

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ

**Отчёт по лабораторной работе №6**

**Курс: «Операционные системы»**

**Тема: «Средства межпроцессорного взаимодействия Windows»**

Выполнил студент:

Волкова М.Д.

Группа: 43501/3

Проверил:

Мальшев И.А.

Санкт-Петербург  
2017 г.

# Лабораторная работа №6

## 1.1 Цель работы

Изучить средства межпроцессорного взаимодействия в ОС Windows.

### Глава 1. Неименованные каналы

1. Создать клиент-серверное приложение, позволяющее набираемые символы в терминальном окне командной строки (сервер) отображать их в окно процесса-потомка (клиент).
2. Создать эхо-сервер, взаимодействующий с клиентом посредством pipe.

### Глава 2. Именованные каналы

1. Программа, обеспечивающая взаимодействие процессов посредством именованных каналов. Реализовать между одним клиентом и сервером обмен данными, вводимыми с консоли на стороне клиента и возвращаемыми сервером обратно до получения команды exit.
2. Программа, обеспечивающая взаимодействие процессов посредством именованных каналов – аналогичная программа с эхо-сервером, но с множеством клиентов и принудительной блокировкой обмена до завершения каждой операции. Реализовать между сервером и множеством клиентов обмен данными, вводимыми с консоли на стороне клиента и возвращаемыми сервером обратно до получения команды exit.
3. Модифицируем приложение из предыдущего примера для сетевого обмена информацией.

### Глава 3. Сокеты

1. Программа локального обмена сокетами с использованием потокового протокола с установлением соединения (TCP в стеке TCP/IP).
2. Модифицировать программу для локального обмена с множеством клиентов и с доступом к общему ресурсу.
3. Сетевая передача данных с помощью сокетов.
4. Реализовать сетевую передачу сообщений с помощью сокетов одновременно многих клиентов одному серверу с разных операционных систем.
5. Привести примеры использования портов завершения. Привести пример приложения с большим количеством клиентов до 1000 (когда порты завершения оправданы), общее количество потоков не более 10.
6. Оформить приложение с сокетами в виде службы.
7. Реализовать обмен на основе UDP.

### Глава 4. Сигналы в Windows

1. Создание обработчика сигналов завершения для консольного приложения.

## Глава 5. Разделяемая память

1. Взаимодействие двух процессов через совместно используемую именованную память, при котором первый процесс записывает данные, а второй считывает их. Создать программу, в которой первый процесс генерирует случайное число и записывает его в буфер, доступный второму процессу, откуда он его и считывает с последующим выводом.

## Глава 6. Почтовые слоты

1. Предложить собственную реализацию приложения, иллюстрирующую обмен информацией почтовыми слотами.
2. Продемонстрировать возможность локального и удаленного доступа.
3. Выполнить широкополосную передачу данных.

## 1.2 Ход работы

### 1.2.1 Глава 1. Неименованные каналы

Посредством pipe-канала можно передавать данные только между двумя процессами. В основе взаимодействия лежит так называемая файловая модель функционирования. Один из процессов создает канал, другой открывает его. После этого оба процесса могут передавать данные через канал в одну или обе стороны, используя для этого функции, предназначенные для работы с файлами, такие как ReadFile и WriteFile.

Анонимные каналы (anonymous channels) Windows обеспечивают однонаправленное (полудуплексное) по-символьное межпроцессное взаимодействие. Каждый канал имеет два дескриптора: дескриптор чтения (read handle) и дескриптор записи (write handle).

После создания канала необходимо передать клиентскому процессу его дескрипторы (или один из них), что обычно делается с помощью механизма наследования. Для наследования описателя нужно, чтобы процесс-потомок создавался функцией CreateProcess с флагом наследования TRUE.

Анонимный канал создается функцией CreatePipe:

```
BOOL WINAPI CreatePipe (
    _Out_ PHANDLE hReadPipe,
    _Out_ PHANDLE hWritePipe,
    _In_opt_ LPSECURITY_ATTRIBUTES lpPipeAttributes,
    _In_ DWORD nSize
);
```

## 1. Клиент-серверное приложение на основе анонимных каналов

На сервере создается неименованный канал для связи с процессом-потомком и порождается процесс-потомок. Также на сервере производится запись из консоли в канал. При нажатии на ESC приложение завершается:

```
1 #include <windows.h>
2 #include <conio.h>
3 #include <string.h>
4 #include <iostream>
5 #include <tchar.h>
6
7 // Путь к процессу потомку
8 static const char* CHILD_NAME = "p1.1.c.exe";
9
10 // Особые коды нажатых клавиш
11 static const short CARRIAGE_CODE = 0xD;
12 static const short EXIT_CODE = 0x1B;
13
14 int main() {
15     // Установка атрибутов для пайпа
16     SECURITY_ATTRIBUTES securityAttributes;
17     securityAttributes.nLength = sizeof(securityAttributes);
18     securityAttributes.lpSecurityDescriptor = nullptr;
19     securityAttributes.bInheritHandle = true;
20
21     HANDLE readPipe, writePipe;
22
23     // Создание анонимного канала
24     if (!CreatePipe(&readPipe, &writePipe, &securityAttributes, NULL)) {
25         std::cerr << "It's impossible to create pipe." << std::endl;
26         return 0x1;
27     }
28
29     std::cout << "Pipe has been created." << std::endl
30         << "Handle of pipe " << readPipe << std::endl;
31
32     // Подменяем стандартный дескриптор ввода дескриптором ввода канала
33     STARTUPINFO startupInfo;
34     ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
35     startupInfo.cb = sizeof(STARTUPINFO);
36     startupInfo.dwFlags = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
37     startupInfo.wShowWindow = SW_NORMAL;
38     startupInfo.hStdInput = readPipe;
39     startupInfo.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
40     startupInfo.hStdError = startupInfo.hStdOutput;
41
42     PROCESS_INFORMATION processInformation;
43
44     // Создадим новый процесс
45     if (!CreateProcess(nullptr, _tcsdup(TEXT(CHILD_NAME)), nullptr, nullptr, TRUE,
46         CREATE_NEW_CONSOLE, nullptr, nullptr, &startupInfo, &processInformation)) {
47         std::cerr << "It's impossible to create process." << std::endl;
48         return 0x2;
49     }
50
51     std::cout << "Process has been created." << std::endl;
```

```

51
52 std::cout << "STD INPUT HANDLE " << GetStdHandle(STD_INPUT_HANDLE) << std::endl
53     << "STD OUTPUT HANDLE " << GetStdHandle(STD_OUTPUT_HANDLE) << std::endl;
54
55 char buffer;
56 while(true) {
57     buffer = _getch();
58
59     std::cout.put(buffer);
60
61     if(buffer == CARRIAGE_CODE) {
62         buffer = '\n';
63         std::cout.put(buffer);
64     }
65
66     DWORD countOfWrittenBytes;
67
68     // Записываем в канал один байт
69     WriteFile(writePipe, &buffer, 1, &countOfWrittenBytes, nullptr);
70
71     // При нажатии на ESC завершаем цикл
72     if(buffer == EXIT_CODE)
73         break;
74 }
75
76 // Завершаем процесс потомок
77 TerminateProcess(processInformation.hProcess, NULL);
78
79 CloseHandle(processInformation.hThread);
80 CloseHandle(processInformation.hProcess);
81
82 CloseHandle(readPipe);
83 CloseHandle(writePipe);
84
85 std::cout << "Press \"Enter\" to exit." << std::endl;
86 std::getchar();
87
88 return 0x0;
89 }

```

Клиент открывает неименованный канал, и считывает из него сообщения. Запись и чтение производится с помощью стандартных потоков stdin и stdout.

```

1 #include <iostream>
2 #include <stdio.h>
3 #include <windows.h>
4
5 int main() {
6     std::cout << "STD INPUT HANDLE = " << GetStdHandle(STD_INPUT_HANDLE) << std::endl
7         << "STD OUTPUT HANDLE = " << GetStdHandle(STD_OUTPUT_HANDLE) << std::endl;
8
9     while(true) {
10         DWORD totalBytesAvailable;
11
12         // Получаем данные из канала
13         PeekNamedPipe(GetStdHandle(STD_INPUT_HANDLE), nullptr, NULL, nullptr, &
14             totalBytesAvailable, nullptr);
15
16         if(totalBytesAvailable) {
17             char buffer;
18             DWORD countOfBytesRead;
19
20             // Считываем один байт из канала
21             ReadFile(GetStdHandle(STD_INPUT_HANDLE), &buffer, 1, &countOfBytesRead, nullptr);
22             std::cout << buffer;
23         }
24     }
25 }

```

```

24
25     return 0x0;
26 }

```

Результат передачи от родителя потомку:

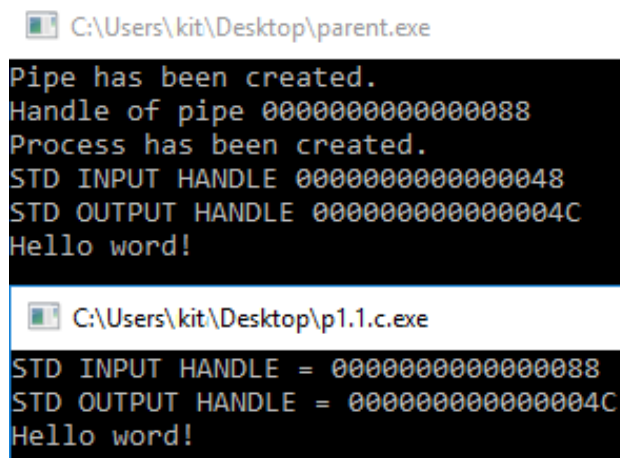


Рис. 1.1

Сообщение было успешно передано.

## 2. Эхо-сервер, взаимодействующий с клиентом посредством анонимных каналов

В программе используется передача дескрипторов через наследование. По причине того, что анонимный канал является полудуплексным, для организации эхо-сервера необходимо создавать 2 канала (для передачи от клиента-серверу и обратно):

```

1 #include <windows.h>
2 #include <stdio.h>
3 #include <ostream>
4 #include <iostream>
5 #include <tchar.h>
6
7 // Путь к процессу потомку
8 static const char* CHILD_NAME = "p1.2.c.exe";
9 // Размер буфера
10 static const int BUFFER_SIZE = 1024;
11 // Количество отсылаемых сообщений
12 static const int MESSAGES_COUNT = 5;
13 // Задержка перед завершением
14 static const int TERMINATE_DELAY = 6000;
15
16 int main() {
17     // Установка атрибутов для пайпов
18     SECURITY_ATTRIBUTES securityAttributes;
19     ZeroMemory(&securityAttributes, sizeof(securityAttributes));
20     securityAttributes.nLength = sizeof(SECURITY_ATTRIBUTES);
21     securityAttributes.lpSecurityDescriptor = nullptr;
22     securityAttributes.bInheritHandle = TRUE;
23
24     // Создаем канал для передачи от сервера клиенту
25     HANDLE readPipeFromServerToClient, writePipeFromServerToClient;
26     if (!CreatePipe(&readPipeFromServerToClient, &writePipeFromServerToClient, &
27         securityAttributes, NULL)) {
28         std::cerr << "It's impossible to create pipe from server to client." << std::endl;
29         return 0x1;
30     }
31
32     // Создаем канал для передачи от клиента серверу
33     HANDLE readPipeFromClientToServer, writePipeFromClientToServer;

```

```

33 if(!CreatePipe(&readPipeFromClientToServer, &writePipeFromClientToServer, &
34 securityAttributes, NULL)) {
35     std::cerr << "It's impossible to create pipe from client to server." << std::endl;
36     return 0x2;
37 }
38
39 std::cout << "Server started." << std::endl;
40
41 // Подменяем стандартный дескриптор ввода и вывода
42 STARTUPINFO startupInfo;
43 ZeroMemory(&startupInfo, sizeof(startupInfo));
44 GetStartupInfo(&startupInfo);
45 startupInfo.hStdInput = readPipeFromServerToClient;
46 startupInfo.hStdOutput = writePipeFromClientToServer;
47 startupInfo.hStdError = GetStdHandle(STD_ERROR_HANDLE);
48 startupInfo.dwFlags = STARTF_USESTDHANDLES;
49
50 PROCESS_INFORMATION processInformation;
51
52 // Создадим новый процесс
53 if(!CreateProcess(nullptr, _tcsdup(TEXT(CHILD_NAME)), nullptr, nullptr, TRUE,
54 CREATE_NEW_CONSOLE, nullptr, nullptr, &startupInfo, &processInformation)) {
55     std::cerr << "It's impossible to create process." << std::endl;
56     return 0x3;
57 }
58
59 CloseHandle(readPipeFromServerToClient);
60 CloseHandle(writePipeFromClientToServer);
61
62 std::string message;
63 char buffer[BUFFER_SIZE];
64
65 for(int index = 0; index < MESSAGES_COUNT; ++index) {
66     ZeroMemory(buffer, sizeof(buffer));
67     message.clear();
68
69     // Получаем сообщение от клиента
70     DWORD countOfBytesRead;
71     if(!ReadFile(readPipeFromClientToServer, buffer, BUFFER_SIZE, &countOfBytesRead,
72 nullptr)) {
73         std::cerr << "It's impossible to read file." << std::endl;
74         return 0x4;
75     }
76
77     message = buffer;
78
79     std::cout << "Message from client: " << message.data() << std::endl;
80
81     // Отправляем сообщение обратно клиенту
82     DWORD countOfBytesWrite;
83     if(!WriteFile(writePipeFromServerToClient, message.data(), DWORD(message.size()), &
84 countOfBytesWrite, nullptr)) {
85         std::cerr << "It's impossible to write file." << std::endl;
86         return 0x5;
87     }
88 }
89
90 std::cout << "Client terminate across few seconds." << std::endl;
91 Sleep(TERMINATE_DELAY);
92
93 CloseHandle(readPipeFromClientToServer);
94 CloseHandle(writePipeFromServerToClient);
95
96 return 0x0;
97 }

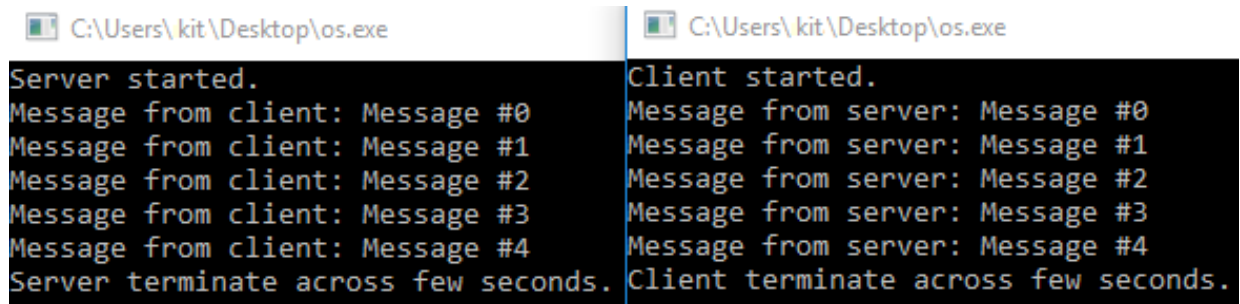
```

Клиентская часть симметрична серверной:

```
1 #include <stdio.h>
2 #include <windows.h>
3 #include <ostream>
4 #include <iostream>
5 #include <string>
6
7 // Размер буфера
8 static const int BUFFER_SIZE = 1024;
9 // Количество отсылаемых сообщений
10 static const int MESSAGES_COUNT = 5;
11 // Задержка перед завершением
12 static const int TERMINATE_DELAY = 5000;
13
14 int main() {
15     // Пишем в поток ошибок, так как ввод и вывод занят
16     std::cerr << "Client started." << std::endl;
17
18     std::string message;
19     char buffer[BUFFER_SIZE];
20
21     for(int index = 0; index < MESSAGES_COUNT; ++index) {
22         message = "Message #";
23         message += std::to_string(index);
24
25         // Отправляем сообщение на сервер
26         DWORD countOfBytesWrite;
27         if(!WriteFile(GetStdHandle(STD_OUTPUT_HANDLE), message.data(), DWORD(message.size()),
28             &countOfBytesWrite, nullptr)) {
29             std::cerr << "It's impossible to write file." << std::endl;
30             return 0x1;
31         }
32
33         ZeroMemory(buffer, sizeof(buffer));
34
35         // Принимаем ответ с сервера
36         DWORD countOfBytesRead;
37         if(!ReadFile(GetStdHandle(STD_INPUT_HANDLE), buffer, BUFFER_SIZE, &countOfBytesRead,
38             nullptr)) {
39             std::cerr << "It's impossible to read file." << std::endl;
40             return 0x2;
41         }
42
43         // Пишем в поток ошибок, так как ввод и вывод занят
44         std::cerr << "Message from server: " << buffer << std::endl;
45     }
46
47     // Пишем в поток ошибок, так как ввод и вывод занят
48     std::cerr << "Client terminate across few seconds." << std::endl;
49     Sleep(TERMINATE_DELAY);
50     return 0x0;
51 }
```



Результат обмена сообщениями:



```
C:\Users\kit\Desktop\os.exe
Server started.
Message from client: Message #0
Message from client: Message #1
Message from client: Message #2
Message from client: Message #3
Message from client: Message #4
Server terminate across few seconds.

C:\Users\kit\Desktop\os.exe
Client started.
Message from server: Message #0
Message from server: Message #1
Message from server: Message #2
Message from server: Message #3
Message from server: Message #4
Client terminate across few seconds.
```

Рис. 1.2

Сообщения были сгенерированы на клиентской стороне, отправлены на сервер. После этого сервер успешно принял сообщения и отослал их назад клиенту.

## 1.2.2 Глава 2. Именованные каналы

Именованные каналы являются дуплексными, ориентированы на обмен сообщениями и обеспечивают взаимодействие через сеть. Кроме того, один именованный канал может иметь несколько открытых дескрипторов. В сочетании с удобными, ориентированными на выполнение транзакций функциями эти возможности делают именованные каналы пригодными для создания клиент-серверных систем. Обмен данными может быть синхронным и асинхронным.

Для создания именованного канала используется функция `CreateNamedPipe`. При первом вызове функции `CreateNamedPipe` происходит создание самого именованного канала, а не просто его экземпляра. Заккрытие последнего открытого дескриптора экземпляра именованного канала приводит к уничтожению этого экземпляра (обычно существует по одному дескриптору на каждый экземпляр). Уничтожение последнего экземпляра именованного канала приводит к уничтожению самого канала, в результате чего имя канала становится вновь доступным для повторного использования.

```
HANDLE WINAPI CreateNamedPipe(
    _In_ LPCTSTR lpName,
    _In_ DWORD dwOpenMode,
    _In_ DWORD dwPipeMode,
    _In_ DWORD nMaxInstances,
    _In_ DWORD nOutBufferSize,
    _In_ DWORD nInBufferSize,
    _In_ DWORD nDefaultTimeout,
    _In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

После создания именованного канала сервер может ожидать подключения клиента, вызывая функцию `ConnectNamedPipe`.

```
BOOL WINAPI ConnectNamedPipe(
    _In_ HANDLE hNamedPipe,
    _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

Для подключения клиента к именованному каналу применяется функция `CreateFile`. С помощью функции `WaitNamedPipe` процесс может выполнять ожидание момента, когда канал Pipe будет доступен для соединения.

```
BOOL WINAPI WaitNamedPipe(
    _In_ LPCTSTR lpNamedPipeName,
    _In_ DWORD nTimeout
);
```

## 1. Программа, обеспечивающая взаимодействие процессов посредством именованных каналов

Сервер создает именованный канал для двустороннего использования и ожидает подключения программы-клиента. В цикле сервер получает сообщения от клиента и отправляет их назад:

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <tchar.h>
4 #include <iostream>
5
6 // Имя именованного канала
7 static const char* PIPE_NAME = "\\.\pipe\\$MyPipe$";
8 // Размер буфера
9 static const int BUFFER_SIZE = 1024;
10 // Сообщение для выхода
11 static const char* EXIT_MESSAGE = "@EXIT@";
12
13 int main() {
14     std::cout << "Server started." << std::endl;
15
16     // Создаем именованный канал
17     HANDLE namedPipe = CreateNamedPipe(_tcscdup(TEXT(PIPE_NAME)), PIPE_ACCESS_DUPLEX,
18     PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT, PIPE_UNLIMITED_INSTANCES,
19     BUFFER_SIZE, BUFFER_SIZE, NULL, nullptr);
20     if(namedPipe == INVALID_HANDLE_VALUE) {
21         std::cerr << "It's impossible to create pipe." << std::endl;
22         return 0x1;
23     }
24
25     // Подключаемся к именованному каналу
26     if(!ConnectNamedPipe(namedPipe, nullptr)) {
27         std::cerr << "It's impossible to connect to pipe." << std::endl;
28         return 0x2;
29     }
30
31     std::cout << "Pipe has been successfully created." << std::endl;
32
33     char buffer[BUFFER_SIZE];
34
35     while(true) {
36         ZeroMemory(buffer, sizeof(buffer));
37
38         // Получаем сообщение от клиента
39         DWORD countOfBytesRead;
40         if(!ReadFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesRead, nullptr)) {
41             std::cerr << "It's impossible to read file." << std::endl;
42             return 0x3;
43         }
44
45         if(!strcmp(buffer, EXIT_MESSAGE))
46             break;
47
48         std::cout << "Message from client: " << buffer << std::endl;
49
50         // Отправляем сообщение обратно клиенту
51         DWORD countOfBytesWrite;
52         if(!WriteFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesWrite, nullptr)) {
53             std::cerr << "It's impossible to write file." << std::endl;
54             return 0x4;
55         }
56     }
57
58     CloseHandle(namedPipe);
59     return 0x0;
60 }
```

Клиент на своей стороне открывает канал, пишет в него и читает эхо-ответ. При отправке сообщения "@EXIT@" программа завершается:

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <string>
4 #include <tchar.h>
5 #include <ostream>
6 #include <iostream>
7
8 // Имя именованного канала
9 static const char* PIPE_NAME = "\\.\pipe\\$MyPipe$";
10 // Размер буфера
11 static const int BUFFER_SIZE = 1024;
12 // Сообщение для выхода
13 static const char* EXIT_MESSAGE = "@EXIT@";
14
15 int main() {
16     std::cout << "Client started." << std::endl;
17
18     // Подключаемся к именованному каналу
19     HANDLE namedPipe = CreateFile(_tcsdup(TEXT(PIPE_NAME)), GENERIC_READ | GENERIC_WRITE,
20     NULL, nullptr, OPEN_EXISTING, NULL, nullptr);
21     if(namedPipe == INVALID_HANDLE_VALUE) {
22         std::cerr << "It's impossible to create file." << std::endl;
23         return 0x1;
24     }
25
26     std::cout << "Connection established." << std::endl;
27
28     std::string message;
29     char buffer[BUFFER_SIZE];
30
31     while(true) {
32         std::getline(std::cin, message);
33
34         // Отправляем сообщение на сервер
35         DWORD countOfBytesWrite;
36         if(!WriteFile(namedPipe, message.data(), DWORD(message.size()), &countOfBytesWrite,
37         nullptr)) {
38             std::cerr << "It's impossible to write file." << std::endl;
39             return 0x2;
40         }
41
42         if(message == EXIT_MESSAGE)
43             break;
44
45         ZeroMemory(buffer, sizeof(buffer));
46
47         // Получаем сообщение обратно от сервера
48         DWORD countOfBytesRead;
49         if(!ReadFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesRead, nullptr)) {
50             std::cerr << "It's impossible to read file." << std::endl;
51             return 0x3;
52         }
53
54         std::cout << "Message from server: " << buffer << std::endl;
55     }
56
57     CloseHandle(namedPipe);
58     return 0x0;
59 }
```

Результат клиент-серверного взаимодействия:

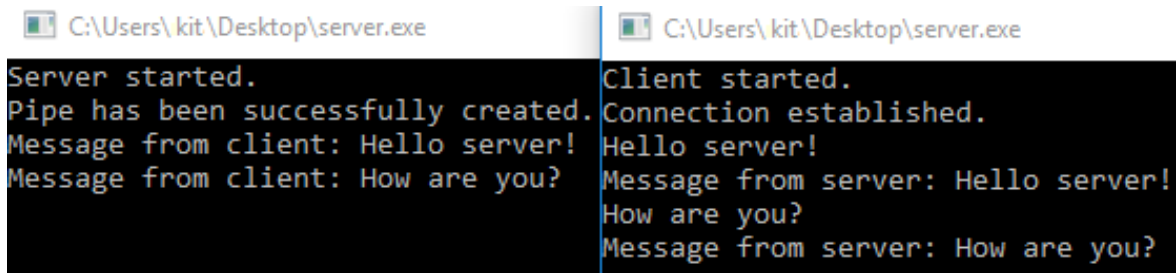


Рис. 1.3

Сообщения отправленные на сервер сразу же возвращаются назад, что соответствует корректной работе эхо-сервера.

## 2. Поддержка множества клиентов, блокировка обмена до завершения операций

Сервер, как и ранее, создает все необходимые ресурсы и переходит в состояние ожидания соединений. Именованный канал создается для чтения и записи. Передача происходит сообщениями, функции передачи и приема блокируются до их окончания:

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <tchar.h>
4 #include <iostream>
5
6 // Имя именованного канала
7 static const char* PIPE_NAME = "\\.\pipe\\$MyPipe$";
8 // Размер буфера
9 static const int BUFFER_SIZE = 1024;
10 // Максимальное количество клиентов
11 static const int CLIENT_COUNT = 10;
12 // Сообщение для выхода
13 static const char* EXIT_MESSAGE = "@EXIT@";
14
15 static bool interrupt = false;
16
17 DWORD WINAPI threadExecutor(LPVOID param);
18
19 int main() {
20     std::cout << "Server started." << std::endl;
21
22     while(!interrupt) {
23         // Создаем именованный канал
24         HANDLE namedPipe = CreateNamedPipe(_tcscdup(TEXT(PIPE_NAME)), PIPE_ACCESS_DUPLEX,
25         PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT, CLIENT_COUNT, BUFFER_SIZE,
26         BUFFER_SIZE, 5000, nullptr);
27         if(namedPipe == INVALID_HANDLE_VALUE) {
28             std::cerr << "It's impossible to create pipe." << std::endl;
29             return 0x1;
30         }
31
32         // Подключаемся к именованному каналу
33         if(!ConnectNamedPipe(namedPipe, nullptr)) {
34             std::cerr << "It's impossible to connect to pipe." << std::endl;
35             return 0x2;
36         }
37
38         std::cout << "Pipe has been successfully created." << std::endl;
39
40         if(!CreateThread(nullptr, NULL, threadExecutor, LPVOID(namedPipe), NULL, nullptr)) {
41             std::cerr << "It's impossible to create thread." << std::endl;
42             return 0x3;
43         }
44     }
45 }
```

```

42 }
43
44 std::cout << "Press \"Enter\" to exit." << std::endl;
45 std::getchar();
46
47 return 0x0;
48 }
49
50 DWORD WINAPI threadExecutor(LPVOID param) {
51     HANDLE namedPipe = HANDLE(param);
52
53     char buffer[BUFFER_SIZE];
54     while(!interrupt) {
55         ZeroMemory(buffer, sizeof(buffer));
56
57         // Получаем сообщение от клиента
58         DWORD countOfBytesRead;
59         if(!ReadFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesRead, nullptr)) {
60             std::cerr << "It's impossible to read file." << std::endl;
61             exit(0x3);
62         }
63
64         std::cout << "Message from client: " << buffer << std::endl;
65
66         // Отправляем сообщение обратно клиенту
67         DWORD countOfBytesWrite;
68         if(!WriteFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesWrite, nullptr)) {
69             std::cerr << "It's impossible to write file." << std::endl;
70             exit(0x4);
71         }
72
73         // Завершаем работу приложения
74         if(!strcmp(buffer, EXIT_MESSAGE))
75             interrupt = true;
76     }
77
78     CloseHandle(namedPipe);
79     ExitThread(NULL);
80 }

```

Клиентский код практически не изменился, за исключением добавления функции WaitNamedPipe:

```

1 #include <windows.h>
2 #include <stdio.h>
3 #include <tchar.h>
4 #include <iostream>
5 #include <string>
6
7 // Имя именованного канала
8 static const char* PIPE_NAME = "\\.\pipe\\$MyPipe$";
9 // Размер буфера
10 static const int BUFFER_SIZE = 1024;
11 // Сообщение для выхода
12 static const char* EXIT_MESSAGE = "@EXIT@";
13
14 int main() {
15     std::cout << "Client started." << std::endl;
16
17     // Ожидаем пока канал освободится
18     if(!WaitNamedPipe(_tcsdup(TEXT(PIPE_NAME)), NMPWAIT_WAIT_FOREVER)) {
19         std::cerr << "It's impossible to wait pipe." << std::endl;
20         return 0x1;
21     }
22
23     HANDLE namedPipe = CreateFile(_tcsdup(TEXT(PIPE_NAME)), GENERIC_READ | GENERIC_WRITE,
24     NULL, nullptr, OPEN_EXISTING, NULL, nullptr);
25     if(namedPipe == INVALID_HANDLE_VALUE) {

```

```

25     std::cerr << "It's impossible to create file." << std::endl;
26     return 0x2;
27 }
28
29 std::cout << "Connection established." << std::endl;
30
31 std::string message;
32 char buffer[BUFFER_SIZE];
33
34 while(true) {
35     std::getline(std::cin, message);
36
37     // Отправляем сообщение на сервер
38     DWORD countOfBytesWrite;
39     if(!WriteFile(namedPipe, message.data(), DWORD(message.size()), &countOfBytesWrite,
40     nullptr)) {
41         std::cerr << "It's impossible to write file." << std::endl;
42         return 0x3;
43     }
44
45     // Завершаем работу приложения
46     if(message == EXIT_MESSAGE)
47         break;
48
49     ZeroMemory(buffer, sizeof(buffer));
50
51     // Получаем сообщение обратно от сервера
52     DWORD countOfBytesRead;
53     if(!ReadFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesRead, nullptr)) {
54         std::cerr << "It's impossible to read file." << std::endl;
55         return 0x4;
56     }
57
58     std::cout << "Message from server: " << buffer << std::endl;
59 }
60
61 CloseHandle(namedPipe);
62
63 std::cout << "Press \"Enter\" to exit." << std::endl;
64 std::getchar();
65
66 return 0x0;
67 }

```

Результат клиент-серверного взаимодействия с множеством клиентов:

```

C:\Users\kit\Desktop\server.exe
Server started.
Pipe has been successfully created.
Message from client: Hello from client 1!
Pipe has been successfully created.
Message from client: Hello from client 2!

C:\Users\kit\Desktop\client.exe
Client started.
Connection established.
Hello from client 1!
Message from server: Hello from client 1!

C:\Users\kit\Desktop\client.exe
Client started.
Connection established.
Hello from client 2!
Message from server: Hello from client 2!

```

Рис. 1.4

Был запущен сервер и два клиентских окна, после этого каждый клиент послал по сообщению, которые сервер успешно обработал.

### 3. Модификация для сетевого обмена информацией

Для совместной работы компьютеры нужно подсоединить к одной домашней группе. Так же необходимо установить поле DACL защиты объекта в NULL. Параметры защиты именованного канала задаются с помощью структуры SECURITY\_ATTRIBUTES, которая указывается последним параметром в функции CreateNamedPipe.

Модификация сервера для работы по сети:

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <tchar.h>
4 #include <iostream>
5
6 // Имя именованного канала
7 static const char* PIPE_NAME = "\\\\.\\pipe\\$MyPipe$";
8 // Размер буфера
9 static const int BUFFER_SIZE = 1024;
10 // Максимальное количество клиентов
11 static const int CLIENT_COUNT = 10;
12 // Сообщение для выхода
13 static const char* EXIT_MESSAGE = "@EXIT@";
14
15 static bool interrupt = false;
16
17 DWORD WINAPI threadExecutor(LPVOID param);
18
19 int main() {
20     std::cout << "Server started." << std::endl;
21
22     while(!interrupt) {
23         // Инициализация дескриптора значениями по умолчанию
24         SECURITY_DESCRIPTOR securityDescriptor;
25         if(!InitializeSecurityDescriptor(&securityDescriptor, SECURITY_DESCRIPTOR_REVISION))
26         {
27             std::cerr << "It's impossible to initialize security descriptor." << std::endl;
28             return 0x1;
29         }
30
31         // Обнуление поля DACL
32         if(!SetSecurityDescriptorDacl(&securityDescriptor, TRUE, nullptr, FALSE)) {
33             std::cerr << "It's impossible to set security descriptor." << std::endl;
34             return 0x2;
35         }
36
37         // Установка атрибутов для именованного канала
38         SECURITY_ATTRIBUTES securityAttributes;
39         ZeroMemory(&securityAttributes, sizeof(securityAttributes));
40         securityAttributes.nLength = sizeof(SECURITY_ATTRIBUTES);
41         securityAttributes.lpSecurityDescriptor = &securityDescriptor;
42         securityAttributes.bInheritHandle = FALSE;
43
44         // Создаем именованный канал
45         HANDLE namedPipe = CreateNamedPipe(_tcsdup(TEXT(PIPE_NAME)), PIPE_ACCESS_DUPLEX,
46             PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT, CLIENT_COUNT, BUFFER_SIZE,
47             BUFFER_SIZE, 5000, &securityAttributes);
48         if(namedPipe == INVALID_HANDLE_VALUE) {
49             std::cerr << "It's impossible to create pipe." << std::endl;
50             return 0x3;
51         }
52
53         // Подключаемся к именованному каналу
54         if(!ConnectNamedPipe(namedPipe, nullptr)) {
55             std::cerr << "It's impossible to connect to pipe." << std::endl;
56             return 0x4;
57         }
58
59         std::cout << "Pipe has been successfully created." << std::endl;
```

```

57     if(!CreateThread(nullptr, NULL, threadExecutor, LPVOID(namedPipe), NULL, nullptr)) {
58         std::cerr << "It's impossible to create thread." << std::endl;
59         return 0x5;
60     }
61 }
62 }
63
64 std::cout << "Press \"Enter\" to exit." << std::endl;
65 std::getchar();
66
67 return 0x0;
68 }
69
70 DWORD WINAPI threadExecutor(LPVOID param) {
71     HANDLE namedPipe = HANDLE(param);
72
73     char buffer[BUFFER_SIZE];
74     while(!interrupt) {
75         ZeroMemory(buffer, sizeof(buffer));
76
77         // Получаем сообщение от клиента
78         DWORD countOfBytesRead;
79         if(!ReadFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesRead, nullptr)) {
80             std::cerr << "It's impossible to read file." << std::endl;
81             exit(0x6);
82         }
83
84         std::cout << "Message from client: " << buffer << std::endl;
85
86         // Отправляем сообщение обратно клиенту
87         DWORD countOfBytesWrite;
88         if(!WriteFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesWrite, nullptr)) {
89             std::cerr << "It's impossible to write file." << std::endl;
90             exit(0x7);
91         }
92
93         // Завершаем работу приложения
94         if(!strcmp(buffer, EXIT_MESSAGE))
95             interrupt = true;
96     }
97
98     CloseHandle(namedPipe);
99     ExitThread(NULL);
100 }

```

Модификация клиента для работы по сети:

```

1 #include <windows.h>
2 #include <stdio.h>
3 #include <tchar.h>
4 #include <iostream>
5 #include <string>
6
7 // Имя именованного канала
8 static const char* PIPE_NAME = "\\\\.\\pipe\\$MyPipe$";
9 // Размер буфера
10 static const int BUFFER_SIZE = 1024;
11 // Сообщение для выхода
12 static const char* EXIT_MESSAGE = "@EXIT@";
13
14 int main() {
15     std::cout << "Client started." << std::endl;
16
17     // Ожидаем пока канал освободится
18     if(!WaitNamedPipe(_tcsdup(TEXT(PIPE_NAME)), NMPWAIT_WAIT_FOREVER)) {
19         std::cerr << "It's impossible to wait pipe." << std::endl;
20         return 0x1;

```



```

21 }
22
23 HANDLE namedPipe = CreateFile(_tcsdup(TEXT(PIPE_NAME)), GENERIC_READ | GENERIC_WRITE,
24 NULL, nullptr, OPEN_EXISTING, NULL, nullptr);
25 if(namedPipe == INVALID_HANDLE_VALUE) {
26     std::cerr << "It's impossible to create file." << std::endl;
27     return 0x2;
28 }
29
30 std::cout << "Connection established." << std::endl;
31
32 std::string message;
33 char buffer[BUFFER_SIZE];
34
35 while(true) {
36     std::getline(std::cin, message);
37
38     // Отправляем сообщение на сервер
39     DWORD countOfBytesWrite;
40     if(!WriteFile(namedPipe, message.data(), DWORD(message.size()), &countOfBytesWrite,
41     nullptr)) {
42         std::cerr << "It's impossible to write file." << std::endl;
43         return 0x3;
44     }
45
46     // Завершаем работу приложения
47     if(message == EXIT_MESSAGE)
48         break;
49
50     ZeroMemory(buffer, sizeof(buffer));
51
52     // Получаем сообщение обратно от сервера
53     DWORD countOfBytesRead;
54     if(!ReadFile(namedPipe, buffer, BUFFER_SIZE, &countOfBytesRead, nullptr)) {
55         std::cerr << "It's impossible to read file." << std::endl;
56         return 0x4;
57     }
58
59     std::cout << "Message from server: " << buffer << std::endl;
60 }
61
62 CloseHandle(namedPipe);
63
64 std::cout << "Press \"Enter\" to exit." << std::endl;
65 std::getchar();
66
67 return 0x0;
68 }

```

Сетевые параметры клиента:

```

DNS-суффикс подключения . . . . . :
Локальный IPv6-адрес канала . . . : fe80::b5e4:68bb:4d4c:fad7%10
IPv4-адрес. . . . . : 192.168.0.106
Маска подсети . . . . . : 255.255.255.0
Основной шлюз. . . . . : 192.168.0.1

```

Рис. 1.5

Сетевые параметры сервера:

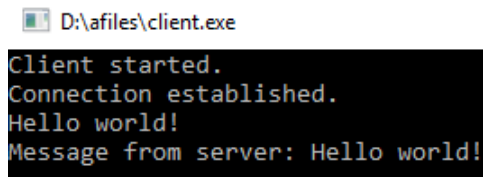
```

DNS-суффикс подключения . . . . . :
Локальный IPv6-адрес канала . . . : fe80::b5e4:68bb:4d4c:fad7%10
IPv4-адрес. . . . . : 192.168.0.105
Маска подсети . . . . . : 255.255.255.0
Основной шлюз. . . . . : 192.168.0.1

```

Рис. 1.6

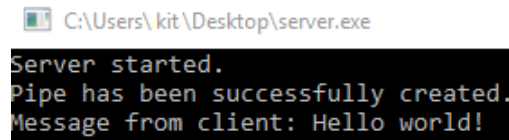
Отправка клиентского сообщения:



```
D:\afiles\client.exe
Client started.
Connection established.
Hello world!
Message from server: Hello world!
```

Рис. 1.7

Прием сообщения удаленным сервером:



```
C:\Users\kit\Desktop\server.exe
Server started.
Pipe has been successfully created.
Message from client: Hello world!
```

Рис. 1.8

Компьютеры были объединены в общую домашнюю группу, что позволило им обмениваться сообщениями удаленно при помощи именованных каналов.

### 1.2.3 Глава 3. Сокеты

WinSock или Windows socket - это интерфейс программного программирования (API) созданный для реализации приложений в сети на основе протокола TCP/IP. Для работы используется WSOCK32.DLL. Windows socket разрабатывался на основе интерфейса Беркли для UNIX, но к ним добавлены функции поддержки событий Windows.

Есть две версии библиотеки WinSock:

- *WinSock 1.1* - поддержка только TCP/IP.
- *WinSock 2.0* - Поддержка дополнительного программного обеспечения.

Спецификация WinSock разделяет функции на три типа:

- Функции Беркли.
- Информационные функции (получение информации о наименовании доменов, службах, протоколах Internet).
- Расширения Windows для функций Беркли.

#### 1 - 3. Многопоточный сервер, обслуживающая множество клиентов, поддерживающий сетевой обмен информацией

Разработаем программу *TCP* сервера, который в бесконечном цикле ожидает подключения клиентов, создает для каждого из них новый поток, принимает сообщения клиента и отправляет их назад. Также реализована обработка сигнала прерывания для корректного завершения работы всех потоков и закрытия сокетов:

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <thread>
6 #include <map>
7 #include <winsock2.h>
8 #include <ws2tcpip.h>
9 #include <iostream>
10 #include <string>
11
12 #pragma comment (lib , "Ws2_32.lib")
```

```

13
14 static const char* PORT = "65100";
15
16 static const int BACKLOG = 5;
17 static const int BUFFER_SIZE = 1000;
18 static const int FLAGS = 0;
19
20 // Коллекция для хранения пар значений:
21 // сокет + идентификатор потока
22 std::map<SOCKET, std::shared_ptr<std::thread>> threads;
23 // Серверный сокет
24 SOCKET serverSocket;
25
26 // Обработчик сигнала прерывания корректное( завершение приложения)
27 void signalHandler(int sig);
28 // Обработчик клиентского потока
29 void* clientExecutor(void* clientSocket);
30 // Функция считывания строки символов с клиента
31 int readLine(SOCKET socket, char* buffer, int bufferSize, int flags);
32 // Функция отправки строки символов клиенту
33 int sendLine(SOCKET socket, char* buffer, int flags);
34 // Корректное закрытие сокета
35 void clearSocket(SOCKET socket);
36 // Завершение работы клиентского потока
37 void destroyClient(SOCKET socket);
38
39 int main(int argc, char** argv) {
40     std::string port = PORT;
41     if(argc < 2)
42         std::cout << "Using default port: " << port << "." << std::endl;
43     else
44         port = argv[1];
45
46     WSADATA wsaData;
47     int wsaStartup = WSASStartup(MAKEWORD(2, 2), &wsaData);
48     if(wsaStartup != 0) {
49         std::cerr << "It's impossible to startup wsa." << std::endl;
50         return 0x1;
51     }
52
53     struct addrinfo hints;
54     ZeroMemory(&hints, sizeof(hints));
55     hints.ai_family = AF_INET;
56     hints.ai_socktype = SOCK_STREAM;
57     hints.ai_protocol = IPPROTO_TCP;
58     hints.ai_flags = AI_PASSIVE;
59
60     struct addrinfo* result = nullptr;
61     int addressInfo = getaddrinfo(nullptr, port.data(), &hints, &result);
62     if(addressInfo != 0) {
63         std::cerr << "It's impossible to get address info." << std::endl;
64         WSACleanup();
65         return 0x2;
66     }
67
68     // Создание серверного сокета
69     serverSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
70     if(serverSocket == INVALID_SOCKET) {
71         std::cerr << "It's impossible to create socket." << std::endl;
72         WSACleanup();
73         return 0x3;
74     }
75
76     std::cout << "Server socket " << serverSocket << " created." << std::endl;
77
78     // Биндим сервер на определенный адрес

```

```

79  int serverBind = bind(serverSocket, result->ai_addr, int(result->ai_addrlen));
80  if(serverBind == SOCKET_ERROR) {
81      std::cerr << "It's impossible to bind socket." << std::endl;
82      closesocket(serverSocket);
83      WSACleanup();
84      return 0x4;
85  }
86
87  freeaddrinfo(result);
88
89  // Слушаем сокет
90  int serverListen = listen(serverSocket, BACKLOG);
91  if(serverListen == SOCKET_ERROR) {
92      std::cerr << "It's impossible to listen socket." << std::endl;
93      closesocket(serverSocket);
94      WSACleanup();
95      return 0x5;
96  }
97
98  // Обработка прерывания для корректного завершения приложения
99  signal(SIGINT, signalHandler);
100
101  std::cout << "Wait clients." << std::endl;
102
103  while(true) {
104      // Ждем подключения клиентов
105      SOCKET clientSocket = accept(serverSocket, nullptr, nullptr);
106
107      if(clientSocket == INVALID_SOCKET)
108          continue;
109
110      // Пробуем создать поток обработки клиентских сообщений
111      auto bindThread = std::bind(&clientExecutor, std::placeholders::_1);
112      auto sharedThread = std::make_shared<std::thread>(bindThread, &clientSocket);
113
114      // Добавляем в коллекцию пару значений: сокет + идентификатор потока
115      threads.insert(std::pair<SOCKET, std::shared_ptr<std::thread>>(clientSocket,
116      sharedThread));
117  }
118
119  void signalHandler(int sig) {
120      // Для всех элементов коллекции
121      for(auto& current: threads) {
122          std::cout << "Try to finish client with socket " << current.first << "." << std::endl;
123          ;
124          // Закрываем клиентские сокеты
125          clearSocket(current.first);
126          std::cout << "Client socket " << current.first << " closed." << std::endl;
127      }
128
129      // Закрываем серверный сокет
130      clearSocket(serverSocket);
131      std::cout << "Server socket " << serverSocket << " closed." << std::endl;
132
133      WSACleanup();
134
135      exit(0x0);
136  }
137
138  void* clientExecutor(void* socket) {
139      SOCKET clientSocket = *((SOCKET*) socket);
140
141      std::cout << "Client thread with socket " << clientSocket << " created." << std::endl;
142
143      char buffer[BUFFER_SIZE];

```

```

143 while(true) {
144     // Ожидаем прибытия строки
145     int result = readLine(clientSocket, buffer, BUFFER_SIZE, FLAGS);
146     if(result < 0) {
147         destroyClient(clientSocket);
148         break;
149     }
150
151     if(strlen(buffer) <= 1) {
152         destroyClient(clientSocket);
153         break;
154     }
155
156     std::cout << "Client message: " << buffer << std::endl;
157
158     // Отправляем строку назад
159     result = sendLine(clientSocket, buffer, FLAGS);
160     if(result < 0) {
161         destroyClient(clientSocket);
162         break;
163     }
164 }
165
166 return nullptr;
167 }
168
169 int readLine(SOCKET socket, char* buffer, int bufferSize, int flags) {
170     // Очищаем буфер
171     ZeroMemory(buffer, bufferSize);
172
173     char resolvedSymbol = ' ';
174     for(int index = 0; index < BUFFER_SIZE; ++index) {
175         // Считываем по одному символу
176         int readSize = recv(socket, &resolvedSymbol, 1, flags);
177         if(readSize <= 0)
178             return -1;
179
180         if(resolvedSymbol == '\n')
181             break;
182
183         if(resolvedSymbol != '\r')
184             buffer[index] = resolvedSymbol;
185     }
186
187     return 0x0;
188 }
189
190 int sendLine(SOCKET socket, char* buffer, int flags) {
191     auto length = strlen(buffer);
192
193     // Перед отправкой сообщения добавляем в конец перевод строки
194     if(length == 0)
195         return -1;
196
197     if(buffer[length - 1] != '\n') {
198         if(length >= BUFFER_SIZE)
199             return -1;
200
201         buffer[length] = '\n';
202     }
203
204     length = strlen(buffer);
205
206     // Отправляем строку клиенту
207     int result = send(socket, buffer, int(length), flags);
208     return result;

```

```

209 }
210
211 void clearSocket(SOCKET socket) {
212     // Завершение работы сокета
213     int socketShutdown = shutdown(socket, SD_BOTH);
214     if(socketShutdown != 0)
215         std::cerr << "It's impossible to shutdown socket." << std::endl;
216
217     // Закрытие сокета
218     closesocket(socket);
219 }
220
221 void destroyClient(SOCKET socket) {
222     std::cout << "It's impossible to receive message from client or send message to client."
223         << std::endl;
224     // Завершение работы сокета
225     clearSocket(socket);
226     // Отсоединение потока для удаления из коллекции
227     threads.at(socket)->detach();
228     // Удаление пары значений из коллекции по ключу
229     threads.erase(socket);
230
231     std::cout << "Client socket " << socket << " closed." << std::endl;
232 }

```

Клиент подключается к серверу, считывает сообщения из консоли и отправляет их серверу, после этого ожидает ответного сообщения сервера. Если было введено пустое сообщение, то клиент завершает работу:

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <signal.h>
5 #include <winsock2.h>
6 #include <ws2tcpip.h>
7 #include <string>
8 #include <iostream>
9
10 #pragma comment (lib, "Ws2_32.lib")
11
12 static const char* PORT = "65100";
13 static const char* IP = "127.0.0.1";
14
15 static const int BACKLOG = 5;
16 static const int BUFFER_SIZE = 1000;
17 static const int FLAGS = 0;
18
19 // Клиентский сокет
20 SOCKET clientSocket;
21
22 // Обработчик сигнала прерывания корректное( завершение приложения)
23 void signalHandler(int sig);
24 // Функция считывания строки символов с сервера
25 int readLine(SOCKET socket, char* buffer, int bufferSize, int flags);
26 // Функция отправки строки символов серверу
27 int sendLine(SOCKET socket, std::string message, int flags);
28 // Корректное закрытие сокета
29 void clearSocket(SOCKET socket);
30
31 int main(int argc, char** argv) {
32     std::string port = PORT;
33     std::string ip = IP;
34
35     if(argc < 3) {
36         std::cout << "Using default ip: " << ip << "." << std::endl;
37         std::cout << "Using default port: " << port << "." << std::endl;
38     }
39     else {

```

```

40     ip = argv[1];
41     port = argv[2];
42 }
43
44 WSADATA wsaData;
45 int wsaStartup = WSAStartup(MAKEWORD(2, 2), &wsaData);
46 if(wsaStartup != 0) {
47     std::cerr << "It's impossible to startup wsa." << std::endl;
48     return 0x1;
49 }
50
51 struct addrinfo hints;
52 ZeroMemory(&hints, sizeof(hints));
53 hints.ai_family = AF_INET;
54 hints.ai_socktype = SOCK_STREAM;
55 hints.ai_protocol = IPPROTO_TCP;
56
57 struct addrinfo* addressResult = nullptr;
58 int addressInfo = getaddrinfo(ip.data(), port.data(), &hints, &addressResult);
59 if(addressInfo != 0) {
60     std::cerr << "It's impossible to get address info." << std::endl;
61     WSACleanup();
62     return 0x2;
63 }
64
65 for(auto currentPtr = addressResult; currentPtr != nullptr; currentPtr = currentPtr->
66     ai_next) {
67     clientSocket = socket(currentPtr->ai_family, currentPtr->ai_socktype, currentPtr->
68     ai_protocol);
69     if(clientSocket == INVALID_SOCKET) {
70         std::cerr << "It's impossible to create socket." << std::endl;
71         WSACleanup();
72         return 0x3;
73     }
74
75     auto serverConnection = connect(clientSocket, currentPtr->ai_addr, int(currentPtr->
76     ai_addrlen));
77     if(serverConnection != SOCKET_ERROR)
78         break;
79
80     closesocket(clientSocket);
81     clientSocket = INVALID_SOCKET;
82 }
83
84 freeaddrinfo(addressResult);
85
86 if(clientSocket == INVALID_SOCKET) {
87     std::cerr << "It's impossible to create connection." << std::endl;
88     WSACleanup();
89     return 0x4;
90 }
91
92 std::cout << "Socket has been successfully created." << std::endl
93     << "Connection established." << std::endl;
94
95 // Обработка прерывания для корректного завершения приложения
96 signal(SIGINT, signalHandler);
97
98 std::string message;
99 char buffer[BUFFER_SIZE];
100
101 while(true) {
102     message.clear();
103     std::getline(std::cin, message);
104
105     if(message.empty() || (message.size() == 1 && message.back() == '\n')) {

```

```

103     std::cout << "Empty message, try to close client socket." << std::endl;
104     clearSocket(clientSocket);
105     WSACleanup();
106     return 0x0;
107 }
108
109 // Отправляем строку на сервер
110 int result = sendLine(clientSocket, message, FLAGS);
111 if(result < 0) {
112     std::cout << "It's impossible to send message, try to close client socket." << std
113     ::endl;
114     clearSocket(clientSocket);
115     WSACleanup();
116     return 0x0;
117 }
118
119 // Ожидаем ответ строки
120 result = readLine(clientSocket, buffer, BUFFER_SIZE, FLAGS);
121 if(result < 0) {
122     std::cout << "It's impossible to receive message, try to close client socket." <<
123     std::endl;
124     clearSocket(clientSocket);
125     WSACleanup();
126     return 0x0;
127 }
128
129 std::cout << "Server message: " << buffer << std::endl;
130 }
131
132 void signalHandler(int sig) {
133     // Закрываем клиентский сокет
134     clearSocket(clientSocket);
135     std::cout << "Client socket " << clientSocket << " closed." << std::endl;
136
137     WSACleanup();
138
139     exit(0x0);
140 }
141
142 int readLine(SOCKET socket, char* buffer, int bufferSize, int flags) {
143     // Очищаем буфер
144     ZeroMemory(buffer, bufferSize);
145
146     char resolvedSymbol = ' ';
147     for(int index = 0; index < BUFFER_SIZE; ++index) {
148         // Считываем по одному символу
149         int readSize = recv(socket, &resolvedSymbol, 1, flags);
150         if(readSize <= 0)
151             return -1;
152
153         if(resolvedSymbol == '\n')
154             break;
155
156         if(resolvedSymbol != '\r')
157             buffer[index] = resolvedSymbol;
158     }
159
160     return 0x0;
161 }
162
163 int sendLine(SOCKET socket, std::string message, int flags) {
164     // Перед отправкой сообщения добавляем в конец перевод строки
165     if(message.empty())
166         return -1;

```



```

167 if(message.back() != '\n')
168     message += '\n';
169
170 // Отправляем строку на сервер
171 return send(socket, message.data(), int(message.size()), flags);
172 }
173
174 void clearSocket(SOCKET socket) {
175     // Завершение работы сокета
176     int socketShutdown = shutdown(socket, SD_BOTH);
177     if(socketShutdown == SOCKET_ERROR)
178         std::cerr << "It's impossible to shutdown socket." << std::endl;
179
180     // Закрытие сокета
181     int socketClose = closesocket(socket);
182 }

```

Результат одновременной локальной работы нескольких клиентов:

```

C:\Users\kit\Desktop\server.exe
Using default port: 65100.
Server socket 164 created.
Wait clients.
Client thread with socket 168 created.
Client message: Hello server!
Client message: How are you?
Client thread with socket 184 created.
Client message: I'm the second client!

C:\Users\kit\Desktop\client.exe
Using default ip: 127.0.0.1.
Using default port: 65100.
Socket has been successfully created.
Connection established.
Hello server!
Server message: Hello server!
How are you?
Server message: How are you?

C:\Users\kit\Desktop\client.exe
Using default ip: 127.0.0.1.
Using default port: 65100.
Socket has been successfully created.
Connection established.
I'm the second client!
Server message: I'm the second client!

```

Рис. 1.9

Клиенты успешно установили соединение с сервером, а также осуществили обмен сообщениями.

Однако, в основном сокеты используют для сетевого обмена пакетами. Вышеописанному клиенту и серверу не нужны дополнительные модификации для сетевого обмена. Однако необходимо узнать ip адрес компьютера с сервером и указать его в качестве аргумента командной строки клиента.

Сетевые параметры клиента:

```

DNS-суффикс подключения . . . . . :
Локальный IPv6-адрес канала . . . : fe80::b5e4:68bb:4d4c:fad7%10
IPv4-адрес. . . . . : 192.168.0.106
Маска подсети . . . . . : 255.255.255.0
Основной шлюз. . . . . : 192.168.0.1

```

Рис. 1.10

Сетевые параметры сервера:

```

DNS-суффикс подключения . . . . . :
Локальный IPv6-адрес канала . . . : fe80::b5e4:68bb:4d4c:fad7%10
IPv4-адрес. . . . . : 192.168.0.105
Маска подсети . . . . . : 255.255.255.0
Основной шлюз. . . . . : 192.168.0.1

```

Рис. 1.11

Отправка клиентского сообщения:

```
Администратор: C:\Windows\system32\cmd.exe - client.exe 192.168.0.106 65100

C:\Users\kit\Desktop>client.exe 192.168.0.106 65100
Socket has been successfully created.
Connection established.
Hello server!
Server message: Hello server!
Connection established!!!
Server message: Connection established!!!
```

Рис. 1.12

Прием сообщения удаленным сервером:

```
C:\Users\kit\Desktop>server.exe

Using default port: 65100.
Server socket 156 created.
Wait clients.
Client thread with socket 164 created.
Client message: Hello server!
Client message: Connection established!!!
```

Рис. 1.13

В первую очередь был получен внутренний ip адрес серверного компьютера. После этого был он был указан в качестве аргумента командной строки для клиента. После этого был осуществлен обмен сообщениями.

#### 4. Обмен сообщениями между разными операционными системами

Была реализована сетевая передача сообщений между разными операционными системами. Серверное приложение было запущено на Windows 10 (полные характеристики системы приведены в начале работы), клиентское приложение было запущено на Ubuntu 16.1 (клиентское приложение и система взяты из лабораторной работы "Средства межпроцессорного взаимодействия Linux").

Клиентское приложение для Linux:

```
1 #include <arpa/inet.h>
2 #include <netinet/in.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include <stdlib.h>
6 #include <unistd.h>
7 #include <sys/types.h>
8 #include <signal.h>
9
10 #define PORT 65100
11 #define IP "127.0.0.1"
12
13 #define BACKLOG 5
14 #define BUFFER_SIZE 1000
15 #define IP_SIZE 16
16 #define FLAGS 0
17
18 // Клиентский сокет
19 int clientSocket;
20
21 // Обработчик сигнала прерывания корректное( завершение приложения)
22 void signalHandler(int sig);
23 // Функция считывания строки символов с сервера
24 int readLine(int socket, char* buffer, int bufferSize, int flags);
25 // Функция отправки строки символов серверу
26 int sendLine(int socket, char* buffer, int flags);
27 // Корректное закрытие сокета
28 void closeSocket(int socket);
```

```

29
30 int main(int argc, char** argv) {
31     int port = PORT;
32     char ip[IP_SIZE];
33
34     strcpy(ip, IP);
35     if(argc < 3) {
36         printf("Using default ip: %s.\n", ip);
37         printf("Using default port: %d.\n", port);
38     }
39     else {
40         strcpy(ip, argv[1]);
41         port = atoi(argv[2]);
42     }
43
44     // Создание клиентского сокета
45     clientSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
46     if(clientSocket < 0) {
47         perror("It's impossible to create socket");
48         return 0x1;
49     }
50
51     printf("Client socket %d created.\n", clientSocket);
52
53     // Структура, задающая адресные характеристики
54     struct sockaddr_in info;
55     info.sin_family = AF_INET;
56     info.sin_port = htons(port);
57     info.sin_addr.s_addr = inet_addr(ip);
58
59     // Подключение к серверу
60     int clientConnect = connect(clientSocket, (struct sockaddr *) &info, sizeof(info));
61     if(clientConnect) {
62         perror("It's impossible to connect");
63         return 0x2;
64     }
65
66     printf("Connection established.\n");
67
68     // Обработка прерывания для корректного завершения приложения
69     signal(SIGINT, signalHandler);
70
71     printf("Ready to send messages.\n");
72
73     char buffer[BUFFER_SIZE];
74     while(1) {
75         bzero(buffer, BUFFER_SIZE);
76         fgets(buffer, BUFFER_SIZE, stdin);
77
78         if(strlen(buffer) == 0 || buffer[0] == '\n') {
79             printf("Empty message, try to close client socket.\n");
80             closeSocket(clientSocket);
81             return 0x0;
82         }
83
84         // Отправляем строку на сервер
85         int result = sendLine(clientSocket, buffer, FLAGS);
86         if(result < 0) {
87             printf("It's impossible to send message, try to close client socket.\n");
88             closeSocket(clientSocket);
89             return 0x0;
90         }
91
92         // Ожидаем ответ строки
93         result = readLine(clientSocket, buffer, BUFFER_SIZE, FLAGS);
94         if(result < 0) {

```

```

95     printf("It's impossible to receive message, try to close client socket.\n");
96     closeSocket(clientSocket);
97     return 0x0;
98 }
99
100     printf("Server message: %s\n", buffer);
101 }
102
103     return 0x0;
104 }
105
106 void signalHandler(int sig) {
107     // Закрываем клиентский сокет
108     closeSocket(clientSocket);
109     printf("Client socket %d closed.\n", clientSocket);
110
111     exit(0x0);
112 }
113
114 int readLine(int socket, char* buffer, int bufferSize, int flags) {
115     // Очищаем буфер
116     bzero(buffer, bufferSize);
117
118     char resolvedSymbol = ' ';
119     for(int index = 0; index < BUFFER_SIZE; ++index) {
120         // Считываем по одному символу
121         int readSize = recv(socket, &resolvedSymbol, 1, flags);
122         if(readSize <= 0)
123             return -1;
124         else if(resolvedSymbol == '\n')
125             break;
126         else if(resolvedSymbol != '\r')
127             buffer[index] = resolvedSymbol;
128     }
129
130     return 0x0;
131 }
132
133 int sendLine(int socket, char* buffer, int flags) {
134     unsigned int length = strlen(buffer);
135
136     // Перед отправкой сообщения добавляем в конец перевод строки
137     if(length == 0)
138         return -1;
139     else if(buffer[length - 1] != '\n') {
140         if(length >= BUFFER_SIZE)
141             return -1;
142         else
143             buffer[length] = '\n';
144     }
145
146     length = strlen(buffer);
147
148     // Отправляем строку серверу
149     int result = send(socket, buffer, length, flags);
150     return result;
151 }
152
153 void closeSocket(int socket) {
154     // Завершение работы сокета
155     int socketShutdown = shutdown(socket, SHUT_RDWR);
156     if(socketShutdown != 0)
157         perror("It's impossible to shutdown socket");
158
159     // Закрытие сокета
160 }

```

```

161 int socketClose = close(socket);
162     if(socketClose != 0)
163         perror("It's impossible to close socket");
164 }

```

Сетевые параметры клиента:

```

wlp2s0    Link encap:Ethernet  HWaddr 74:de:2b:64:22:23
          inet addr:192.168.0.105  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr: fe80::47b3:2637:b9d2:ef23/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4109 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1737 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4455873 (4.4 MB)  TX bytes:190204 (190.2 KB)

```

Рис. 1.14

Результат работы клиента:

```

kit@kit:~/Desktop/ezsem/OS/4/report/listings$ ./client 192.168.0.106 65
100
Client socket 3 created.
Connection established.
Ready to send messages.
Hello from ubuntu!
Server message: Hello from ubuntu!

```

Рис. 1.15

Результат работы сервера:

```

C:\Users\kit\Desktop\server.exe
Using default port: 65100.
Server socket 156 created.
Wait clients.
Client thread with socket 160 created.
Client message: Hello from ubuntu!

```

Рис. 1.16

Сетевой обмен был успешно организован, никаких проблем при реализации не возникло.

## 5. Использование портов завершения при большом количестве клиентов

Порт завершения представляет собой специальный механизм в составе ОС, с помощью которого приложение использует объединение нескольких потоков, предназначенных единственно для цели обработки асинхронных операций ввода/вывода с перекрытием. Для функционирования этой модели необходимо создание специального программного объекта ядра системы, который и был назван "порт завершения". Это осуществляется с помощью функции `CreateIoCompletionPort`, которая ассоциирует этот объект с одним или несколькими файловыми дескрипторами и который будет управлять перекрывающимися I/O операциями, используя определенное количество потоков для обслуживания завершенных запросов.

```

HANDLE WINAPI CreateIoCompletionPort(
    _In_      HANDLE      FileHandle,
    _In_opt_  HANDLE      ExistingCompletionPort,
    _In_      ULONG_PTR   CompletionKey,
    _In_      DWORD       NumberOfConcurrentThreads
);

```

Реализация сервера, использующего порт завершения:

```

1 #define _WINSOCK_DEPRECATED_NO_WARNINGS 0;
2
3 #include <stdio.h>
4 #include <string.h>

```

```

5 #include <stdlib.h>
6 #include <signal.h>
7 #include <thread>
8 #include <map>
9 #include <winsock2.h>
10 #include <ws2tcpip.h>
11 #include <iostream>
12 #include <string>
13
14 #pragma comment (lib , "Ws2_32.lib")
15
16 // Порт
17 static const char* PORT = "65100";
18
19 // Количество обрабатывающих потоков
20 static const int COUNT_OF_THREADS = 8;
21 // Размер буфера для передачи
22 static const int BUFFER_SIZE = 1024;
23 // Флаги
24 static const int FLAGS = 0;
25
26 // Структура посылки
27 struct OverlappedConnection: public OVERLAPPED {
28     int client_number;
29     SOCKET socket_handle;
30     char* buffer;
31
32     enum {
33         op_type_send,
34         op_type_recv
35     } op_type;
36 };
37
38 // Список всех клиентских соединений для корректного завершения их работы
39 std::map<SOCKET, OverlappedConnection*> clients;
40 // Серверный сокет
41 SOCKET serverSocket;
42
43 // Обработчик сигнала прерывания корректное( завершение приложения)
44 void signalHandler(int sig);
45 // Закрывает все соединения и завершает работу сервера
46 void closeClientConnection(SOCKET clientSocket);
47 // Закрывает соединение с конкретным клиентом
48 void closeAllConnections();
49
50 // Функция потока сервера для обслуживания порта завершения
51 DWORD WINAPI threadExecutor(HANDLE ioPort);
52
53 int main(int argc, char** argv) {
54     std::string port = PORT;
55     if(argc < 2)
56         std::cout << "Using default port: " << port << "." << std::endl;
57     else
58         port = argv[1];
59
60     // Инициализация библиотеки
61     WSADATA wsaData;
62     int wsaStartup = WSAStartup(MAKEWORD(2, 2), &wsaData);
63     if(wsaStartup == SOCKET_ERROR) {
64         std::cerr << "It's impossible to startup wsa." << std::endl;
65         return 0x1;
66     }
67
68     // Создаем порт завершения
69     HANDLE ioPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, nullptr, NULL, NULL);
70     if(!ioPort) {

```

```

71     std::cerr << "It's impossible to create competition port." << std::endl;
72     WSACleanup();
73     return 0x2;
74 }
75
76 // Задаем параметры прослушивающего сокета сервера
77 serverSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, nullptr, NULL,
78     WSA_FLAG_OVERLAPPED);
79 if(serverSocket == INVALID_SOCKET) {
80     std::cerr << "It's impossible to create socket." << std::endl;
81     WSACleanup();
82     return 0x3;
83 }
84
85 std::cout << "Server socket " << serverSocket << " created." << std::endl;
86
87 // Используем ранее созданный порт завершения
88 if(!CreateIoCompletionPort(HANDLE(serverSocket), ioPort, NULL, NULL)) {
89     std::cerr << "It's impossible to create io competition port." << std::endl;
90     WSACleanup();
91     return 0x4;
92 }
93
94 // Заполнение адресной структуры
95 SOCKADDR_IN sockaddrIn;
96 ZeroMemory(&sockaddrIn, sizeof(sockaddrIn));
97 sockaddrIn.sin_family = AF_INET;
98 sockaddrIn.sin_port = htons(std::stoi(port.data()));
99 sockaddrIn.sin_addr.s_addr = INADDR_ANY;
100
101 // Биндим сервер на определенный адрес
102 int serverBind = bind(serverSocket, LPSOCKADDR(&sockaddrIn), sizeof(sockaddrIn));
103 if(serverBind == SOCKET_ERROR) {
104     std::cerr << "It's impossible to bind socket." << std::endl;
105     closesocket(serverSocket);
106     WSACleanup();
107     return 0x5;
108 }
109
110 // Прослушиваем сокет
111 int serverListen = listen(serverSocket, SOMAXCONN);
112 if(serverListen == SOCKET_ERROR) {
113     std::cerr << "It's impossible to listen socket." << std::endl;
114     closesocket(serverSocket);
115     WSACleanup();
116     return 0x6;
117 }
118
119 // Создаем набор потоков для обслуживания сообщений от порта завершения
120 DWORD threadId;
121 for(int threadIndex = 0; threadIndex < COUNT_OF_THREADS; ++threadIndex)
122     if(!CreateThread(nullptr, NULL, LPTHREAD_START_ROUTINE(threadExecutor), ioPort, NULL,
123         &threadId)) {
124         std::cerr << "It's impossible to create thread." << std::endl;
125         closesocket(serverSocket);
126         WSACleanup();
127         return 0x7;
128     }
129
130 // Обработка прерывания для корректного завершения приложения
131 signal(SIGINT, signalHandler);
132
133 std::cout << "Wait clients." << std::endl;
134
135 int clientIndex = 0;
136 while(true) {

```

```

135 // Ждем подключения клиентов
136 SOCKET clientSocket = accept(serverSocket, nullptr, nullptr);
137
138 if(clientSocket == INVALID_SOCKET)
139     continue;
140
141 if(!CreateIoCompletionPort(HANDLE(clientSocket), ioPort, NULL, NULL)) {
142     std::cerr << "It's impossible to create io competition port." << std::endl;
143     closesocket(serverSocket);
144     WSACleanup();
145     return 0x8;
146 }
147
148 // Создаем и заполняем overlapped структуру
149 OverlappedConnection* overlappedConnection = new OverlappedConnection;
150 overlappedConnection->socket_handle = clientSocket;
151 overlappedConnection->op_type = OverlappedConnection::op_type_recv;
152 overlappedConnection->buffer = new char[BUFFER_SIZE];
153 overlappedConnection->hEvent = nullptr;
154 overlappedConnection->client_number = ++clientIndex;
155
156 // Добавляем в список для корректного завершения
157 clients.insert(std::pair<SOCKET, OverlappedConnection*>(clientSocket,
158 overlappedConnection));
159
160 std::cout << "Client #" << clientIndex << " has been connected. Client socket " <<
161 clientSocket << "." << std::endl;
162
163 WSABUF buffer;
164 buffer.buf = overlappedConnection->buffer;
165 buffer.len = BUFFER_SIZE;
166
167 DWORD countOfBytesRecv;
168 if(!WSARecv(overlappedConnection->socket_handle, &buffer, 1, &countOfBytesRecv,
169 nullptr, overlappedConnection, nullptr))
170     std::cout << "WSA recv error." << std::endl;
171 }
172 }
173
174 DWORD WINAPI threadExecutor(HANDLE ioPort) {
175     DWORD countOfBytesTransferred, key;
176     LPOVERLAPPED overlapped;
177
178     while(true) {
179         // Получаем информацию о завершении операции
180         if(GetQueuedCompletionStatus(ioPort, &countOfBytesTransferred, PULONG_PTR(&key), &
181 overlapped, INFINITE)) {
182             // Операция завершена успешно
183             OverlappedConnection* overlappedConnection = (OverlappedConnection*) overlapped;
184
185             // Определяем тип завершенной операции и выполняем соответствующие действия
186             switch(overlappedConnection->op_type) {
187                 case OverlappedConnection::op_type_send:
188                     //Завершена отправка данных
189                     delete [] overlappedConnection->buffer;
190                     delete overlappedConnection;
191                     break;
192
193                 case OverlappedConnection::op_type_recv:
194                     //Завершен приём данных
195                     if(!countOfBytesTransferred)
196                         closeClientConnection(overlappedConnection->socket_handle);
197
198                     overlappedConnection->buffer[countOfBytesTransferred] = '\\0';
199
200                     std::cout << "Client #" << overlappedConnection->client_number << " received

```



```

197     message: " << overlappedConnection->buffer << std::endl;
198     if (send(overlappedConnection->socket_handle, overlappedConnection->buffer,
199             countOfBytesTransferred, NULL) == SOCKET_ERROR)
200         std::cout << "Send message error." << std::endl;
201
202     WSABUF buffer;
203     buffer.buf = overlappedConnection->buffer;
204     buffer.len = BUFFER_SIZE;
205
206     DWORD countOfBytesRecv;
207     if (!WSARecv(overlappedConnection->socket_handle, &buffer, 1, &countOfBytesRecv,
208                  nullptr, overlappedConnection, nullptr))
209         std::cout << "WSA recv error." << std::endl;
210
211     break;
212 }
213 else {
214     if (!overlapped)
215         closeAllConnections();
216     else
217         closeClientConnection(((OverlappedConnection*) overlapped)->socket_handle);
218 }
219 }
220
221 void signalHandler(int sig) {
222     // Закрываем все соединения на сигнал прерывания
223     closeAllConnections();
224 }
225
226 void closeClientConnection(SOCKET clientSocket) {
227     // Закрытие конкретного клиентского соединения
228
229     std::cout << "Client #" << clients.at(clientSocket)->client_number << " disconnected.
230     Client socket " << clientSocket << "." << std::endl;
231
232     delete clients.at(clientSocket);
233     clients.erase(clientSocket);
234     closesocket(clientSocket);
235 }
236
237 void closeAllConnections() {
238     // Закрытие всех клиентских соединений, завершение работы приложения
239     for (auto& current: clients) {
240         SOCKET clientSocket = current.second->socket_handle;
241
242         std::cout << "Client #" << current.second->client_number << " disconnected. Client
243         socket " << clientSocket << "." << std::endl;
244
245         delete clients.at(clientSocket);
246         closesocket(clientSocket);
247     }
248
249     clients.clear();
250
251     closesocket(serverSocket);
252     WSACleanup();
253
254     exit(0x0);
255 }

```

Для анализа работоспособности на большом количестве клиентов был написан bat скрипт:

## 6. Оформление приложения в виде службы

В операционной системе Windows можно запускать приложения в качестве службы. Такое приложение не имеет рабочего терминала и выполняется в фоновом режиме.

Для начала работы службы необходимо дать знать менеджеру служб о том, что мы хотим добавить наше приложение в таблицу служб. Для этого необходимо указать точку входа.

Функция StartServiceCtrlDispatcher связывает службу с SCM (Service Control Manager):

```
BOOL WINAPI StartServiceCtrlDispatcher(  
    _In_ const SERVICE_TABLE_ENTRY *lpServiceTable  
);
```

Функция OpenSCManager позволяет получить экземпляр SCM:

```
SC_HANDLE WINAPI OpenSCManager(  
    _In_opt_ LPCTSTR lpMachineName,  
    _In_opt_ LPCTSTR lpDatabaseName,  
    _In_     DWORD   dwDesiredAccess  
);
```

Служба создается функцией CreateService:

```
SC_HANDLE WINAPI CreateService(  
    _In_         SC_HANDLE hSCManager,  
    _In_         LPCTSTR lpServiceName,  
    _In_opt_     LPCTSTR lpDisplayName,  
    _In_         DWORD dwDesiredAccess,  
    _In_         DWORD dwServiceType,  
    _In_         DWORD dwStartType,  
    _In_         DWORD dwErrorControl,  
    _In_opt_     LPCTSTR lpBinaryPathName,  
    _In_opt_     LPCTSTR lpLoadOrderGroup,  
    _Out_opt_    LPDWORD lpdwTagId,  
    _In_opt_     LPCTSTR lpDependencies,  
    _In_opt_     LPCTSTR lpServiceStartName,  
    _In_opt_     LPCTSTR lpPassword  
);
```

Функция OpenService в зависимости от параметров запускает или останавливает службу:

```
SC_HANDLE WINAPI OpenService(  
    _In_ SC_HANDLE hSCManager,  
    _In_ LPCTSTR lpServiceName,  
    _In_ DWORD dwDesiredAccess  
);
```

Сервер был модифицирован следующим образом: в зависимости от аргумента командной строки install, start, remove устанавливается, запускается и удаляется служба соответственно:

```
1 #define _WINSOCK_DEPRECATED_NO_WARNINGS 0  
2  
3 #include <stdio.h>  
4 #include <string.h>  
5 #include <stdlib.h>  
6 #include <signal.h>  
7 #include <thread>  
8 #include <map>  
9 #include <winsock2.h>  
10 #include <ws2tcpip.h>  
11 #include <iostream>  
12 #include <fstream>  
13 #include <string>
```

```

14
15 #pragma comment (lib , "Ws2_32.lib")
16
17 // Порт
18 static const char* PORT = "65100";
19
20 // Количество обрабатывающих потоков
21 static const int COUNT_OF_THREADS = 8;
22 // Размер буфера для передачи
23 static const int BUFFER_SIZE = 1024;
24 // Флаги
25 static const int FLAGS = 0;
26
27 static const int SERVICE_ERROR = -1;
28
29 // Аргументы командной строки для управления службой
30 static const char* INSTALL_PARAMETR = "install";
31 static const char* REMOVE_PARAMETR = "remove";
32 static const char* START_PARAMETR = "start";
33
34 // Название службы, путь к службе, путь к файлу логирования
35 static constexpr char* SERVICE_NAME = "ECHO_SERVER";
36 static const char* SERVICE_PATH = "D:\\temp\\p3.6.p.exe";
37 static const char* LOG_PATH = "D:\\temp\\p3.6.p.log";
38
39 // Структура посылки
40 struct OverlappedConnection: public OVERLAPPED {
41     int client_number;
42     SOCKET socket_handle;
43     char* buffer;
44
45     enum {
46         op_type_send,
47         op_type_recv
48     } op_type;
49 };
50
51 // Список всех клиентских соединений для корректного завершения их работы
52 std::map<SOCKET, OverlappedConnection*> clients;
53 // Серверный сокет
54 SOCKET serverSocket;
55
56 SERVICE_STATUS serverStatus;
57 SERVICE_STATUS_HANDLE handleServerStatus;
58
59 std::ofstream* out = nullptr;
60
61 void log(char* message);
62 int installEchoServer();
63 int removeEchoServer();
64 int startEchoServer();
65 int serviceMain(int argc, char** argv);
66
67 // Обработчик сигнала прерывания корректное( завершение приложения)
68 void signalHandler(int sig);
69 // Закрывает все соединения и завершает работу сервера
70 void closeClientConnection(SOCKET clientSocket);
71 // Закрывает соединение с конкретным клиентом
72 void closeAllConnections();
73
74 // Функция потока сервера для обслуживания порта завершения
75 DWORD WINAPI threadExecutor(HANDLE ioPort);
76
77 int main(int argc, char** argv) {
78     if(argc == 1) {
79         // Выполнение основных задач службы

```

```

80     SERVICE_TABLE_ENTRY serviceTable[] = {
81         { SERVICE_NAME, LPSERVICE_MAIN_FUNCTION(serviceMain) },
82         { nullptr, nullptr }
83     };
84
85     if(!StartServiceCtrlDispatcher(serviceTable)) {
86         std::cerr << "It's impossible to start service dispatcher." << std::endl;
87         return 0x1;
88     }
89 }
90 else if(!strcmp(argv[argc - 1], INSTALL_PARAMETR))
91     installEchoServer();
92 else if(!strcmp(argv[argc - 1], REMOVE_PARAMETR))
93     removeEchoServer();
94 else if(!strcmp(argv[argc - 1], START_PARAMETR))
95     startEchoServer();
96
97     return 0x0;
98 }
99
100 void log(char* message) {
101     // Логирование
102
103     if(out == nullptr)
104         out = new std::ofstream();
105
106     if(!out->is_open())
107         out->open(LOG_PATH);
108
109     if(!out->is_open())
110         return;
111
112     *out << " PID " << GetCurrentProcessId() << ": " << message << std::endl;
113
114     out->close();
115 }
116
117 int installEchoServer() {
118     // Создание службы
119
120     SC_HANDLE serviceManager = OpenSCManager(nullptr, nullptr, SC_MANAGER_CREATE_SERVICE);
121     if(!serviceManager) {
122         log("It's impossible to open service control manager.");
123         return SERVICE_ERROR;
124     }
125
126     SC_HANDLE service = CreateService(
127         serviceManager, SERVICE_NAME, SERVICE_NAME,
128         SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS, SERVICE_DEMAND_START,
129         SERVICE_ERROR_NORMAL, SERVICE_PATH, nullptr, nullptr, nullptr, nullptr);
130     if(!service) {
131         log("It's impossible to create service.");
132         CloseServiceHandle(serviceManager);
133         return SERVICE_ERROR;
134     }
135
136     CloseServiceHandle(service);
137     CloseServiceHandle(serviceManager);
138
139     log("Service has been successfully installed.");
140     return NULL;
141 }
142
143 int removeEchoServer() {
144     // Удаление службы
145

```

```

146 SC_HANDLE serviceManager = OpenSCManager(nullptr, nullptr, SC_MANAGER_ALL_ACCESS);
147 if(!serviceManager) {
148     log("It's impossible to open service control manager.");
149     return SERVICE_ERROR;
150 }
151
152 SC_HANDLE service = OpenService(serviceManager, SERVICE_NAME, SERVICE_STOP | DELETE);
153 if(!serviceManager) {
154     log("It's impossible to remove service.");
155     CloseServiceHandle(serviceManager);
156     return SERVICE_ERROR;
157 }
158
159 DeleteService(service);
160 CloseServiceHandle(service);
161 CloseServiceHandle(serviceManager);
162
163 log("Service has been successfully removed.");
164 return NULL;
165 }
166
167 int startEchoServer() {
168     // Запуск службы
169
170     SC_HANDLE serviceManager = OpenSCManager(nullptr, nullptr, SC_MANAGER_CREATE_SERVICE);
171     if(!serviceManager) {
172         log("It's impossible to open service control manager.");
173         return SERVICE_ERROR;
174     }
175
176     SC_HANDLE service = OpenService(serviceManager, SERVICE_NAME, SERVICE_START);
177     if(!serviceManager) {
178         log("It's impossible to start service.");
179         CloseServiceHandle(serviceManager);
180         return SERVICE_ERROR;
181     }
182
183     CloseServiceHandle(serviceManager);
184     CloseServiceHandle(serviceManager);
185
186     log("Service has been successfully started.");
187     return NULL;
188 }
189
190 int serviceMain(int argc, char** argv) {
191     std::string port = PORT;
192     if(argc < 2)
193         std::cout << "Using default port: " << port << "." << std::endl;
194     else
195         port = argv[1];
196
197     // Инициализация библиотеки
198     WSADATA wsaData;
199     int wsaStartup = WSASStartup(MAKEWORD(2, 2), &wsaData);
200     if(wsaStartup == SOCKET_ERROR) {
201         std::cerr << "It's impossible to startup wsa." << std::endl;
202         return 0x1;
203     }
204
205     // Создаем порт завершения
206     HANDLE ioPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, nullptr, NULL, NULL);
207     if(!ioPort) {
208         std::cerr << "It's impossible to create competition port." << std::endl;
209         WSACleanup();
210         return 0x2;
211     }

```

```

212
213 // Задаем параметры прослушивающего сокета сервера
214 serverSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, nullptr, NULL,
215     WSA_FLAG_OVERLAPPED);
216 if(serverSocket == INVALID_SOCKET) {
217     std::cerr << "It's impossible to create socket." << std::endl;
218     WSACleanup();
219     return 0x3;
220 }
221
222 std::cout << "Server socket " << serverSocket << " created." << std::endl;
223
224 // Используем ранее созданный порт завершения
225 if(!CreateIoCompletionPort(HANDLE(serverSocket), ioPort, NULL, NULL)) {
226     std::cerr << "It's impossible to create io competition port." << std::endl;
227     WSACleanup();
228     return 0x4;
229 }
230
231 // Заполнение адресной структуры
232 SOCKADDR_IN sockaddrIn;
233 ZeroMemory(&sockaddrIn, sizeof(sockaddrIn));
234 sockaddrIn.sin_family = AF_INET;
235 sockaddrIn.sin_port = htons(std::stoi(port.data()));
236 sockaddrIn.sin_addr.s_addr = INADDR_ANY;
237
238 // Биндим сервер на определенный адрес
239 int serverBind = bind(serverSocket, LPSOCKADDR(&sockaddrIn), sizeof(sockaddrIn));
240 if(serverBind == SOCKET_ERROR) {
241     std::cerr << "It's impossible to bind socket." << std::endl;
242     closesocket(serverSocket);
243     WSACleanup();
244     return 0x5;
245 }
246
247 // Прослушиваем сокет
248 int serverListen = listen(serverSocket, SOMAXCONN);
249 if(serverListen == SOCKET_ERROR) {
250     std::cerr << "It's impossible to listen socket." << std::endl;
251     closesocket(serverSocket);
252     WSACleanup();
253     return 0x6;
254 }
255
256 // Создаем набор потоков для обслуживания сообщений от порта завершения
257 DWORD threadId;
258 for(int threadIndex = 0; threadIndex < COUNT_OF_THREADS; ++threadIndex)
259     if(!CreateThread(nullptr, NULL, LPTHREAD_START_ROUTINE(threadExecutor), ioPort, NULL,
260         &threadId)) {
261         std::cerr << "It's impossible to create thread." << std::endl;
262         closesocket(serverSocket);
263         WSACleanup();
264         return 0x7;
265     }
266
267 // Обработка прерывания для корректного завершения приложения
268 signal(SIGINT, signalHandler);
269
270 std::cout << "Wait clients." << std::endl;
271
272 int clientIndex = 0;
273 while(true) {
274     // Ждем подключения клиентов
275     SOCKET clientSocket = accept(serverSocket, nullptr, nullptr);
276
277     if(clientSocket == INVALID_SOCKET)

```

```

276     continue;
277
278     if(!CreateIoCompletionPort(HANDLE(clientSocket), ioPort, NULL, NULL)) {
279         std::cerr << "It's impossible to create io competition port." << std::endl;
280         closesocket(serverSocket);
281         WSACleanup();
282         return 0x8;
283     }
284
285     // Создаем и заполняем overlapped структуру
286     OverlappedConnection* overlappedConnection = new OverlappedConnection;
287     overlappedConnection->socket_handle = clientSocket;
288     overlappedConnection->op_type = OverlappedConnection::op_type_recv;
289     overlappedConnection->buffer = new char[BUFFER_SIZE];
290     overlappedConnection->hEvent = nullptr;
291     overlappedConnection->client_number = ++clientIndex;
292
293     // Добавляем в список для корректного завершения
294     clients.insert(std::pair<SOCKET, OverlappedConnection*>(clientSocket,
295     overlappedConnection));
296
297     std::cout << "Client #" << clientIndex << " has been connected. Client socket " <<
298     clientSocket << "." << std::endl;
299
300     WSABUF buffer;
301     buffer.buf = overlappedConnection->buffer;
302     buffer.len = BUFFER_SIZE;
303
304     DWORD countOfBytesRecv;
305     if(!WSARecv(overlappedConnection->socket_handle, &buffer, 1, &countOfBytesRecv,
306     nullptr, overlappedConnection, nullptr))
307         std::cout << "WSA recv error." << std::endl;
308     }
309 }
310
311 DWORD WINAPI threadExecutor(HANDLE ioPort) {
312     DWORD countOfBytesTransferred, key;
313     LPOVERLAPPED overlapped;
314
315     while(true) {
316         // Получаем информацию о завершении операции
317         if(GetQueuedCompletionStatus(ioPort, &countOfBytesTransferred, PULONG_PTR(&key), &
318         overlapped, INFINITE)) {
319             // Операция завершена успешно
320             OverlappedConnection* overlappedConnection = (OverlappedConnection*) overlapped;
321
322             // Определяем тип завершённой операции и выполняем соответствующие действия
323             switch(overlappedConnection->op_type) {
324                 case OverlappedConnection::op_type_send:
325                     //Завершена отправка данных
326                     delete [] overlappedConnection->buffer;
327                     delete overlappedConnection;
328                     break;
329
330                 case OverlappedConnection::op_type_recv:
331                     //Завершен приём данных
332                     if(!countOfBytesTransferred)
333                         closeClientConnection(overlappedConnection->socket_handle);
334
335                     overlappedConnection->buffer[countOfBytesTransferred] = '\0';
336
337                     std::cout << "Client #" << overlappedConnection->client_number << " received
338                     message: " << overlappedConnection->buffer << std::endl;
339
340                     if(send(overlappedConnection->socket_handle, overlappedConnection->buffer,
341                     countOfBytesTransferred, NULL) == SOCKET_ERROR)

```

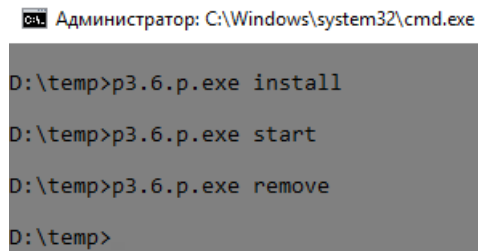
```

336         std::cout << "Send message error." << std::endl;
337
338     WSABUF buffer;
339     buffer.buf = overlappedConnection->buffer;
340     buffer.len = BUFFER_SIZE;
341
342     DWORD countOfBytesRecv;
343     if(!WSARecv(overlappedConnection->socket_handle, &buffer, 1, &countOfBytesRecv,
344     nullptr, overlappedConnection, nullptr))
345         std::cout << "WSA recv error." << std::endl;
346
347     break;
348 }
349 else {
350     if(!overlapped)
351         closeAllConnections();
352     else
353         closeClientConnection(((OverlappedConnection*) overlapped)->socket_handle);
354 }
355 }
356 }
357
358 void signalHandler(int sig) {
359     // Закрываем все соединения на сигнал прерывания
360     closeAllConnections();
361 }
362
363 void closeClientConnection(SOCKET clientSocket) {
364     // Закрытие конкретного клиентского соединения
365
366     std::cout << "Client #" << clients.at(clientSocket)->client_number << " disconnected.
367     Client socket " << clientSocket << "." << std::endl;
368
369     delete clients.at(clientSocket);
370     clients.erase(clientSocket);
371     closesocket(clientSocket);
372 }
373
374 void closeAllConnections() {
375     // Закрытие всех клиентских соединений, завершение работы приложения
376     for(auto& current: clients) {
377         SOCKET clientSocket = current.second->socket_handle;
378
379         std::cout << "Client #" << current.second->client_number << " disconnected. Client
380         socket " << clientSocket << "." << std::endl;
381
382         delete clients.at(clientSocket);
383         closesocket(clientSocket);
384     }
385
386     clients.clear();
387
388     closesocket(serverSocket);
389     WSACleanup();
390
391     exit(0x0);
392 }

```



Установка, запуск и удаление службы:



```
Администратор: C:\Windows\system32\cmd.exe
D:\temp>p3.6.p.exe install
D:\temp>p3.6.p.exe start
D:\temp>p3.6.p.exe remove
D:\temp>
```

Рис. 1.17

Созданная служба после установки:

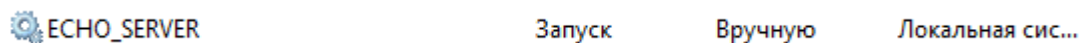


Рис. 1.18

Процесс привязанный к службе:

p3.6.p.exe	24652	Выполняется	СИСТЕМА	00	Обычный	10	p3.6.p.exe
------------	-------	-------------	---------	----	---------	----	------------

Рис. 1.19

Стоит отметить, что СИСТЕМА является владельцем этого процесса.

## 7. Реализация обмена на основе UDP

Разработаем программу *UDP* сервера. В отличие от *TCP*, соединение не устанавливается, поэтому отсутствуют функции *listen* и *accept*. Также не имеет большого смысла регистрировать отдельный поток для каждого клиента. Сервер в бесконечном цикле принимает сообщения от клиента и отправляет их назад:

```
1 #include <iostream>
2 #include <stdio.h>
3 #include <string.h>
4 #include <stdlib.h>
5 #include <signal.h>
6 #include <winsock2.h>
7 #include <ws2tcpip.h>
8 #include <string>
9
10 #pragma comment (lib , "Ws2_32.lib")
11
12 static const int PORT = 65100;
13
14 static const int BACKLOG = 5;
15 static const int BUFFER_SIZE = 1000;
16 static const int FLAGS = 0;
17
18 // Серверный сокет
19 SOCKET serverSocket;
20
21 // Обработчик сигнала прерывания корректное( завершение приложения)
22 void signalHandler(int sig);
23 // Функция отправки строки символов клиенту
24 int sendLine(char* buffer , int flags , const sockaddr_in* address);
25
26 int main(int argc , char** argv) {
27     int port = PORT;
28     if(argc < 2)
29         std::cout << "Using default port: " << port << "." << std::endl;
30     else
31         port = std::stoi(argv[1]);
```

```

32
33 // Инициализация библиотеки
34 WSADATA wsaData;
35 int wsaStartup = WSAStartup(MAKEWORD(2, 2), &wsaData);
36 if(wsaStartup == SOCKET_ERROR) {
37     std::cerr << "It's impossible to startup wsa." << std::endl;
38     return 0x1;
39 }
40
41 // Создание серверного сокета
42 serverSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
43 if(serverSocket == INVALID_SOCKET) {
44     std::cerr << "It's impossible to create socket." << std::endl;
45     return 0x2;
46 }
47
48 std::cout << "Server socket " << serverSocket << " created." << std::endl;
49
50 // Структура, задающая адресные характеристики
51 struct sockaddr_in info;
52 info.sin_family = AF_INET;
53 info.sin_port = htons(port);
54 info.sin_addr.s_addr = htonl(INADDR_ANY);
55
56 // Биндим сервер на определенный адрес
57 int serverBind = bind(serverSocket, (struct sockaddr *)&info, sizeof(info));
58 if(serverBind == SOCKET_ERROR) {
59     std::cerr << "It's impossible to bind socket." << std::endl;
60     return 0x3;
61 }
62
63 // Обработка прерывания для корректного завершения приложения
64 signal(SIGINT, signalHandler);
65
66 std::cout << "Wait clients." << std::endl;
67
68 char buffer[BUFFER_SIZE];
69 while(true) {
70     struct sockaddr_in* address = new sockaddr_in;
71     size_t size = sizeof(struct sockaddr_in);
72
73     // Ожидаем прибытия строки
74     ZeroMemory(buffer, BUFFER_SIZE);
75     int result = recvfrom(serverSocket, buffer, BUFFER_SIZE, FLAGS, (sockaddr *)&address,
76         (socklen_t *)&size);
77     if(result < 0) {
78         delete address;
79         std::cerr << "It's impossible to receive message from client." << std::endl;
80         continue;
81     }
82
83     if(strlen(buffer) <= 1) {
84         delete address;
85         std::cerr << "Message is empty." << std::endl;
86         continue;
87     }
88
89     std::cout << "Client message: " << buffer;
90
91     // Отправляем строку назад
92     result = sendLine(buffer, FLAGS, address);
93     if(result < 0) {
94         delete address;
95         std::cerr << "It's impossible to send message to client." << std::endl;
96         continue;
97     }
98 }

```

```

97     delete address;
98 }
99
100
101     return 0x0;
102 }
103
104 void signalHandler(int sig) {
105     // Закрываем серверный сокет
106     int socketClose = closesocket(serverSocket);
107     if(socketClose != 0)
108         std::cerr << "It's impossible to close socket." << std::endl;
109     else
110         std::cout << "Server socket " << serverSocket << " closed." << std::endl;
111
112     WSACleanup();
113     exit(0x0);
114 }
115
116 int sendLine(char* buffer, int flags, const sockaddr_in* address) {
117     size_t length = strlen(buffer);
118
119     // Перед отправкой сообщения добавляем в конец перевод строки
120     if(length == 0)
121         return -1;
122
123     if(buffer[length - 1] != '\n') {
124         if(length >= BUFFER_SIZE)
125             return -1;
126
127         buffer[length] = '\n';
128     }
129
130     length = strlen(buffer);
131
132     // Отправляем строку клиенту
133     int result = sendto(serverSocket, buffer, int(length), flags, (struct sockaddr *)
134         address, sizeof(struct sockaddr_in));
135     return result;
136 }

```

Клиент подключается к серверу, считывает сообщения из консоли и отправляет их серверу, после этого ожидает ответного сообщения сервера. В отличие от *TCP* отсутствует функция *connect*, отвечающая за установление соединения. Если было введено пустое сообщение, то клиент завершает работу:

```

1 #define _CRT_SECURE_NO_WARNINGS 0
2 #define _WINSOCK_DEPRECATED_NO_WARNINGS 0
3
4 #include <iostream>
5 #include <stdio.h>
6 #include <string.h>
7 #include <stdlib.h>
8 #include <signal.h>
9 #include <winsock2.h>
10 #include <ws2tcpip.h>
11 #include <string>
12
13 #pragma comment (lib, "Ws2_32.lib")
14
15 static const int PORT = 65100;
16 static const char* IP = "127.0.0.1";
17
18 static const int BACKLOG = 5;
19 static const int BUFFER_SIZE = 1000;
20 static const int IP_SIZE = 16;
21 static const int FLAGS = 0;
22

```

```

23 // Клиентский сокет
24 SOCKET clientSocket;
25
26 // Обработчик сигнала прерывания корректное( завершение приложения)
27 void signalHandler(int sig);
28 // Функция отправки строки символов серверу
29 int sendLine(char* buffer, int flags, const struct sockaddr_in* address);
30 // Корректное закрытие сокета
31 void clearSocket(SOCKET socket);
32
33 int main(int argc, char** argv) {
34     int port = PORT;
35     char ip[IP_SIZE];
36
37     strcpy(ip, IP);
38     if(argc < 3) {
39         std::cout << "Using default ip: " << ip << "." << std::endl
40             << "Using default port: " << port << "." << std::endl;
41     }
42     else {
43         strcpy(ip, argv[1]);
44         port = std::stoi(argv[2]);
45     }
46
47     // Инициализация библиотеки
48     WSADATA wsaData;
49     int wsaStartup = WSASStartup(MAKEWORD(2, 2), &wsaData);
50     if(wsaStartup == SOCKET_ERROR) {
51         std::cerr << "It's impossible to startup wsa." << std::endl;
52         return 0x1;
53     }
54
55     // Создание клиентского сокета
56     clientSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
57     if(clientSocket == INVALID_SOCKET) {
58         std::cerr << "It's impossible to create socket." << std::endl;
59         return 0x1;
60     }
61
62     std::cout << "Client socket " << clientSocket << " created." << std::endl;
63
64     // Структура, задающая адресные характеристики
65     struct sockaddr_in address;
66     address.sin_family = AF_INET;
67     address.sin_port = htons(port);
68     address.sin_addr.s_addr = inet_addr(ip);
69
70     std::cout << "Connection established." << std::endl;
71
72     // Обработка прерывания для корректного завершения приложения
73     signal(SIGINT, signalHandler);
74
75     std::cout << "Ready to send messages." << std::endl;
76
77     char buffer[BUFFER_SIZE];
78
79     while(true) {
80         ZeroMemory(buffer, BUFFER_SIZE);
81         fgets(buffer, BUFFER_SIZE, stdin);
82
83         if(strlen(buffer) == 0 || buffer[0] == '\n') {
84             std::cerr << "Empty message, trying to close client socket." << std::endl;
85             clearSocket(clientSocket);
86             return 0x0;
87         }
88

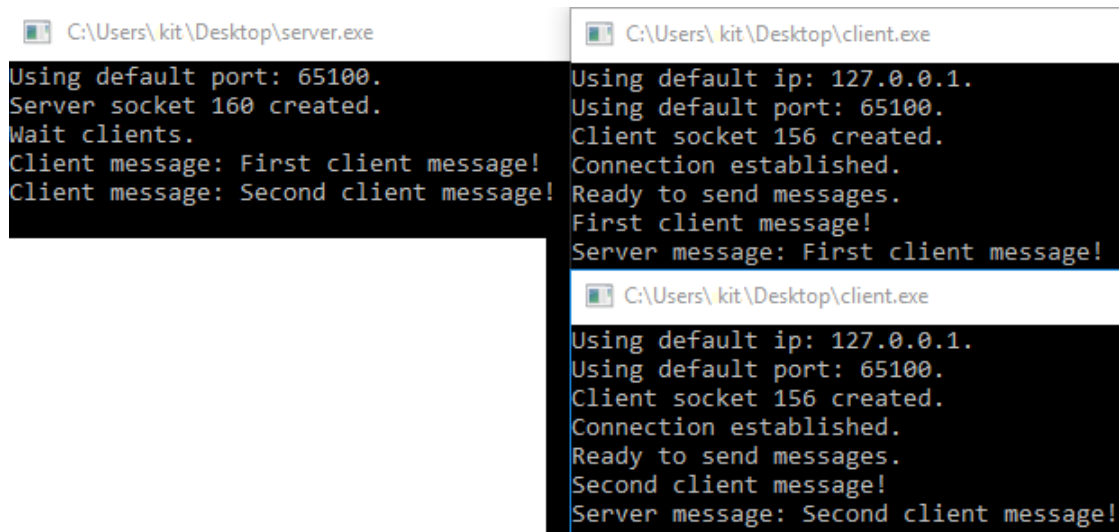
```

```

89 // Отправляем строку на сервер
90 int result = sendLine(buffer, FLAGS, &address);
91 if(result < 0) {
92     std::cerr << "It's impossible to send message to server." << std::endl;
93     clearSocket(clientSocket);
94     return 0x0;
95 }
96
97 size_t size = sizeof(struct sockaddr_in);
98
99 // Ожидаем ответ сервера
100 ZeroMemory(buffer, BUFFER_SIZE);
101 result = recvfrom(clientSocket, buffer, BUFFER_SIZE, FLAGS, (struct sockaddr *) &
102 address, (socklen_t *) &size);
103 if(result < 0) {
104     std::cerr << "It's impossible to receive message from server." << std::endl;
105     clearSocket(clientSocket);
106     return 0x0;
107 }
108
109 std::cout << "Server message: " << buffer;
110 }
111
112 return 0x0;
113 }
114
115 void signalHandler(int sig) {
116     // Закрываем клиентский сокет
117     clearSocket(clientSocket);
118     std::cout << "Client socket " << clientSocket << " closed." << std::endl;
119
120     WSACleanup();
121     exit(0x0);
122 }
123
124 int sendLine(char* buffer, int flags, const struct sockaddr_in* address) {
125     size_t length = strlen(buffer);
126
127     // Перед отправкой сообщения добавляем в конец перевод строки
128     if(length == 0)
129         return -1;
130
131     if(buffer[length - 1] != '\n') {
132         if(length >= BUFFER_SIZE)
133             return -1;
134
135         buffer[length] = '\n';
136     }
137
138     length = strlen(buffer);
139
140     // Отправляем строку серверу
141     int result = sendto(clientSocket, buffer, int(length), flags, (struct sockaddr *)
142 address, sizeof(struct sockaddr_in));
143     return result;
144 }
145
146 void clearSocket(SOCKET socket) {
147     // Закрытие сокета
148     int socketClose = closesocket(socket);
149     if(socketClose != 0)
150         std::cerr << "It's impossible to close socket." << std::endl;
151 }

```

Протестируем клиент-серверное приложение:



```
C:\Users\kit\Desktop\server.exe
Using default port: 65100.
Server socket 160 created.
Wait clients.
Client message: First client message!
Client message: Second client message!

C:\Users\kit\Desktop\client.exe
Using default ip: 127.0.0.1.
Using default port: 65100.
Client socket 156 created.
Connection established.
Ready to send messages.
First client message!
Server message: First client message!

C:\Users\kit\Desktop\client.exe
Using default ip: 127.0.0.1.
Using default port: 65100.
Client socket 156 created.
Connection established.
Ready to send messages.
Second client message!
Server message: Second client message!
```

Рис. 1.20

В первую очередь был запущен сервер два клиента. Клиенты отправляют сообщения серверу, после чего сервер завершает свою работу. В связи с тем что отсутствует установление соединения *UDP*, клиенты не узнают о том, что сервер отключился. Для обхода такой ситуации обычно с какой то очередностью серверу отправляется пакет, если от сервера пакет не пришел - значит сервер недоступен.

#### 1.2.4 Глава 4. Сигналы в Windows

Данное средство IPC в Windows не поддерживается в том смысле, в котором оно используется в Unix-системах. Однако, например, консольному приложению можно посылать сигналы CTRL+C и CTRL+BREAK. Система может посылать приложению сигналы: CTRL\_CLOSE\_EVENT, CTRL\_LOGOFF\_EVENT и CTRL\_SHUTDOWN\_EVENT, когда пользователь закрывает консоль, выходит из системы или когда система завершает работу. По получению данных сигналов процесс может произвести корректное завершение.

С помощью функции SetConsoleCtrlHandler можно установить обработчик на эти сигналы, но отправить сигнал другому приложению невозможно.

```
BOOL WINAPI SetConsoleCtrlHandler(
    _In_opt_ PHANDLER_ROUTINE HandlerRoutine,
    _In_     BOOL Add
);
```

Обработчики сигналов объединены в список. Когда приходит сигнал, вызывается последний зарегистрированный обработчик (при этом запускается отдельный поток). Если этот обработчик возвращает FALSE (он не обрабатывает этот сигнал), то вызывается следующий. Если все обработчики вернули FALSE, вызовется обработчик по умолчанию, который по умолчанию завершает процесс.

#### 1. Создание обработчика сигналов завершения для консольного приложения

Была написана программа, которая устанавливает обработчик сигнала функцией SetConsoleCtrlHandler и выводит название завершающего сигнала. Также в зависимости от сигнала производится звуковой сигнал определенной частоты:

```
1 #include <windows.h>
2 #include <iostream>
3
4 // Обработчик сигналов прерывания
5 BOOL ctrlHandler(DWORD signalType);
6
7 int main() {
8     if(SetConsoleCtrlHandler(PHANDLER_ROUTINE(ctrlHandler), TRUE)) {
9         std::cout << "Control handler is installed. Waiting interrupt signals." << std::endl;
10        while(true);
11    }
```

```

12
13     std::cerr << "It's impossible to set control handler." << std::endl;
14     return 0x1;
15 }
16
17 BOOL ctrlHandler(DWORD signalType) {
18     // Выводим сообщение и звук определенной частоты, в зависимости от прерывания
19     switch(signalType) {
20     case CTRL_C_EVENT:
21         std::cout << "Ctrl-C event." << std::endl;
22         Beep(750, 300);
23         return TRUE;
24
25     case CTRL_CLOSE_EVENT:
26         std::cout << "Ctrl-Close event." << std::endl;
27         Beep(600, 200);
28         return TRUE;
29
30     case CTRL_BREAK_EVENT:
31         std::cout << "Ctrl-Break event." << std::endl;
32         Beep(900, 200);
33         return FALSE;
34
35     case CTRL_LOGOFF_EVENT:
36         std::cout << "Ctrl-Logoff event." << std::endl;
37         Beep(1000, 200);
38         return FALSE;
39
40     case CTRL_SHUTDOWN_EVENT:
41         std::cout << "Ctrl-Shutdown event." << std::endl;
42         Beep(750, 500);
43         return FALSE;
44
45     default:
46         return FALSE;
47     }
48 }

```

Результат работы программы:

```

C:\Users\kit\Desktop>os.exe
Control handler is installed. Waiting interrupt signals.
Ctrl-C event.
Ctrl-C event.
Ctrl-Break event.
^C

```

Рис. 1.21

Реализуем собственный обработчик прерывания. При завершении приложения комбинацией клавиш Ctrl+C реализована проверка случайного нажатия. Если пользователь нажал комбинацию случайно, то приложение продолжает работу, а если нет то запрашивается еще одно нажатие:

```

1 ...
2 case CTRL_C_EVENT:
3     std::cout << "Ctrl-C event." << std::endl;
4     Beep(750, 300);
5
6     std::cout << "Are you sure you want to close application?" << std::endl
7         << "Press Ctrl+C to close it." << std::endl;
8
9     SetConsoleCtrlHandler(PHANDLER_ROUTINE(ctrlHandler), TRUE);
10    return TRUE;
11 ...

```

## 1.2.5 Глава 5. Разделяемая память

Потоки одного процесса могут разделять общую память этого процесса. У каждого процесса – свое изолированное адресное пространство. Кроме рассмотренных выше средств передачи информации между процессами или потоками разных процессов, одно из наиболее эффективных – использование общей памяти, доступ к которой обеспечивается со стороны каждого процесса. ОС Windows поддерживает такое средство, как именованная, совместно используемая память.

Первый участвующий в обмене информацией процесс создает объект проекции файла при помощи вызова функции `CreateFileMapping`. Используя флажок `PAGE_READWRITE`, задается доступ по чтению и записи в память через представление данных файла в адресном пространстве процесса.

```
HANDLE WINAPI CreateFileMapping(  
    _In_      HANDLE      hFile,  
    _In_opt_ LPSECURITY_ATTRIBUTES lpAttributes,  
    _In_      DWORD       flProtect,  
    _In_      DWORD       dwMaximumSizeHigh,  
    _In_      DWORD       dwMaximumSizeLow,  
    _In_opt_ LPCTSTR      lpName  
);
```

Процесс затем использует дескриптор объекта возвращаемый функцией `CreateFileMapping`, при вызове функции `MapViewOfFile`. Эта функция создает представление файла в адресном пространстве процесса и возвращает указатель на представление данных файла для их дальнейшего использования.

```
LPVOID WINAPI MapViewOfFile(  
    _In_      HANDLE      hFileMappingObject,  
    _In_      DWORD       dwDesiredAccess,  
    _In_      DWORD       dwFileOffsetHigh,  
    _In_      DWORD       dwFileOffsetLow,  
    _In_      SIZE_T      dwNumberOfBytesToMap  
);
```

Другой процесс может получить доступ к тем же данным при помощи вызова функции `OpenFileMapping` с тем же самым именем, что и первый процесс, а затем использовать функцию `MapViewOfFile`, чтобы получить свой указатель на представление данных файла.

```
HANDLE WINAPI OpenFileMapping(  
    _In_      DWORD       dwDesiredAccess,  
    _In_      BOOL        bInheritHandle,  
    _In_      LPCTSTR     lpName  
);
```

## 1. Взаимодействие двух процессов через совместно используемую именованную память

Процесс-читатель считывает определенное количество сообщений (`ITERATIONS_COUNT`) от процесса-писателя, после чего завершает свою работу:

```
1 #include <windows.h>  
2 #include <iostream>  
3  
4 // Размер буфера разделяемой памяти  
5 static const int BUFFER_SIZE = 256;  
6 // Количество сообщений  
7 static const int ITERATIONS_COUNT = 15;  
8  
9 static const char* MUTEX_NAME = "SYNC_MUTEX";  
10 static const char* MAPPING_NAME = "FILE_MAPPING_OBJECT";  
11  
12 int main() {  
13     // Открываем мьютекс  
14     HANDLE mutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, TEXT(MUTEX_NAME));  
15     if(mutex == nullptr) {  
16         std::cerr << "It's impossible to open mutex." << std::endl;  
17         return 0x1;  
18     }  
19 }
```



```

19
20 HANDLE mapFile = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, TEXT(MAPPING_NAME));
21 if(mapFile == nullptr) {
22     std::cerr << "It's impossible to open mapping file." << std::endl;
23     return 0x2;
24 }
25
26 // Получаем представление проецированной области памяти
27 LPCTSTR viewFile = LPTSTR(MapViewOfFile(mapFile, FILE_MAP_ALL_ACCESS, NULL, NULL,
28     BUFFER_SIZE));
29 if(viewFile == nullptr) {
30     std::cerr << "It's impossible to view map file." << std::endl;
31     return 0x3;
32 }
33
34 for(int index = 0; index < ITERATIONS_COUNT; ++index) {
35     // Ожидаем освобождения мьютекса
36     WaitForSingleObject(mutex, INFINITE);
37     // Считываем разделяемую память
38     std::cout << "Read message: " << (char *) viewFile << std::endl;
39     ReleaseMutex(mutex);
40 }
41
42 UnmapViewOfFile(viewFile);
43 CloseHandle(mapFile);
44 return 0x0;
45 }

```

Процесс-писатель бесконечно генерирует случайные числа и записывает их в разделяемую память:

```

1 #include <windows.h>
2 #include <iostream>
3 #include <ctime>
4 #include <string>
5
6 // Размер буфера разделяемой памяти
7 static const int BUFFER_SIZE = 256;
8 // Количество сообщений
9 static const int ITERATIONS_COUNT = 15;
10
11 static const int DELAY = 1000;
12
13 static const char* MESSAGE = "WRITER_MESSAGE";
14
15 static const char* MUTEX_NAME = "SYNC_MUTEX";
16 static const char* MAPPING_NAME = "FILE_MAPPING_OBJECT";
17
18 int main() {
19     // Создаем мьютекс
20     HANDLE mutex = CreateMutex(nullptr, FALSE, TEXT(MUTEX_NAME));
21     if(mutex == nullptr) {
22         std::cerr << "It's impossible to create mutex." << std::endl;
23         return 0x1;
24     }
25
26     HANDLE mapFile = CreateFileMapping(INVALID_HANDLE_VALUE, nullptr, PAGE_READWRITE, NULL,
27         BUFFER_SIZE, TEXT(MAPPING_NAME));
28     if(mapFile == nullptr || mapFile == INVALID_HANDLE_VALUE) {
29         std::cerr << "It's impossible to create mapping file." << std::endl;
30         return 0x2;
31     }
32
33     // Получаем представление проецированной области памяти
34     LPCTSTR viewFile = LPTSTR(MapViewOfFile(mapFile, FILE_MAP_ALL_ACCESS, NULL, NULL,
35         BUFFER_SIZE));
36     if(viewFile == nullptr) {
37         std::cerr << "It's impossible to view map file." << std::endl;
38     }
39 }

```

```

36     return 0x3;
37 }
38
39 srand(time(nullptr));
40
41 std::string message;
42
43 int number;
44 while(true) {
45     // Пишем в разделяемую память случайные цифры
46     number = rand();
47     message = std::to_string(number);
48     CopyMemory(PVOID(viewFile), message.data(), message.size());
49     std::cout << "Write message: " << (char *) viewFile << std::endl;
50
51     ReleaseMutex(mutex);
52 }
53
54 UnmapViewOfFile(viewFile);
55 CloseHandle(mapFile);
56 CloseHandle(mutex);
57 return 0x0;
58 }

```

Результат совместной работы процесса-писателя и процесса-читателя:

```

Администратор: C:\Windows\system32\cmd.exe
Write message: 13439
Write message: 19440
Write message: 13082
Write message: 17682
Write message: 30783
Write message: 16230
Write message: 13338
Write message: 29430
Write message: 11561
Write message: 28966
Write message: 22086
Write message: 98086
Write message: 14760
Write message: 31272

```

Рис. 1.22

Процесс-писатель успешно сгенерировал набор случайных чисел, а процесс-читатель успешно считал их из разделяемой памяти.

## 1.2.6 Глава 6. Почтовые слоты

Для решения задачи передачи данных посредством почтовых слотов необходимо воспользоваться функциями `CreateMailslot` и `CreateFile`.

```

HANDLE WINAPI CreateMailslot(
    _In_      LPCTSTR          lpName,
    _In_      DWORD             nMaxMessageSize,
    _In_      DWORD             lReadTimeout,
    _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes
);

```

Сервер создает почтовый слот и считывает оттуда данные при помощи функции `ReadFile`. Клиент использует функцию `CreateFile` с именем почтового слота в качестве имени файла. Таким образом, клиент

присоединяется к уже существующему почтовому слоту, созданному сервером. Запись данных в почтовый слот происходит при помощи функции WriteFile.

## 1. Обмен сообщениями посредством почтовых слотов на одном компьютере

Сервер считывает клиентские сообщения посредством почтовых слотов:

```
1 #include <windows.h>
2 #include <iostream>
3
4 // Название почтового слота
5 static const char* MAILSLLOT_NAME = "\\\\.\\mailslot\\MyMailSlot";
6 // Размер буфера
7 static const int BUFFER_SIZE = 1024;
8
9 int main() {
10     // Создаем почтовый слот
11     HANDLE mailslot = CreateMailslot(MAILSLLOT_NAME, BUFFER_SIZE, MAILSLLOT_WAIT_FOREVER,
12         nullptr);
13     if(mailslot == INVALID_HANDLE_VALUE) {
14         std::cerr << "It's impossible to create mailslot." << std::endl;
15         return 0x1;
16     }
17
18     std::cout << "Mailslot successfully created." << std::endl;
19
20     char buffer[BUFFER_SIZE];
21     while(true) {
22         ZeroMemory(buffer, BUFFER_SIZE);
23
24         // Считываем клиентское сообщение
25         DWORD countOfBytesRead;
26         if(!ReadFile(mailslot, buffer, BUFFER_SIZE, &countOfBytesRead, nullptr)) {
27             std::cerr << "It's impossible to read file." << std::endl;
28             return 0x2;
29         }
30
31         std::cout << "Client message: " << buffer << std::endl;
32     }
33
34     CloseHandle(mailslot);
35     return 0x0;
36 }
```

Клиентская программа считывает сообщения с консоли и отправляет их на сервер:

```
1 #include <stdio.h>
2 #include <windows.h>
3 #include <iostream>
4 #include <string>
5
6 // Название почтового слота
7 static const char* MAILSLLOT_NAME = "\\\\.\\mailslot\\MyMailSlot";
8 // Размер буфера
9 static const int BUFFER_SIZE = 1024;
10
11 int main() {
12     // Создаем почтовый слот
13     HANDLE mailslot = CreateFile(MAILSLLOT_NAME, GENERIC_WRITE, FILE_SHARE_READ, nullptr,
14         OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, nullptr);
15     if(mailslot == INVALID_HANDLE_VALUE) {
16         std::cerr << "It's impossible to create file." << std::endl;
17         return 0x1;
18     }
19
20     std::cout << "Ready to send messages." << std::endl;
```

```

21 std::string message;
22 while(true) {
23     message.clear();
24     std::getline(std::cin, message);
25
26     // Отправляем сообщение на сервер
27     DWORD countOfBytesWrite;
28     if(!WriteFile(mailslot, message.data(), int(message.size()), &countOfBytesWrite,
29         nullptr)) {
30         std::cerr << "It's impossible to write file." << std::endl;
31         return 0x2;
32     }
33 }
34 CloseHandle(mailslot);
35 return 0x0;
36 }

```

Результат передачи сообщения в пределах одного компьютера:

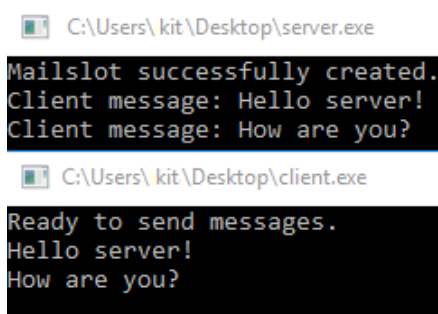


Рис. 1.23

## 1. Обмен сообщениями посредством почтовых слотов в локальной сети

Для выполнения удаленного доступа, то есть использования почтовых слотов для передачи данных по сети, необходимо указать вместо точки в имени почтового слота адрес компьютера в сети:

```
static const char* MAILSLLOT_NAME = "\\DESKTOP-MK2E3M0\\mailslot\\MyMailSlot";
```

После этого запускаем сервер и клиент на разных компьютерах. Результат приема сообщения сервером:

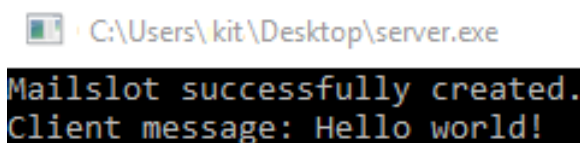


Рис. 1.24

Рассмотрим пришедший пакет программой Wireshark:

26152	52.856216	192.168.0.105	192.168.0.106	SMB Mailslot	228 Write Mail Slot
-------	-----------	---------------	---------------	--------------	---------------------

Рис. 1.25

Содержимое пришедшего пакета:

```

0000 ff ff ff ff ff ff 4c cc 6a 25 cc 59 08 00 45 00 .....L. j%.Y..E.
0010 00 d6 4e 86 00 00 80 11 00 00 c0 a8 00 6a c0 a8 ..N.....j..
0020 00 ff 00 8a 00 8a 00 c2 83 8d 11 02 f0 20 c0 a8 ..... ..
0030 00 6a 00 8a 00 ac 00 00 20 45 45 45 46 46 44 45 .j..... EEEFFDE
0040 4c 46 45 45 50 46 41 43 4e 45 4e 45 4c 44 43 45 LFEEPFAC NENELDCE
0050 46 44 44 45 4e 45 50 41 41 00 20 46 48 45 50 46 FDDNEPA A. FHEPF
0060 43 45 4c 45 48 46 43 45 50 46 46 46 41 43 41 43 CELEHFCE PFFFACAC
0070 41 43 41 43 41 43 41 43 41 41 41 00 ff 53 4d 42 ACACACAC AAA..SMB
0080 25 00 00 00 00 18 04 00 00 00 00 00 00 00 00 %.....
0090 00 00 00 00 00 00 ff fe 00 00 00 00 11 00 00 0c .....
00a0 00 02 00 00 00 00 02 00 00 00 00 00 00 00 00 .....
00b0 00 5c 00 0c 00 5c 00 03 00 01 00 00 00 02 00 23 .\...\.. ..#
00c0 00 5c 4d 41 49 4c 53 4c 4f 54 5c 4d 79 4d 61 69 .\MAILSL OT\MyMai
00d0 6c 53 6c 6f 74 00 00 00 48 65 6c 6c 6f 20 77 6f lslot... Hello wo
00e0 72 6c 64 21 rld!

```

Рис. 1.26

Как видно из записей трафика на машину сервер (192.168.0.106) пришли данные от клиента (192.168.0.105) по протоколу SMB Mailslot. Если подробно рассмотреть сами передаваемые данные, то в конце можно заметить текст, который успешно был передан.

## 2. Широковещательная передача данных посредством почтовых слотов

При создании почтовых слотов с одинаковым именем на нескольких компьютерах домена возможна широковещательная рассылка сообщений клиентов. Один клиентский процесс может посылать сообщения сразу всем этим серверным процессам. Для этого заменим в клиенте имя компьютера на символ "\*":

```
static const char* MAILSLT_NAME = "\\*\\mailslot\\MyMailSlot";
```

Запустим на каждой из машин по серверу и отправим широковещательный пакет. Результат приема сообщения серверами:

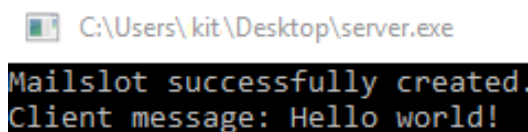


Рис. 1.27

Рассмотрим пришедший пакет программой Wireshark:

18871 30.119225	192.168.0.106	192.168.0.255	SMB Mailslot	228 Write Mail Slot
-----------------	---------------	---------------	--------------	---------------------

Рис. 1.28

Содержимое пришедшего пакета:

```

0000 ff ff ff ff ff ff 4c cc 6a 25 cc 59 08 00 45 00 .....L. j%.Y..E.
0010 00 d6 4e 86 00 00 80 11 00 00 c0 a8 00 6a c0 a8 ..N.....j..
0020 00 ff 00 8a 00 8a 00 c2 83 8d 11 02 f0 20 c0 a8 ..... ..
0030 00 6a 00 8a 00 ac 00 00 20 45 45 45 46 46 44 45 .j..... EEEFFDE
0040 4c 46 45 45 50 46 41 43 4e 45 4e 45 4c 44 43 45 LFEEPFAC NENELDCE
0050 46 44 44 45 4e 45 50 41 41 00 20 46 48 45 50 46 FDDNEPA A. FHEPF
0060 43 45 4c 45 48 46 43 45 50 46 46 46 41 43 41 43 CELEHFCE PFFFACAC
0070 41 43 41 43 41 43 41 43 41 41 41 00 ff 53 4d 42 ACACACAC AAA..SMB
0080 25 00 00 00 00 18 04 00 00 00 00 00 00 00 00 %.....
0090 00 00 00 00 00 00 ff fe 00 00 00 00 11 00 00 0c .....
00a0 00 02 00 00 00 00 02 00 00 00 00 00 00 00 00 .....
00b0 00 5c 00 0c 00 5c 00 03 00 01 00 00 00 02 00 23 .\...\.. ..#
00c0 00 5c 4d 41 49 4c 53 4c 4f 54 5c 4d 79 4d 61 69 .\MAILSL OT\MyMai
00d0 6c 53 6c 6f 74 00 00 00 48 65 6c 6c 6f 20 77 6f lslot... Hello wo
00e0 72 6c 64 21 rld!

```

Рис. 1.29

Как и требовалось ожидать, широковещательные сообщения были успешно отправлены клиентами и приняты серверами.

## 1.3 Вывод

В ходе данной работы были рассмотрены следующие средства межпроцессного взаимодействия в операционных системах семейства Windows: именованные и неименованные каналы, сокет, разделяемая память, почтовые слоты.

- Неименованные каналы Windows обеспечивают однонаправленное посимвольное межпроцессное взаимодействие. Каждый канал имеет два дескриптора: дескриптор чтения и дескриптор записи. Дескрипторы каналов часто бывают наследуемыми. Чтобы канал можно было использовать для IPC, должен существовать еще один процесс, и для этого процесса требуется один из дескрипторов канала. Анонимные каналы обеспечивают только однонаправленное взаимодействие, для двухстороннего взаимодействия необходимы два канала.
- Именованные каналы обеспечивают межпроцессное взаимодействие между сервером и одним или несколькими клиентами. Они предоставляют больше функциональных возможностей, чем анонимные каналы, которые обеспечивают межпроцессное взаимодействие на локальном компьютере. Именованные каналы поддерживают дуплексную связь по сети, несколько экземпляров сервера, взаимодействие, основанное на сообщениях и олицетворение клиента, что позволяет подключаемым процессам использовать собственные наборы разрешений на удаленных серверах. Использовать именованные каналы для связи по сети возможно только для компьютеров с ОС Windows, подключенных к одной домашней группе.
- Возможность взаимодействия с другими системами обеспечивается в Windows поддержкой сокетов. Сокет – это оконечная точка соединения, которая идентифицируется четырьмя значениями: IP адрес отправителя, порт отправителя, IP адрес получателя, порт получателя.
- Механизм сигналов как IPC отсутствует в ОС Windows. Процессы не могут отправлять сигналы другим процессам для обмена информацией. Присутствует 2 сигнала которые пользователь может отправлять приложению с клавиатуры: Ctrl+C и Ctrl+Break. Так же система может посылать приложению сигналы, когда пользователь закрывает консоль, выходит из системы или, когда система завершается.
- ОС Windows поддерживает такое средство, как именованная, совместно используемая память. Разделяемая память позволяет обмениваться информацией между двумя процессами.
- Почтовый слот – механизм синхронизации, иначе называемый почтовый ящик. Каждый слот реализуется как псевдофайл в оперативной памяти и содержит некоторое количество записей, которые могут быть прочтены всеми компьютерами в сетевом домене. Используя почтовые слоты можно передавать данные между компьютерами в локальной сети. При создании почтовых слотов с одинаковым именем на нескольких компьютерах домена возможна широковещательная рассылка сообщений клиентов. Один клиентский процесс может посылать сообщения сразу всем этим серверным процессам посредством широковещательных запросов.