexs-MacBook-Pro src % ./main
Start
create 1 -1
Ok: 6664
create 3 1
Ok: 6668
create 2 1
Ok: 6672
exec 3 time
Ok:3: 0
exec 3 start
Ok:3
exec 3 time
Ok:3: 3676
create 4 -1
Ok: 6685
remove 2
Ok
exec 3 time
Ok:3: 17862
remove 4
Ok
create 5 4
Error: Parent not found
create 5 3
Ok: 6701
exec 3 time
Ok:3: 45909
exec 3 stop
Ok:3
exec 2 start
Error:id: Not found
exec 4 start
Error:id: Not found
exec 5 start
Ok:5
exec 3 time
Ok:3: 50904
exec 5 time
Ok:5: 8391
exec 3 time
Ok:3: 50904
q
Stop
alex@Alexs-MacBook-Pro src %
```

Вывод

В данной лабораторной работе я познакомился с очередями сообщений. Это был мой первый опыт работы с серверами сообщений, но я знал о слышал о них и раньше и было интересно поработать с ними самостоятельно. Писать это оказалось немного легче чем я думал, скорее всего из-за выбора подходящей библиотеки, чем я не разу не жалел.