Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Операционные системы

Отчет по лабораторной работе №6
Средства межпроцессного взаимодействия в ОС Windows

Работу выполнил:
Черевичник Андрей
Группа: 43501/3
Преподаватель:
Малышев Игорь Алексеевич

Санкт-Петербург
2017

1 Цель работы

Изучить средства межпроцессного взаимодействия в ОС Windows.

2 Программа работы

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

2. Именованные каналы.
   2.1.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.
   2.2. 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.
   2.3. Модифицируем приложение из предыдущего примера (2.2) для сетевого обмена информацией.

3. Сокеты.
   3.1. Программа локального обмена сокетами с использованием потокового протокола с установлением соединения (TCP в стеке TCP/IP).
   3.2. Модифицировать программу для локального обмена с множеством клиентов и с доступом к общему ресурсу.
   3.3. Сетевая передача данных с помощью сокетов.

4. Провести эксперимент с множеством клиентов при сетевом обмене, представить результаты для виртуальной и реальной сетей;

5. Проанализировать пример применения сокетов (сетевой обмен «мгновенными» сообщениями). Представить архитектуру приложения, алгоритмы сервера и клиента, схему и диаграмму их взаимодействия. Составить спецификацию функций (имя, назначение, параметры, файлы). Дополнить приложение, предоставив возможность обмениваться информацией клиентам в Linux и Windows. Исходный код в приложении. Настроить функционирование в лабораторной сети. Описать необходимые настройки.

6. Привести примеры использования портов завершения. Привести пример приложения с большим количеством клиентов до 1000 (когда порты завершения оправданы), общее количество потоков не более 10.

7. Оформить приложение с сокетами в виде службы.

8. Реализовать обмен на основе UDP

9. Сигналы в Windows
   9.1. В качестве примера рассмотрим код из msdn. В нем происходит перехват сигналов CTRL+C, CTRL+BREAK. При этом обработчик смотрит, какой сигнал ему передан, и выводит его название. В качестве звуковой индикации работы приложение вызывает функцию Beep.
   9.2. Предложить собственную реализацию обработчика сигнала.

10. 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.

11. Предложить собственную реализацию приложения, иллюстрирующую обмен информацией почтовыми слотами. Продемонстрировать возможность локального и удаленного доступа. Выполнить широковещательную передачу данных.

3 Ход работы

Основная рабочая станция:

```
OS Name:                   Microsoft Windows 10 Pro
OS Version:                10.0.10586 N/A Build 10586
...
Wireless LAN adapter Wi-Fi 2:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::58c4:2b5b:1f7:4fa7%6
   IPv4 Address. . . . . . . . . . . : 192.168.1.27
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

Дополнительная рабочая станция:

```
OS Name:                   Microsoft Windows 10 Pro
OS Version:                10.0.10586 N/A Build 10586
...
Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::a593:35fa:9f2:9f2c%4
   IPv4 Address. . . . . . . . . . . : 192.168.1.36
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.1
```

Стоит заметить, что дополнительная рабочая станция является виртуальной машиной с прямым доступом в локальную сеть. Использовался компилятор MinGW 4.9.2 x32.

3.1 Неименованные каналы (Pipe)

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) посимвольное 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.

Задание Создать клиент-серверное приложение, позволяющее набираемые символы в терминальном окне командной строки (сервер) отображать их в окно процесса-потомка (клиент). В программе-сервере master создается неименованный канал для связи с процессом-потомком, порождается сам процесс-потомок (программа-клиент) slave. На стороне сервера производится запись из консоли в канал. В slave открывается неименованный канал и осуществляется считывание из него в новое окно. Запись/чтение канала производится с помощью стандартных потоков `std_in` и `std_out`.

Исходный код программы pipe_master:

```
1  #include <windows.h>
2  #include <stdio.h>
3  #include <conio.h>
4  #include <string.h>
5  #include <iostream>
6  using namespace std;
7  int main() { // инициализируем необходимые структуры
8      STARTUPINFO si = { sizeof(si) };
9      SECURITY_ATTRIBUTES sa;
10     PROCESS_INFORMATION pi;
11     char buf[1024];
12     char t = '\n';
13     HANDLE newstdread, newstdwrite;//хэндлы потоков для пайпа
14     //инициализируем нужные поля SECURITY_ATTRIBUTES
```

```
    sa.nLength = sizeof(sa);
    sa.lpSecurityDescriptor = NULL;
    sa.bInheritHandle = true; //разрешаем наследование дескрипторов
    //создаем анонимный каналсоздаем( пайп для stdin)
    if (!CreatePipe(&newstdread, //указатель на пееменнуюр типа dword, котоаяр получит хэндл//
    ↪ конца чтения пайпа
                    &newstdwrite, //указатель на пееменнуюр типа dword, котоаяр получит хэндл//
    ↪ на конец записи пайпа
                    &sa, // указатель на структуру атрибутов безти–
                    0))//размер буфера, испся– по умолч.
    {
        cout << "I can't CreatePipe";
        getch();
        return 0;
    }
    else
        cout << "\nPipe Created!\n";
    //выводим на экран дескриптор потока ввода анонимного канала
    cout << "The read HANDLE of PIPE = " << newstdread << endl;
    //обнуляем поля STARTUPINFO и задаем нужные значения
    ZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);
    si.dwFlags = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
    si.wShowWindow = SW_NORMAL;
    //подменяем стандартный дескриптор ввода дескриптором ввода канала
    si.hStdInput = newstdread;
    si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
    si.hStdError = si.hStdOutput;
    TCHAR czCommandLine[] = L"C:\\OS\\Lab6\\pipe_slave.exe";
    if (!CreateProcess(NULL, czCommandLine, NULL, NULL, TRUE, CREATE_NEW_CONSOLE, NULL, NULL
    ↪ , &si, &pi)) {
        cout << "Error: Can't CreateProcess";
        getch();
        CloseHandle(newstdread);
        CloseHandle(newstdwrite);
        return 0;
    }
    else
    cout << "\nProcess Created!!!\n";
    memset(buf, '\0', sizeof(buf));
    cout << buf;
    unsigned long bread;
    cout << "STD INPUT HANDLE = " << GetStdHandle(STD_INPUT_HANDLE) << endl;
    cout << "STD OUTPUT HANDLE = " << GetStdHandle(STD_OUTPUT_HANDLE) << endl;
    while (1)
    {
        memset(buf, '\0', sizeof(buf));
        *buf = (char)getch();
        cout.put(*buf);
        if (*buf == 13)
        {
            *buf = '\n';
            cout.put(*buf);
        }
            WriteFile(newstdwrite, //указатель на пишущих хэндл канала
                    buf, // указатель на буфер
                    1, //колво– байт данныхзаписываемых, в буфер
                    &bread,// указатель на переменную, хранящую колво– байт, записанных в буфер
                    NULL);//тк.. й1 аргумент не был открытс флагом FILE_FLAG_OVERLAPPED
        if (*buf == 27)
            break;
    }
    TerminateProcess(pi.hProcess, 0); // завершение процесса
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
    CloseHandle(newstdread);
    CloseHandle(newstdwrite);
    system("PAUSE");
    return 0;
}
```

Исходный код программы pipe_slave:

```
#include <iostream>
#include <conio.h>
#include <stdio.h>
#include <windows.h>
```

```
using namespace std;

int main()
{
    char buf[2];
    unsigned long avail;
    cout << "STD INPUT HANDLE = " << GetStdHandle(STD_INPUT_HANDLE) << "\n";
    cout << "STD OUTPUT HANDLE = " << GetStdHandle(STD_OUTPUT_HANDLE) << "\n";
    unsigned long bread; //колво– прочитанных байт
    while (1){
        PeekNamedPipe( // получаем данные из анонимного канала
            GetStdHandle(STD_INPUT_HANDLE), // идентификатор канала Pipe
            NULL, //адрес буфера для прочитанных данных(NULL нет– данных для чтения)
            NULL, //размер буфера в байтах, параметр игнорируется, если буфер NULL
            NULL, //указатель на переменную, которая получает число считанных байт параметр// =
    ↪ NULL, если нет данных для чтения
            &avail, //адрес переменной, в которую будет записано общее количество байт//
    ↪ данныхдоступных, в канале для чтения
            NULL); // адрес переменной, в которую будет записано количество непрочитанных// байт в
    ↪ данном сообщении
        if (avail)
        {
            memset(buf, '\0', sizeof(buf));
            ReadFile(
                GetStdHandle(STD_INPUT_HANDLE), //handle канала
                buf, //указатель на буферв, который пишем считанные из канала// данные
                1, //колво– байтов для чтения
                &bread, //указатель на переменную колво(– считанных байт)
                NULL); // если handle не OVERLAPPED, то равен NULL
            cout << buf;
        }
    }
    return 0;
}
```

При запуске "сервера"в командной строке выводится текст "Pipe Created! затем сервером порождается процесс "slave"в новом окне, после чего любые символы, которые пишем в окне сервера, моментально появляются в окне клиента.

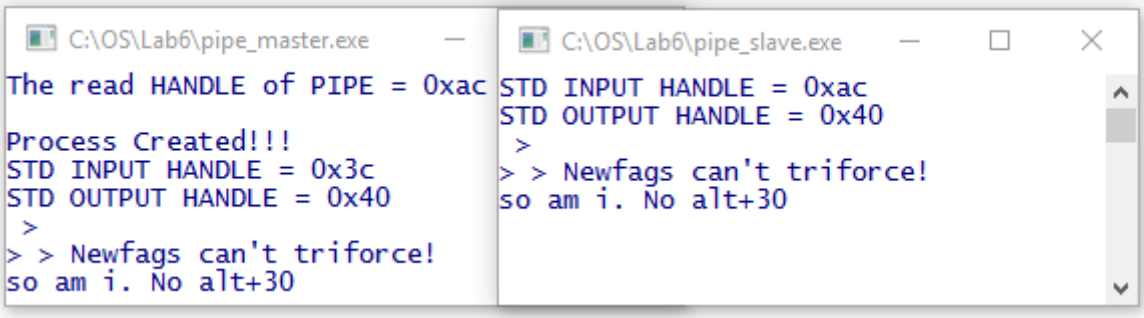

Рис. 1: Результат работы

Задание Создать эхо-сервер, взаимодействующий с клиентом посредством pipe.

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). При этом ненужные дескрипторы каналов закрываются только на стороне сервера (т.к. клиент наследует 4 дескриптора, а явно мы передаем только 2 дескриптора).

Дескрипторы каналов связываются со стандартным вводом и выводом клиентского процесса. Поэтому клиент выводит информацию в поток ошибок (что приведет к выводу в консоль процесса-клиента). Клиент передает сообщение, например, вида: «message num 1». Сервер передает данное сообщение обратно. Процессы завершаются после передачи 10 сообщений.

Исходный код программы сервера:

```
#include <windows.h>
#include <stdio.h>

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

```
5       HANDLE hReadPipeFromServToClient, hWritePipeFromServToClient;
6       //дескрипторы канала для
7       //передачи от сервера клиенту
8       HANDLE hReadPipeFromClientToServ, hWritePipeFromClientToServ;
9       //дескрипторы канала для
10      //передачи от сервера клиенту
11      SECURITY_ATTRIBUTES PipeSA = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
12      //чтобы сделать
13      //дескрипторы наследуемыми
14      //создаем канал для передачи от сервера клиенту, сразу делаем дескрипторы наследуемыми
15      if(CreatePipe(&hReadPipeFromServToClient,&hWritePipeFromServToClient,&PipeSA,0)==0) {
16          printf("impossible to create anonymous pipe from serv to client\n");
17          getchar();
18          return 1000;
19      }
20      //создаем канал для передачи от клиента серверу, сразу делаем дескрипторы наследуемыми
21      if(CreatePipe(&hReadPipeFromClientToServ,&hWritePipeFromClientToServ,&PipeSA,0)==0) {
22          printf("impossible to create anonymous pipe from client to serv\n");
23          getchar();
24          return 1001;
25      }
26      PROCESS_INFORMATION processInfo_Client;
27      // информация о процессеклиенте–
28      STARTUPINFO startupInfo_Client;
29      //структура, которая описывает внешний вид основного
30      //окна и содержит дескрипторы стандартных устройств нового процесса, используем для установки
31      //процессклиент– будет иметь те же параметры запуска, что и сервер, за исключением
32      //дескрипторов ввода, вывода и ошибок
33      GetStartupInfo(&startupInfo_Client);
34      startupInfo_Client.hStdInput = hReadPipeFromServToClient; //устанавливаем поток ввода
35      startupInfo_Client.hStdOutput=hWritePipeFromClientToServ; //установим поток вывода
36      startupInfo_Client.hStdError=GetStdHandle(STD_ERROR_HANDLE); //установим поток ошибок
37      startupInfo_Client.dwFlags = STARTF_USESTDHANDLES; //устанавливаем наследование
38      //создаем процесс клиента
39      TCHAR czCommandLine[] = L"C:\\OS\\Lab6\\pipe_client.exe";
40      CreateProcess(NULL, czCommandLine, NULL, NULL, TRUE, CREATE_NEW_CONSOLE, NULL, NULL, &
        ↪ startupInfo_Client, &processInfo_Client);
41      CloseHandle(processInfo_Client.hThread);
42      //закрываем дескрипторы созданного процесса и его
43      //потока
44      CloseHandle(processInfo_Client.hProcess);
45      //закрываем ненужные дескрипторы каналов, которые не использует сервер
46      CloseHandle(hReadPipeFromServToClient);
47      CloseHandle(hWritePipeFromClientToServ);
48  #define BUF_SIZE 100 //размер буфера для сообщений
49      BYTE buf[BUF_SIZE]; //буфер приемапередачи/
50      DWORD readbytes,writebytes; //число прочитанныхпереданных/ байт
51      for(int i=0;i<10;i++) { //читаем данные из канала от клиента
52          if(!ReadFile(hReadPipeFromClientToServ,buf,BUF_SIZE,&readbytes,NULL)) {
53              printf("impossible to use readfile\n GetLastError= %d\n",GetLastError());
54              getchar();
55              return 10000;
56          }
57          printf("get from client: \"%s\"\n",buf);
58          if(!WriteFile(hWritePipeFromServToClient,buf,readbytes,&writebytes,NULL)) {
59              printf("impossible to use writefile\n GetLastError= %d\n",GetLastError());
60              getchar();
61              return 10001;
62          } //пишем данные в канал клиенту
63      }
64      //закрываем HANDLE каналов
65      CloseHandle(hReadPipeFromClientToServ);
66      CloseHandle(hWritePipeFromServToClient);
67      printf("server ended work\n Press any key");
68      getchar();
69      return 0;
70  }
```

Исходный код программы клиента:

```
1  #include <stdio.h>
2  #include <Windows.h>
3
4  int main(int argc, char* argv[]) {
5      char strtosend[100]; //строка для передачи
6      char getbuf[100]; //буфер приема
7      int bytestosend; //число передаваемых байт
```

```
8       DWORD bytessended,bytesreaded; //число переданных и принятых байт
9       for(int i=0;i<10;i++) {
10          bytestosend=sprintf(strtosend,"message num %d",i+1);
11          //формирование строки для
12          //передачи
13          strtosend[bytestosend]=0;
14          fprintf(stderr,"client sended: \"%s\"\n",strtosend);
15          if(!WriteFile(GetStdHandle(STD_OUTPUT_HANDLE),strtosend,bytestosend+1,&bytesreaded,
        ↪ NULL)) {
16              fprintf(stderr,"Error with writeFile\n Wait 5 sec GetLastError= %d\n",
        ↪ GetLastError());
17              Sleep(5000);
18              return 1000;
19          }
20          if (!ReadFile(GetStdHandle(STD_INPUT_HANDLE), getbuf, 100, &bytesreaded, NULL)) {
21              fprintf(stderr,"Error with readFile\n Wait 5 sec GetLastError= %d\n",
        ↪ GetLastError());
22              Sleep(5000);
23              return 1001;
24          }
25          fprintf(stderr,"Get msg from server: \"%s\"\n",getbuf);
26      }
27      fprintf(stderr,"client ended work\n Wait 5 sec");
28      Sleep(5000);
29      return 0;
30  }
```

Результат выполнения программ: Сервер:

```
1  get from client: "message num 1"
2  get from client: "message num 2"
3  get from client: "message num 3"
4  get from client: "message num 4"
5  get from client: "message num 5"
6  get from client: "message num 6"
7  get from client: "message num 7"
8  get from client: "message num 8"
9  get from client: "message num 9"
10 get from client: "message num 10"
11 server ended work
12  Press any key
```

Клиент:

```
1  client sended: "message num 1"
2  Get msg from server: "message num 1"
3  client sended: "message num 2"
4  Get msg from server: "message num 2"
5  client sended: "message num 3"
6  Get msg from server: "message num 3"
7  client sended: "message num 4"
8  Get msg from server: "message num 4"
9  client sended: "message num 5"
10 Get msg from server: "message num 5"
11 client sended: "message num 6"
12 Get msg from server: "message num 6"
13 client sended: "message num 7"
14 Get msg from server: "message num 7"
15 client sended: "message num 8"
16 Get msg from server: "message num 8"
17 client sended: "message num 9"
18 Get msg from server: "message num 9"
19 client sended: "message num 10"
20 Get msg from server: "message num 10"
21 client ended work
22 Wait 5 sec
```

3.2 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. Закрытие 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.

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

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.

Задание Реализовать между одним клиентом и сервером обмен данными, вводимыми с консоли на стороне клиента и возвращаемыми сервером обратно до получения команды exit.

Программа-сервер создает именованный канал для двунаправленного использования и ожидает подключения программы-клиента.

Проверяем, корректно ли произошло подключение, затем входим в цикл получения команд от "клиента"с последующими эхо-ответами. При появлении команды exit со стороны клиента, сервер завершает работу, закрывает канал.

Клиент на своей стороне открывает канал, пишет в него и читает эхо-ответ. При вводе exit программа завершается.

Исходный код программы npipe_server:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
int main()
{
    // Флаг успешного создания канала
    BOOL fConnected;
    // Идентификатор канала Pipe
    HANDLE hNamedPipe;
    // Имя создаваемого канала Pipe
    LPCWSTR lpszPipeName = L"\\\\.\\pipe\\$MyPipe$";
    // Буфер для передачи данных через канал
    char szBuf[512];
    // Количество байт данных, принятых через канал
    DWORD cbRead;
    // Количество байт данных, переданных через канал
    DWORD cbWritten;
    printf("Named pipe was creating \n");
    // Создаем канал Pipe, имеющий имя lpszPipeName
    hNamedPipe = CreateNamedPipe(
        lpszPipeName, //имя канала
        PIPE_ACCESS_DUPLEX, // режим котрытия канала - двунаправленный
        //параметры режима pipe:
        PIPE_TYPE_MESSAGE //Данные записываются в канал в виде потока сообщений
        | PIPE_READMODE_MESSAGE //Данные считываются в виде потока сообщений
        | PIPE_WAIT, // блокирующий режим
        PIPE_UNLIMITED_INSTANCES, //неограниченное колво- подключений"" к каналу
        512, 512, //колво- байт входного и вызодного буферов
        0, //таймаут- в 50 миллисекунд по( умолчанию)
        NULL); // дескриптор безопасности по умолчанию
    // Если возникла ошибка, выводим ее код и
    // завершаем работу приложения
    if (hNamedPipe == INVALID_HANDLE_VALUE)
    {
        fprintf(stdout, "CreateNamedPipe: Error %ld\n",
            GetLastError());
        getch();
        return 0;
    }
    // Выводим сообщение о начале процесса создания канала
    fprintf(stdout, "Waiting for connect...\n");
    // Ожидаем соединения со стороны клиента
    fConnected = ConnectNamedPipe(hNamedPipe, // имя канала
        NULL); // overlapped=null
    // При возникновении ошибки выводим ее код
    if (!fConnected)
    {
        switch (GetLastError())
```

```
49          {
50          case ERROR_NO_DATA:
51              fprintf(stdout, "ConnectNamedPipe: ERROR_NO_DATA");
52              getch();
53              CloseHandle(hNamedPipe);
54              return 0;
55              break;
56          case ERROR_PIPE_CONNECTED:
57              fprintf(stdout,
58                  "ConnectNamedPipe: ERROR_PIPE_CONNECTED");
59              getch();
60              CloseHandle(hNamedPipe);
61              return 0;
62              break;
63          case ERROR_PIPE_LISTENING:
64              fprintf(stdout,
65                  "ConnectNamedPipe: ERROR_PIPE_LISTENING");
66              getch();
67              CloseHandle(hNamedPipe);
68              return 0;
69              break;
70          case ERROR_CALL_NOT_IMPLEMENTED:
71              fprintf(stdout,
72                  "ConnectNamedPipe: ERROR_CALL_NOT_IMPLEMENTED");
73              getch();
74              CloseHandle(hNamedPipe);
75              return 0;
76              break;
77          default:
78              fprintf(stdout, "ConnectNamedPipe: Error %ld\n",
79                  GetLastError());
80              getch();
81              CloseHandle(hNamedPipe);
82              return 0;
83              break;
84          }
85          CloseHandle(hNamedPipe);
86          getch();
87          return 0;
88      } // Выводим сообщение об успешном создании канала
89      fprintf(stdout, "\nConnected. Waiting for command...\n");
90      // Цикл получения команд из канала
91      while (1)
92      {
93          if (ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL))
94          {
95              // Выводим принятую команду на консоль
96              printf("Received: <%s>\n", szBuf);
97              // Если пришла команда "exit",
98              // завершаем работу приложения
99              if (!strcmp(szBuf, "exit"))
100                 break;
101             // отправляем эхоответ–
102             if (!WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1, &cbWritten, NULL)) break;
103         }
104         else
105         {
106             getch();
107             break;
108         }
109     }
110     CloseHandle(hNamedPipe);
111     return 0;
112 }
```

Исходный код программы npipe_client:

```
1  #include <windows.h>
2  #include <stdio.h>
3  #include <conio.h>
4  int main(int argc, char *argv[])
5  { // Идентификатор канала Pipe
6      HANDLE hNamedPipe;
7      // Количество байт, переданных через канал
8      DWORD cbWritten;
9      // Количество байт, принятых через канал
10     DWORD cbRead;
```

```
11      // Буфер для передачи данных
12      char szBuf[256];
13      // Буфер для имени канала Pipe
14      LPCWSTR szPipeName = L"\\\\.\\pipe\\$MyPipe$";
15      printf("Named pipe client demo\n");
16      printf("Syntax: pipec [servername]\n");
17      // открываем handle нашего именованого pipe
18      hNamedPipe = CreateFile(
19          szPipeName, // pipe name
20          GENERIC_READ | // read and write access
21          GENERIC_WRITE,
22          0, // no sharing
23          NULL, // default security attributes
24          OPEN_EXISTING, // opens existing pipe
25          0, // default attributes
26          NULL); // no template file
27      // Если возникла ошибка, выводим ее код и
28      // завершаем работу приложения
29      if (hNamedPipe == INVALID_HANDLE_VALUE)
30      {
31          fprintf(stdout, "CreateFile: Error %ld\n",
32              GetLastError());
33          getch();
34          return 0;
35      }
36      // Выводим сообщение о создании канала
37      fprintf(stdout, "\nConnected. Type 'exit' to terminate\n");
38      // Цикл обмена данными с серверным процессом
39      while (1)
40      { // Выводим приглашение для ввода команды
41          printf("cmd>");
42          // Вводим текстовую строку
43          gets(szBuf);
44          // Передаем введенную строку серверному процессу
45          // в качестве команды
46          if (!WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1, &cbWritten, NULL)) break;
47          // Получаем эту же команду обратно от сервера
48          if (ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL))
49              printf("Received back: <%s>\n", szBuf);
50          // В ответ на команду "exit" завершаем цикл
51          // обмена данными с серверным процессом
52          if (!strcmp(szBuf, "exit"))
53              break;
54      }
55      // Закрываем идентификатор канала
56      CloseHandle(hNamedPipe);
57      return 0;
58  }
```

Результат выполнения программы приведен на рисунке 2.

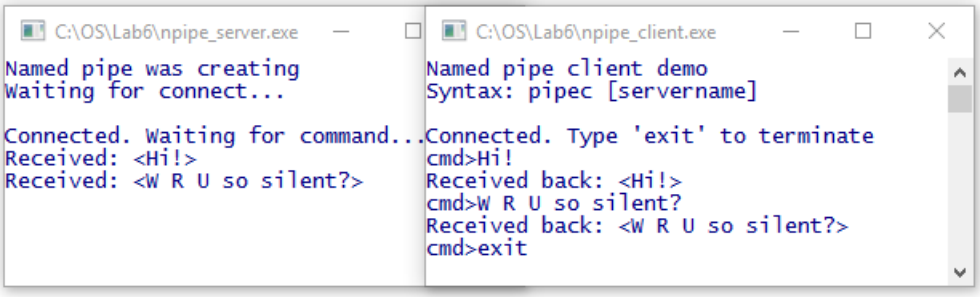

Рис. 2: Результат работы

Запускаем в одном окне сервер (на рис. слева). Он выводит сообщения о том, что канал создан и ожидает подключения клиента. Запускаем в другом окне клиента (справа). Затем на стороне сервера мы получаем уведомление о том, что создано новое подключение, теперь мы можем получать сообщения от клиента.

Клиент после запуска выводит уведомления о том, что он подключился к серверу и ожидает ввода команд в строке cmd> . Вводимые в командной строке символы он пересылает серверу и сразу выводит возвращенный эхо-ответ от него. Для завершения сеанса обмена следует ввести зарезервированную

команду exit на стороне клиента, после ее доставки серверу, он завершает работу.

Сервер работает с одним клиентом, поэтому ему не нужно вызывать функции отсоединения клиента (по завершению сервера, клиент тоже завершается).

Задание Реализовать между сервером и множеством клиентов обмен данными, вводимыми с консоли на стороне клиента и возвращаемыми сервером обратно до получения команды exit.

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.

Клиент после соединения с сервером начинает чтение сообщений с консоли, пока не встретит слово «exit». По данному слову и клиент и сервер завершают свою работу.

Обратить внимание на использование функции WaitNamedPipe, а также на возможность использования количества экземпляров каналов, равного количеству потенциальных клиентов.

Исходный код программы npipe_server_bi:

```
#include <windows.h>
#include <stdio.h>
#define SIZE_OF_BUF 1000

int main() {
    HANDLE hNamedPipe; // Идентификатор канала Pipe
    LPCWSTR lpszPipeName = L"\\\\.\\pipe\\$$MyPipe$$"; // Имя создаваемого канала  Pipe
    char buf[SIZE_OF_BUF]; // Буфер для передачи данных через канал
    DWORD readbytes,writebytes; //число байт прочитанных и переданных
    printf("Server is started. Try to create named pipe\n"); // Создаем канал Pipe, имеющий имя lpszPipeName
    hNamedPipe = CreateNamedPipe(lpszPipeName, //имя канала
                                 PIPE_ACCESS_DUPLEX, //доступ и на чтение и на запись
                                 PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT, //передача сообщений как( чтение, так и запись)
                                 5, //максимальное число экземпляров каналов равно 5 число( клиентов)
                                 SIZE_OF_BUF,
                                 SIZE_OF_BUF,
                                 5000,
                                 NULL); //размеры выходного и входного буферов канала, 5 секунд − длительность для функции WaitNamedPipe
    if(hNamedPipe == INVALID_HANDLE_VALUE) {
        fprintf(stdout,"CreateNamedPipe: Error %ld\n",GetLastError());
        getchar();
        return 1000;
    }
    printf("Named pipe created successfully\n"); // Выводим сообщение о начале процесса создания канала
    printf("waiting for connect\n"); // Ожидаем соединения со стороны клиента
    if(!ConnectNamedPipe(hNamedPipe, NULL)) { // При возникновении ошибки выводим ее код
        printf("error with function ConnectNamedPipe\n");
        getchar();
        CloseHandle(hNamedPipe);
        return 1001;
    } // Выводим сообщение об успешном создании канала
    fprintf(stdout,"Client connected\n"); // Цикл получения команд через канал
    while(1) { // Получаем очередную команду через канал Pipe
        if(ReadFile(hNamedPipe, buf, SIZE_OF_BUF, &readbytes, NULL)) { // Посылаем эту команду обратно клиентскому // приложению
            if(!WriteFile(hNamedPipe, buf, strlen(buf) + 1, &writebytes, NULL)) break; // Выводим принятую команду на консоль
            printf("Get client msg: %s\n", buf); // Если пришла команда "exit", // завершаем работу приложения
            if(!strncmp(buf, "exit",4))
                break;
        } else {
            fprintf(stdout,"ReadFile: Error %ld\n",GetLastError());
            getchar();
            break;
        }
    }
    CloseHandle(hNamedPipe);
    printf("server is ending\n press any key\n");
    getchar();
    return 0;
}
```

Исходный код программы npipe_client_bi:

```
#include <windows.h>
#include <stdio.h>
#define SIZE_OF_BUF 1000

int main(int argc, char *argv[]) {
    HANDLE hNamedPipe; // Идентификатор канала Pipe
    DWORD readbytes,writebytes; // количество байт принятых и переданных в канал
    char buf[SIZE_OF_BUF]; // Буфер для передачи данных
    printf("Client is started\n Try to use WaitNamedPipe\n"); //ожидаем"" пока освободится
    ↪ канал
    LPCWSTR szPipeName = L"\\\\.\\pipe\\$$MyPipe$$";
    if (!WaitNamedPipe(szPipeName, NMPWAIT_WAIT_FOREVER)) {
        printf("pipe wasn't created\n getlasterror = %d", GetLastError());
        getchar();
        return 1000;
    } // Создаем канал с процессомсервером–
    hNamedPipe = CreateFile(szPipeName, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING
    ↪ , 0, NULL); // Если возникла ошибка, выводим ее код и
    // завершаем работу приложения
    if(hNamedPipe == INVALID_HANDLE_VALUE) {
        fprintf(stdout,"CreateFile: Error %ld\n",GetLastError());
        getchar();
        return 1001;
    } // Выводим сообщение о создании канала
    printf("successfully connected\n input message\n"); // Цикл обмена данными с серверным
    ↪ процессом
    while(1) { // Выводим приглашение для ввода команды printf("cmd>"); // Вводим текстовую
    ↪ строку
        fgets(buf,SIZE_OF_BUF,stdin); // Передаем введенную строку серверному процессу // в
    ↪ качестве команды
        if(!WriteFile(hNamedPipe, buf, strlen(buf) + 1, &writebytes, NULL)) {
            printf("connection refused\n");
            break;
        } // Получаем эту же команду обратно от сервера
        if(ReadFile(hNamedPipe, buf, SIZE_OF_BUF, &readbytes, NULL))
            printf("Received from server: %s\n", buf); // Если произошла ошибка, выводим ее
    ↪ код и // завершаем работу приложения
        else {
            fprintf(stdout,"ReadFile: Error %ld\n", GetLastError());
            getchar();
            break;
        } // В ответ на команду "exit" завершаем цикл // обмена данными с серверным процессом
        if(!strncmp(buf, "exit",4))
            break;
    } // Закрываем идентификатор канала
    CloseHandle(hNamedPipe);
    printf("client is ending\n Press any key\n");
    getchar();
    return 0;
}
```

Результат выполнения программы приведен на рисунке 3.

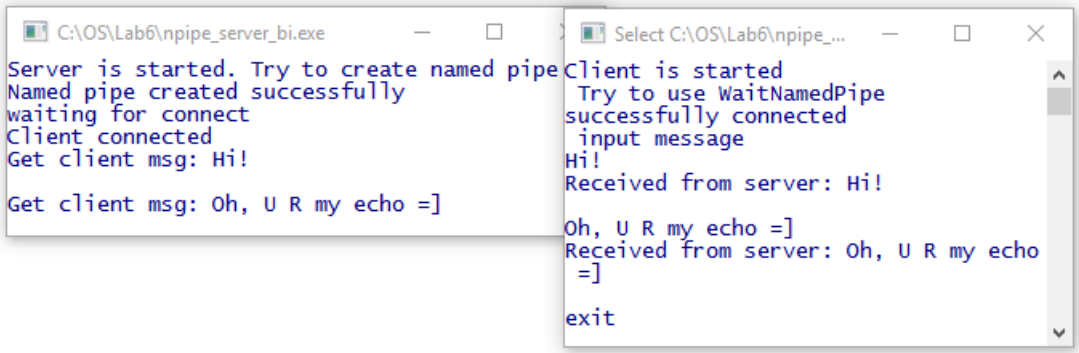

Рис. 3: Результат работы

Для подключения нескольких клиентов сервер был изменен:

```
#include <windows.h>
#include <stdio.h>
#define SIZE_OF_BUF 1000

DWORD WINAPI threadHandler(LPVOID param){
    HANDLE hNamedPipe = (HANDLE)param;
    DWORD readbytes, writebytes; //число байт прочитанных и переданных
    char buf[SIZE_OF_BUF]; // Буфер для передачи данных через канал
    while (1)
    {
        // Получаем очередную команду через канал Pipe
        if (ReadFile(hNamedPipe, buf, SIZE_OF_BUF, &readbytes, NULL))
        {
            // Посылаем эту команду обратно клиентскому
            // приложению
            if (!WriteFile(hNamedPipe, buf, strlen(buf) + 1, &writebytes, NULL))
                break;
            // Выводим принятую команду на консоль
            printf("Get client msg: %s\n", buf);
            // Если пришла команда "exit",
            // завершаем работу приложения
            if (!strncmp(buf, "exit", 4))
                break;
        }
        else
        {
            fprintf(stdout, "ReadFile: Error %ld\n", GetLastError());
            getchar();
            break;
        }
    }
    CloseHandle(hNamedPipe);
    ExitThread(0);
}



int main()
{
    HANDLE hNamedPipe; // Идентификатор канала Pipe
    HANDLE t; //Для потока
    LPCWSTR lpszPipeName = L"\\\\.\\pipe\\$$MyPipe$$"; // Имя создаваемого канала Pipe
    printf("Server is started. Try to create named pipe\n");
    // Создаем канал Pipe, имеющий имя lpszPipeName
    while (1){
        hNamedPipe = CreateNamedPipe(
            lpszPipeName, //имя канала
            PIPE_ACCESS_DUPLEX, //доступ и на чтение и на запись
            PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT, //передача сообщений как(
    ↪ чтение,
            //так и запись)
            5, //максимальное число экземпляров каналов равно 5 число( клиентов)
            SIZE_OF_BUF, SIZE_OF_BUF, 5000, NULL); //размеры выходного и входного буферов
    ↪ канала, 5
        //секунд - длительность для функции WaitNamedPipe
        if (hNamedPipe == INVALID_HANDLE_VALUE) // Если возникла ошибка, выводим ее код и
    ↪ завершаем
            //работу приложения
        {
            fprintf(stdout, "CreateNamedPipe: Error %ld\n", GetLastError());
            getchar();
            return 1000;
        }
        printf("Named pipe created successfully\n");
        // Выводим сообщение о начале процесса создания канала
        printf("waiting for connect\n");
        // Ожидаем соединения со стороны клиента

        if (!ConnectNamedPipe(hNamedPipe, NULL))
        {
            // При возникновении ошибки выводим ее код
            printf("error with function ConnectNamedPipe\n");
            getchar();
            CloseHandle(hNamedPipe);
            return 1001;
        }
```

```
74          // Выводим сообщение об успешном создании канала
75          fprintf(stdout, "Client connected\n");
76          t = CreateThread(NULL, 0, threadHandler, (LPVOID)hNamedPipe, 0, NULL);
77          CloseHandle(t);
78      }
79      printf("server is ending\n press any key\n");
80      getchar();
81      return 0;
82  }
```

Результат выполнения программы приведен на рисунке 4.

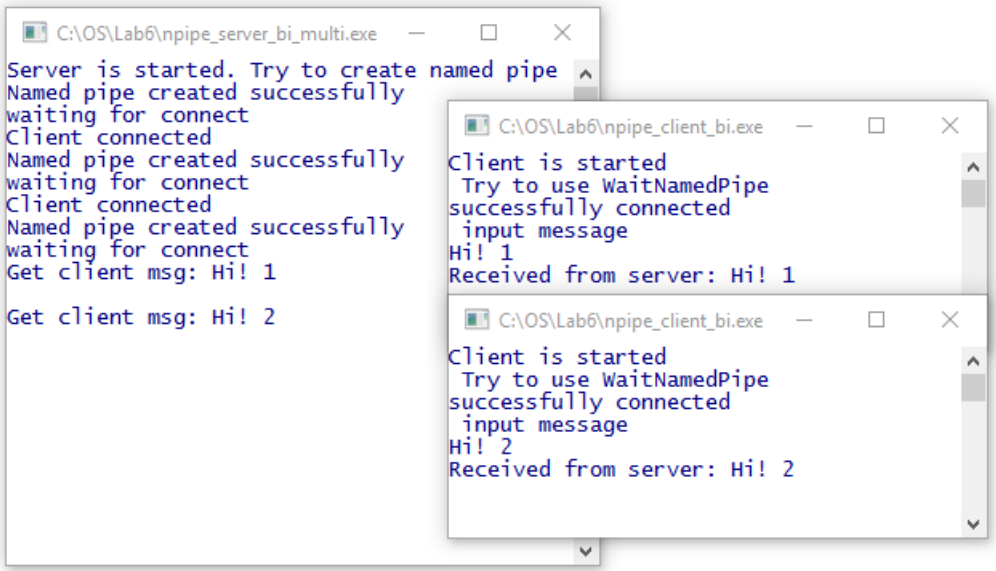

Рис. 4: Результат работы

Именованные каналы позволяют осуществлять обмен между процессами, выполняющимися на разных компьютерах в сети. Для этого необходимо выполнить определенный ряд условий и настроек. Но сетевые именованные каналы не являются промышленным стандартом и используются в этом качестве скорее как исключение. В ОС семейства Windows это возможно, в отличие, например, от Unix-подобных систем, где обмен данными осуществляется через ядро.

Для совместной работы компьютеры нужно подсоединить к одной домашней группе. Так же необходимо установить поле DACL (Discretionary Access Control List) защиты объекта в NULL (разрешение всем пользователям и группам доступа к объекту). Параметры защиты именованного канала задаются с помощью структуры SECURITYATTRIBUTES, которая указывается последним параметром в функции CreateNamedPipe.

Server для работы по сети:

```
1  #include <windows.h>
2  #include <stdio.h>
3  #define SIZE_OF_BUF 1000
4
5  DWORD WINAPI threadHandler(LPVOID param){
6       HANDLE hNamedPipe = (HANDLE)param;
7       DWORD readbytes, writebytes; //число байт прочитанных и переданных
8       char buf[SIZE_OF_BUF]; // Буфер для передачи данных через канал
9       while (1)
10      {
11          // Получаем очередную команду через канал Pipe
12          if (ReadFile(hNamedPipe, buf, SIZE_OF_BUF, &readbytes, NULL))
13          {
14              // Посылаем эту команду обратно клиентскому
15              // приложению
16              if (!WriteFile(hNamedPipe, buf, strlen(buf) + 1, &writebytes, NULL))
17                  break;
18              // Выводим принятую команду на консоль
19              printf("Get client msg: %s\n", buf);
20              // Если пришла команда "exit",
21              // завершаем работу приложения
22              if (!strncmp(buf, "exit", 4))
```

```
23                  break;
24          }
25          else
26          {
27              fprintf(stdout, "ReadFile: Error %ld\n", GetLastError());
28              getchar();
29              break;
30          }
31      }
32      CloseHandle(hNamedPipe);
33      ExitThread(0);
34  }
35
36
37
38  int main()
39  {
40      HANDLE hNamedPipe; // Идентификатор канала Pipe
41      HANDLE t; //Для потока
42      LPCWSTR lpszPipeName = L"\\\\.\\pipe\\$$MyPipe$$"; // Имя создаваемого канала Pipe
43      printf("Server is started. Try to create named pipe\n");
44      // Создаем канал Pipe, имеющий имя lpszPipeName
45      while (1){
46          // Создание SECURITY_ATTRIBUTES и SECURITY_DESCRIPTOR объектов
47          SECURITY_ATTRIBUTES sa;
48          SECURITY_DESCRIPTOR sd;
49          // Инициализация SECURITY_DESCRIPTOR значениями поумолчанию–
50          if (InitializeSecurityDescriptor(&sd, SECURITY_DESCRIPTOR_REVISION) == 0) {
51              printf("InitializeSecurityDescriptor failed with error %ld\n", GetLastError());
52              return 50000;
53          }
54          // Установка поля DACL в SECURITY_DESCRIPTOR в NULL
55          if (SetSecurityDescriptorDacl(&sd, TRUE, NULL, FALSE) == 0) {
56              printf("SetSecurityDescriptorDacl failed with error %ld\n", GetLastError());
57              return 50001;
58          } // Установка SECURITY_DESCRIPTOR в структуре SECURITY_ATTRIBUTES
59          sa.nLength = sizeof(SECURITY_ATTRIBUTES);
60          sa.lpSecurityDescriptor = &sd;
61          sa.bInheritHandle = FALSE; //запрещение наследования
62          hNamedPipe = CreateNamedPipe( lpszPipeName, //имя канала
63                                        PIPE_ACCESS_DUPLEX, //доступ и на чтение и на запись
64                                        PIPE_TYPE_MESSAGE | PIPE_READMODE_MESSAGE | PIPE_WAIT,
    ↪ //передача сообщений как( чтение, так и запись)
65                                        5, //максимальное число экземпляров каналов равно 5 число(
    ↪ клиентов)
66                                        SIZE_OF_BUF,
67                                        SIZE_OF_BUF,
68                                        5000,
69                                        &sa); //размеры выходного и входного буферов канала, 5
    ↪ секунд – длительность для функции WaitNamedPipe
70          if (hNamedPipe == INVALID_HANDLE_VALUE) // Если возникла ошибка, выводим ее код и
    ↪ завершаем
71              //работу приложения
72          {
73              fprintf(stdout, "CreateNamedPipe: Error %ld\n", GetLastError());
74              getchar();
75              return 1000;
76          }
77          printf("Named pipe created successfully\n");
78          // Выводим сообщение о начале процесса создания канала
79          printf("waiting for connect\n");
80          // Ожидаем соединения со стороны клиента
81
82          if (!ConnectNamedPipe(hNamedPipe, NULL))
83          {
84              // При возникновении ошибки выводим ее код
85              printf("error with function ConnectNamedPipe\n");
86              getchar();
87              CloseHandle(hNamedPipe);
88              return 1001;
89          }
90          // Выводим сообщение об успешном создании канала
91          fprintf(stdout, "Client connected\n");
92          t = CreateThread(NULL, 0, threadHandler, (LPVOID)hNamedPipe, 0, NULL);
93          CloseHandle(t);
94      }
```

```
95      printf("server is ending\n press any key\n");
96      getchar();
97      return 0;
98  }
```

Client для работы по сети:

```
1  #include <windows.h>
2  #include <stdio.h>
3  #define SIZE_OF_BUF 1000
4
5  int main(int argc, char *argv[]) {
6      HANDLE hNamedPipe; // Идентификатор канала Pipe
7      DWORD readbytes,writebytes; // количество байт принятых и переданных в канал
8      char buf[SIZE_OF_BUF]; // Буфер для передачи данных
9      printf("Client is started\n Try to use WaitNamedPipe\n"); //ожидаем"" пока освободится
       ↪ канал
10     //адрес сервера и имя канала
11     LPCWSTR szPipeName = L"\\\\WIN-IS-HARD\\pipe\\$$MyPipe$$";
12     //ожидаем"" пока освободится канал
13     if (!WaitNamedPipe(szPipeName, NMPWAIT_WAIT_FOREVER)) {
14         printf("pipe wasn't created\n getlasterror = %ld", GetLastError());
15         getchar();
16         return 1000;
17     } // Создаем канал с процессомсервером-
18     hNamedPipe = CreateFile(szPipeName, GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING
       ↪ , 0, NULL); // Если возникла ошибка, выводим ее код и
19     // завершаем работу приложения
20     if(hNamedPipe == INVALID_HANDLE_VALUE) {
21         fprintf(stdout,"CreateFile: Error %ld\n",GetLastError());
22         getchar();
23         return 1001;
24     } // Выводим сообщение о создании канала
25     printf("successfully connected\n input message\n"); // Цикл обмена данными с серверным
       ↪ процессом
26     while(1) { // Выводим приглашение для ввода команды printf("cmd>"); // Вводим текстовую
       ↪ строку
27         fgets(buf,SIZE_OF_BUF,stdin); // Передаем введенную строку серверному процессу // в
       ↪ качестве команды
28         if(!WriteFile(hNamedPipe, buf, strlen(buf) + 1, &writebytes, NULL)) {
29             printf("connection refused\n");
30             break;
31         } // Получаем эту же команду обратно от сервера
32         if(ReadFile(hNamedPipe, buf, SIZE_OF_BUF, &readbytes, NULL))
33             printf("Received from server: %s\n", buf); // Если произошла ошибка, выводим ее
       ↪ код и // завершаем работу приложения
34         else {
35             fprintf(stdout,"ReadFile: Error %ld\n", GetLastError());
36             getchar();
37             break;
38         } // В ответ на команду "exit" завершаем цикл // обмена данными с серверным процессом
39         if(!strncmp(buf, "exit",4))
40             break;
41     } // Закрываем идентификатор канала
42     CloseHandle(hNamedPipe);
43     printf("client is ending\n Press any key\n");
44     getchar();
45     return 0;
46 }
```

Программы были протестированы в локальной сети. Программы npipe_server_bi_multi_net и npipe_client_bi_net запускались на разных машинах с ОС Windows 10. Компьютеры были объединены в общую домашнюю группу.

Результат работы npipe_server_bi_multi_net на основной машине приведен на рисунке 5.

Результат работы npipe_client_bi_net на дополнительной машине приведен на рисунке 6.

Из-за сложности получения в программе собственных сетевых адресов для подтверждения общения по сети, было решено отловить трафик утилитой WireShark. Трафик отлавливался на основной машине (сторона сервера)

Инициализация общения представлена на рисунке 7.

Пример записи и последующего чтения представлен на рисунке 8.

Этап завершения общения представлен на рисунке 9.

Из записи трафика видно, что общение происходит между двумя машинами, а нагрузка на сеть достаточно велика.

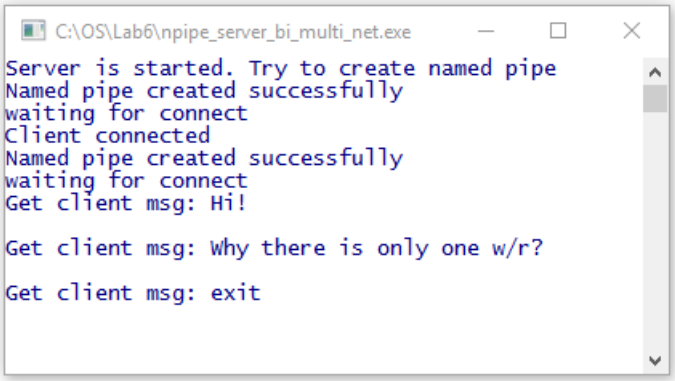

Рис. 5: Результат работы npipe_server_bi_multi_net

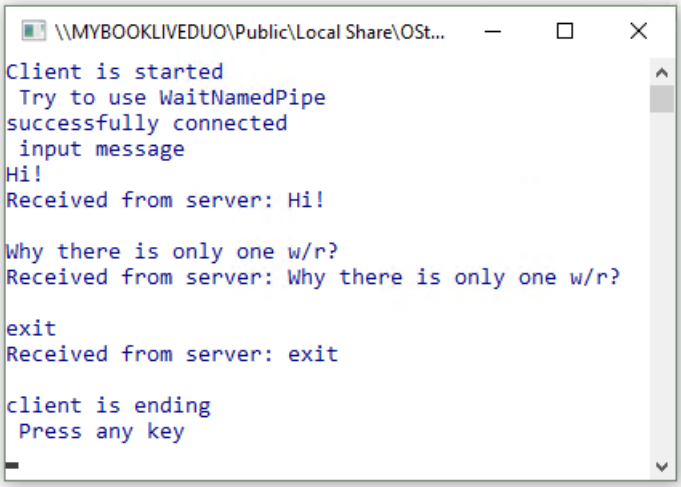

Рис. 6: Результат работы npipe_client_bi_net

Source	Destination	Protoc	Length	Info
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	526	Negotiate Protocol Response
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	252	Negotiate Protocol Request
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	586	Negotiate Protocol Response
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	240	Session Setup Request, NTLMSSP_NEGOTIATE
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	379	Session Setup Response, Error: STATUS_MORE_PROCESSING_REQUI
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	695	Session Setup Request, NTLMSSP_AUTH, User: WIN-IS-HARD\Home
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	179	Session Setup Response
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	186	Tree Connect Request Tree: \\WIN-IS-HARD\IPC$
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	158	Tree Connect Response
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	198	Ioctl Request FSCTL_QUERY_NETWORK_INTERFACE_INFO
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	494	Ioctl Response FSCTL_QUERY_NETWORK_INTERFACE_INFO
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	232	Ioctl Request FSCTL_PIPE_WAIT Pipe: $$MyPipe$$
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	190	Ioctl Response FSCTL_PIPE_WAIT
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	218	Create Request File: $$MyPipe$$
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	230	Create Response File: $$MyPipe$$
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	182	GetInfo Request FILE_INFO/SMB2_FILE_STANDARD_INFO File: $$M
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	174	GetInfo Response

Рис. 7: Инициализация общения сервера и клиента

fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	195	Write Request Len:5 Off:0 File: $$MyPipe$$
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	158	Write Response
fe80::a593:35fa:96a9:9f2c	fe80::58c4:2b5b:1f7:4fa7	SMB2	191	Read Request Len:1000 Off:0 File: $$MyPipe$$
fe80::58c4:2b5b:1f7:4fa7	fe80::a593:35fa:96a9:9f2c	SMB2	163	Read Response

Рис. 8: Один цикл обмена данными

```
fe80::a593:35fa:96a9:9f2c  fe80::58c4:2b5b:1f7:4fa7  SMB2    196 Write Request Len:6 Off:0 File: $$MyPipe$$
fe80::58c4:2b5b:1f7:4fa7   fe80::a593:35fa:96a9:9f2c SMB2    158 Write Response
fe80::a593:35fa:96a9:9f2c  fe80::58c4:2b5b:1f7:4fa7  SMB2    191 Read Request Len:1000 Off:0 File: $$MyPipe$$
fe80::58c4:2b5b:1f7:4fa7   fe80::a593:35fa:96a9:9f2c SMB2    164 Read Response
fe80::a593:35fa:96a9:9f2c  fe80::58c4:2b5b:1f7:4fa7  SMB2    166 Close Request File: $$MyPipe$$
fe80::58c4:2b5b:1f7:4fa7   fe80::a593:35fa:96a9:9f2c SMB2    202 Close Response
fe80::a593:35fa:96a9:9f2c  fe80::58c4:2b5b:1f7:4fa7  SMB2    146 Tree Disconnect Request
fe80::58c4:2b5b:1f7:4fa7   fe80::a593:35fa:96a9:9f2c SMB2    146 Tree Disconnect Response
fe80::a593:35fa:96a9:9f2c  fe80::58c4:2b5b:1f7:4fa7  SMB2    146 Session Logoff Request
fe80::58c4:2b5b:1f7:4fa7   fe80::a593:35fa:96a9:9f2c SMB2    146 Session Logoff Response
```

Рис. 9: Инициализация общения сервера и клиента

3.3 Сокеты

Возможность взаимодействия с другими системами обеспечивается в Windows поддержкой сокетов (sockets) Windows Sockets — совместимого и почти точного аналога сокетов Berkeley Sockets, де-факто играющих роль промышленного стандарта.

Winsock API поддерживается библиотекой DLL (WS232.DLL), для получения доступа к которой следует подключить к программе библиотеку WS232.LIB. Эту DLL следует инициализировать с помощью нестандартной, специфической для Winsock функции WSAStartup, которая должна быть первой из функций Winsock, вызываемых программой. Когда необходимость в использовании функциональных возможностей Winsock отпадает, следует вызывать функцию WSACleanup. В QtCreator подключение происходит с помощью добавления в файл проекта (.pro) строчки "LIBS += -lws2_32".

После инициализации библиотеки сокетов можно использовать стандартные функции работы с сокетами.

Взаимодействие в сети осуществляется между клиентом и сервером. Клиент посылает серверу некоторый запрос. Сервер обрабатывает запрос и шлет ответ. Сам по себе сокет – это оконечная точка соединения, которая идентифицируется 4 значениями: IP адрес отправителя, порт отправителя, IP адрес получателя, порт получателя. Порт – идентификатор процесса в ОС с точки зрения сетевого взаимодействия. Порт напрямую связан с протоколом. Например, в ОС могут быть два процесса с одинаковым номером порта, но использующие при этом разные протоколы.

Программа локального обмена сокетами с использованием потокового протокола с установлением соединения (TCP в стеке TCP/IP). Для потоковых протоколов (к которым относится протокол TCP в стеке TCP/IP) необходимо применять средства, позволяющие определить границы сообщений в передаваемых данных, так как данный вид протоколов имеет дело с доставкой только потока байт (при этом гарантируется порядок доставки).

Исходный код программы tcp_server:

```
1  #include <winsock2.h>
2  #include <ws2tcpip.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  // Need to link with Ws2_32.lib, Mswsock.lib, and Advapi32.lib
7  #define MAX_STR_LEN 255
8  #define SIZE_OF_BUF 255
9
10
11 int recvn(SOCKET fd, char *bp, size_t len) {
12     return recv(fd, bp, len, MSG_WAITALL);
13 }
14
15
16 int sendn(SOCKET s, char* buf, int lenbuf, int flags) {
17     int bytesSended = 0; //
18     int n; //
19     while (bytesSended < lenbuf)  {
20         n = send(s, buf + bytesSended, lenbuf - bytesSended, flags);
21         if (n < 0) {
22             printf("Error with send in sendn\n");
23             break;
24         }
25         bytesSended += n;
26     }
27     return (n == -1 ? -1 : bytesSended);
28 }
29
30 int recvLine(SOCKET sock, char* buffer, int buffSize) { //функция приема сообщения
```

```
31      char* buff = buffer; //указатель на начало внешнего буфера
32      char* currPosPointer; //указатель для работы со временным буфером
33      int count = 0; //число прочитанных символов без( удаления из буфера сокета)
34      char tempBuf[100]; //временный буфер для приема
35      char currChar; //текущий анализируемый символ ищем( разделитель)
36      int tmpcount = 0;
37      while (--buffSize > 0){
38          if (--count <= 0) {
39              recvn(sock, tempBuf, tmpcount);
40              count = recv(sock, tempBuf, sizeof (tempBuf), MSG_PEEK);
41              if (count <= 0) { return count; }
42              currPosPointer = tempBuf;
43              tmpcount = count;
44          }
45          currChar = *currPosPointer++;
46          *buffer++ = currChar;
47          if (currChar == '\n') {
48              *(buffer - 1) = '\0';
49              recvn(sock, tempBuf, tmpcount - count + 1);
50              return buffer - buff-1;
51          }
52      }
53      return -1;
54  }
55
56  int sendLine(int sock, char* str) {
57      char tempBuf[MAX_STR_LEN];
58      strcpy(tempBuf, str);
59      if(tempBuf[strlen(tempBuf)-1]!='\n')
60          strcat(tempBuf, "\n");
61      return sendn(sock, tempBuf, strlen(tempBuf), 0);
62  }
63
64  DWORD WINAPI threadHandler(LPVOID param){
65      SOCKET client_socket = (SOCKET)param;
66      if (client_socket == INVALID_SOCKET) {
67          printf("error with accept socket. GetLasterror= %ld\n", GetLastError());
68          return 1003;
69      }
70      char buf[SIZE_OF_BUF]; //буфер приема и передачи сообщения
71      int readbytes; //число прочитанных байт
72      while (1) {
73          if ((readbytes = recvLine(client_socket, buf, SIZE_OF_BUF)) == 0) {
74              printf("Connection refused\n");
75              break;
76          }
77          else if (readbytes == -1) {
78              printf("buf is small\n");
79              return 2000;
80          }
81          printf("get msg from client \"%s\" with size= %d\n", buf, readbytes);
82          sendLine(client_socket, buf); //sendn(client_socket,buf,readbytes,0); шлем//
        ↪ сообщение обратно клиенту
83          if (strncmp(buf, "exit", 4) == 0) break;
84      }
85      closesocket(client_socket);
86      return 0;
87  }
88
89
90  int main(void) {
91      //используется для инициализации библиотеки сокетов
92      WSADATA WSStartData; //Инициализация WinSock и проверка его запуска
93      if (WSAStartup(MAKEWORD(2, 0), &WSStartData) != 0) {
94          printf("WSAStartup failed with error: %ld\n", GetLastError());
95          return 100;
96      } //создание сокета
97      SOCKET server_socket; //по умолчанию используется протокол tcp
98      printf("Server is started.\nTry to create socket ----------------------");
99      if((server_socket = socket( AF_INET, SOCK_STREAM, 0 )) ==INVALID_SOCKET) {
100         printf("error with creation socket. GetLasterror= %ld\n",GetLastError());
101         return 1000;
102     }
103     printf("CHECK\n"); //Привязывание сокета конкретному IP и номеру порта
104     struct sockaddr_in sin; sin.sin_addr.s_addr=inet_addr("127.0.0.1"); // используем
        ↪ локальную машину
```

```
105     sin.sin_port=htons(7500); // может быть любым кроме зарезервированных
106     sin.sin_family=AF_INET; printf("Try to bind socket ————————————————————");
107     if ( bind( server_socket, (struct sockaddr *)&sin, sizeof(sin) ) !=0 ) {
108         printf("error with bind socket. GetLasterror= %ld\n",GetLastError());
109         return 1001;
110     }
111     printf("CHECK\n"); //делаем сокет прослушиваемым
112     printf("Try to set socket listening ————————");
113     if(listen(server_socket,5 )!=0) {
114         printf("error with listen socket. GetLasterror= %ld\n",GetLastError());
115         return 1002;
116     }
117     printf("CHECK\n");
118     printf("Server starts listening\n"); //Ждем клиента. Создаем пустую структуру, которая
    ↪ будет содержать параметры сокета, инициирующего соединение
119     struct sockaddr_in from;
120     int fromlen=sizeof(from); // начинаем слушать"" входящие запросы на подключение
121     SOCKET client_socket=accept(server_socket,(struct sockaddr*)&from,&fromlen);
122     if(client_socket==INVALID_SOCKET) {
123         printf("error with accept socket. GetLasterror= %ld\n",GetLastError());
124         return 1003;
125     }
126     printf("get client with IP= %s, port = %d\n",inet_ntoa(from.sin_addr),ntohs(from.
    ↪ sin_port));
127     char buf[SIZE_OF_BUF]; //буфер приема и передачи сообщения
128     int readbytes; //число прочитанных байт
129     while(1) {
130         if((readbytes=recvLine(client_socket,buf,SIZE_OF_BUF))==0) {
131             printf("Connection refused\n");
132             break;
133         }
134         else if(readbytes==-1) {
135             printf("buf is small\n");
136             return 2000;
137         }
138         printf("get msg from client \"%s\" with size= %d\n",buf,readbytes);
139         sendLine(client_socket,buf); //sendn(client_socket,buf,readbytes,0); шлем//
    ↪ сообщение обратно клиенту
140         if (strncmp(buf, "exit", 4) == 0) break;
141     }
142     closesocket(client_socket);
143     closesocket(server_socket);
144     return 0;
145 }
```

Основная функция программы tcp_client:

```
1   int main(void) { //используется для инициализации библиотеки сокетов
2     WSADATA WSStartData; //Инициализация WinSock и проверка его запуска
3     if (WSAStartup(MAKEWORD(2, 0), &WSStartData) != 0) {
4         printf("WSAStartup failed with error: %ld\n", GetLastError());
5         return 100;
6     }
7     int er_code=0; // инициализация клиентского сокета
8     printf("Client is started.\nTry to create socket\n");
9     int client_socket = socket( AF_INET, SOCK_STREAM, 0 );
10    printf("socket created successfully\n");
11    struct sockaddr_in sin;
12    sin.sin_addr.s_addr=inet_addr("127.0.0.1");
13    sin.sin_port=htons(7500);
14    sin.sin_family=AF_INET; // установливаем TCPсоединение–
15    printf("try to connect to server\n");
16    if(connect(client_socket, (struct sockaddr *) &sin,sizeof(sin))!=0) {
17        printf("connect failed with error: %d\n", er_code);
18        return SOCKET_ERROR;
19    }
20    printf("Client connected sucessfully\nEnter msg to send\n————————————————————\n");
21    char buf[SIZE_OF_BUF]; //буфер для приема и передачи сообщений
22    while(1) {
23        fgets(buf,SIZE_OF_BUF,stdin);
24        printf("client sended msg: %s",buf);
25        sendLine(client_socket,buf);
26        recvLine(client_socket,buf,SIZE_OF_BUF);
27        printf("get msg from serv: \"%s\"\n**********\n",buf);
28    } // заканчиваем работу с сокетом клиента
29    closesocket(client_socket);
30    return 0;
```

```
}
```

Результат выполнения программы приведен на рисунке 10.

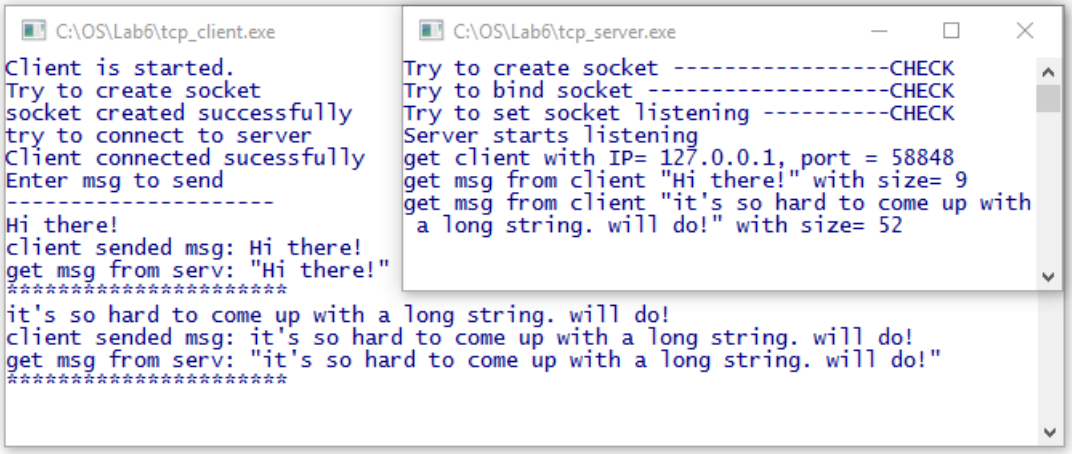

Рис. 10: Результат работы

Для поддержки подключения нескольких клиентов код сервера был немного изменен:

```
DWORD WINAPI threadHandler(LPVOID param){
  SOCKET client_socket = (SOCKET)param;
  if (client_socket == INVALID_SOCKET) {
    printf("error with accept socket. GetLasterror= %d\n", GetLastError());
    return 1003;
  }
  char buf[SIZE_OF_BUF]; //буфер приема и передачи сообщения
  int readbytes; //число прочитанных байт
  while (1) {
    if ((readbytes = recvLine(client_socket, buf, SIZE_OF_BUF)) == 0) {
      printf("Connection refused\n");
      break;
    }
    else if (readbytes == -1) {
      printf("buf is small\n");
      return 2000;
    }
    printf("get msg from client \"%s\" with size= %d\n", buf, readbytes);
    sendLine(client_socket, buf); //sendn(client_socket,buf,readbytes,0); шлем// сообщение
    ↪ обратно клиенту
    if (strncmp(buf, "exit", 4) == 0) break;
  }
  closesocket(client_socket);
  return 0;
}
//.............
// main:
  while (SOCKET client_socket = accept(server_socket, (struct sockaddr*)&from, &fromlen)){
    HANDLE t;
    t = CreateThread(NULL, 0, threadHandler, (LPVOID)client_socket, 0, NULL);
  }
  closesocket(server_socket);

```

С помощью сокетов часто реализовывают сетевые приложения. Для этого необходимо внести незначительные изменения в код сервера и клиента: слушающий сокет сервера необходимо привязывать к адресу `INADDR_ANY`, чтобы он мог принимать соединения с любых адресов; клиенту необходимо указать IP адрес компьютера, на котором запущен сервер.

Результат выполнения сервера и двух клиентов рис. 11. По этому рисунку видно, что общение происходило между разными компьютерами в сети.

Результат выполнения двух удалённых клиентов 12.

Так мы видим, что с одним сервером могут работать сразу несколько клиентов, причём доступ открыт и из сети. Открытие доступа требует дополнительных прав при запуске программы, но получив права, сервер успешно запускается. Обеспечение доступа всех клиентов к общему ресурсу с помощью мьютекса является самым простым и надёжным решением, однако наименее эффективным.

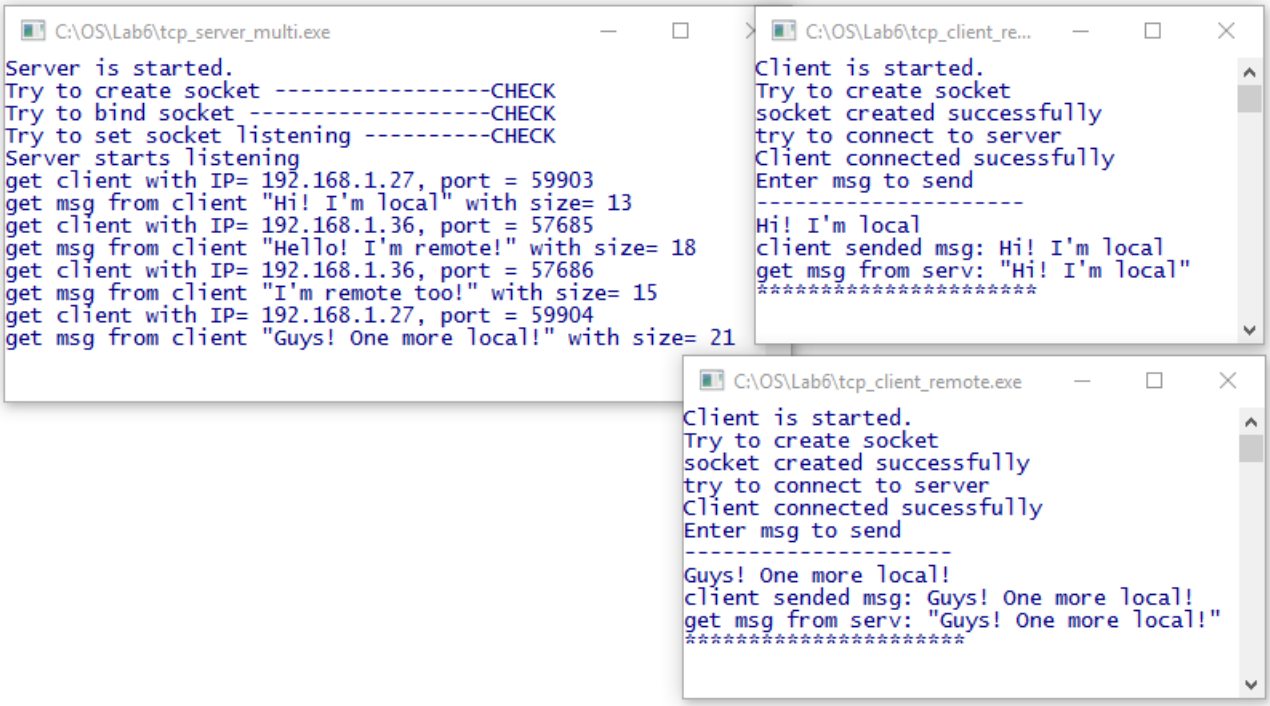

Рис. 11: Основная машина

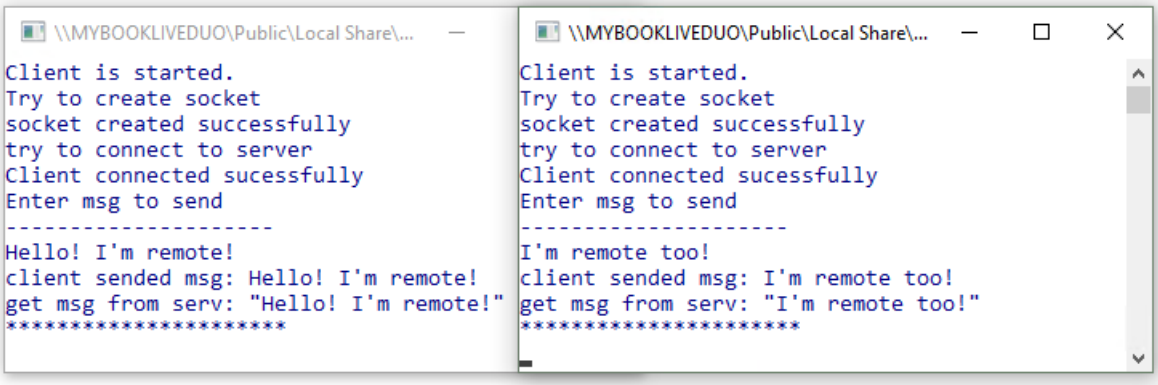

Рис. 12: Дополнительная (удалённая) машина

3.4 Порты завершения

Порт завершения представляет собой специальный механизм в составе ОС, с помощью которого приложение использует объединение (пул) 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 (сокетными) дескрипторами и который будет управлять перекрывающимися I/O операциями, используя определенное количество потоков для обслуживания завершенных запросов.

Опишем основной каркас прикладной программы-приложения, использующей для ввода/вывода модель порта завершения. Это простой ECHO-сервер, получающий информацию от клиента и ее же обратно отправляющего. Предлагается следующие шаги:

1. Создать порт завершения. Четвертый параметр функции оставлен как 0 - только одному рабочему потоку на процессоре будут позволено выполняться в данное время на порте завершения.

2. Создать рабочие потоки для обслуживания завершенных I/O-запросов на порте завершения. Когда вызывается функция создания потока CreateThread(), необходимо указать ту функцию,

которая будет исполняться в рабочем потоке.

3. Сформировать слушающий сокет, чтобы принимать входные подключений на порту 7500.
4. Принять поступившие подключения функцией accept().
5. Создать структуру данных, чтобы представить "per-handle data"и сохранить дескриптор присоединенного сокета в структуре.
6. Связать новый дескриптор сокета, возвращенный из accept(), с портом завершения, вызывая CreateIoCompletionPort(). Передать структуру с "per-handle data"в функцию CreateIoCompletionPort() через параметр ключа завершения.
7. Начать выполнять операции I/O на принятом подключении. По существу, мы будем выдавать один или более асинхронных вызовов WSARec() или WSASend() на новом сокете, используя механизм ввода/вывода с перекрытием. Вызовы этих функций будут исполняться асинхронно и с перекрытием (т.е. одновременно). Когда эти функции завершатся (с тем или иным результатом), рабочий поток обслужит исполненные запросы и будет ждать следующие вызовы.

Исходный код программы tcp_sever_complition:

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <Winsock2.h>
#include <list>  // Используем STL

using namespace std;  // Использовать пространство имен std
#define BUFF_SIZE  128  // Размер буфера
#define PORT 7500  // Номер порта
#define BUF_LEN 128
DWORD WINAPI ServerPool(HANDLE hp);

SOCKET server_sock;  // Прослушивающий сокет сервера
int ClientCount;  // Счетчик клиентов
list <SOCKET> ClientList;  // Список клиентов
//-------------------------------------------------------------------------
struct ovpConnection: public OVERLAPPED
{
    int client_number;  // Номер клиента
    SOCKET sock_handle;  // Сокет клиента
    char * buffer;  // Буфер сообщений
    enum
    {
        op_type_send,  // Посылка
        op_type_recv  // Прием
    }op_type;  // Тип операции
};
//-------------------------------------------------------------------------
int main(int argc, char *argv[])
{
    int err;  // Возвращаемое значение
    //  char buffer[128]; // Буфер для сообщений
    WORD wVersionRequested;  // Запрашиваемая версия
    WSADATA wsaData;  // Структура инфции– о сокетах
    HANDLE hCp;  // Описатель порта завершения
    //  LPOVERLAPPED overlapped; // Структура асинхронного I/O
    HANDLE hThread;  // Хендл потока
    DWORD ThreadId;  // Идентификатор потока
    DWORD flags;  // Флаги фции– WSARecv
    //Инициализация библиотеки ws2_32.dll
    wVersionRequested = MAKEWORD(2, 2);
    err = WSAStartup(wVersionRequested, &wsaData);
    if (err == SOCKET_ERROR)
    {
        printf("Error on WSAStartup %d\n", WSAGetLastError());
        WSACleanup(); // Завершение работы
        _getch();
        return 2;
    }
    //Создаем порт завершения
    hCp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);
    if (hCp == NULL)
    {
        printf("Error on CreateIoCompletionPort %ld\n", GetLastError());
```

```
55          WSACleanup(); // Завершение работы
56          _getch();
57          return 3;
58      }
59      // Задаем параметры для прослушивающего сокета сервера
60      server_sock = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL, NULL,
61          WSA_FLAG_OVERLAPPED);
62      if (server_sock == INVALID_SOCKET)
63      {
64          printf("Error on WSASocket %d\n", WSAGetLastError());
65          WSACleanup(); //Завершение работы с сокетами
66          _getch();
67          return 4;
68      }
69      else
70      {
71          // Используем ранее созданный порт завершения
72          if (CreateIoCompletionPort((HANDLE)server_sock, hCp, 0, 0) == NULL)
73          {
74              printf("Error on CreateIoCompletionPort %ld\n", GetLastError());
75              WSACleanup(); //Завершение работы
76              _getch();
77              return 5;
78          }
79      }
80      //Заполняем структуру адреса и подключаем сокет к коммуникационной среде
81      SOCKADDR_IN sinServer;
82      sinServer.sin_family = AF_INET;
83      sinServer.sin_port = htons(PORT);
84      sinServer.sin_addr.s_addr = INADDR_ANY;
85      err = bind(server_sock, (LPSOCKADDR)&sinServer, sizeof(sinServer));
86      if (err == -1)
87      {
88          printf("Error on bind %ld\n", GetLastError());
89          WSACleanup(); //Завершение работы
90          _getch();
91          return 6;
92      }
93      //Создаем очередь для ожидания запросов от клиентов на соединение
94      err = listen(server_sock, SOMAXCONN);
95      if (err == -1) {
96          printf("Error on listen № %ld\n", GetLastError());
97          WSACleanup(); // Завершение работы
98          _getch();
99          return 7;
100     }
101     //оздаемC рабочий поток для обслуживания сообщений от порта завершения
102     for (int i=0; i < 8;i++)
103         hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ServerPool, hCp, 0,
104             &ThreadId);
105     ClientCount = 0;
106     printf("Server started \n");
107     //Бесконечный цикл для многократного обслушивания запросов от клиентов
108     while (true)
109     {
110         //Принимаем запрос от программыклиента– на установление связи
111         SOCKADDR_IN sinClient;
112         int lenClient = sizeof(sinClient);
113         SOCKET client = accept(server_sock, (struct sockaddr*)&sinClient,
114             &lenClient);
115         CreateIoCompletionPort((HANDLE)client, hCp, 0, 0);
116         //Добавляем клиента в список
117         ClientList.insert(ClientList.end(), client);
118         // Создаем overlappedструктуру–
119         ovpConnection * op = new ovpConnection;
120         //Заполняем overlappedструктуру–
121         op->sock_handle = client;
122         op->op_type = ovpConnection::op_type_recv;
123         op->buffer = new char[BUFF_SIZE];
124         op->hEvent = 0;
125         op->client_number = ++ClientCount;
126         printf("Client #%d connected, number of active clients "
127             "%d\n",ClientCount,ClientList.size());
128         unsigned long b;
129         WSABUF buf;
130         buf.buf = op->buffer;
```

```
131             buf.len = BUFF_SIZE;
132             flags = 0;
133             err = WSARecv(op->sock_handle, &buf, 1, &b, &flags, op, 0);
134             if (!err)
135             {
136                 printf("Error on WSARecv %d\n", WSAGetLastError());
137             }
138         }
139         return 0;
140 }
141 //-----------------------------------------------------------------------------
142 //Функция потока сервера для обслуживания порта завершения
143 //-----------------------------------------------------------------------------
144 DWORD WINAPI ServerPool(HANDLE hp)
145 {
146     int err;  // Возвращаемое значение
147     unsigned long bytes;  // Колво- байтов
148     unsigned long key;  // Значение, асоциированное с хендлом порта
149     char buffer[BUF_LEN];  // Буфер для сообщений
150     LPOVERLAPPED overlapped; // Структура асинхронного I/O
151     HANDLE hport = hp;  // Дескриптор порта
152     DWORD flags;  // Флаги фции- WSARecv()
153     ZeroMemory(buffer, BUF_LEN);
154     while (true)
155     {
156         // Получаем информацию о завершении операции
157         if (GetQueuedCompletionStatus(hport, &bytes, &key, &overlapped, INFINITE))
158         {
159             // Операция завершена успешно
160             ovpConnection * op = (ovpConnection*)overlapped;
161             // Определяем тип завершенной операции и выполняем соответствующие
162             //действия
163                 switch (op->op_type)
164             {
165                 //Завершена отправка данных
166                 case ovpConnection::op_type_send:
167                     delete[] op->buffer;
168                     delete op;
169                     break;
170                     //Завершен приём данных
171                 case ovpConnection::op_type_recv:
172                     if (!bytes) {
173                         //Соединение с данным клиентом закрыто
174                         //ClientList.remove(op->sock_handle);
175                         closesocket(op->sock_handle);
176                         printf("Client #%d disconnected, number of active "
177                             "clients %d\n",op->client_number,ClientList.size());
178                             break;
179                     }
180                     op->buffer[bytes] = '\0';
181
182                     printf("From client #%d recieved message %s\n", op->client_number, op->
    ↪ buffer);
183                     if (send(op->sock_handle, op->buffer, bytes, 0) ==
184                         SOCKET_ERROR) {
185                         printf("error while sending: %d\n",
186                             WSAGetLastError());
187                     }
188
189                     unsigned long b;
190                     WSABUF buf;
191                     buf.buf = op->buffer;
192                     buf.len = BUFF_SIZE;  // buffer_len - постоянная величина
193                     flags = 0;
194                     err = WSARecv(op->sock_handle, &buf, 1, &b, &flags, op, 0);
195                     if (!err)
196                     {
197                         printf("Error on WSARecv %d\n", WSAGetLastError());
198                     }
199             }
200         }
201         else
202         {
203             if (!overlapped)
204             {
205                 // Ошибка с портом
```

```
206                     // Закрываем все сокеты, закрываем порт, очищаем список
207                     for (list<SOCKET>::iterator
208                         i = ClientList.begin(); i != ClientList.end(); i++)
209                     {
210                         closesocket(*i);
211                     }
212                     ClientList.clear();
213                     closesocket(server_sock);
214                     CloseHandle(hport);
215                     printf("Port error %ld, сервер завершает"
216                         " работу\n",GetLastError());
217                         _getch();
218                     exit(0);
219                 }
220                 else
221                 {
222                     //Закрываем соединение с клиентом
223                     closesocket(((ovpConnection*)overlapped)->sock_handle);
224                     //ClientList.remove(((ovpConnection*)overlapped)->sock_handle);
225                     printf("Client #%d disconnected, number of active clients "
226                         "%d\n",((ovpConnection*)overlapped)->client_number, ClientList.size());
227                 }
228             }
229         }
230         return 0;
231 }
```

Проверим работоспособности сервера в пределах локальной машины и пары ранее рассмотренных tcp клиентов клиентов, рисунок 13.

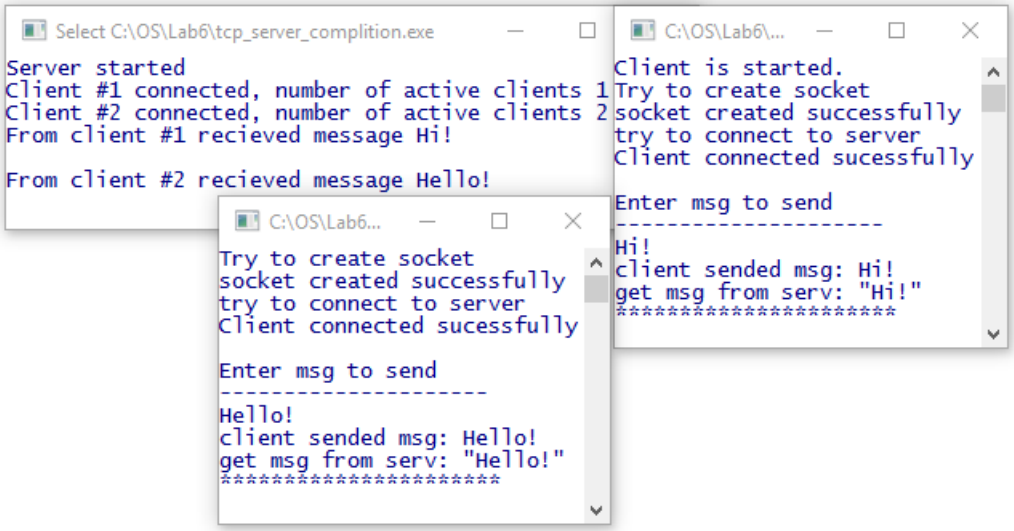

Рис. 13: Локальная проверка

Теперь запустим сервер в нагруженной ситуации. Для этого перепишем клиент, изменив адрес на адрес дополнительной рабочей станции, а ручной ввод текста заменим на вывод заданной строки через раз в 20 секунд. Напишем bat скрипт для запуска 1000 клиентов:

```
1 @ECHO OFF
2 ECHO Script is started
3 set N=1000
4 FOR /l %%i IN (1, 1, %N%) DO (
5   START "TCP client %%i" "tcp_stress.exe"
6 )
7 ECHO Script is finished
8 ECHO Number of process with image "client.exe"
9 Exit
```

На дополнительной машине запустим сервер, а скрип на основной. Будем наблюдать за нагрузкой на дополнительной машине в установившийся период:

Для осознания эффективности проведём такой же эксперимент но с обычным, многопоточным, tcp сервером:

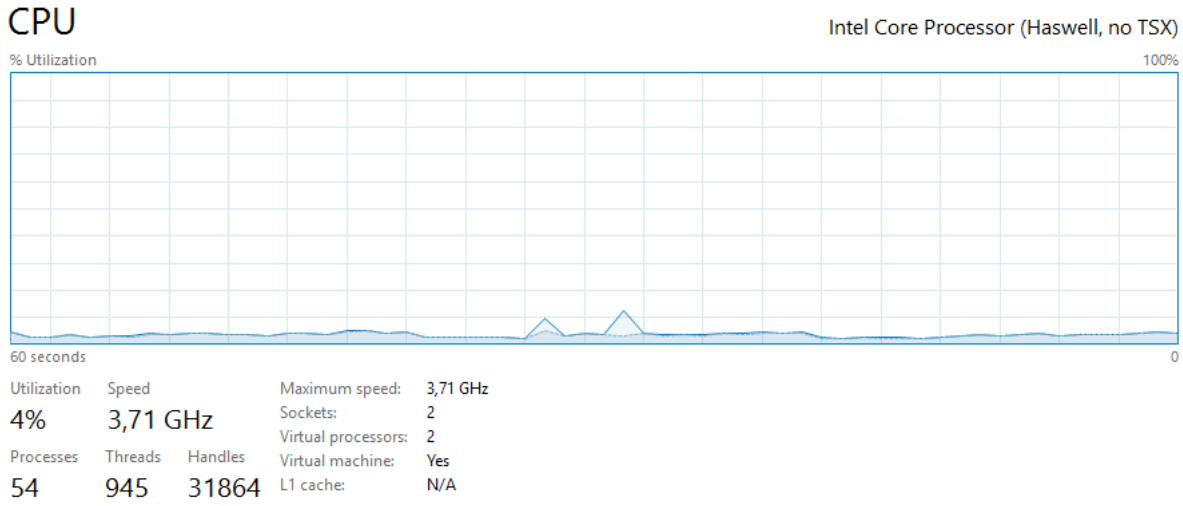

Рис. 14: Нагрузка при использование завершающих портов

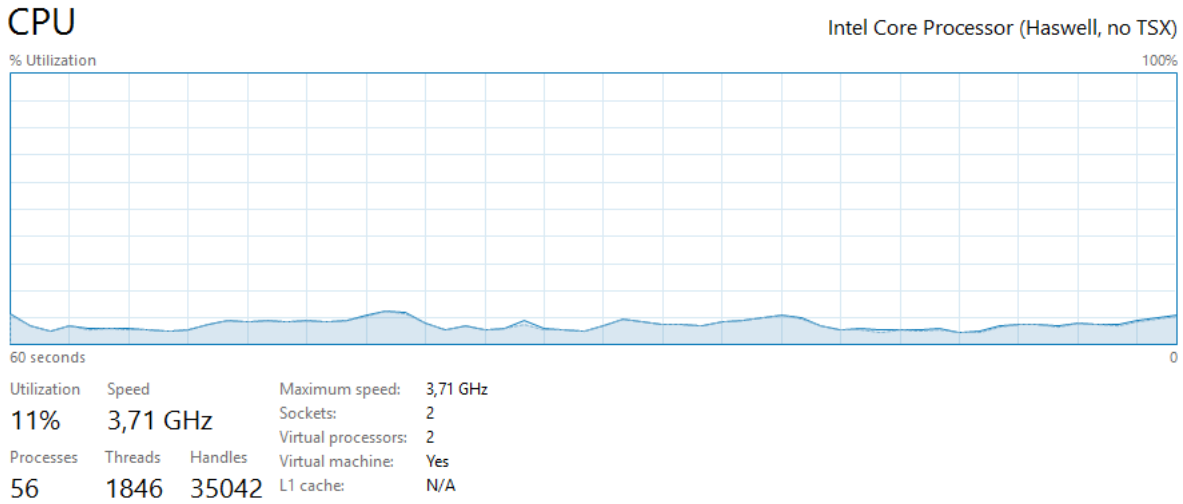

Рис. 15: Нагрузка без использования завершающих портов

При сравнение заметна разница в количестве потоков. А нагрузка на процессор выросла больше чем в два раза, это при том, что часть ресурсов использует и сама система. Таким образом порты завершения являются эффективным решением для многоклиентских приложений.

3.5 Сигналы в Windows

Данное средство IPC в Windows не поддерживается. 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 завершается. По получению данных сигналов процесс может произвести корректное завершение.

С помощью функции SetConsoleCtrlHandler можно установить обработчик на данные сигналы, но отправить сигнал другому приложению мы не можем.

Зарегистрированный обработчик должен проверять тип сигнала на возможность его обработки. 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.

В качестве примера рассмотрим код из msdn. В нем происходит перехват сигналов CTRL+C, CTRL+BREAK. При этом обработчик смотрит, какой сигнал ему передан, и выводит его название. В качестве звуковой индикации работы приложение вызывает функцию Beep. Данная функция вос-

производит звуковой сигнал через динамик консоли с разной частотой и длительностью, задаваемыми ей через параметры.

В функции main регистрируется обработчик сигналов, затем главный поток работает в бесконечном цикле.

Исходный код:

```
#include <windows.h>
#include <stdio.h>

BOOL CtrlHandler( DWORD fdwCtrlType ) {
    switch( fdwCtrlType )  {
        // Handle the CTRL-C signal.
    case CTRL_C_EVENT:
        printf( "Ctrl-C event\n\n" );
        Beep( 750, 300 );
        return( TRUE ); //CTRL-CLOSE: confirm that the user wants to exit.
    case CTRL_CLOSE_EVENT:
        Beep( 600, 200 );
        printf( "Ctrl-Close event\n\n" );
        return(TRUE); // Pass other signals to the next handler.
    case CTRL_BREAK_EVENT:
        Beep( 900, 200 );
        printf( "Ctrl-Break event\n\n" );
        return FALSE;
    case CTRL_LOGOFF_EVENT:
        Beep( 1000, 200 );
        printf( "Ctrl-Logoff event\n\n" );
        return FALSE;
    case CTRL_SHUTDOWN_EVENT:
        Beep( 750, 500 );
        printf( "Ctrl-Shutdown event\n\n" );
        return FALSE;
    default:
        return FALSE;
    }
}

int main( void ) {
    if( SetConsoleCtrlHandler( (PHANDLER_ROUTINE) CtrlHandler, TRUE ) ) {
        printf( "\nThe Control Handler is installed.\n" );
        printf( "\n -- Now try pressing Ctrl+C or Ctrl+Break, or" );
        printf( "\n try logging off or closing the console...\n" );
        printf( "\n(...waiting in a loop for events...)\n\n" );
        while( 1 ){ }
    } else {
        printf( "\nERROR: Could not set control handler");
        return 1;
    }
    return 0;
}
```

Результат выполнения программы приведен на рисунке 16. Так как сигналы не являются сред-

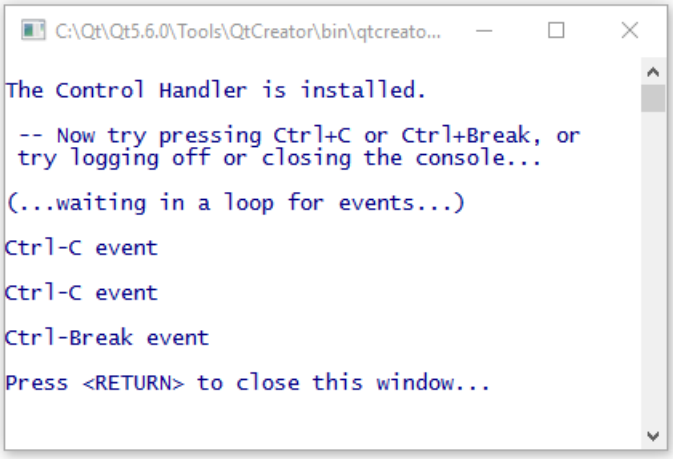

Рис. 16: Результат работы

ством IPC в Windows, а средством пользователя и системы с процессом, то можно предположить что собственные обработчики сигналов задуманы для корректного завершения программы, например, закрытия всех поток и корректное окончание записи файлов, для которых простой обрыв неприемлем. Так реализуем популярный в программах с графическим интерфейсом вопрос "Уверены ли вы, что хотите закрыть программу". Для этого изменим обработчик:

```
1   printf( "Do you want to close this app?\nPress Ctrl-C one more time to close it...\n\n" );
2   SetConsoleCtrlHandler((PHANDLER_ROUTINE)CtrlHandler, FALSE);
3   return( TRUE ); //CTRL-CLOSE: confirm that the user wants to exit.
```

Результат выполнения программы при двухкратном нажатие Ctrl-C приведен на рисунке 17.

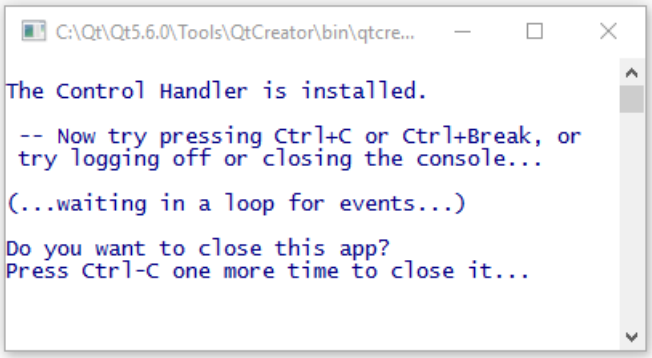

Рис. 17: Результат работы с собственным обработчиком

Программа при первом сигнале попадает в свой обработчик, где выводит инструкцию пользователю и устанавливает обработчик по умолчанию. При втором сигнале вызывается обработчик по умолчанию, а он приводит к завершению процесса.

3.6 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. Приведем системные функции, которые позволяют запрограммировать такое взаимодействие.

Первый участвующий в обмене информацией процесс создает объект "проекция 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. Процесс затем использует дескриптор объекта "проекция файла 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.

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.

Для записи данных в память используется функция CopyMemory(), первый аргумент которой – возвращаемый функцией MapViewOfFile() указатель, а следующие – характеризуют записываемые данные.

Когда процессу больше не нужен доступ к объекту "проекция файла в память он должен вызвать функцию CloseHandle() для дальнейшего освобождения ресурса. При условии, что все дескрипторы закрыты (не осталось процессов, использующих этот ресурс), система может освободить секцию файла подкачки, используемого объектом.

Задание 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.

Исходный код первой:

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <conio.h>
4 #define BUF_SIZE 256
5 TCHAR szName[] = TEXT("MyFileMappingObject");
```

```
6  TCHAR szMsg[] = TEXT("Message from first process");
7  HANDLE WINAPI mutex;
8  int main()
9  {
10     HANDLE hMapFile;
11     LPCTSTR pBuf;
12     mutex = CreateMutex(NULL, false, TEXT("SyncMutex"));
13     // create a memory, wicth two proccess will be working
14     hMapFile = CreateFileMapping(
15         INVALID_HANDLE_VALUE, // использование файла подкачки
16         NULL, // защита по умолчанию
17         PAGE_READWRITE, // доступ к чтениюзаписи/
18         0, // макс. размер объекта
19         BUF_SIZE, // размер буфера
20         szName); // имя отраженного в памяти объекта
21     if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE)
22     {
23         printf("Не может создать отраженный в памяти объект (%ld).\n",
24             GetLastError());
25         return 1;
26     }
27     pBuf = (LPTSTR)MapViewOfFile(hMapFile, //дескриптор проецируемого в памяти объекта
28         FILE_MAP_ALL_ACCESS, // разрешение чтениязаписирежим/( доступа)
29         0, //Старшее слово смещения файла, где начинается отображение
30         0, //Младшее слово смещения файла, где начинается отображение
31         BUF_SIZE); //Число отображаемых байтов файла
32     if (pBuf == NULL)
33     {
34         printf("Представление проецированного файла невозможно (%ld).\n",
35             GetLastError());
36         return 2;
37     }
38     int i = 0;
39     while (true)
40     {
41         i = rand();
42         itoa(i, (char *)szMsg, 10);
43         WaitForSingleObject(mutex, INFINITE);
44         CopyMemory((PVOID)pBuf, szMsg, sizeof(szMsg));
45         printf("write message: %s\n", (char *)pBuf);
46         Sleep(1000); //необходимо только для отладки – для удобства представления и анализа
   ↪ результатов//
47         ReleaseMutex(mutex);
48     }
49     // освобождение памяти и закрытие описателя handle
50     UnmapViewOfFile(pBuf);
51     CloseHandle(hMapFile);
52     CloseHandle(mutex);
53 }
```

Исходный код второй программы:

```
1  #include <windows.h>
2  #include <stdio.h>
3  #include <conio.h>
4  #define BUF_SIZE 256
5  #define TIME 15 // number of reading operation in this process
6  TCHAR szName[] = TEXT("MyFileMappingObject");
7  HANDLE WINAPI mutex;
8  int main()
9  {
10     HANDLE hMapFile;
11     LPCTSTR pBuf;
12     mutex = OpenMutex(
13         MUTEX_ALL_ACCESS, // request full access
14         FALSE, // handle not inheritable
15         TEXT("SyncMutex")); // object name
16     if (mutex == NULL)
17         printf("OpenMutex error: %ld\n", GetLastError());
18     else printf("OpenMutex successfully opened the mutex.\n");
19     hMapFile = OpenFileMapping(
20         FILE_MAP_ALL_ACCESS, // доступ к чтениюзаписи/
21         FALSE, // имя не наследуется
22         szName); // имя проецируемого" " объекта
23     if (hMapFile == NULL)
24     {
25         printf("Невозможно открыть объект проекция файла (%ld).\n", GetLastError());
```

```
26          return 1;
27      }
28      pBuf = (LPTSTR) MapViewOfFile(hMapFile, // дескриптор проецируемого"" объекта
29          FILE_MAP_ALL_ACCESS, // разрешение чтениязаписи/
30          0,
31          0,
32          BUF_SIZE);
33      if (pBuf == NULL)
34      {
35          printf("Представление проецированного файла (%ld) невозможно .\n", GetLastError());
36          return 2;
37      }
38      for (int i = 0; i<TIME; i++)
39      {
40          WaitForSingleObject(mutex, INFINITE);
41          printf("read message: %s\n", (char *)pBuf);
42          ReleaseMutex(mutex);
43      }
44      UnmapViewOfFile(pBuf);
45      CloseHandle(hMapFile);
46      return 0;
47  }
```

Результат выполнения программы приведен на рисунке 18.

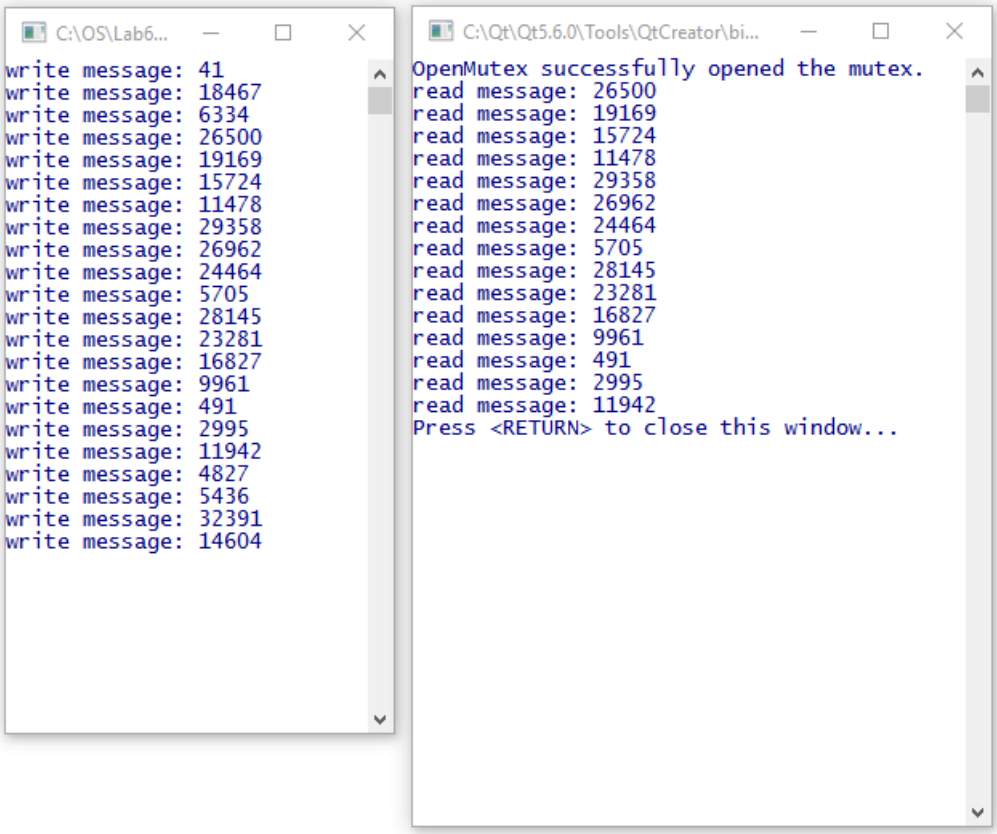

Рис. 18: Результат работы

Для организации синхронизации доступа к памяти в данном примере рассматривается использование мьютексов. В первом процессе создается именованный мьютекс, который "защищает"критические участки кода, в данном случае - запись в общую разделяемую память.

Записываемая в буфер информация - это случайное число, которое генерируется функцией rand()перед каждой записью.

Сообщения извлекаются из памяти правильно, следовательно, система разделения доступа к памяти работает корректно.

3.7 Почтовые слоты

MailSlot – механизм синхронизации, иначе называемый «почтовый ящик». Каждый слот реализуется как псевдофайл в оперативной памяти и содержит некоторое количество записей («сообщений»), которые могут быть прочтены всеми компьютерами в сетевом домене. Общий размер данных не дможет превышать 64K. В отличие от дисковых файлов, файлы MailSlot временные. Когда все указатели на MailSlot закрываются, MailSlot и все данные, которые он содержит, удаляются. Для обмена посредством MailSlot создается клиент-серверное приложение. MailSlot сервер – является процессом, который создает и владеет MailSlot. При создании сервер получает указатель, используемый затем при чтении или записи данных им самим или другим процессом, получившим указатель на MailSlot. Создать слот можно только локально, а получать доступ (обращаться) и локально и удалённо.

Для демонстрации работы с почтовыми слотами были написаны программы сервера и клиента. Исходный сервера:

```
1  #include <windows.h>
2  #include <stdio.h>
3  #include <conio.h>
4
5  int main() {
6      // Код возврата из функций
7      BOOL   fReturnCode;
8      // Размер сообщения в байтах
9      DWORD  cbMessages;
10     // Количество сообщений в канале Mailslot
11     DWORD  cbMsgNumber;
12     // Идентификатор канала Mailslot
13     HANDLE hMailslot;
14     // Имя создаваемого канала Mailslot
15     LPCWSTR  lpszMailslotName = L"\\\\.\\mailslot\\$Channel$";
16     // Буфер для передачи данных через канал
17     char   szBuf[512];
18     // Количество байт данных, принятых через канал
19     DWORD  cbRead;
20
21     // Создаем канал Mailslot, имеющий имя lpszMailslotName
22     hMailslot = CreateMailslot(
23         lpszMailslotName, 0,
24         MAILSLOT_WAIT_FOREVER, NULL);
25
26     // Если возникла ошибка, выводим ее код и зваершаем
27     // работу приложения
28     if (hMailslot == INVALID_HANDLE_VALUE)
29     {
30         fprintf(stdout, "CreateMailslot: Error %ld\n",
31             GetLastError());
32         getch();
33         return 0;
34     }
35
36     // Выводим сообщение о создании канала
37     fprintf(stdout, "Mailslot created\n");
38
39     // Цикл получения команд через канал
40     while (1)
41     {
42         // Определяем состояние канала Mailslot
43         fReturnCode = GetMailslotInfo(
44             hMailslot, NULL, &cbMessages,
45             &cbMsgNumber, NULL);
46
47         if (!fReturnCode)
48         {
49             fprintf(stdout, "GetMailslotInfo: Error %ld\n",
50                 GetLastError());
51             getch();
52             break;
53         }
54
55         // Если в канале есть Mailslot сообщения,
56         // читаем первое из них и выводим на экран
57         if (cbMsgNumber != 0)
58         {
59             if (ReadFile(hMailslot, szBuf, 512, &cbRead, NULL))
60             {
```

```
61                  // Выводим принятую строку на консоль
62                  printf("Received: <%s>\n", szBuf);
63
64                  // Если пришла команда "exit",
65                  // завершаем работу приложения
66                  if (!strcmp(szBuf, "exit"))
67                      break;
68              }
69              else
70              {
71                  fprintf(stdout, "ReadFile: Error %ld\n",
72                      GetLastError());
73                  getch();
74                  break;
75              }
76          }
77
78          // Выполняем задержку на  500 миллисекунд
79          Sleep(500);
80      }
81
82      // Перед завершением приложения закрываем
83      // идентификатор канала Mailslot
84      CloseHandle(hMailslot);
85      return 0;
86  }
```

Исходный код клиента:

```
1  #include <windows.h>
2  #include <stdio.h>
3  #include <conio.h>
4
5  int main(int argc, char *argv[])
6  {
7      // Идентификатор канала Mailslot
8      HANDLE hMailslot;
9      // Буфер для имени канала Mailslot
10     LPCWSTR  szMailslotName = L"\\\\.\\mailslot\\$Channel$";
11     // Буфер для передачи данных через канал
12     char  szBuf[512];
13     // Количество байт, переданных через канал
14     DWORD  cbWritten;
15     // Создаем канал с процессом MSLOTS
16     hMailslot = CreateFile(
17         szMailslotName, GENERIC_WRITE,
18         FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
19
20     // Если возникла ошибка, выводим ее код и
21     // завершаем работу приложения
22     if (hMailslot == INVALID_HANDLE_VALUE)
23     {
24         fprintf(stdout, "CreateFile: Error %ld\n",
25             GetLastError());
26         getch();
27         return 0;
28     }
29
30     // Выводим сообщение о создании канала
31     fprintf(stdout, "\nConnected. Type 'exit' to terminate\n");
32
33     // Цикл посылки команд через канал
34     while (1)
35     {
36         // Выводим приглашение для ввода команды
37         printf("cmd>");
38
39         // Вводим текстовую строку
40         gets(szBuf);
41
42         // Передаем введенную строку серверному процессу
43         // в качестве команды
44         if (!WriteFile(hMailslot, szBuf, strlen(szBuf) + 1,
45             &cbWritten, NULL))
46             break;
47
48         // В ответ на команду "exit" завершаем цикл
```

```
49          // обмена данными с серверным процессом
50          if (!strcmp(szBuf, "exit"))
51              break;
52      }
53
54      // Закрываем идентификатор канала
55      CloseHandle(hMailslot);
56      return 0;
57  }
```

Результат локального выполнения программ приведен на рисунке 19.

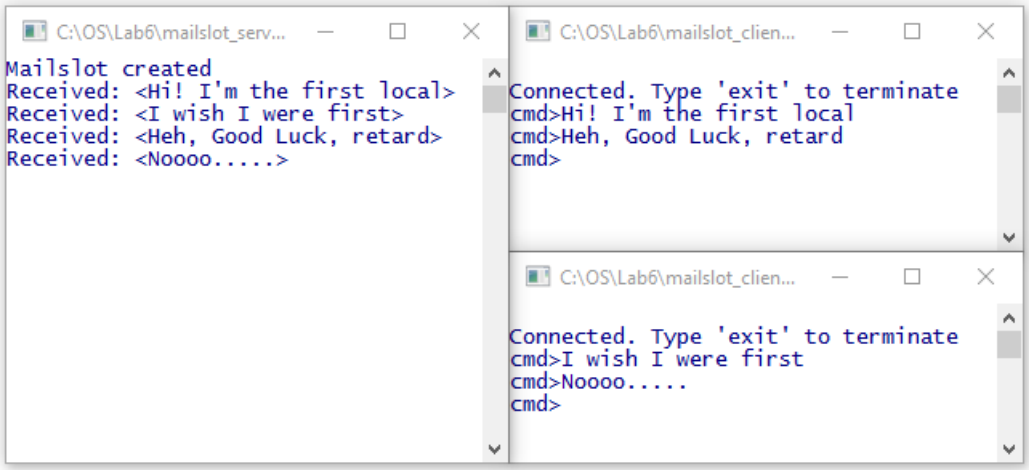

Рис. 19: Локальный запуск

Используя почтовые слоты можно передавать данные между компьютерами в локальной сети. Для этого клиенту укажем имя основной машины "WIN-IS-HARD".

Результат сетевого выполнения программ приведен на рисунках 20 и 21.

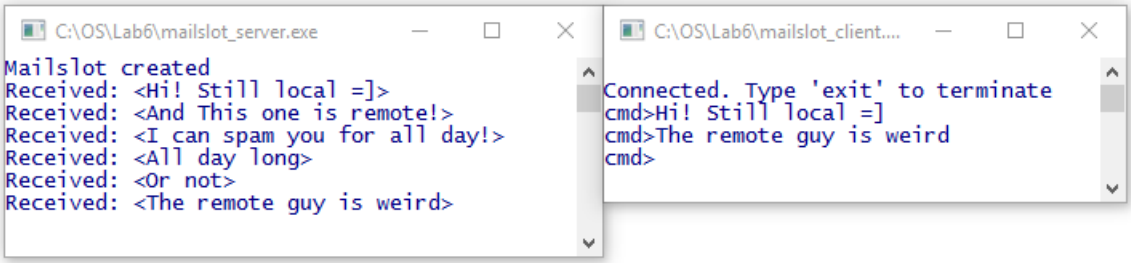

Рис. 20: Сетевой запуск на локальной машине

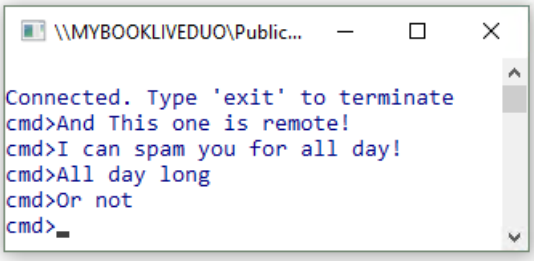

Рис. 21: Сетевой запуск на дополнительной машине

Для подтверждения работы по сети был записан соответствующий трафик. Он представлен на рисунке 22

Source	Destination	Protocol	Length	Info
192.168.1.36	192.168.1.27	SMB Mailslot	240	Write Mail Slot
192.168.1.36	192.168.1.27	SMB Mailslot	244	Write Mail Slot
192.168.1.36	192.168.1.27	SMB Mailslot	229	Write Mail Slot
192.168.1.36	192.168.1.27	SMB Mailslot	223	Write Mail Slot

Рис. 22: Запись трафика основной машиной

Учитывая, что фильтрация была по протоколу, мы видим, что локальный трафик не выходит в сеть, а все попавшие в сеть пакеты соответствуют своему отправленному из дополнительной машины сообщению.

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 '*'. Запустим на каждой из машин по серверу и по широковещательному клиенту:
Результат широковещательного выполнения программ приведен на рисунках 23 и 24.

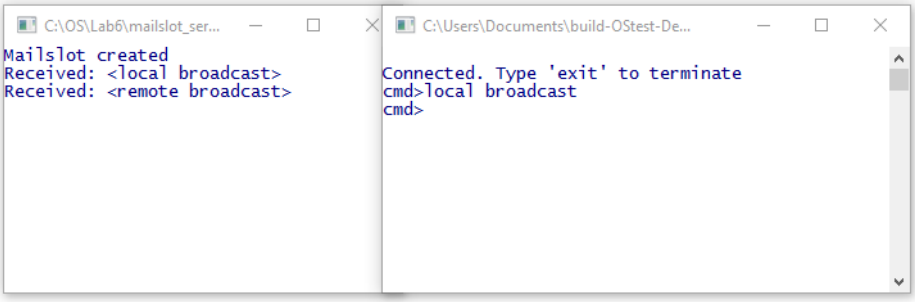

Рис. 23: Широковещательный запуск на локальной машине

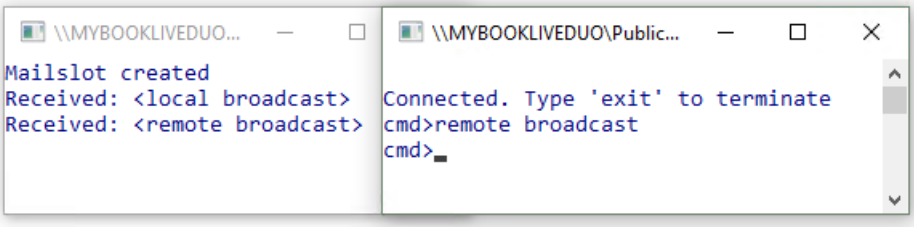

Рис. 24: Широковещательный запуск на дополнительной машине

Для подтверждения работы по сети был записан соответствующий трафик. Он представлен на рисунке 25

Source	Destination	Protocol	Length	Info
192.168.1.27	192.168.1.255	SMB Mailslot	232	Write Mail Slot
192.168.1.36	192.168.1.255	SMB Mailslot	233	Write Mail Slot

Рис. 25: Запись трафика основной машиной

4 Вывод

В ОС Windows реализовано несколько средств межпроцессного взаимодействия. Некоторые из них совпадают с IPC в UNIX: неименованные и именованные каналы, сокеты, разделяемая память. Некоторые уникальны, например, почтовые слоты.

Неименованные каналы Windows обеспечивают однонаправленное (полудуплексное) 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. Поэтому, именованные каналы редко используются для клиент-серверных приложений.

Возможность взаимодействия с другими системами обеспечивается в Windows поддержкой сокетов. Сокет – это оконечная точка соединения, которая идентифицируется 4 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.

MailSlot – механизм синхронизации, иначе называемый «почтовый ящик». Каждый слот реализуется как псевдофайл в оперативной памяти и содержит некоторое количество записей («сообщений»), 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.