Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математикиКафедра вычислительной математики и программирования

Лабораторная работа №6-8 по курсу

«Операционные системы»

Группа: М80 – 201Б-19
Студент: Цыкин И.А.
Преподаватель: Миронов Е.С
Оценка:

_____

_____

Дата:

_____

Москва,
2020.

Содержание

Постановка задачи

Целью является приобретение практических навыков в:

- Управлении серверами сообщений (No6)
- Применение отложенных вычислений (No7)
- Интеграция программных систем друг с другом (No8)

Вариант 45

1. Топология — отсортированное бинарное дерево (Но у меня не вышло реализовать это).
2. Тип вычислительной команды — локальный таймер.
3. Тип проверки узлов на доступность — heartbeat time.

Общие сведения о программе

Программа состоит из двух основных файлов и библиотеки, реализующей взаимодействия с узлами. Помимо этого используется библиотека zmq, которая реализует очередь сообщений.
1) main.cpp — программа управляющего узлов
3) command.hpp - реализация библиотеки для взаимодействия между узлами
5) child.cpp – файл дочернего узла

Очередь сообщений - компонент, используемый для межпроцессного или межпотокового взаимодействия внутри одного процесса. Для обмена сообщениями используется очередь. Очереди сообщений предоставляют асинхронный протокол передачи данных, означая, что отправитель и получатель сообщения не обязаны взаимодействовать с очередью сообщений одновременно. Размещённые в очереди сообщения хранятся до тех пор, пока получатель не получит их.

ZMQ - библиотека асинхронных сообщений, предназначенная для использования в распределенных или параллельных приложениях. Он обеспечивает очередь сообщений, но в отличие от промежуточного программного обеспечения, ориентированного на сообщения, система ZMQ может работать без выделенного посредника сообщений.

Сокеты - название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет — абстрактный объект, представляющий конечную точку соединения.

Общий метод и алгоритм решения

1. Управляющий узел принимает команды, обрабатывает их и пересылает дочерниму узлу.
2. 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.
3. Если узел недоступен, то по истечении таймаута будет сгенерировано сообщение о недоступности узла и оно будет передано вверх по дереву, к управляющему узлу. При удалении узла, все его потомки уничтожаются.

Листинг программы

main.cpp

```
#include "command.hpp"

#include <csignal>
#include <vector>
#include <map>

using namespace std;

int main(){
	vector<int> vec;
	map<int, int> works;
	zmq::context_t context(1);
	zmq::socket_t main_socket(context, ZMQ_REQ);
	int port = bind_socket(main_socket);
	string cmd;
	string msg;
	string sub_cmd;
	string result;
	int input_id;
	int n = 5;
	int child_pid = 0;
	int child_id = 0;

	auto begin = chrono::steady_clock::now();
	auto end = chrono::steady_clock::now();
	auto elapsed_ms = 0;

	for(;;){
		cin >> cmd;
		if(cmd == "create") {
```

```
        cin >> input_id;
        if (child_pid == 0) {
              child_pid = fork();
              if (child_pid == 0) {
                    create_node(input_id, port);
              } else{
                    child_id = input_id;
                    msg = "pid";
                    message_send(main_socket, msg);
                    result = message_recieve(main_socket);
                    if(result.substr(0,2) == "OK"){
                          vec.push_back(input_id);
                    }
              }
        }else {
              ostringstream msg_stream;
              msg_stream << "create " << input_id;
              message_send(main_socket, msg_stream.str());
              result = message_recieve(main_socket);
              if(result.substr(0,2) == "OK"){
                    vec.push_back(input_id);
              }
        }
        cout << result << endl;
  }else if(cmd == "remove") {
        if (child_pid == 0) {
              cout << "Error: Not found" << endl;
              continue;
        }
        cin >> input_id;
        if (input_id == child_id) {
              msg = "kill_child";
              message_send(main_socket, msg);
              result = message_recieve(main_socket);
              if(result == "OK"){
                    kill(child_pid, SIGTERM);
                    kill(child_pid, SIGKILL);
                    child_id = 0;
                    child_pid = 0;
                    cout << result << endl;
                    vec.clear();
              }else{
                    cout << "Error: exit" << endl;
              }
              continue;
        }
        ostringstream msg_stream;
        msg_stream << "remove " << input_id;
        message_send(main_socket, msg_stream.str());
        result = message_recieve(main_socket);
        cout << result << endl;
```

```
            if(result.substr(0,2) == "OK"){
                  for(int i = vec.size() - 1; i >= 0; --i){
                        if(vec[i] != input_id){
                        vec.pop_back();
                  }else{
                        vec.pop_back();
                        break;
                  }
            }
      }
}else if(cmd == "all"){
      if (child_pid == 0) {
            cout << "Error: Not found" << endl;
            continue;
      }
      for(int i = 0; i < vec.size(); i++){
            cout << vec[i] << " ";
      }
      cout << endl;
}else if(cmd == "exec"){
      if (child_pid == 0) {
            cout << "Error: Not found" << endl;
            continue;
      }
      cin >> input_id;
      cin >> sub_cmd;
      ostringstream msg_stream;
      msg_stream << "exec " << input_id << " " << sub_cmd;
      message_send(main_socket, msg_stream.str());
      result = message_recieve(main_socket);
      cout << result << endl;
}else if(cmd == "heartbeat"){
      if (child_pid == 0) {
            cout << "Error: Not found" << endl;
            continue;
      }
      works.clear();
      int time;
      cin >> time;
      cmd = cmd + " " + to_string(time);
      sleep(time/1000);
      message_send(main_socket, cmd);
      result = message_recieve(main_socket);
      istringstream is = istringstream(result);
      while(is){
            is >> input_id;
            works.insert(make_pair(input_id, 1));
      }
      cout << "OK" << endl;
}else if (cmd == "ping") {
      if(works.size() == 0){
```

```
                continue;
            }
            cin >> input_id;
            if(works[input_id] == 1){
                cout << "OK: 1" << endl;
            }else{
                cout << "OK: -1" << endl;
            }
        }else if(cmd == "exit"){
            if(child_pid == 0){
                cout << "OK\n";
                return 0;
            }
            msg = "kill_child";
            message_send(main_socket, msg);
            result = message_recieve(main_socket);
            if(result == "OK"){
                kill(child_pid, SIGTERM);
                kill(child_pid, SIGKILL);
                child_id = 0;
                child_pid = 0;
                cout << result << endl;
            }else{
                cout << "Error: exit" << endl;
            }
                return 0;
            }else{
                cout << "Error: bad command" << endl;
            }
        }
}
```

child.cpp

```
#include <csignal>
#include <chrono>


using namespace std;

int main(int argc, char* argv[]){
    if(argc != 2) {
        cout << "Error: child's parametrs" << endl;
        return -1;
    }

    int id = stoi(argv[0]);
    int port = stoi(argv[1]);
    zmq::context_t context(2);
    zmq::socket_t parent_socket(context, ZMQ_REP);
```

```
zmq::socket_t child_socket(context, ZMQ_REQ);

parent_socket.connect(get_port(port));
int child_port = bind_socket(child_socket);

string request;
string cmd;
string sub_cmd;
string msg;
string result;
int input_id;
int child_pid = 0;
int child_id = 0;
int send_child = 0;
int last_heartbit_time = -1;

auto begin = chrono::steady_clock::now();
auto end = chrono::steady_clock::now();
auto elapsed_ms = 0;

for(;;){
	request = message_recieve(parent_socket);
	istringstream cmd_stream(request);
	cmd_stream >> cmd;
		if(cmd == "pid") {
			msg = "OK: " + to_string(getpid());
			message_send(parent_socket, msg);
		} else if (cmd == "kill_child") {
			if (child_pid == 0) {
				msg = "OK";
				message_send(parent_socket, msg);
			} else {
				msg = "kill_child";
				message_send(child_socket, msg);
				result = message_recieve(child_socket);
				if(result == "OK"){
					message_send(parent_socket, result);
				}else{
					cout << "Error: kill" << endl;
				}
				kill(child_pid, SIGTERM);
				kill(child_pid, SIGKILL);
				message_send(parent_socket, result);
			}
		}else if(cmd == "ping"){
			if(child_pid == 0){
				msg = "OK: ";
				message_send(parent_socket, msg);
			}else{
				message_send(child_socket, cmd);
				string str = message_recieve(child_socket);
```

```
                result = str + to_string(child_id) + " ";
                message_send(parent_socket, result);
        }
    }else if(cmd == "create") {
        cmd_stream >> input_id;
        if (input_id == id) {
                msg = "Error: Already exists";
                message_send(parent_socket, msg);
        } else if (child_pid == 0) {
                child_pid = fork();
                        if (child_pid == 0) {
                        create_node(input_id, child_port);
                } else {
                        child_id = input_id;
                        msg = "pid";
                        message_send(child_socket, msg);
                        result = message_recieve(child_socket);
                        message_send(parent_socket, result);
                }
        } else {
                message_send(child_socket, request);
                result = message_recieve(child_socket);
                message_send(parent_socket, result);
        }
    }else if(cmd == "remove"){
        cmd_stream >> input_id;
        if(child_pid == 0){
                msg = "Error: Not found";
                message_send(parent_socket, msg);
        }else if(child_id == input_id){
                msg = "kill_child";
                message_send(child_socket, msg);
                result = message_recieve(child_socket);
        if(result == "OK"){
                message_send(parent_socket, result);
        }else{
                cout << "Error: kill" << endl;
        }
        kill(child_pid, SIGTERM);
        kill(child_pid, SIGKILL);
        child_pid = 0;
        child_id = 0;
        message_send(parent_socket, result);
        } else{
                message_send(child_socket, request);
                result = message_recieve(child_socket);
                message_send(parent_socket, result);
        }
    }else if(cmd == "exec"){
        cmd_stream >> input_id;
        if(id == input_id){
```

```
                              cmd_stream >> sub_cmd;
                              if(sub_cmd == "start"){
                                    begin = std::chrono::steady_clock::now();
                                    result = "OK: start";
                              }else if(sub_cmd == "stop"){
                                    end = std::chrono::steady_clock::now();
                                    elapsed_ms =
chrono::duration_cast<std::chrono::milliseconds>(end - begin).count();
                              result = "OK: stop";
                              }else if(sub_cmd == "time"){
                                    result = "OK: " + to_string(elapsed_ms) +
" ms";
                                    elapsed_ms = 0;
                              }else{
                                    result = "Error: bad subcommand";
                              }
                              message_send(parent_socket, result);
                        }else{
                              if(child_pid == 0){
                              msg = "Error: Not found";
                              message_send(child_socket, msg);
                        }else{
                              message_send(child_socket, request);
                              result = message_recieve(child_socket);
                              message_send(parent_socket, result);
                        }
                  }
            }else if(cmd == "heartbeat"){
                  int time;
                  cmd_stream >> time;
                  if(child_pid == 0){
                        msg = to_string(id);
                  }else{
                        auto t1 = std::chrono::steady_clock::now();
                        message_send(child_socket, request);
                        result = message_recieve(child_socket);
                        auto t2 = std::chrono::steady_clock::now();
                        auto T =
chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
                        if(T > 4*time){
                              msg = to_string(id);
                        }else{
                              msg = result + " " + to_string(id);
                        }
                  }
                  message_send(parent_socket, msg);
            }
      }
}
```

command.cpp

```
#include <iostream>

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

using namespace std;

void create_node(int& id, int& port) {
	char* arg_id = strdup((to_string(id)).c_str());
	char* arg_port = strdup((to_string(port)).c_str());
	char* args[] = {arg_id, arg_port, NULL};
	execv("./child", args);
}

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

int bind_socket(zmq::socket_t& socket) {
	int port = 3000;
	while (true) {
		try {
			socket.bind(get_port(port));
			break;
		} catch(zmq::error_t &e) {
			++port;
		}
	}
	return port;
}

bool message_send(zmq::socket_t& socket, const string& msg) {
	int msg_size = msg.size();
	zmq::message_t message(msg_size);
	memcpy(message.data(), msg.c_str(), msg_size);
	try {
		socket.send(message, zmq::send_flags::none);
		return true;
	} catch(...) {
		return false;
	}
}

string message_recieve(zmq::socket_t& socket) {
	zmq::message_t request;
	zmq::send_result_t answer;
	try {
		answer = socket.recv(request, zmq::recv_flags::none);
```

```
	} catch(zmq::error_t &e) {
		answer = false;
	}
	string recieve_msg(static_cast<char*>(request.data()), request.size());
	if (recieve_msg.empty() || !answer)
		return "Error: Node is not available";
	else
		return recieve_msg;
}
```

Результаты работы программы

```
vaney@vaney-VirtualBox:~/OS/lab6$ g++ -o main main.cpp -lzmq
vaney@vaney-VirtualBox:~/OS/lab6$ g++ -o child child.cpp -lzmq
vaney@vaney-VirtualBox:~/OS/lab6$ ./main
create 12
OK: 6406
create 66
OK: 6410
create 88
OK: 6414
create 90
OK: 6419
all
12 66 88 90
exec 12 start
OK: start
heartbeat 100
OK
ping 12
OK: 1
ping 66
OK: 1
remove 66
OK
all
12
heartbeat 250
OK
ping 66
OK: -1
ping 12
OK: 1
exec 12 stop
OK: stop
exec 12 time
OK: 59899 ms
remove 12
OK
```

```
all
Error: Not found
heartbeat 100
Error: Not found
Error: bad command
exit
OK
vaney@vaney-VirtualBox:~/OS/lab6$
```

Выводы

В результате данной лабораторной работы я научился работать с технологией очереди сообщений, создающие и связывающие процессы в определенные топологии, понимать клиент-серверную архитектуру, читать документацию и осваивать новые библиотеки (zmq) в кратчайшие сроки.