

Санкт-Петербургский государственный политехнический университет  
Институт Информационных Технологий и Управления  
Кафедра компьютерных систем и программных технологий

---

Отчёт по расчётной работе № 2  
по предмету «Системное программное обеспечение»

**МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ В ОС WINDOWS**

Работу выполнил студент гр. 53501/3 \_\_\_\_\_ Мартынов С. А.

Работу принял преподаватель \_\_\_\_\_ Душутина Е. В.

Санкт-Петербург  
2014

# Оглавление

Постановка задачи	3
Введение	4
Анонимные каналы	7
Именованные каналы	14
Почтовые ящики	23
Shared memory	31
Сокеты	37
Порты завершения	49
Сигналы	62
Заключение	65
Список литературы	67

# Постановка задачи

В рамках данной работы необходимо ознакомиться с основными механизмами межпроцессное взаимодействие в ОС Windows

1. Анонимные каналы;
2. Именованные каналы  
(локальная/сетевая реализация);
3. Почтовые ящики;
4. Shared memory;
5. Сокеты;
6. Порты завершения;
7. Сигналы.

В процессе изучения предполагается разработать простой (консольный) мгновенный обмен сообщениями.

# Введение

Для тестирования сетевых реализаций используются два виртуальные машины (Win7) под управлением гипервизора VirtualBox. Сетевое подключение осуществляется в режиме bridge. Топология представлена на рисунке 1. Разницы между виртуальной и физической средой быть не должно.

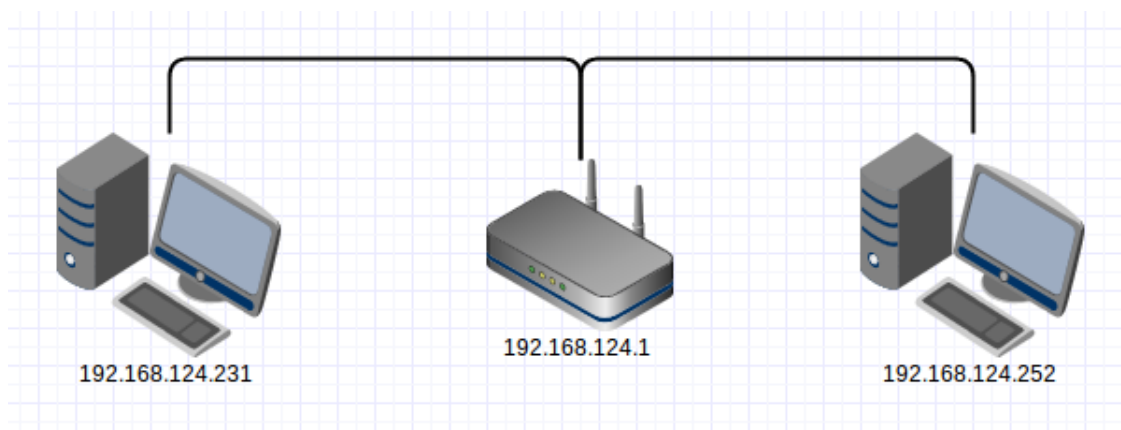


Рис. 1: Топология сети.

Все результаты, представленные в данном отчёте получены с использованием Microsoft Windows 7 Ultimate Service Pack 1 64-bit (build 7601). Для разработки использовалась Microsoft Visual Studio Express 2013 for Windows Desktops (Version 12.0.30723.00 Update 3). В качестве отладчика использовался Microsoft WinDbg (release 6.3.9600.16384).

Для логирования использовался код, приведённый в листинге 1. Этот файл подключается в первых строках каждого проекта. Он создаёт файл, с именем идентичным исполняемой программе, и записывает туда все события.

Листинг 1: Логер, используемый во всех проектах

(src/InterProcessCommunication/AnonymousPipeServer/logger.h)

```
1 #pragma once
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <tchar.h>
6 #include <stdarg.h>
7 #include <time.h>
8
9 // log
10 FILE* logfile;
11
12 void initlog(const _TCHAR* prog);
13 void closelog();
14 void writelog(_TCHAR* format, ...);
15
16 void initlog(const _TCHAR* prog) {
17     _TCHAR logname[255];
18     wcscpy_s(logname, prog);
19     // replace extension
20     _TCHAR* extension;
21     extension = wcsstr(logname, _T(".exe"));
22     wcsncpy_s(extension, 5, _T(".log"), 4);
23     // Try to open log file for append
24     if (_wfopen_s(&logfile, logname, _T("a+"))) {
25         _wprintf(_T("The following error occurred"));
26         _tprintf(_T("Can't open log file %s\n"), logname);
27         exit(-1);
28     }
29     writelog(_T("%s is starting."), prog);
30 }
31 void closelog() {
32     writelog(_T("Shutting down.\n"));
33     fclose(logfile);
34 }
35 void writelog(_TCHAR* format, ...) {
36     _TCHAR buf[255];
37     va_list ap;
38     struct tm newtime;
39     __time64_t long_time;
40     // Get time as 64-bit integer.
41     _time64(&long_time);
42     // Convert to local time.
```

```

43 _localtime64_s(&newtime, &long_time);
44 // Convert to normal representation.
45 swprintf_s(buf, _T("[%d/%d/%d %d:%d:%d] "), newtime.tm_mday,
46     newtime.tm_mon + 1, newtime.tm_year + 1900, newtime.tm_hour,
47     newtime.tm_min, newtime.tm_sec);
48 // Write date and time
49 fwprintf(logfile, _T("%s"), buf);
50 // Write all params
51 va_start(ap, format);
52 _vsnwprintf_s(buf, sizeof(buf) - 1, format, ap);
53 fwprintf(logfile, _T("%s"), buf);
54 va_end(ap);
55 // New sting
56 fwprintf(logfile, _T("\n"));
57 }

```

В данном файле содержатся следующие методы:

- `initlog` – принимает в качестве параметра имя приложения, заменяет `exe` на `log` и создаёт лог-файл;
- `closelog` – обеспечивает корректное завершение работы, освобождая дескриптор файла;
- `writelog` – функция получает форматированную строку и набор параметров, из которых формируется строка лога по аналогии с библиотечной функцией `printf`. Интересным моментом является тот факт, что строка дописывает текущее время, что упрощает анализ параллельных программ.

Часть кода и лог-файлов приведена в листингах по ходу работы, однако более полная версия находится в папках `src/InterProcessCommunication` и `logs/InterProcessCommunication`.

# Анонимные каналы

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). Функция, с помощью которой создаются анонимные каналы, имеет следующий прототип [1, 2]:

```
BOOL CreatePipe(PHANDLE phRead, PHANDLE phWrite,  
                LPSECURITY_ATTRIBUTES lpsa, DWORD cbPipe)
```

Дескрипторы каналов часто бывают наследуемыми; причины этого станут понятными из приведенного ниже примера. Значение параметра cbPipe, указывающее размер канала в байтах, носит рекомендательный характер, причем значению 0 соответствует размер канала по умолчанию.

Чтобы канал можно было использовать для IPC, должен существовать еще один процесс, и для этого процесса требуется один из дескрипторов канала. Предположим, например, что родительскому процессу, вызвавшему функцию CreatePipe, необходимо вывести данные, которые нужны дочернему процессу. Тогда возникает вопрос о том, как передать дочернему процессу дескриптор чтения (phRead). Родительский процесс осуществляет это, устанавливая дескриптор стандартного ввода в структуре STARTUPINFO для дочерней процедуры равным \*phRead.

Чтение с использованием дескриптора чтения канала блокируется, если канал пуст. В противном случае в процессе чтения будет воспринято столько байтов, сколько имеется в канале, вплоть до количества, указанного при вызове функции ReadFile. Операция записи в заполненный канал, которая выполняется с использованием буфера в памяти, также будет блокирована.

Наконец, анонимные каналы обеспечивают только однонаправленное взаимодействие. Для двустороннего взаимодействия необходимы два канала.

В листинге 2 и 3 демонстрируются серверный и клиентский модуль программы, в которой используется передача дескрипторов через наследование. Анонимный канал является по-

лудуплексным, поэтому для организации эхо-сервера было необходимо создавать 2 канала (для передачи от клиента к серверу и в обратном направлении). При этом ненужные дескрипторы каналов закрываются только на стороне сервера (т.к. клиент наследует 4 дескриптора, а явно передаются только 2). Дескрипторы каналов связываются со стандартным вводом и выводом клиентского процесса. Для вывода информации клиенту остаётся только поток ошибок[1].

## Листинг 2: Сервер анонимных каналов

(src/InterProcessCommunication/AnonymousPipeServer/main.cpp)

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <tchar.h>
4 #include "logger.h"
5
6 //размер буфера для сообщений
7 #define BUF_SIZE 100
8
9 int _tmain(int argc, _TCHAR* argv[]) {
10     //Init log
11     initlog(argv[0]);
12
13     //буфер приема/передачи
14     _TCHAR buf[BUF_SIZE];
15     //число прочитанных/переданных байт
16     DWORD readbytes, writebytes;
17
18     //дескрипторы канала для передачи от сервера клиенту
19     HANDLE hReadPipeFromServToClient, hWritePipeFromServToClient;
20     //дескрипторы канала для передачи от клиента серверу
21     HANDLE hReadPipeFromClientToServ, hWritePipeFromClientToServ;
22
23     //чтобы сделать дескрипторы наследуемыми
24     SECURITY_ATTRIBUTES PipeSA = { sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
25     //создаем канал для передачи от сервера клиенту, сразу делаем дескрипторы
        наследуемыми
26     if (CreatePipe(&hReadPipeFromServToClient, &hWritePipeFromServToClient, &
        PipeSA, 0) == 0) {
27         double errorcode = GetLastError();
28         writelog(_T("Can't create anonymous pipe from server to client, GLE=%d."
        ), errorcode);
29         _tprintf(_T("Can't create anonymous pipe from server to client, GLE=%d."
        ), errorcode);
30         closelog();
31         exit(1);
    }
```



```

32 }
33 writelog(_T("Anonymous pipe from server to client created"));
34 _tprintf(_T("Anonymous pipe from server to client created\n"));
35
36 //создаем канал для передачи от клиента серверу, сразу делаем дескрипторы
наследуемыми
37 if (CreatePipe(&hReadPipeFromClientToServ, &hWritePipeFromClientToServ, &
    PipeSA, 0) == 0) {
38     double errorcode = GetLastError();
39     writelog(_T("Can't create anonymous pipe from client to server, GLE=%d."
        ), errorcode);
40     _tprintf(_T("Can't create anonymous pipe from client to server, GLE=%d."
        ), errorcode);
41     closelog();
42     exit(2);
43 }
44 writelog(_T("Anonymous pipe from client to server created"));
45 _tprintf(_T("Anonymous pipe from client to server created\n"));
46
47 PROCESS_INFORMATION processInfo_Client; // информация о процессе-клиенте
48 //структура, которая описывает внешний вид основного окна и содержит
49 // дескрипторы стандартных устройств нового процесса
50 STARTUPINFO startupInfo_Client;
51
52 //процесс-клиент будет иметь те же параметры запуска, что и сервер,
53 // за исключением дескрипторов ввода, вывода и ошибок
54 GetStartupInfo(&startupInfo_Client);
55 //устанавливаем поток ввода
56 startupInfo_Client.hStdInput = hReadPipeFromServToClient;
57 //установим поток вывода
58 startupInfo_Client.hStdOutput = hWritePipeFromClientToServ;
59 //установим поток ошибок
60 startupInfo_Client.hStdError = GetStdHandle(STD_ERROR_HANDLE);
61 //устанавливаем наследование
62 startupInfo_Client.dwFlags = STARTF_USESTDHANDLES;
63 //создаем процесс клиента
64 if (!CreateProcess(_T("AnonymousPipeClient.exe"), // имя исполняемого моду
    ля
65     NULL, //командная строка
66     NULL, //атрибуты безопасности процесса
67     NULL, //атрибуты безопасности потока
68     TRUE, //флаг наследования описателя
69     CREATE_NEW_CONSOLE, //флаги создания
70     NULL, //новый блок окружения
71     NULL, //имя текущей директории

```

```

72     &startupInfo_Client, // STARTUPINFO
73     &processInfo_Client)) { //PROCESS_INFORMATION
74         writelog(_T("Can't create process, GLE=%d."), GetLastError());
75         _wprintf(_T("Create process"));
76         closelog();
77         exit(3);
78     }
79     writelog(_T("New process created"));
80     _tprintf(_T("New process created\n"));
81
82     //закрываем дескрипторы созданного процесса и его потока
83     CloseHandle(processInfo_Client.hThread);
84     CloseHandle(processInfo_Client.hProcess);
85     //закрываем ненужные дескрипторы каналов, которые не использует сервер
86     CloseHandle(hReadPipeFromServToClient);
87     CloseHandle(hWritePipeFromClientToServ);
88
89     // Начинаем взаимодействие с клиентом через анонимный канал
90     for (int i = 0; i < 10; i++) {
91         //читаем данные из канала от клиента
92         if (!ReadFile(hReadPipeFromClientToServ, buf, sizeof(buf), &readbytes,
93             NULL)) {
94             double errorcode = GetLastError();
95             writelog(_T("Impossible to use readfile, GLE=%d."), errorcode);
96             _tprintf(_T("Impossible to use readfile, GLE=%d."), errorcode);
97             closelog();
98             exit(4);
99         }
100         _tprintf(_T("Get from client: \"%s\"\n"), buf);
101         writelog(_T("Get from client: \"%s\""), buf);
102         //пишем данные в канал клиенту
103         if (!WriteFile(hWritePipeFromServToClient, buf, readbytes, &writebytes,
104             NULL)) {
105             double errorcode = GetLastError();
106             writelog(_T("Impossible to use writefile, GLE=%d."), errorcode);
107             _tprintf(_T("Impossible to use writefile, GLE=%d."), errorcode);
108             closelog();
109             exit(5);
110         }
111     }
112     //закрываем HANDLE каналов
113     CloseHandle(hReadPipeFromClientToServ);
114     CloseHandle(hWritePipeFromServToClient);
115     closelog();

```

```
115     exit(0);
116 }
```

Клиент передаёт сообщение, например, вида: «message num 1», ждёт 1 секунду, и пытается прочесть его от сервера. Сервер, получив сообщение от клиента, передаёт его обратно. Процессы завершаются после передачи 10 сообщений.

Листинг 3: Клиент анонимных каналов

(src/InterProcessCommunication/AnonymousPipeClient/main.cpp)

```
1 #include <stdio.h>
2 #include <Windows.h>
3 #include "logger.h"
4
5 int _tmain(int argc, _TCHAR* argv[]) {
6     //Init log
7     initlog(argv[0]);
8
9     _TCHAR strtosend[100]; //строка для передачи
10    _TCHAR getbuf[100]; //буфер приема
11    //число переданных и принятых байт
12    DWORD bytessended, bytesreaded;
13
14    // Начинаем взаимодействие с сервером через анонимный канал
15    for (int i = 0; i < 10; i++) {
16        //формирование строки для передачи
17        bytessended = swprintf_s(strtosend, _T("Message num %d"), i + 1);
18        strtosend[bytessended++] = _T('\0');
19
20        writelog(_T("Client sended: \"%s\""), strtosend);
21        fwprintf(stderr, _T("Client sended: \"%s\"\n"), strtosend);
22
23        //передача данных
24        if (!WriteFile(GetStdHandle(STD_OUTPUT_HANDLE), strtosend, bytessended *
25            sizeof(wchar_t), &bytesreaded, NULL)) {
26            double errorcode = GetLastError();
27            writelog(_T("Error with writeFile, GLE=%d."), errorcode);
28            fwprintf(stderr, _T("Error with writeFile, GLE=%d."), errorcode);
29            closelog();
30            exit(1);
31        }
32        Sleep(1000);
33        //примем ответа от сервера
34        if (!ReadFile(GetStdHandle(STD_INPUT_HANDLE), getbuf, 100, &bytesreaded,
35            NULL)) {
```

```

34     double errorcode = GetLastError();
35     writelog(_T("Error with readfile, GLE=%d."), errorcode);
36     fwprintf(stderr, _T("Error with readfile, GLE=%d."), errorcode);
37     closelog();
38     exit(2);
39 }
40 writelog(_T("Get msg from server: \"%s\""), getbuf);
41 fwprintf(stderr, _T("Get msg from server: \"%s\\n\""), getbuf);
42 }
43 Sleep(10000);
44 closelog();
45 return 0;
46 }

```

The image displays two overlapping Windows command prompt windows. The top window, titled 'C:\Windows\system32\cmd.exe', shows the execution of a server process. It displays the following output:

```

bug>
C:\Users\win7\Documents\Visual Studio 2013\Projects\InterProcessCommunication\De
bug>
C:\Users\win7\Documents\Visual Studio 2013\Projects\InterProcessCommunication\De
bug>
C:\Users\win7\Documents\Visual Studio 2013\Projects\InterProcessCommunication\De
bug>
C:\Users\win7\Documents\Visual Studio 2013\Projects\InterProcessCommunication\De
bug>AnonymousPipeServer.exe
Anonymous pipe from server to client created
Anonymous pipe from client to server created
New process created
Get from client: "Message num 1"
Get from client: "Message num 2"
Get from client: "Message num 3"
Get from client: "Message num 4"
Get from client: "Message num 5"
Get from client: "Message num 6"
Get from client: "Message num 7"
Get from client: "Message num 8"
Get from client: "Message num 9"
Get from client: "Message num 10"
C:\Users\win7\Documents\Visual Studio 2013\Projects\InterProcessCommunication\De
bug>

```

The bottom window, titled 'AnonymousPipeServer.exe', shows the execution of a client process. It displays the following output:

```

Client sended: "Message num 3"
Get msg from server: "Message num 3"
Client sended: "Message num 4"
Get msg from server: "Message num 4"
Client sended: "Message num 5"
Get msg from server: "Message num 5"
Client sended: "Message num 6"
Get msg from server: "Message num 6"
Client sended: "Message num 7"
Get msg from server: "Message num 7"
Client sended: "Message num 8"
Get msg from server: "Message num 8"
Client sended: "Message num 9"
Get msg from server: "Message num 9"
Client sended: "Message num 10"
Get msg from server: "Message num 10"
-

```

Рис. 2: Работа с анонимными каналами.

Результаты работы показаны на рисунке 2. Листинг 4 и листинг 5 содержат протоколы работы сервера и клиента.

Листинг 4: Протокол работы серверного модуля программы работы с анонимными каналами

```
1 [14/2/2015 3:46:36] AnonymousPipeServer.exe is starting.
2 [14/2/2015 3:46:36] Anonymous pipe from server to client created
3 [14/2/2015 3:46:36] Anonymous pipe from client to server created
4 [14/2/2015 3:46:36] New process created
5 [14/2/2015 3:46:36] Get from client: "Message num 1"
6 [14/2/2015 3:46:37] Get from client: "Message num 2"
7 [14/2/2015 3:46:38] Get from client: "Message num 3"
8 [14/2/2015 3:46:39] Get from client: "Message num 4"
9 [14/2/2015 3:46:40] Get from client: "Message num 5"
10 [14/2/2015 3:46:41] Get from client: "Message num 6"
11 [14/2/2015 3:46:42] Get from client: "Message num 7"
12 [14/2/2015 3:46:43] Get from client: "Message num 8"
13 [14/2/2015 3:46:44] Get from client: "Message num 9"
14 [14/2/2015 3:46:45] Get from client: "Message num 10"
15 [14/2/2015 3:46:45] Shutting down.
```

Листинг 5: Протокол работы клиентского модуля программы работы с анонимными каналами

```
1 [14/2/2015 3:46:36] AnonymousPipeClient.exe is starting.
2 [14/2/2015 3:46:36] Client sended: "Message num 1"
3 [14/2/2015 3:46:37] Get msg from server: "Message num 1"
4 [14/2/2015 3:46:37] Client sended: "Message num 2"
5 [14/2/2015 3:46:38] Get msg from server: "Message num 2"
6 [14/2/2015 3:46:38] Client sended: "Message num 3"
7 [14/2/2015 3:46:39] Get msg from server: "Message num 3"
8 [14/2/2015 3:46:39] Client sended: "Message num 4"
9 [14/2/2015 3:46:40] Get msg from server: "Message num 4"
10 [14/2/2015 3:46:40] Client sended: "Message num 5"
11 [14/2/2015 3:46:41] Get msg from server: "Message num 5"
12 [14/2/2015 3:46:41] Client sended: "Message num 6"
13 [14/2/2015 3:46:42] Get msg from server: "Message num 6"
14 [14/2/2015 3:46:42] Client sended: "Message num 7"
15 [14/2/2015 3:46:43] Get msg from server: "Message num 7"
16 [14/2/2015 3:46:43] Client sended: "Message num 8"
17 [14/2/2015 3:46:44] Get msg from server: "Message num 8"
18 [14/2/2015 3:46:44] Client sended: "Message num 9"
19 [14/2/2015 3:46:45] Get msg from server: "Message num 9"
20 [14/2/2015 3:46:45] Client sended: "Message num 10"
21 [14/2/2015 3:46:46] Get msg from server: "Message num 10"
22 [14/2/2015 3:46:56] Shutting down.
```

# Именованные каналы

Именованные каналы (named pipes) предлагают ряд возможностей, которые делают их полезными в качестве универсального механизма реализации приложений на основе ИРС, включая приложения, требующие сетевого доступа к файлам, и клиент-серверные системы, хотя для реализации простых вариантов ИРС, ориентированных на байтовые потоки, как в предыдущем примере, в котором взаимодействие процессов ограничивается рамками одной системы, анонимных каналов вам будет вполне достаточно. К числу упомянутых возможностей (часть которых обеспечивается дополнительно) относятся следующие[2]:

- Именованные каналы ориентированы на обмен сообщениями, поэтому процесс, выполняющий чтение, может считывать сообщения переменной длины именно в том виде, в каком они были посланы процессом, выполняющим запись.
- Именованные каналы являются двунаправленными, что позволяет осуществлять обмен сообщениями между двумя процессами посредством единственного канала.
- Допускается существование нескольких независимых экземпляров канала, имеющих одинаковые имена. Например, с единственной серверной системой могут связываться одновременно несколько клиентов, использующих каналы с одним и тем же именем. Каждый клиент может иметь собственный экземпляр именованного канала, и сервер может использовать этот же канал для отправки ответа клиенту.
- Каждая из систем, подключенных к сети, может обратиться к каналу, используя его имя. Взаимодействие посредством именованного канала осуществляется одинаковым образом для процессов, выполняющихся как на одной и той же, так и на разных машинах.
- Имеется несколько вспомогательных и связанных функций, упрощающих обслуживание взаимодействия "запрос/ответ" и клиент-серверных соединений.

Как правило, именованные каналы являются более предпочтительными по сравнению с анонимными, хотя существуют ситуации, когда анонимные каналы оказываются исключительно полезными. Во всех случаях, когда требуется, чтобы канал связи был двунаправлен-

ным, ориентированным на обмен сообщениями или доступным для нескольких клиентских процессов, следует применять именованные каналы.

Реализация работы с именованными каналами представлена в листинге 8 (серверный модуль) и 9 (клиентский модуль). 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.

Листинг 6: Сервер именованного каналов

(src/InterProcessCommunication/NamedPipeServer/main.cpp)

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <conio.h>
4 #include <tchar.h>
5 #include <strsafe.h>
6 #include "logger.h"
7
8 #define BUFSIZE 512
9
10 DWORD WINAPI InstanceThread(LPVOID);
11
12 int _tmain(int argc, _TCHAR* argv[]) {
13     //Init log
14     initlog(argv[0]);
15     _tprintf(_T("Server is started.\n\n"));
16
17     BOOL fConnected = FALSE; // Флаг наличия подключенных клиентов
18     DWORD dwThreadId = 0; // Номер обслуживающего потока
19     HANDLE hPipe = INVALID_HANDLE_VALUE; // Идентификатор канала
20     HANDLE hThread = NULL; // Идентификатор обслуживающего потока
21     LPTSTR lpszPipename = _T("\\\\.\\pipe\\$$$MyPipe$$"); // Имя создаваемого к
        анала
22
23     // Цикл ожидает клиентов и создаёт для них потоки обработки
24     for (;;) {
25         writelog(_T("Try to create named pipe on %s"), lpszPipename);
26         _tprintf(_T("Try to create named pipe on %s\n"), lpszPipename);
27
28         // Создаем канал:
29         if ((hPipe = CreateNamedPipe(
30             lpszPipename, // имя канала,
31             PIPE_ACCESS_DUPLEX, // режим открытия канала - двунаправленный,
```

```

32     PIPE_TYPE_MESSAGE | // данные записываются в канал в виде потока сообщ
        ений,
33     PIPE_READMODE_MESSAGE | // данные считываются в виде потока сообщений,
34     PIPE_WAIT, // функции передачи и приема блокируются до их окончания,
35     PIPE_UNLIMITED_INSTANCES, // максимальное число экземпляров каналов не
        ограничено,
36     BUFSIZE, //размеры выходного и входного буферов канала,
37     BUFSIZE,
38     5000, // 5 секунд - длительность для функции WaitNamedPipe,
39     NULL))// дескриптор безопасности по умолчанию.
40     == INVALID_HANDLE_VALUE) {
41     double errorcode = GetLastError();
42     writelog(_T("CreateNamedPipe failed, GLE=%d."), errorcode);
43     _tprintf(_T("CreateNamedPipe failed, GLE=%d.\n"), errorcode);
44     closelog();
45     exit(1);
46 }
47 writelog(_T("Named pipe created successfully!"));
48 _tprintf(_T("Named pipe created successfully!\n"));
49
50 // Ожидаем соединения со стороны клиента
51 writelog(_T("Waiting for connect..."));
52 _tprintf(_T("Waiting for connect...\n"));
53 fConnected = ConnectNamedPipe(hPipe, NULL) ?
54 TRUE :
55     (GetLastError() == ERROR_PIPE_CONNECTED);
56
57 // Если произошло соединение
58 if (fConnected) {
59     writelog(_T("Client connected!"));
60     writelog(_T("Creating a processing thread..."));
61     _tprintf(_T("Client connected!\nCreating a processing thread...\n"));
62
63     // Создаём поток для обслуживания клиента
64     hThread = CreateThread(
65         NULL, // дескриптор защиты
66         0, // начальный размер стека
67         InstanceThread, // функция потока
68         (LPVOID)hPipe, // параметр потока
69         0, // опции создания
70         &dwThreadId); // номер потока
71
72     // Если поток создать не удалось - сообщаем об ошибке
73     if (hThread == NULL) {
74         double errorcode = GetLastError();

```



```

75     writelog(_T("CreateThread failed, GLE=%d."), errorcode);
76     _tprintf(_T("CreateThread failed, GLE=%d.\n"), errorcode);
77     closelog();
78     exit(1);
79 }
80 else CloseHandle(hThread);
81 }
82 else {
83     // Если клиенту не удалось подключиться, закрываем канал
84     CloseHandle(hPipe);
85     writelog(_T("There are not connection requests."));
86     _tprintf(_T("There are not connection requests.\n"));
87 }
88 }
89
90 closelog();
91 exit(0);
92 }
93
94 DWORD WINAPI InstanceThread(LPVOID lpvParam) {
95     writelog(_T("Thread %d started!"), GetCurrentThreadId());
96     _tprintf(_T("Thread %d started!\n"), GetCurrentThreadId());
97     HANDLE hPipe = (HANDLE)lpvParam; // Идентификатор канала
98     // Буфер для хранения полученного и передаваемого сообщения
99     _TCHAR* chBuf = (_TCHAR*)HeapAlloc(GetProcessHeap(), 0, BUFSIZE * sizeof(
        _TCHAR));
100     DWORD readbytes, writebytes; // Число байт прочитанных и переданных
101
102     while (1) {
103         // Получаем очередную команду через канал Pipe
104         if (ReadFile(hPipe, chBuf, BUFSIZE*sizeof(_TCHAR), &readbytes, NULL)) {
105             // Посылаем эту команду обратно клиентскому приложению
106             if (!WriteFile(hPipe, chBuf, (lstrlen(chBuf) + 1)*sizeof(_TCHAR), &
                writebytes, NULL))
107                 break;
108             // Выводим принятую команду на консоль
109             writelog(_T("Thread %d: Get client msg: %s"), GetCurrentThreadId(),
                chBuf);
110             _tprintf(TEXT("Get client msg: %s\n"), chBuf);
111             // Если пришла команда "exit", завершаем работу приложения
112             if (!_tcsncmp(chBuf, L"exit", 4))
113                 break;
114         } else {
115             double errorcode = GetLastError();

```

```

116     writelog(_T("Thread %d: GReadFile: Error %ld"), GetCurrentThreadId(),
        errorcode);
117     _tprintf(TEXT("ReadFile: Error %ld\n"), errorcode);
118     _getch();
119     break;
120 }
121 }
122
123 // Освобождение ресурсов
124 FlushFileBuffers(hPipe);
125 DisconnectNamedPipe(hPipe);
126 CloseHandle(hPipe);
127 HeapFree(GetProcessHeap(), 0, chBuf);
128
129 writelog(_T("Thread %d: InstanceThread exiting."), GetCurrentThreadId());
130 _tprintf(TEXT("InstanceThread exiting.\n"));
131 return 0;
132 }

```

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.

Для данной программы были внесены изменения в логер. Теперь каждый клиентский процесс создаёт свой отдельный файл с протоколом.

Листинг 7: Клиент именованного каналов

(src/InterProcessCommunication/NamedPipeClient/main.cpp)

```

1 #include <windows.h>
2 #include <stdio.h>
3 #include <conio.h>
4 #include <tchar.h>
5 #include <strsafe.h>
6 #include "logger.h"
7
8 #define BUFSIZE 512
9
10 int _tmain(int argc, _TCHAR* argv[]) {
11     //Init log
12     initlog(argv[0]);
13     _tprintf(_T("Client is started!\n\n"));
14
15     HANDLE hPipe = INVALID_HANDLE_VALUE; // Идентификатор канала

```

```

16 LPTSTR lpszPipename = _T("\\\\.\\pipe\\$$MyPipe$$"); // Имя создаваемого к
    анала Pipe
17 _TCHAR chBuf[BUFSIZE]; // Буфер для передачи данных через канал
18 DWORD readbytes, writebytes; // Число байт прочитанных и переданных
19
20 writelog(_T("Try to use WaitNamedPipe..."));
21 _tprintf(_T("Try to use WaitNamedPipe...\n"));
22 // Пытаемся открыть именованный канал, если надо - ожидаем его освобождени
    я
23 while (1) {
24     // Создаем канал с процессом-сервером:
25     hPipe = CreateFile(
26         lpszPipename, // имя канала,
27         GENERIC_READ // текущий клиент имеет доступ на чтение,
28         | GENERIC_WRITE, // текущий клиент имеет доступ на запись,
29         0, // тип доступа,
30         NULL, // атрибуты защиты,
31         OPEN_EXISTING, // открывается существующий файл,
32         0, // атрибуты и флаги для файла,
33         NULL); // доступа к файлу шаблона.
34
35     // Продолжаем работу, если канал создать удалось
36     if (hPipe != INVALID_HANDLE_VALUE)
37         break;
38
39     // Выход, если ошибка связана не с занятым каналом.
40     double errorcode = GetLastError();
41     if (errorcode != ERROR_PIPE_BUSY) {
42         writelog(_T("Could not open pipe. GLE=%d\n"), errorcode);
43         _tprintf(_T("Could not open pipe. GLE=%d\n"), errorcode);
44         closelog();
45         exit(1);
46     }
47
48     // Если все каналы заняты, ждём 20 секунд
49     if (!WaitNamedPipe(lpszPipename, 20000)) {
50         writelog(_T("Could not open pipe: 20 second wait timed out.));
51         _tprintf(_T("Could not open pipe: 20 second wait timed out.));
52         closelog();
53         exit(2);
54     }
55 }
56
57 // Выводим сообщение о создании канала
58 writelog(_T("Successfully connected!));

```

```

59 _tprintf(_T("Successfully connected!\n\nInput message...\n"));
60 // Цикл обмена данными с серверным процессом
61 while (1) {
62     // Выводим приглашение для ввода команды
63     _tprintf(_T("cmd>"));
64     // Вводим текстовую строку
65     _fgetts(chBuf, BUFSIZE, stdin);
66     // Заносим строку в протокол
67     writelog(_T("Client sended: %s"), chBuf);
68     // Передаем введенную строку серверному процессу в качестве команды
69     if (!WriteFile(hPipe, chBuf, (lstrlen(chBuf) + 1)*sizeof(TCHAR), &
70         writebytes, NULL)) {
71         writelog(_T("connection refused\n"));
72         _tprintf(_T("connection refused\n"));
73         break;
74     }
75     // Получаем эту же команду обратно от сервера
76     if (ReadFile(hPipe, chBuf, BUFSIZE*sizeof(TCHAR), &readbytes, NULL)) {
77         writelog(_T("Received from server: %s"), chBuf);
78         _tprintf(_T("Received from server: %s\n"), chBuf);
79     } else {
80         // Если произошла ошибка, выводим ее код и завершаем работу приложения
81         double errorcode = GetLastError();
82         writelog(_T("ReadFile: Error %ld\n"), errorcode);
83         _tprintf(_T("ReadFile: Error %ld\n"), errorcode);
84         _getch();
85         break;
86     }
87     // В ответ на команду "exit" завершаем цикл обмена данными с серверным п
88     // процессом
89     if (!_tcsncmp(chBuf, L"exit", 4)) {
90         writelog(_T("Processing exit code"));
91         break;
92     }
93 }
94 // Закрываем идентификатор канала
95 CloseHandle(hPipe);
96 closelog();
97 return 0;
98 }

```

Результат работы программы показан на рисунке 3. Запущен один сервер и три клиента, с которыми этот сервер взаимодействует.

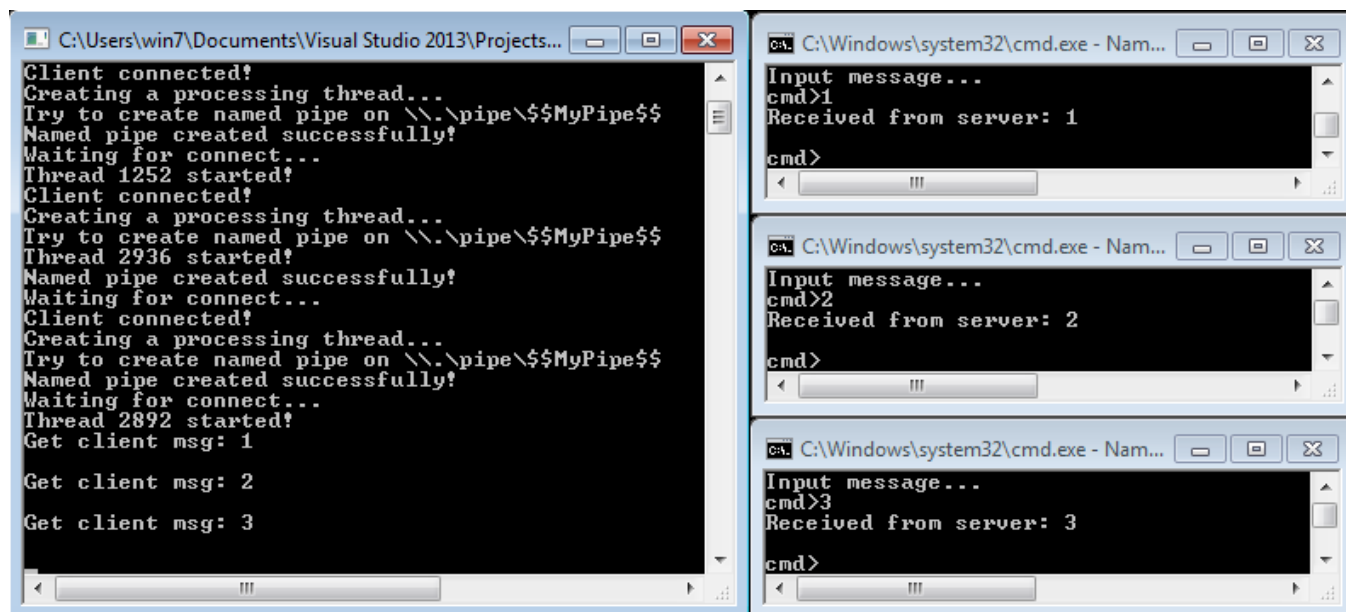


Рис. 3: Работа нескольких клиентов с одним сервером по именованному каналу.

Программа Process Explorer известного разработчика Марка Русиновича позволяет отследить, кто использует именованные каналы. На рисунке 4 видно, что клиентский модуль использует канал MyPipe.

Помимо локального обмена, именованные каналы могут использоваться и для сетевого взаимодействия. Это требует не большой доработки клиента в части указания пути к каналу и изменения настроек безопасности: клиент обращается к именованному каналу указывая имя серверного хоста или его IP адрес[1]. Работа сетевой версии программы показана на рисунке 5.

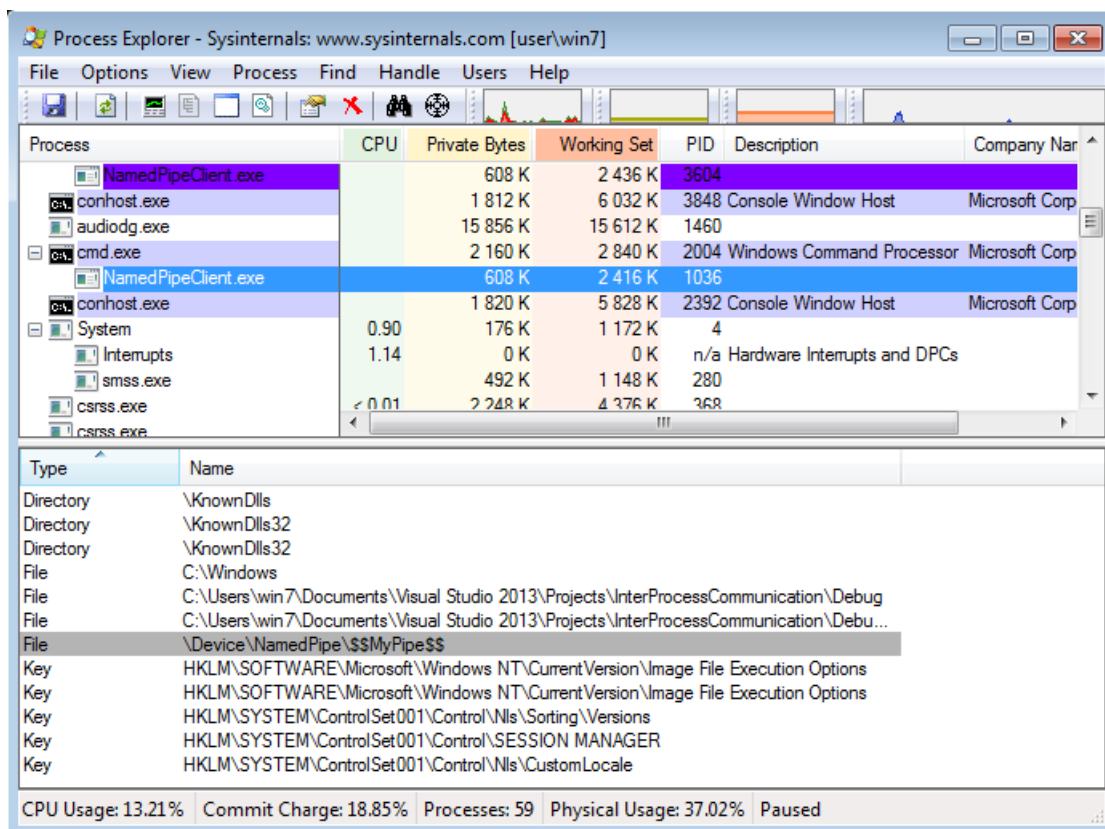


Рис. 4: Отслеживания обращений к локальным (не сетевым) именованным каналам

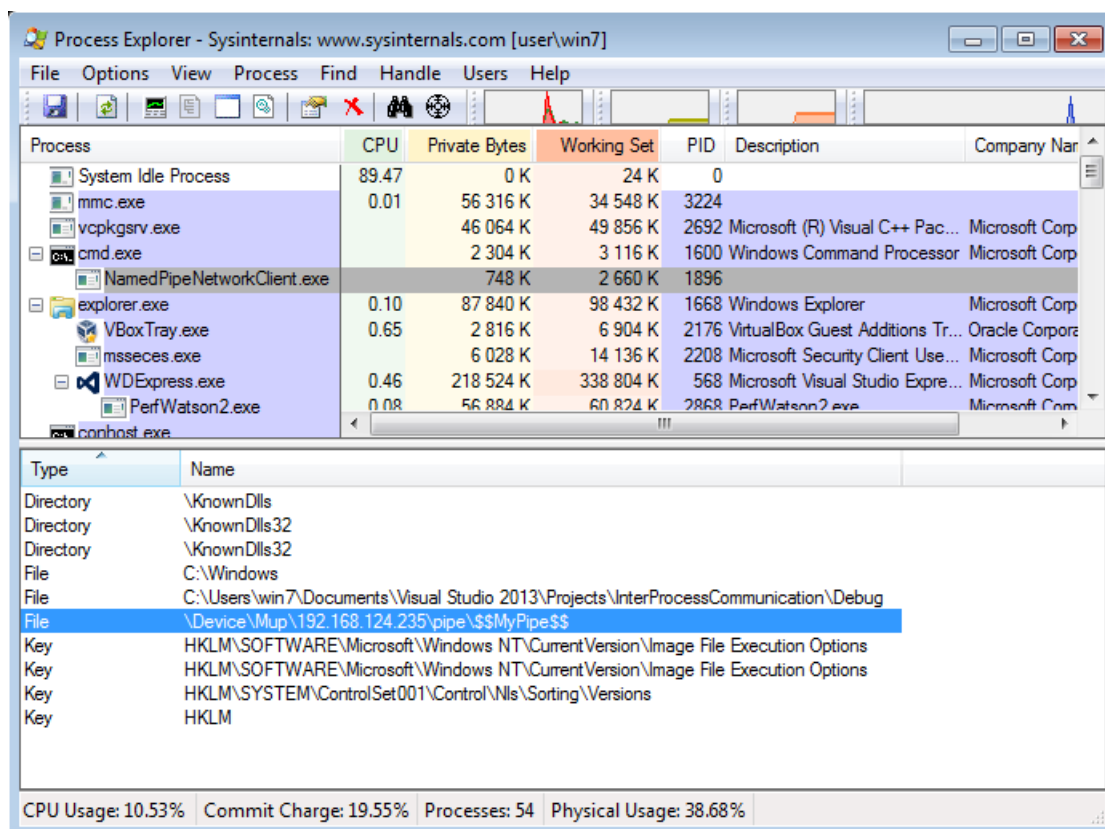


Рис. 5: Отслеживания обращений к сетевым именованным каналам в Process Explorer

# Почтовые ящики

Как и именованные каналы, почтовые ящики (mailslots) Windows снабжаются именами, которые могут быть использованы для обеспечения взаимодействия между независимыми каналами[2]. Почтовые ящики представляют собой широковещательный механизм, и ведут себя иначе по сравнению с именованными каналами, что делает их весьма полезными в ряде ограниченных ситуаций, которые, тем не менее, представляют большой интерес. Из наиболее важных свойств почтовых ящиков можно отметить следующие:

- Почтовые ящики являются однонаправленными.
- С одним почтовым ящиком могут быть связаны несколько записывающих программ (writers) и несколько считывающих программ (readers), но они часто связаны между собой отношениями "один ко многим" в той или иной форме.
- Записывающей программе (клиенту) не известно достоверно, все ли, только некоторые или какая-то одна из программ считывания (сервер) получили сообщение.
- Почтовые ящики могут находиться в любом месте сети.
- Размер сообщений ограничен.

Использование почтовых ящиков требует выполнения следующих операций:

- Каждый сервер создает дескриптор почтового ящика с помощью функции `CreateMailSlot`.
- После этого сервер ожидает получения почтового сообщения, используя функцию `ReadFile`.
- Клиент, обладающий только правами записи, должен открыть почтовый ящик, вызвав функцию `CreateFile`, и записать сообщения, используя функцию `WriteFile`. В случае отсутствия ожидающих программ считывания попытка открытия почтового ящика завершится ошибкой (наподобие "имя не найдено").

Сообщение клиента может быть прочитано всеми серверами; все серверы получают одно и то же сообщение.

Листинг 8 и 9 демонстрируют реализацию приложения, иллюстрирующую обмен информацией почтовыми слотами. В процессе экспериментов было протестировано локальное, сетевое взаимодействие[1]. Для широковещательной передач сообщений, адрес заменялся символом звездочки (\*).

Листинг 8: Реализация серверной части почтового ящика  
(src/InterProcessCommunication/MailslotServer/main.cpp)

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include "logger.h"
4
5 LPTSTR SlotName = TEXT("\\\\.\\mailslot\\sample_mailslot");
6
7 BOOL WriteSlot(HANDLE hSlot, LPTSTR lpszMessage)
8 {
9     BOOL fResult;
10    DWORD cbWritten;
11
12    writelog(_T("Text to send: %s"), lpszMessage);
13    _tprintf(_T("Text to send: %s\n"), lpszMessage);
14
15    fResult = WriteFile(hSlot,
16        lpszMessage,
17        (DWORD)(lstrlen(lpszMessage) + 1)*sizeof(TCHAR),
18        &cbWritten,
19        (LPOVERLAPPED)NULL);
20
21    if (!fResult) {
22        double errorcode = GetLastError();
23        writelog(_T("WriteFile failed, GLE=%d."), errorcode);
24        _tprintf(_T("WriteFile failed, GLE=%d."), errorcode);
25        return FALSE;
26    }
27    writelog(_T("Slot written to successfully"));
28    _tprintf(_T("Slot written to successfully\n"));
29
30    return TRUE;
31 }
32
33 int _tmain(int argc, _TCHAR* argv[]) {
34     //Init log
35     initlog(argv[0]);
36
37     HANDLE hFile;
```



```

38
39 hFile = CreateFile(SlotName,
40     GENERIC_WRITE,
41     FILE_SHARE_READ,
42     (LPSECURITY_ATTRIBUTES)NULL,
43     OPEN_EXISTING,
44     FILE_ATTRIBUTE_NORMAL,
45     (HANDLE)NULL);
46
47 if (hFile == INVALID_HANDLE_VALUE) {
48     double errorcode = GetLastError();
49     writelog(_T("CreateFile failed, GLE=%d."), errorcode);
50     _tprintf(_T("CreateFile failed, GLE=%d."), errorcode);
51     closelog();
52     exit(1);
53 }
54 writelog(_T("Mailslot created"));
55 _tprintf(_T("Mailslot created"));
56
57 //for (int i = 0; i != 100; ++i) {
58     WriteSlot(hFile, _T("Message one for mailslot."));
59     WriteSlot(hFile, _T("Message two for mailslot."));
60     Sleep(5000);
61     WriteSlot(hFile, _T("Message three for mailslot."));
62 //}
63 CloseHandle(hFile);
64
65 closelog();
66 exit(0);
67 }

```

Листинг 9: Реализация клиентской части почтового ящика  
(src/InterProcessCommunication/MailslotClient/main.cpp)

```

1 #include <windows.h>
2 #include <tchar.h>
3 #include <stdio.h>
4 #include <strsafe.h>
5 #include <conio.h>
6 #include "logger.h"
7
8 HANDLE hSlot;
9 LPTSTR SlotName = _T("\\\\.\\mailslot\\sample_mailslot");

```

```

10
11 BOOL ReadSlot()
12 {
13     DWORD cbMessage, cMessage, cbRead;
14     BOOL fResult;
15     LPTSTR lpszBuffer;
16     TCHAR achID[80];
17     DWORD cAllMessages;
18     HANDLE hEvent;
19     OVERLAPPED ov;
20
21     cbMessage = cMessage = cbRead = 0;
22
23     hEvent = CreateEvent(NULL, FALSE, FALSE, _T("ExampleSlot"));
24     if (NULL == hEvent)
25         return FALSE;
26     ov.Offset = 0;
27     ov.OffsetHigh = 0;
28     ov.hEvent = hEvent;
29
30     fResult = GetMailslotInfo(hSlot, // mailslot handle
31                             (LPDWORD)NULL, // no maximum message size
32                             &cbMessage, // size of next message
33                             &cMessage, // number of messages
34                             (LPDWORD)NULL); // no read time-out
35
36     if (!fResult) {
37         double errorcode = GetLastError();
38         writelog(_T("GetMailslotInfo failed, GLE=%d."), errorcode);
39         _tprintf(_T("GetMailslotInfo failed, GLE=%d."), errorcode);
40         return FALSE;
41     }
42
43     if (cbMessage == MAILSLOT_NO_MESSAGE) {
44         writelog(_T("Waiting for a message..."));
45         _tprintf(_T("Waiting for a message...\n"));
46         return TRUE;
47     }
48
49     cAllMessages = cMessage;
50
51     while (cMessage != 0) { // retrieve all messages
52         // Create a message-number string.
53         StringCchPrintf((LPTSTR)achID,
54             80,

```

```

55     _T("\nMessage # %d of %d\n"),
56     cAllMessages - cMessage + 1,
57     cAllMessages);
58
59     // Allocate memory for the message.
60
61     lpszBuffer = (LPTSTR)GlobalAlloc(GPTR,
62         lstrlen((LPTSTR)achID)*sizeof(TCHAR) + cbMessage);
63     if (NULL == lpszBuffer)
64         return FALSE;
65     lpszBuffer[0] = '\0';
66
67     fResult = ReadFile(hSlot,
68         lpszBuffer,
69         cbMessage,
70         &cbRead,
71         &ov);
72
73     if (!fResult) {
74         double errorcode = GetLastError();
75         writelog(_T("ReadFile failed, GLE=%d."), errorcode);
76         _tprintf(_T("ReadFile failed, GLE=%d./n"), errorcode);
77         GlobalFree((HGLOBAL)lpszBuffer);
78         return FALSE;
79     }
80
81     // Concatenate the message and the message-number string.
82     StringCbCat(lpszBuffer,
83         lstrlen((LPTSTR)achID)*sizeof(TCHAR) + cbMessage,
84         (LPTSTR)achID);
85
86     // Display the message.
87     writelog(_T("Contents of the mailslot: %s\n"), lpszBuffer);
88     _tprintf(_T("Contents of the mailslot: %s\n"), lpszBuffer);
89
90     GlobalFree((HGLOBAL)lpszBuffer);
91
92     fResult = GetMailslotInfo(hSlot, // mailslot handle
93         (LPDWORD)NULL, // no maximum message size
94         &cbMessage, // size of next message
95         &cMessage, // number of messages
96         (LPDWORD)NULL); // no read time-out
97
98     if (!fResult) {
99         double errorcode = GetLastError();

```

```

100     writelog(_T("GetMailslotInfo failed, GLE=%d."), errorcode);
101     _tprintf(_T("GetMailslotInfo failed, GLE=%d./n"), errorcode);
102     return FALSE;
103 }
104 }
105 CloseHandle(hEvent);
106 return TRUE;
107 }
108
109 BOOL WINAPI MakeSlot(LPTSTR lpszSlotName)
110 {
111     hSlot = CreateMailslot(lpszSlotName,
112         0, // no maximum message size
113         MAILSLT_WAIT_FOREVER, // no time-out for operations
114         (LPSECURITY_ATTRIBUTES)NULL); // default security
115
116     if (hSlot == INVALID_HANDLE_VALUE) {
117         double errorcode = GetLastError();
118         writelog(_T("CreateMailslot failed, GLE=%d."), errorcode);
119         _tprintf(_T("CreateMailslot failed, GLE=%d."), errorcode);
120         return FALSE;
121     }
122     writelog(_T("Mailslot created"));
123     _tprintf(_T("Mailslot created"));
124
125     return TRUE;
126 }
127
128 int _tmain(int argc, _TCHAR* argv[]) {
129     //Init log
130     initlog(argv[0]);
131
132     MakeSlot(SlotName);
133
134     while (TRUE) {
135         ReadSlot();
136         Sleep(3000);
137     }
138
139     closelog();
140     exit(0);
141 }

```

Результат работы программы показан на рисунке 6.

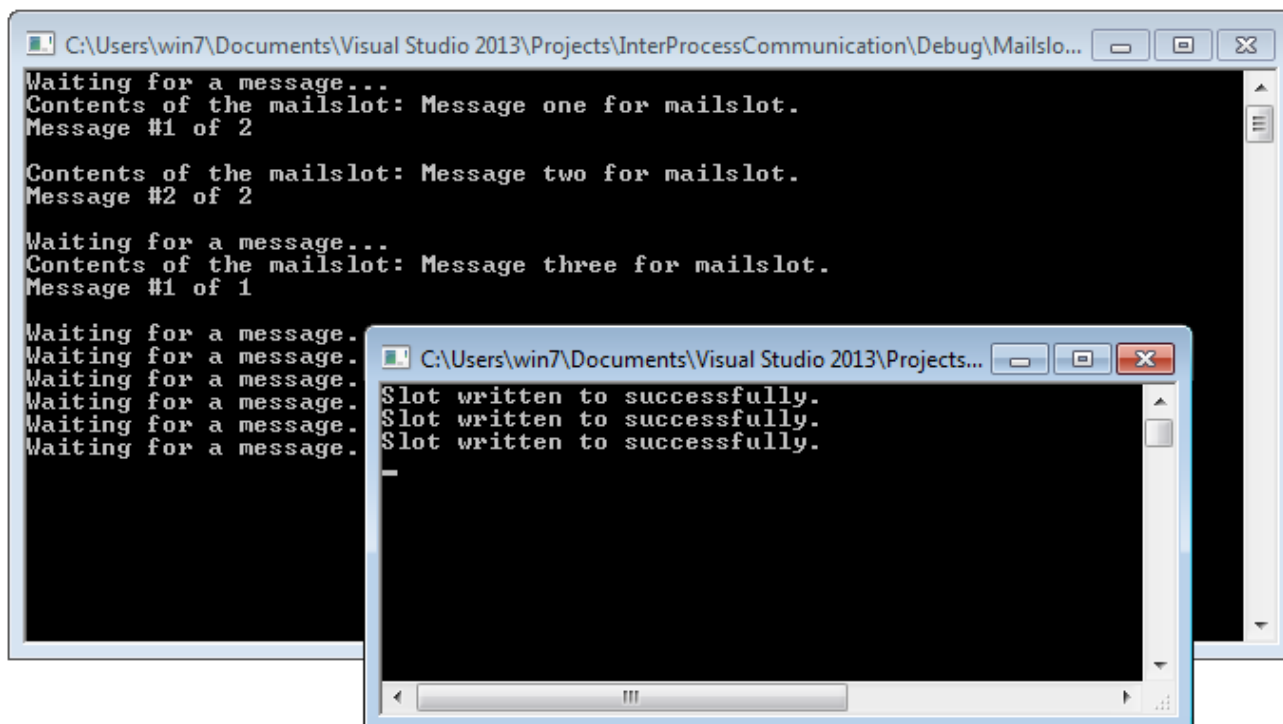


Рис. 6: Работа почтовыми ящиками.

На рисунке 7 показана работа почтового ящика в Process Explorer. Листинг 10 и 11 содержит протокол работы программы.

Листинг 10: Протокол работы серверного модуля программы работы с почтовыми ящиками

```
1 [14/2/2015 7:3:38] C:\Users\win7\Documents\Visual Studio 2013\Projects\
   InterProcessCommunication\Debug\MailslotServer.exe is starting.
2 [14/2/2015 7:3:38] Mailslot created
3 [14/2/2015 7:3:38] Text to send: Message one for mailslot.
4 [14/2/2015 7:3:38] Slot written to successfully
5 [14/2/2015 7:3:38] Text to send: Message two for mailslot.
6 [14/2/2015 7:3:38] Slot written to successfully
7 [14/2/2015 7:3:43] Text to send: Message three for mailslot.
8 [14/2/2015 7:3:43] Slot written to successfully
9 [14/2/2015 7:3:43] Shutting down.
```

Листинг 11: Протокол работы клиентского модуля программы работы с почтовыми ящиками

```
1 [14/2/2015 7:3:38] C:\Users\win7\Documents\Visual Studio 2013\Projects\
   InterProcessCommunication\Debug\MailslotClient.exe is starting.
2 [14/2/2015 7:3:38] Mailslot created
3 [14/2/2015 7:3:38] Waiting for a message...
4 [14/2/2015 7:3:41] Contents of the mailslot: Message one for mailslot.
5 Message #1 of 2
```

```

6 [14/2/2015 7:3:41] Contents of the mailslot: Message two for mailslot.
7 Message #2 of 2
8 [14/2/2015 7:3:44] Contents of the mailslot: Message three for mailslot.
9 Message #1 of 1
10 [14/2/2015 7:3:47] Waiting for a message...

```

Существует еще одна возможность. В вызове функции CreateFile клиент может указать имя почтового ящика в следующем виде:

```
\\*\mailslot\mailslotname
```

При этом символ звездочки (\*) действует в качестве группового символа (wildcard), и клиент может обнаружить любой сервер в пределах имени домена — группы систем, объединенных общим именем, которое назначается администратором сети[1].

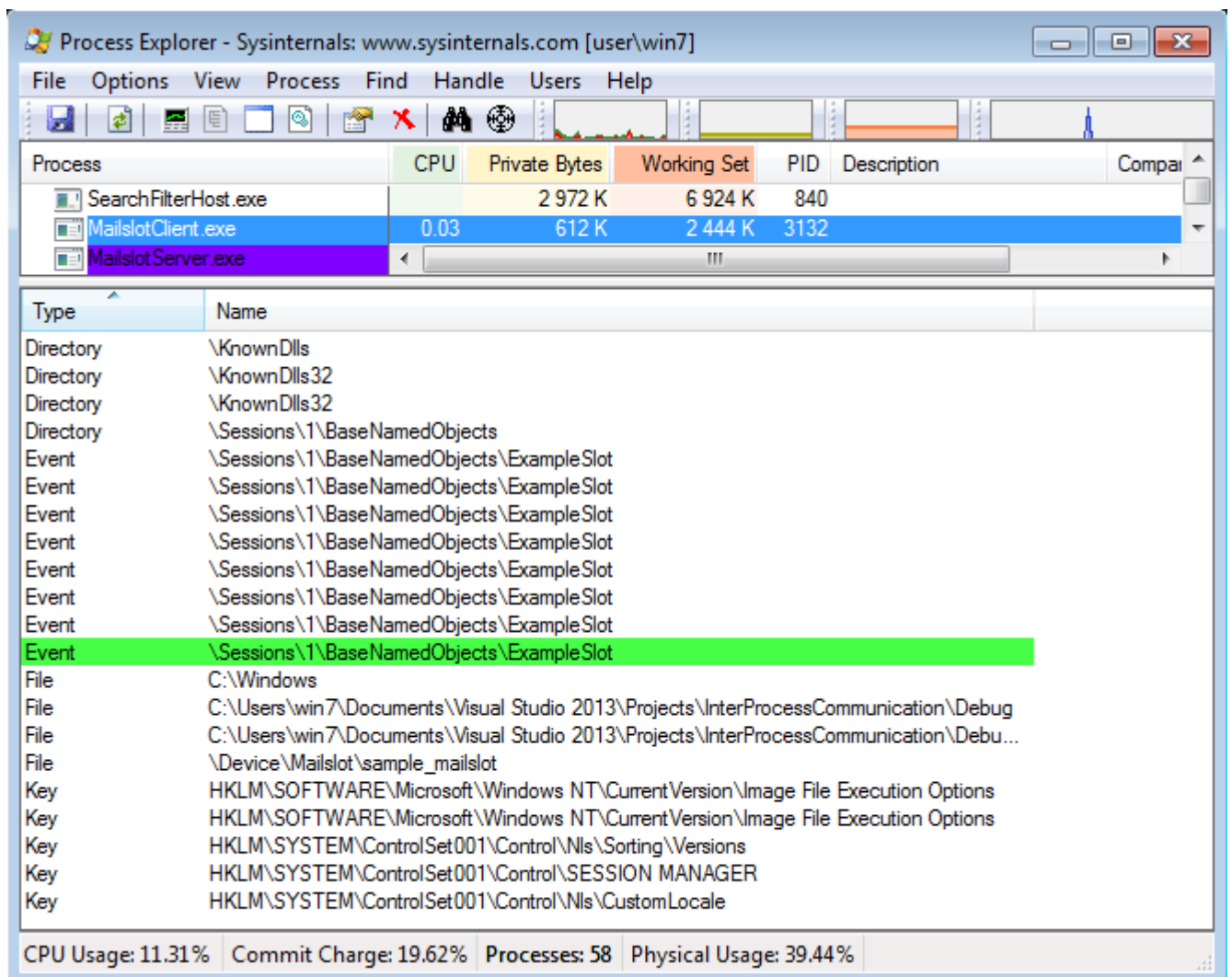


Рис. 7: Работа почтовыми ящиками.

# Shared memory

Этот способ взаимодействия реализуется через технологию File Mapping - отображения файлов на оперативную память. Механизм позволяет осуществлять доступ к файлу таким образом, как будто это обыкновенный массив, хранящийся в памяти (не загружая файл в память явно). Можно создать объект file mapping, но не ассоциировать его с каким-то конкретным файлом[1]. Получаемая область памяти будет общей между процессами. Работая с этой памятью, потоки обязательно должны согласовывать свои действия с помощью объектов синхронизации.

В листинге 12 и 13 представлен код двух программ, одна из которых генерирует случайные числа, а другая их читает и выводит на экран. Взаимодействие осуществляется через разделяемую память, защищённую мьютексом. Рисунок 8 показывает результат такого взаимодействия.

Листинг 12: Программа, генерирующая случайные числа в разделяемую память (src/InterProcessCommunication/SharedMemoryServer/main.cpp)

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <conio.h>
4 #include "logger.h"
5
6 #define BUF_SIZE 256
7 TCHAR szName[] = _T("MyFileMappingObject");
8 TCHAR szMsg[] = _T("Message from first process");
9 HANDLE mutex;
10
11 int _tmain(int argc, _TCHAR* argv[]) {
12     //Init log
13     initlog(argv[0]);
14
15     HANDLE hMapFile;
16     LPCTSTR pBuf;
17     mutex = CreateMutex(NULL, false, TEXT("SyncMutex"));
18     writelog(_T("Mutex created"));
```

```

19 // create a memory, with two process will be working
20 hMapFile = CreateFileMapping(
21     INVALID_HANDLE_VALUE, // использование файла подкачки
22     NULL, // защита по умолчанию
23     PAGE_READWRITE, // доступ к чтению/записи
24     0, // макс. размер объекта
25     BUF_SIZE, // размер буфера
26     szName); // имя отраженного в памяти объекта
27
28 if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE) {
29     double errorcode = GetLastError();
30     writelog(_T("CreateFileMapping failed, GLE=%d"), errorcode);
31     _tprintf(_T("CreateFileMapping failed, GLE=%d"), errorcode);
32     closelog();
33     exit(1);
34 }
35 writelog(_T("FileMappingObject created"));
36
37 pBuf = (LPTSTR)MapViewOfFile(
38     hMapFile, // дескриптор проецируемого в памяти объекта
39     FILE_MAP_ALL_ACCESS, // разрешение чтения/записи (режим доступа)
40     0, // Старшее слово смещения файла, где начинается отображение
41     0, // Младшее слово смещения файла, где начинается отображение
42     BUF_SIZE); // Число отображаемых байтов файла
43
44 if (pBuf == NULL) {
45     double errorcode = GetLastError();
46     writelog(_T("MapViewOfFile failed, GLE=%d"), errorcode);
47     _tprintf(_T("MapViewOfFile failed, GLE=%d"), errorcode);
48     closelog();
49     exit(1);
50 }
51
52 int i = 0;
53 while (true) {
54     i = rand();
55     _itow_s(i, szMsg, sizeof(szMsg), 10);
56     writelog(_T("Wait For Mutex"));
57     WaitForSingleObject(mutex, INFINITE);
58     writelog(_T("Get Mutex"));
59     CopyMemory((PVOID)pBuf, szMsg, sizeof(szMsg));
60     _tprintf(_T("Write message: %s\n"), pBuf);
61     writelog(_T("Write message: %s"), pBuf);
62     Sleep(1000); // необходимо только для отладки - для удобства
63     // представления и анализа результатов

```



```

64     ReleaseMutex(mutex);
65     writelog(_T("Release Mutex"));
66 }
67 // освобождение памяти и закрытие описателя handle
68 UnmapViewOfFile(pBuf);
69 CloseHandle(hMapFile);
70 CloseHandle(mutex);
71
72 closelog();
73 exit(0);
74 }

```

Листинг 13: Программа, читающая случайные числа из разделяемой памяти (src/InterProcessCommunication/SharedMemoryClient/main.cpp)

```

1  #include <windows.h>
2  #include <stdio.h>
3  #include <conio.h>
4  #include "logger.h"
5
6  #define BUF_SIZE 256
7  #define TIME 150
8  // number of reading operation in this process
9  TCHAR szName[] = _T("MyFileMappingObject");
10 HANDLE mutex;
11
12 int _tmain(int argc, _TCHAR* argv[]) {
13     //Init log
14     initlog(argv[0]);
15     Sleep(1000);
16
17     HANDLE hMapFile;
18     LPCTSTR pBuf;
19     mutex = OpenMutex(
20         MUTEX_ALL_ACCESS, // request full access
21         FALSE, // handle not inheritable
22         TEXT("SyncMutex")); // object name
23     if (mutex == NULL) {
24         double errorcode = GetLastError();
25         writelog(_T("OpenMutex error, GLE=%d"), errorcode);
26         _tprintf(_T("OpenMutex error, GLE=%d\n"), errorcode);
27     }
28     writelog(_T("OpenMutex successfully opened the mutex"));

```

```

29  _tprintf(_T("OpenMutex successfully opened the mutex.\n"));
30
31  hMapFile = OpenFileMapping(
32      FILE_MAP_ALL_ACCESS, // доступ к чтению/записи
33      FALSE, // имя не наследуется
34      szName); // имя "проецируемого" объекта
35  if (hMapFile == NULL) {
36      double errorcode = GetLastError();
37      writelog(_T("OpenFileMapping failed, GLE=%d"), errorcode);
38      _tprintf(_T("OpenFileMapping failed, GLE=%d"), errorcode);
39      closelog();
40      exit(1);
41  }
42  pBuf = (LPTSTR)MapViewOfFile(hMapFile,
43      // дескриптор "проецируемого" объекта
44      FILE_MAP_ALL_ACCESS, // разрешение чтения/записи
45      0, 0, BUF_SIZE);
46  if (pBuf == NULL) {
47      double errorcode = GetLastError();
48      writelog(_T("MapViewOfFile failed, GLE=%d"), errorcode);
49      _tprintf(_T("MapViewOfFile failed, GLE=%d"), errorcode);
50      closelog();
51      exit(1);
52  }
53  for (int i = 0; i < TIME; i++) {
54      writelog(_T("Wait For Mutex"));
55      WaitForSingleObject(mutex, INFINITE);
56      writelog(_T("Get Mutex"));
57      _tprintf(_T("Read message: %s\n"), pBuf);
58      writelog(_T("Read message: %s"), pBuf);
59      ReleaseMutex(mutex);
60      writelog(_T("Release Mutex"));
61  }
62  UnmapViewOfFile(pBuf);
63  CloseHandle(hMapFile);
64
65  closelog();
66  exit(0);
67 }

```

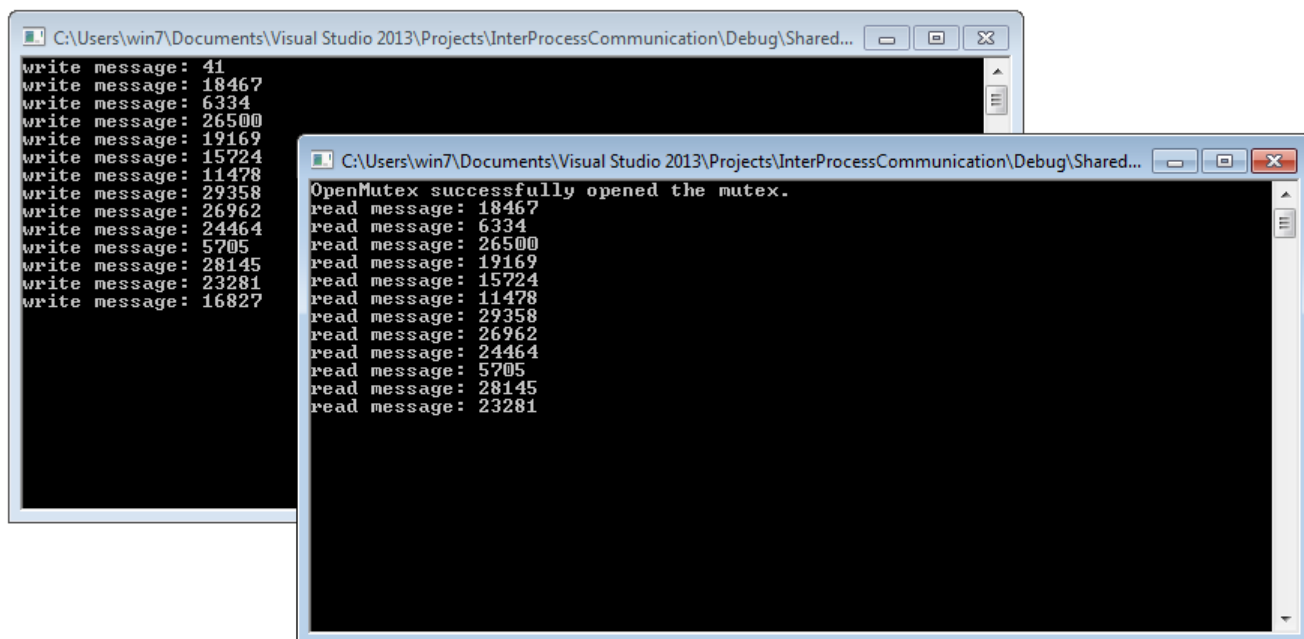


Рис. 8: Работа с разделяемой памятью.

Листинг 14 и 15 содержит протокол (он не много подрезан для краткости) работы программы. Интерес здесь представляет порядок захвата мьютекса.

Листинг 14: Протокол работы серверного модуля программы работы с разделяемой памятью

```

1 [14/2/2015 8:33:21] C:\Users\win7\Documents\Visual Studio 2013\Projects\
   InterProcessCommunication\Debug\SharedMemoryServer.exe is starting.
2 [14/2/2015 8:33:21] Mutex created
3 [14/2/2015 8:33:21] FileMappingObject created
4 [14/2/2015 8:33:21] Wait For Mutex
5 [14/2/2015 8:33:21] Get Mutex
6 [14/2/2015 8:33:21] Write message: 41
7 [14/2/2015 8:33:22] Release Mutex
8 [14/2/2015 8:33:22] Wait For Mutex
9 [14/2/2015 8:33:22] Get Mutex

```

Листинг 15: Протокол работы клиентского модуля программы работы с разделяемой памятью

```

1 [14/2/2015 8:33:21] C:\Users\win7\Documents\Visual Studio 2013\Projects\
   InterProcessCommunication\Debug\SharedMemoryClient.exe is starting.
2 [14/2/2015 8:33:22] OpenMutex successfully opened the mutex
3 [14/2/2015 8:33:22] Wait For Mutex
4 [14/2/2015 8:33:22] Get Mutex
5 [14/2/2015 8:33:22] Read message: 41
6 [14/2/2015 8:33:22] Release Mutex
7 [14/2/2015 8:33:22] Wait For Mutex
8 [14/2/2015 8:33:23] Get Mutex

```

```

9 [14/2/2015 8:33:23] Read message: 18467
10 [14/2/2015 8:33:23] Release Mutex

```

На рисунке 9 видно, как программа клиент работает с ресурсами. Две последние строчки, это мьютекс и общая память.

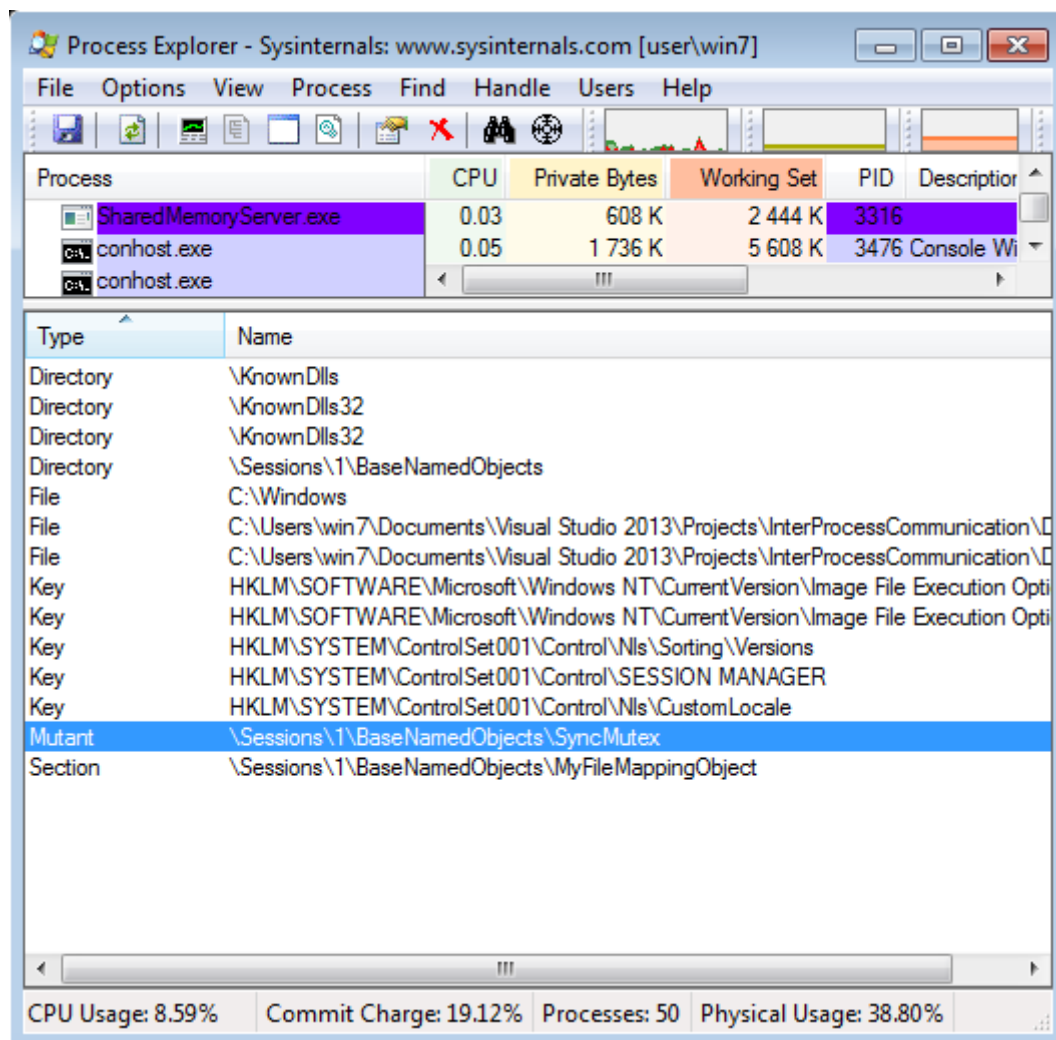


Рис. 9: Мьютекс и общая память

# Сокеты

Winsock API разрабатывался как расширение Berkley Sockets API для среды Windows и поэтому поддерживается всеми системами Windows. К особенностям Winsock можно отнести следующее[2]:

- Перенос уже имеющегося кода, написанного для Berkeley Sockets API, осуществляется непосредственно.
- Системы Windows легко встраиваются в сети, использующие как версию IPv4 протокола TCP/IP, так и постепенно распространяющуюся версию IPv6. Помимо всего остального, версия IPv6 допускает использование более длинных IP-адресов, преодолевая существующий 4-байтовый адресный барьер версии IPv4.
- Сокеты могут использоваться совместно с перекрывающимся вводом/выводом Windows, что, помимо всего прочего, обеспечивает возможность масштабирования серверов при увеличении количества активных клиентов.
- Сокеты можно рассматривать как дескрипторы (типа HANDLE) файлов при использовании функций ReadFile и WriteFile и, с некоторыми ограничениями, при использовании других функций, точно так же, как в качестве дескрипторов файлов сокеты применяются в UNIX. Эта возможность оказывается удобной в тех случаях, когда требуется использование асинхронного ввода/вывода и портов завершения ввода/вывода.
- Существуют также дополнительные, непереносимые расширения.

Работа с сокетами демонстрируется в листинге 16 и 17. Рисунок 6 показывает работу программ, из этих листингов.

Листинг 16: Сервер для работы с Win-сокетами

(src/InterProcessCommunication/WinSocketServer/main.cpp)

```

1 #define _WINSOCK_DEPRECATED_NO_WARNINGS
2 #include <winsock2.h>
3 #include "logger.h"
4
5 #pragma comment(lib, "Ws2_32.lib")
6
7 struct CLIENT_INFO
8 {
9     SOCKET hClientSocket;
10     struct sockaddr_in clientAddr;
11 };
12
13 _TCHAR szServerIPAddr[] = _T("127.0.0.1"); // server IP
14 int nServerPort = 5050; // server port
15 // clients to talk with the server
16
17 bool InitWinSock2_0();
18 BOOL WINAPI ClientThread(LPVOID lpData);
19
20 int _tmain(int argc, _TCHAR* argv[]) {
21     //Init log
22     initlog(argv[0]);
23
24     if (!InitWinSock2_0()) {
25         double errorcode = WSAGetLastError();
26         writelog(_T("Unable to Initialize Windows Socket environment, GLE=%d"),
27             errorcode);
28         _tprintf(_T("Unable to Initialize Windows Socket environment, GLE=%d"),
29             errorcode);
30         closeslog();
31         exit(1);
32     }
33     writelog(_T("Windows Socket environment ready"));
34
35     SOCKET hServerSocket;
36     hServerSocket = socket(
37         AF_INET, // The address family. AF_INET specifies TCP/IP
38         SOCK_STREAM, // Protocol type. SOCK_STREAM specified TCP
39         0 // Protoco Name. Should be 0 for AF_INET address family
40     );
41
42     if (hServerSocket == INVALID_SOCKET) {

```

```

41     writelog(_T("Unable to create Server socket"));
42     _tprintf(_T("Unable to create Server socket"));
43     // Cleanup the environment initialized by WSASStartup()
44     WSACleanup();
45     closelog();
46     exit(2);
47 }
48 writelog(_T("Server socket created"));
49
50 // Create the structure describing various Server parameters
51 struct sockaddr_in serverAddr;
52
53 serverAddr.sin_family = AF_INET;      // The address family. MUST be
54     AF_INET
55 size_t    convtd;
56 char *pMBBuffer = new char[20];
57 wcstombs_s(&convtd, pMBBuffer, 20, szServerIPAddr, 20);
58 //serverAddr.sin_addr.s_addr = inet_addr(pMBBuffer);
59 serverAddr.sin_addr.s_addr = INADDR_ANY;
60 delete[] pMBBuffer;
61 serverAddr.sin_port = htons(nServerPort);
62
63 // Bind the Server socket to the address & port
64 if (bind(hServerSocket, (struct sockaddr *) &serverAddr, sizeof(serverAddr)
65     )) == SOCKET_ERROR) {
66     writelog(_T("Unable to bind to %s on port %d"), szServerIPAddr,
67         nServerPort);
68     _tprintf(_T("Unable to bind to %s on port %d"), szServerIPAddr,
69         nServerPort);
70     // Free the socket and cleanup the environment initialized by WSASStartup
71     ()
72     closesocket(hServerSocket);
73     WSACleanup();
74     closelog();
75     exit(3);
76 }
77 writelog(_T("Bind"));
78
79 // Put the Server socket in listen state so that it can wait for client
80 connections
81 if (listen(hServerSocket, SOMAXCONN) == SOCKET_ERROR) {
82     writelog(_T("Unable to put server in listen state"));
83     _tprintf(_T("Unable to put server in listen state"));
84     // Free the socket and cleanup the environment initialized by WSASStartup
85     ()

```

```

79     closesocket(hServerSocket);
80     WSACleanup();
81     closelog();
82     exit(4);
83 }
84 writelog(_T("Ready for connection"));
85 _tprintf(_T("Ready for connection\n"));
86
87 // Start the infinite loop
88 while (true) {
89     // As the socket is in listen mode there is a connection request pending
90     .
91     // Calling accept( ) will succeed and return the socket for the request.
92     CLIENT_INFO *pClientInfo = new CLIENT_INFO;
93     int nSize = sizeof(pClientInfo->clientAddr);
94
95     pClientInfo->hClientSocket = accept(hServerSocket, (struct sockaddr *) &
96         pClientInfo->clientAddr, &nSize);
97     if (pClientInfo->hClientSocket == INVALID_SOCKET) {
98         writelog(_T("accept() failed"));
99         _tprintf(_T("accept() failed\n"));
100     }
101     else {
102         HANDLE hClientThread;
103         DWORD dwThreadId;
104
105         wchar_t* sin_addr = new wchar_t[20];
106         size_t convtd;
107         mbstowcs_s(&convtd, sin_addr, 20, inet_ntoa(pClientInfo->clientAddr.
108             sin_addr), 20);
109         writelog(_T("Client connected from %s:%d"), sin_addr, pClientInfo->
110             clientAddr.sin_port);
111         _tprintf(_T("Client connected from %s:%d\n"), sin_addr, pClientInfo->
112             clientAddr.sin_port);
113         delete[] sin_addr;
114
115         // Start the client thread
116         hClientThread = CreateThread(NULL, 0,
117             (LPTHREAD_START_ROUTINE)ClientThread,
118             (LPVOID)pClientInfo, 0, &dwThreadId);
119         if (hClientThread == NULL) {
120             writelog(_T("Unable to create client thread"));
121             _tprintf(_T("Unable to create client thread\n"));
122         }
123     }
124 }

```



```

119         CloseHandle(hClientThread);
120     }
121 }
122 }
123
124 closesocket(hServerSocket);
125 WSACleanup();
126 closelog();
127 exit(0);
128 }
129
130 bool InitWinSock2_0() {
131     WSADATA wsaData;
132     WORD wVersion = MAKEWORD(2, 0);
133
134     if (!WSAStartup(wVersion, &wsaData))
135         return true;
136
137     return false;
138 }
139
140 BOOL WINAPI ClientThread(LPVOID lpData) {
141     CLIENT_INFO *pClientInfo = (CLIENT_INFO *)lpData;
142     _TCHAR szBuffer[1024];
143     int nLength;
144
145     while (1) {
146         nLength = recv(pClientInfo->hClientSocket, (char *)szBuffer, sizeof(
            szBuffer), 0);
147         wchar_t* sin_addr = new wchar_t[20];
148         size_t convtd;
149         mbstowcs_s(&convtd, sin_addr, 20, inet_ntoa(pClientInfo->clientAddr.
            sin_addr), 20);
150         if (nLength > 0) {
151             szBuffer[nLength] = '\\0';
152             writelog(_T("Received %s from %s:%d"), szBuffer, sin_addr, pClientInfo
                ->clientAddr.sin_port);
153             _tprintf(_T("Received %s from %s:%d\\n"), szBuffer, sin_addr,
                pClientInfo->clientAddr.sin_port);
154
155             // Convert the string to upper case and send it back, if its not QUIT
156             //_wcsdup(szBuffer);
157             if (wcscmp(szBuffer, _T("QUIT")) == 0) {
158                 closesocket(pClientInfo->hClientSocket);
159                 delete pClientInfo;

```

```

160         return TRUE;
161     }
162     // send() may not be able to send the complete data in one go.
163     // So try sending the data in multiple requests
164     int nCntSend = 0;
165     _TCHAR *pBuffer = szBuffer;
166
167     while ((nCntSend = send(pClientInfo->hClientSocket, (char *)pBuffer,
168         nLength, 0) != nLength)) {
169         if (nCntSend == -1) {
170             writelog(_T("Error sending the data to %s:%d"), sin_addr,
171                 pClientInfo->clientAddr.sin_port);
172             _tprintf(_T("Error sending the data to %s:%d\n"), sin_addr,
173                 pClientInfo->clientAddr.sin_port);
174             break;
175         }
176         if (nCntSend == nLength)
177             break;
178
179         pBuffer += nCntSend;
180         nLength -= nCntSend;
181     }
182
183     else {
184         writelog(_T("Error reading the data from %s:%d"), sin_addr,
185             pClientInfo->clientAddr.sin_port);
186         _tprintf(_T("Error reading the data from %s:%d\n"), sin_addr,
187             pClientInfo->clientAddr.sin_port);
188     }
189
190     delete[] sin_addr;
191 }
192
193 return TRUE;
194 }

```

Клиент отправляет серверу сообщения, и получает эхо-ответ. Слово QUIT зарезервировано для завершения работы.

Листинг 17: Клиент для работы с Win-сокетами

(src/InterProcessCommunication/WinSockClient/main.cpp)

```

1 #define _WINSOCK_DEPRECATED_NO_WARNINGS
2 #include <winsock2.h>
3 #include "logger.h"
4
5 #pragma comment(lib, "Ws2_32.lib")

```

```

6
7 _TCHAR szServerIPAddr[20]; // server IP
8 int nServerPort; // server port
9
10 bool InitWinSock2_0();
11
12 int _tmain(int argc, _TCHAR* argv[]) {
13     //Init log
14     initlog(argv[0]);
15
16     _tprintf(_T("Enter the server IP Address: "));
17     wscanf_s(_T("%19s"), szServerIPAddr, _countof(szServerIPAddr));
18     _tprintf(_T("Enter the server port number: "));
19     wscanf_s(_T("%i"), &nServerPort);
20
21     if (!InitWinSock2_0()) {
22         double errorcode = WSAGetLastError();
23         writelog(_T("Unable to Initialize Windows Socket environment, GLE=%d"),
24             errorcode);
25         _tprintf(_T("Unable to Initialize Windows Socket environment, GLE=%d"),
26             errorcode);
27         closelog();
28         exit(1);
29     }
30     SOCKET hClientSocket;
31     hClientSocket = socket(
32         AF_INET,          // The address family. AF_INET specifies TCP/IP
33         SOCK_STREAM,      // Protocol type. SOCK_STREAM specified TCP
34         0);               // Protocol Name. Should be 0 for AF_INET address family
35
36     if (hClientSocket == INVALID_SOCKET) {
37         writelog(_T("Unable to create Server socket"));
38         _tprintf(_T("Unable to create Server socket"));
39         // Cleanup the environment initialized by WSASStartup()
40         WSACleanup();
41         closelog();
42         exit(2);
43     }
44     writelog(_T("Client socket created"));
45
46     // Create the structure describing various Server parameters
47     struct sockaddr_in serverAddr;
48

```

```

49  serverAddr.sin_family = AF_INET;      // The address family. MUST be
    AF_INET
50  size_t   convtd;
51  char *pMBBuffer = new char[20];
52  wcstombs_s(&convtd, pMBBuffer, 20, szServerIPAddr, 20);
53  serverAddr.sin_addr.s_addr = inet_addr(pMBBuffer);
54  delete[] pMBBuffer;
55  serverAddr.sin_port = htons(nServerPort);
56
57  // Connect to the server
58  if (connect(hClientSocket, (struct sockaddr *) &serverAddr, sizeof(
    serverAddr)) < 0) {
59      writelog(_T("Unable to connect to %s on port %d"), szServerIPAddr,
        nServerPort);
60      _tprintf(_T("Unable to connect to %s on port %d"), szServerIPAddr,
        nServerPort);
61      closesocket(hClientSocket);
62      WSACleanup();
63      closelog();
64      exit(3);
65  }
66  writelog(_T("Connect"));
67
68  _TCHAR szBuffer[1024] = _T("");
69
70  while (wcscmp(szBuffer, _T("QUIT")) != 0) {
71      _tprintf(_T("Enter the string to send (QUIT) to stop: "));
72      wscanf_s(_T("%1023s"), szBuffer, _countof(szBuffer));
73
74      int nLength = (wcslen(szBuffer) + 1) * sizeof(_TCHAR);
75
76      // send( ) may not be able to send the complete data in one go.
77      // So try sending the data in multiple requests
78      int nCntSend = 0;
79      _TCHAR *pBuffer = szBuffer;
80
81      while ((nCntSend = send(hClientSocket, (char *)pBuffer, nLength, 0) !=
        nLength)) {
82          if (nCntSend == -1) {
83              writelog(_T("Error sending the data to server"));
84              _tprintf(_T("Error sending the data to server\n"));
85              break;
86          }
87          if (nCntSend == nLength)
88              break;

```

```

89
90     pBuffer += nCntSend;
91     nLength -= nCntSend;
92 }
93
94 _wcsdup(szBuffer);
95 if (wcscmp(szBuffer, _T("QUIT")) == 0) {
96     break;
97 }
98
99 nLength = recv(hClientSocket, (char *)szBuffer, sizeof(szBuffer), 0);
100 if (nLength > 0) {
101     szBuffer[nLength] = '\\0';
102     writelog(_T("Received %s from server"), szBuffer);
103     _tprintf(_T("Received %s from server\\n"), szBuffer);
104 }
105 }
106
107 closesocket(hClientSocket);
108 WSACleanup();
109 closelog();
110 exit(0);
111 }
112
113 bool InitWinSock2_0() {
114     WSADATA wsaData;
115     WORD wVersion = MAKEWORD(2, 0);
116
117     if (!WSAStartup(wVersion, &wsaData))
118         return true;
119
120     return false;
121 }

```

Листинг 18 и 19 содержит протокол работы программы.

Листинг 18: Протокол работы серверного модуля программы работы с сокетами Windows

```

1 [14/2/2015 11:32:39] C:\Users\win7\Documents\Visual Studio 2013\Projects\
   InterProcessCommunication\Debug\WinSockServer.exe is starting.
2 [14/2/2015 11:32:39] Windows Socket environment ready
3 [14/2/2015 11:32:39] Server socket created
4 [14/2/2015 11:32:39] Bind
5 [14/2/2015 11:32:39] Ready for connection
6 [14/2/2015 11:33:6] Client connected from 192.168.124.235
7 [14/2/2015 11:33:10] Received 1 from 192.168.124.235

```

```

8 [14/2/2015 11:33:14] Received Hello from 192.168.124.235
9 [14/2/2015 11:33:22] Received QUIT from 192.168.124.235
10 [14/2/2015 11:33:28] Shutting down.

```

Листинг 19: Протокол работы клиентского модуля программы работы с сокетами Windows

```

1 [14/2/2015 11:32:41] WinSockClient.exe is starting.
2 [14/2/2015 11:33:6] Windows Socket environment ready
3 [14/2/2015 11:33:6] Client socket created
4 [14/2/2015 11:33:6] Connect
5 [14/2/2015 11:33:10] Received 1 from server
6 [14/2/2015 11:33:14] Received Hello from server
7 [14/2/2015 11:33:22] Shutting down.

```

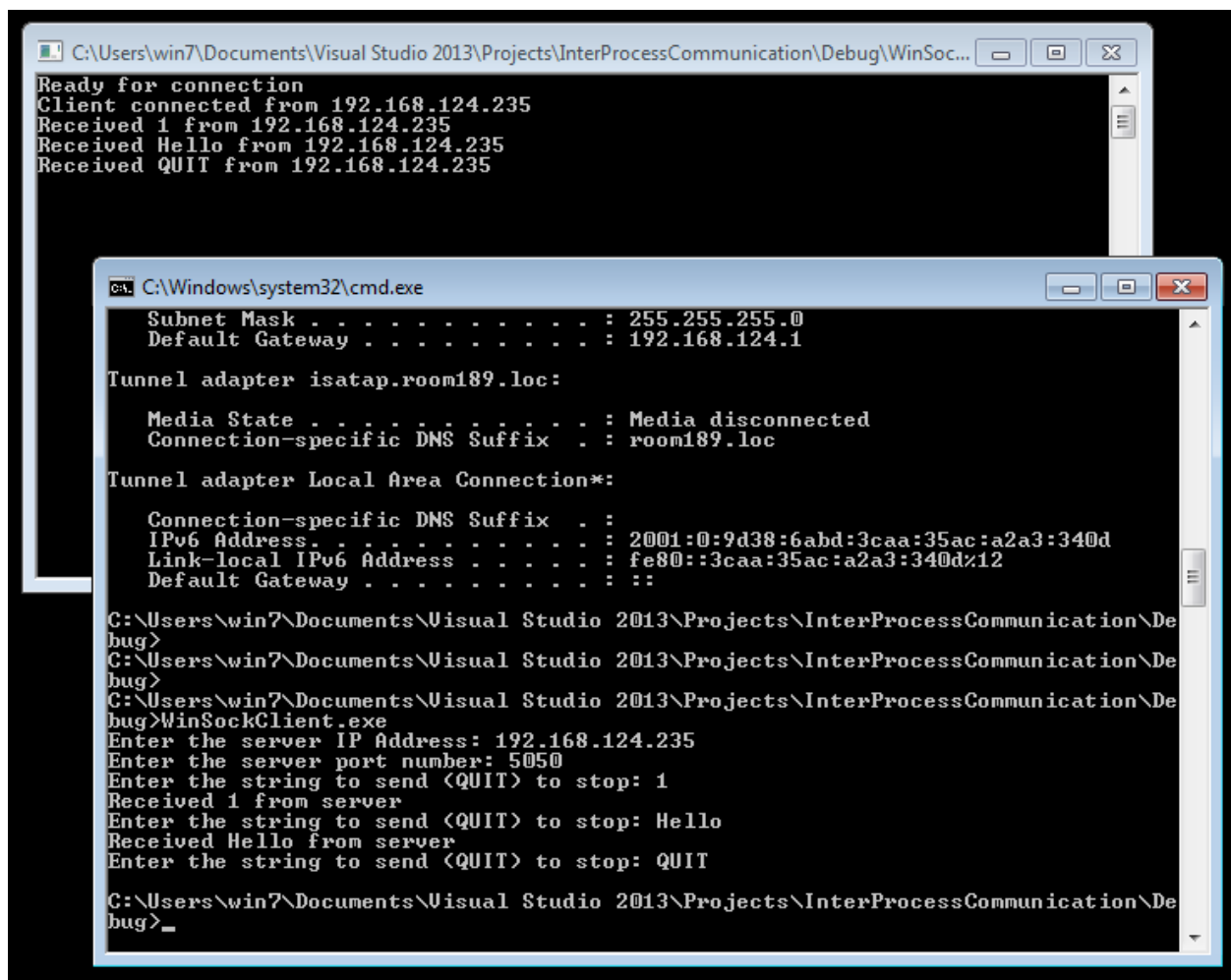


Рис. 10: Работа с сокетами.

Результаты работы программы показаны на рисунке 10. Клиент соединяется с сервером по локальному IP адресу. Серверный слушающий сокет связан с адресом INADDR\_ANY, т.е. прослушивает все сетевые интерфейсы.

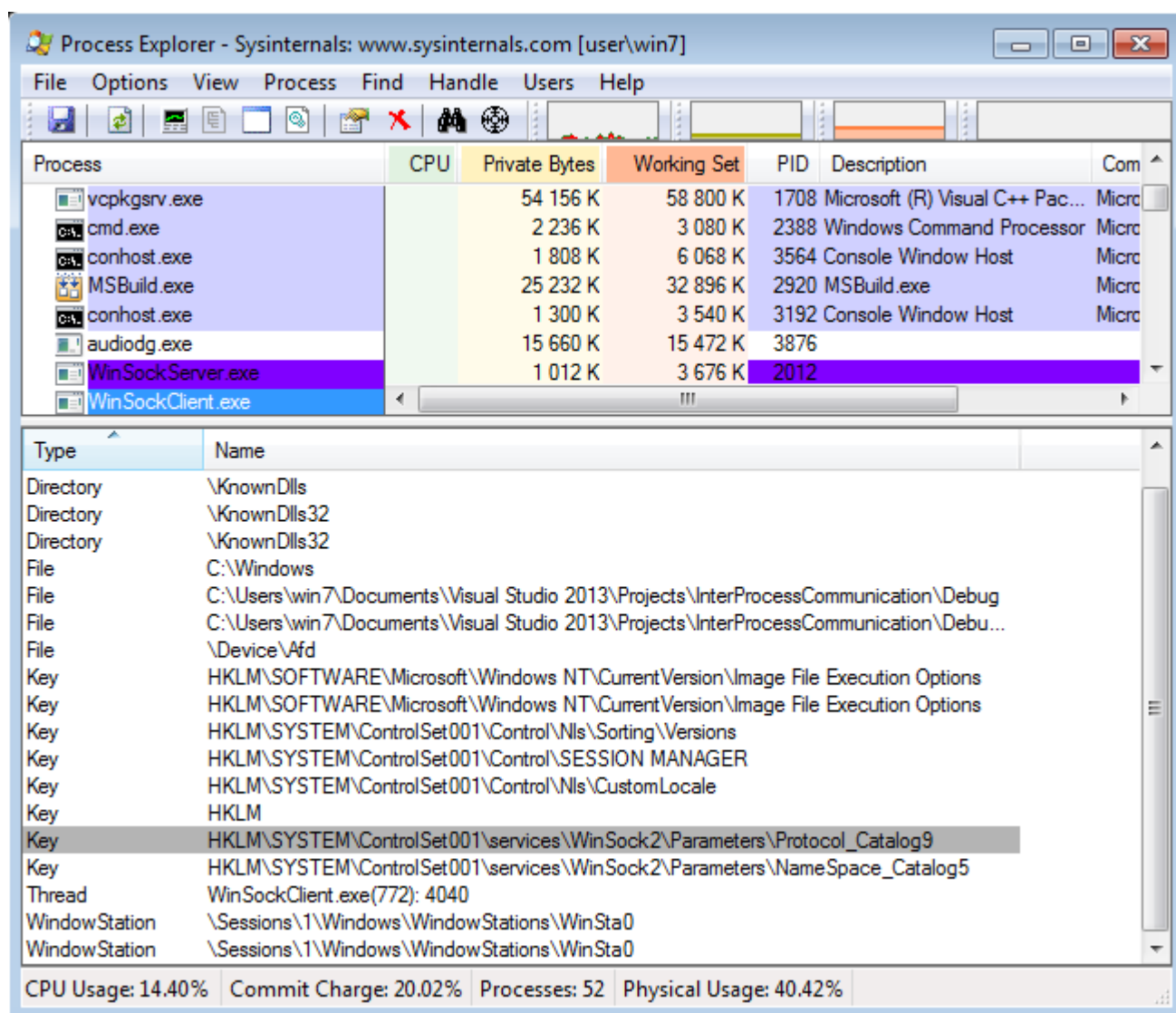


Рис. 11: Сокеты в списке ресурсов

На рисунке 11 показано как сокеты видны системе.

Библиотека Winsock поддерживает два вида сокетов – синхронные (блокируемые) и асинхронные (неблокируемые). Синхронные сокеты задерживают управление на время выполнения операции, а асинхронные возвращают его немедленно, продолжая выполнение в фоновом режиме, и, закончив работу, уведомляют об этом вызывающий код.

Сокеты позволяют работать со множеством протоколов и являются удобным средством межпроцессорного взаимодействия[3], но в данной работе рассматриваются только сокеты семейства протоколов TCP/IP, используемых для обмена данными между узлами сети Интернет.

Независимо от вида, сокеты делятся на два типа – потоковые и дейтаграммные. Потоковые сокеты работают с установкой соединения, обеспечивая надежную идентификацию

обоих сторон и гарантируют целостность и успешность доставки данных. Дейтаграмные сокеты работают без установки соединения и не обеспечивают ни идентификации отправителя, ни контроля успешности доставки данных, зато они заметно быстрее потоковых. Дейтаграмные сокеты опираются на протокол UDP, а потоковые на TCP.

Выбор того или иного типа сокетов определяется транспортным протоколом, на котором работает сервер, – клиент не может по своему желанию установить с дейтаграммным сервером потоковое соединение.



# Порты завершения

Операциям ввода и вывода присуща более медленная скорость выполнения по сравнению с другими видами обработки. Причиной такого замедления являются следующие факторы[4]:

- Задержки, обусловленные затратами времени на поиск нужных дорожек и секторов на устройствах произвольного доступа (диски, компакт-диски).
- Задержки, обусловленные сравнительно низкой скоростью обмена данными между физическими устройствами и системной памятью.
- Задержки при передаче данных по сети с использованием файловых серверов, хранилищ данных и так далее.

Во всех предыдущих примерах операции ввода/вывода выполняются синхронно с потоком, поэтому весь поток вынужден простаивать, пока они не завершатся.

В этом примере показано, каким образом можно организовать продолжение выполнения потока, не дожидаясь завершения операций ввода/вывода, что будет соответствовать выполнению потоками асинхронного ввода/вывода.

Порты завершения оказываются чрезвычайно полезными при построении масштабируемых серверов, способных обеспечивать поддержку большого количества клиентов без создания для каждого из них отдельного потока.

Листинг 20 показывает реализацию порта завершения. Для работы с ним использовался клиент из предыдущего примера.

Листинг 20: Порт завершения (src/InterProcessCommunication/CompletionPortServer/main.cpp)

```
1 #include <winsock2.h>
2 #include <windows.h>
3 #include <stdio.h>
4 #include "logger.h"
5
```

```

6 #pragma comment(lib, "Ws2_32.lib")
7
8 _TCHAR szServerIPAddr[] = _T("127.0.0.1"); // server IP
9 int nServerPort = 5050; // server port
10 // clients to talk with the server
11
12 #define DATA_BUFSIZE 1024
13 #define EMPTY_MSG _T("...")
14
15 typedef struct{
16     OVERLAPPED Overlapped;
17     ///////////////////////////////////
18     WSABUF DataBuf;
19     CHAR Buffer[DATA_BUFSIZE];
20     DWORD BytesSend;
21     DWORD BytesRecv;
22     DWORD TotalBytes;
23     SOCKADDR_IN client;
24 } PER_IO_OPERATION_DATA, *LPPER_IO_OPERATION_DATA;
25
26
27 typedef struct{
28     SOCKET Socket;
29 } PER_HANDLE_DATA, *LPPER_HANDLE_DATA;
30
31
32 DWORD WINAPI ClientThread(LPVOID CompletionPortID);
33
34 int _tmain(int argc, _TCHAR* argv[]) {
35     //Init log
36     initlog(argv[0]);
37
38     SOCKADDR_IN server;
39     SOCKADDR_IN client;
40
41     SOCKET Socket;
42     SOCKET Accept;
43
44     HANDLE CompletionPort;
45     SYSTEM_INFO SysInfo;
46     HANDLE Thread;
47
48     LPPER_HANDLE_DATA PerHandleData;
49     LPPER_IO_OPERATION_DATA PerIoData;
50

```

```

51  DWORD SendBytes;
52  DWORD Flags;
53  DWORD ThreadID;
54  WSADATA wsaData;
55
56  // инициализируем WinSock:
57  if ((WSAStartup(0x0202, &wsaData)) != 0){
58      double errorcode = WSAGetLastError();
59      writelog(_T("Unable to Initialize Windows Socket environment, GLE=%d"),
60              errorcode);
61      _tprintf(_T("Unable to Initialize Windows Socket environment, GLE=%d"),
62              errorcode);
63      closelog();
64      exit(1);
65  }
66  writelog(_T("Windows Socket environment ready"));
67
68  // Создаём порт завершения:
69  if ((CompletionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL,
70      0, 0)) == NULL){
71      double errorcode = WSAGetLastError();
72      writelog(_T("CreateIoCompletionPort failed, GLE=%d"), errorcode);
73      _tprintf(_T("CreateIoCompletionPort failed, GLE=%d"), errorcode);
74      closelog();
75      exit(2);
76  }
77  writelog(_T("Io Completion Port created"));
78
79  // Получаем информацию о системе:
80  GetSystemInfo(&SysInfo);
81  // создаём два потока на процессор:.
82  for (size_t i = 0; i < SysInfo.dwNumberOfProcessors * 2; i++){
83      // создаём рабочий поток, в качестве параметра передаём ей порт заверше
ния
84      if ((Thread = CreateThread(NULL, 0, ClientThread, CompletionPort, 0, &
85          ThreadID)) == NULL) {
86          double errorcode = WSAGetLastError();
87          writelog(_T("CreateThread() failed, GLE=%d"), errorcode);
88          _tprintf(_T("CreateThread() failed, GLE=%d"), errorcode);
89          closelog();
90          exit(3);
91      }
92      CloseHandle(Thread);
93  }

```

```

91 // Создаём слушающий сокет:
92 if ((Socket = WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
    WSA_FLAG_OVERLAPPED)) == INVALID_SOCKET) {
93     double errorcode = WSAGetLastError();
94     writelog(_T("WSASocket() failed, GLE=%d"), errorcode);
95     _tprintf(_T("WSASocket() failed, GLE=%d"), errorcode);
96     closelog();
97     exit(4);
98 }
99
100 server.sin_family = AF_INET;
101 server.sin_addr.s_addr = htonl(INADDR_ANY);
102 server.sin_port = htons(nServerPort);
103
104 if (bind(Socket, (PSOCKADDR)&server, sizeof(server)) == SOCKET_ERROR){
105     double errorcode = WSAGetLastError();
106     writelog(_T("bind() failed, GLE=%d"), errorcode);
107     _tprintf(_T("bind() failed, GLE=%d"), errorcode);
108     closelog();
109     exit(5);
110 }
111
112 if (listen(Socket, 5) == SOCKET_ERROR){
113     double errorcode = WSAGetLastError();
114     writelog(_T("listen() failed, GLE=%d"), errorcode);
115     _tprintf(_T("listen() failed, GLE=%d"), errorcode);
116     closelog();
117     exit(6);
118 }
119 writelog(_T("Ready for connection"));
120 _tprintf(_T("Ready for connection\n"));
121
122 // принимаем соединения и передаём их порту завершения:
123 while (TRUE){
124     // принимаем соединение:
125     if ((Accept = WSAAccept(Socket, (PSOCKADDR)&client, NULL, NULL, 0)) ==
        SOCKET_ERROR){
126         double errorcode = WSAGetLastError();
127         writelog(_T("WSAAccept() failed, GLE=%d"), errorcode);
128         _tprintf(_T("WSAAccept() failed, GLE=%d"), errorcode);
129         continue;
130     }
131
132     // Выделяем память под структуру, которая будет хранить информацию о сок
        ете:

```

```

133     if ((PerHandleData = (LPPER_HANDLE_DATA)GlobalAlloc(GPTR, sizeof(
134         PER_HANDLE_DATA))) == NULL){
135         double errorcode = WSAGetLastError();
136         writelog(_T("GlobalAlloc() failed with error %d"), errorcode);
137         _tprintf(_T("GlobalAlloc() failed with error %d"), errorcode);
138         closelog();
139         exit(7);
140     }
141     writelog(_T("Socket %d connected\n"), Accept);
142     _tprintf(_T("Socket %d connected\n"), Accept);
143
144     PerHandleData->Socket = Accept; // сохраняем описатель сокета
145
146     //привязываем сокет к порту завершения:
147     if (CreateIoCompletionPort((HANDLE)Accept, CompletionPort, (DWORD)
148         PerHandleData, 0) == NULL){
149         double errorcode = WSAGetLastError();
150         writelog(_T("CreateIoCompletionPort() failed with error %d"),
151             errorcode);
152         _tprintf(_T("CreateIoCompletionPort() failed with error %d"),
153             errorcode);
154         closelog();
155         exit(8);
156     }
157
158     // выделяем память под данные операции ввода вывода:
159     if ((PerIoData = (LPPER_IO_OPERATION_DATA)GlobalAlloc(GPTR, sizeof(
160         PER_IO_OPERATION_DATA))) == NULL){
161         double errorcode = WSAGetLastError();
162         writelog(_T("GlobalAlloc() failed with error %d"), errorcode);
163         _tprintf(_T("GlobalAlloc() failed with error %d"), errorcode);
164         closelog();
165         exit(9);
166     }
167
168     ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
169
170     // задаём изначальные данные для операции ввода вывода:
171     PerIoData->BytesSend = 0;
172     PerIoData->BytesRecv = 0;
173     PerIoData->DataBuf.len = (wcslen(EMPTY_MSG) + 1) * sizeof(_TCHAR);
174     PerIoData->DataBuf.buf = (char*) EMPTY_MSG;
175     PerIoData->client = client;
176     PerIoData->TotalBytes = 0;

```

```

173     Flags = 0;
174
175     // отправляем welcome message
176     // остальные операции будут выполняться в рабочем потоке
177     if (WSASend(Accept, &(PerIoData->DataBuf), 1, &SendBytes, 0, &(PerIoData
        ->Overlapped), NULL) == SOCKET_ERROR){
178         if (WSAGetLastError() != ERROR_IO_PENDING){
179             double errorcode = WSAGetLastError();
180             writelog(_T("WSASend() failed with error %d"), errorcode);
181             _tprintf(_T("WSASend() failed with error %d\n"), errorcode);
182             closelog();
183             exit(10);
184         }
185     }
186 }
187 closelog();
188 exit(0);
189 }
190
191 DWORD WINAPI ClientThread(LPVOID CompletionPortID) {
192     HANDLE CompletionPort = (HANDLE)CompletionPortID;
193     DWORD BytesTransferred;
194     LPPER_HANDLE_DATA PerHandleData;
195     LPPER_IO_OPERATION_DATA PerIoData;
196
197     DWORD SendBytes, RecvBytes;
198     DWORD Flags;
199
200     while (TRUE){
201         // ожидание завершения ввода-вывода на любом из сокетов
202         // которые связаны с портом завершения:
203         if (GetQueuedCompletionStatus(CompletionPort, &BytesTransferred,
204             (LPDWORD)&PerHandleData, (LPOVERLAPPED *)&PerIoData, INFINITE) == 0){
205             double errorcode = GetLastError();
206             writelog(_T("WSASend() failed with error %d"), errorcode);
207             _tprintf(_T("WSASend() failed with error %d\n"), errorcode);
208             return 0;
209         }
210
211         // проверяем на ошибки. Если была - значит надо закрыть сокет и очистить
        память за собой:
212         if (BytesTransferred == 0){
213             // тк не было переданно ни одного байта - значит сокет закрыли на той
            стороне
214             // мы должны сделать то же самое:

```

```

215     writelog(_T("Closing socket %d"), PerHandleData->Socket);
216     writelog(_T("Total bytes:%d\n"), PerIoData->TotalBytes);
217     _tprintf(_T("Closing socket %d\nTotal bytes:%d\n"), PerHandleData->
        Socket, PerIoData->TotalBytes);
218
219     // закрываем сокет:
220     if (closesocket(PerHandleData->Socket) == SOCKET_ERROR){
221         double errorcode = WSAGetLastError();
222         writelog(_T("closesocket() failed with error %d"), errorcode);
223         _tprintf(_T("closesocket() failed with error %d\n"), errorcode);
224         return 0;
225     }
226
227     // очищаем память:
228     GlobalFree(PerHandleData);
229     GlobalFree(PerIoData);
230
231     // ждём следующую операцию
232     continue;
233 }
234
235 PerIoData->TotalBytes += BytesTransferred;
236
237 // Проверим значение BytesRecv - если оно равно нулю - значит мы получили
и данные от клиента:
238 if (PerIoData->BytesRecv == 0){
239     PerIoData->BytesRecv = BytesTransferred;
240     PerIoData->BytesSend = 0;
241 }
242 else{
243     PerIoData->BytesSend += BytesTransferred;
244 }
245
246 // мы должны отослать все принятые байты назад:
247 if (PerIoData->BytesRecv > PerIoData->BytesSend){
248     // Шлём данные через WSASend - тк всё сразу может не отослаться
249     // необходимо слать до упора.
250     // Теоретически, за один вызов WSASend все данные могут не отправиться!
251     ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
252
253     PerIoData->DataBuf.buf = PerIoData->Buffer + PerIoData->BytesSend;
254     PerIoData->DataBuf.len = PerIoData->BytesRecv - PerIoData->BytesSend;
255
256     // Convert the string to upper case and send it back, if its not QUIT
257     if (wcscmp((_TCHAR*)PerIoData->Buffer, _T("QUIT")) == 0) {

```

```

258     _tprintf(_T("RCV %s\n"), PerIoData->Buffer);
259     closesocket(PerHandleData->Socket);
260     return TRUE;
261 }
262 if (WSASend(PerHandleData->Socket, &(PerIoData->DataBuf), 1, &
    SendBytes, 0,
263     &(PerIoData->Overlapped), NULL) == SOCKET_ERROR) {
264     if (WSAGetLastError() != ERROR_IO_PENDING) {
265         double errorcode = WSAGetLastError();
266         writelog(_T("WSASend() failed with error %d"), errorcode);
267         _tprintf(_T("WSASend() failed with error %d\n"), errorcode);
268         return 0;
269     }
270 }
271 }
272 else{
273     PerIoData->BytesRecv = 0;
274
275     // ожидаем ещё данные от пользователя:
276     Flags = 0;
277     ZeroMemory(&(PerIoData->Overlapped), sizeof(OVERLAPPED));
278
279     PerIoData->DataBuf.len = DATA_BUFSIZE;
280     PerIoData->DataBuf.buf = PerIoData->Buffer;
281
282     if (WSARecv(PerHandleData->Socket, &(PerIoData->DataBuf), 1, &
        RecvBytes, &Flags,
283     &(PerIoData->Overlapped), NULL) == SOCKET_ERROR){
284         if (WSAGetLastError() != ERROR_IO_PENDING){
285             double errorcode = WSAGetLastError();
286             writelog(_T("WSARecv() failed with error %d"), errorcode);
287             _tprintf(_T("WSASend() failed with error %d\n"), errorcode);
288             return 0;
289         }
290     }
291     writelog(_T("Get task from %d: %s"), PerHandleData->Socket, PerIoData
        ->Buffer);
292     _tprintf(_T("Get task from %d: %s\n"), PerHandleData->Socket,
        PerIoData->Buffer);
293 }
294 }
295 }

```



Объект порт, по сути, представляет собой очередь событий ядра, из которой извлекаются и в которую добавляются сообщения об операциях ввода/вывода. Туда добавляются не все текущие операции, а только те, которые указаны порту. Делается это путем связывания дескриптора файла (сокета, именованного канала, мэйлслота и т.д.) с дескриптором порта. Когда над файлом инициируется асинхронная операция ввода/вывода, то после ее завершения соответствующая запись добавляется в порт.

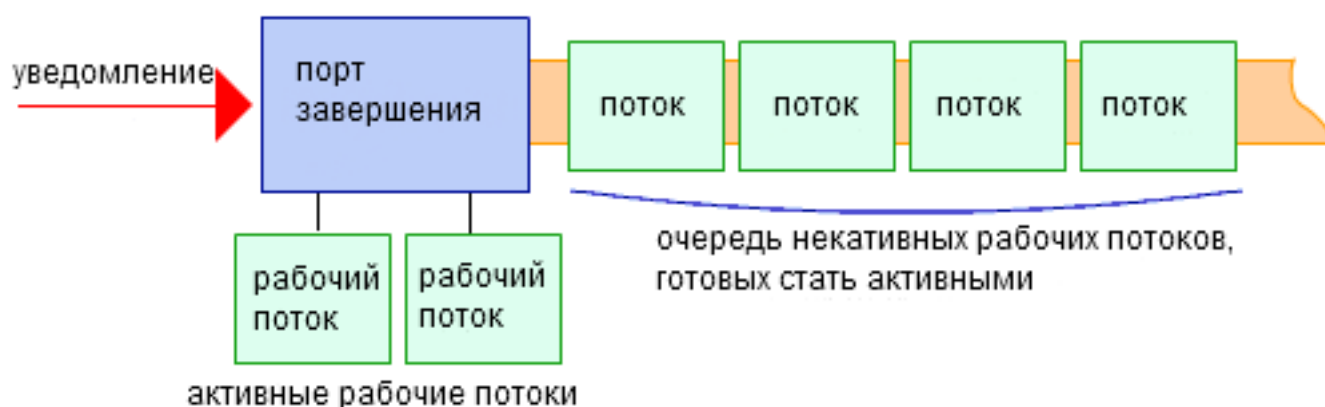


Рис. 12: Схема работы с портом завершения

Для обработки результатов используется пул потоков, количество которых выбирается пользователем, и, как правило, коррелирует с количеством ядер центрального процессора. Когда поток присоединяют к пулу, он извлекает из очереди один результат операции и обрабатывает его. Если на момент присоединения очередь пуста, то поток засыпает до тех пор, пока не появится сообщение для обработки (см. рис. 12).

Порт создаётся командой[5]

```

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

Параметрами являются:

- FileHandle – дескриптор файла открывается для завершения асинхронной операции ввода-вывода (Если FileHandle установлен в INVALID\_HANDLE\_VALUE, функция CreateIoCompletionPort создает порт завершение ввода - вывода, не связывая его с файлом);

- ExistingCompletionPort – дескриптор порта завершения ввода - вывода (Если этот параметр определяет существующий порт завершения I/O, функция связывает его с файлом, указанным параметром FileHandle);
- CompletionKey – код завершения для каждого файла, который включается в каждый блок завершения ввода - вывода для указанного файла;
- NumberOfConcurrentThreads – максимальное число потоков, которым операционная система дает возможность одновременно работать с блоками завершения ввода - вывода для порта завершения ввода - вывода.

Если функция завершается успешно, возвращаемое значение - дескриптор порта завершения ввода-вывода (I/O), который связан с указанным файлом.

Теперь рассмотрим функцию API, которая присоединяет вызывающий ее поток к пулу[6]:

```

BOOL WINAPI GetQueuedCompletionStatus(
    _In_   HANDLE CompletionPort,
    _Out_  LPDWORD lpNumberOfBytes,
    _Out_  PULONG_PTR lpCompletionKey,
    _Out_  LPOVERLAPPED *lpOverlapped,
    _In_   DWORD dwMilliseconds
);

```

Параметры:

- CompletionPort – дескриптор ранее созданного порта завершения;
- lpNumberOfBytes – указатель на переменную, которая получает число байтов, перемещенных в ходе операции ввода-вывода (I/O), которая завершилась;
- lpCompletionKey – указатель на переменную, которая получает значение кода завершения, связанного с дескриптором файла, операция ввода-вывода (I/O) которого завершилась;
- lpOverlapped – указатель на переменную-буфер;
- dwMilliseconds – число миллисекунд, которое вызывающая программа будет ждать пакет завершения, чтобы появиться в порте завершения.

Если функция исключает из очереди пакет окончания работы в следствие успешной операции I/O порта завершения, возвращаемое значение – не ноль.

Теперь рассмотрим внутреннюю структура порта завершения. Фактически, он представляет собой следующую структуру:

```
typedef struct _IO_COMPLETION { KQUEUE Queue; } IO_COMPLETION;
```

Это просто очередь событий ядра. Вот описание структуры KQUEUE:

```
typedef struct _KQUEUE {  
    DISPATCHER_HEADER Header;  
    LIST_ENTRY EnrtyListHead; //очередь пакетов  
    DWORD CurrentCount;  
    DWORD MaximumCount;  
    LIST_ENTRY ThreadListHead; //очередь ожидающих потоков  
} KQUEUE;
```

При создании порта функцией `CreateIoCompletionPort` вызывается внутренний сервис `NtCreateIoCompletion`. Затем происходит его инициализация с помощью функции `KeInitializeQueue`. Когда происходит связывание порта с объектом «файл», Win32-функция `CreateIoCompletionPort` вызывает `NtSetInformationFile`.

```
NtSetInformationFile(  
    HANDLE FileHandle,  
    PIO_STATUS_BLOCK IoStatusBlock,  
    PVOID FileInformation,  
    ULONG Length,  
    FILE_INFORMATION_CLASS FileInformationClass);
```

Для этой функции `FILE_INFORMATION_CLASS` устанавливается как `FileCompletionInformation`, а в качестве параметра `FileInformation` передается указатель на структуру `IO_COMPLETION_CONTEXT` или `FILE_COMPLETION_INFORMATION`.

```
typedef struct _IO_COMPLETION_CONTEXT {  
    PVOID Port;  
    PVOID Key; } IO_COMPLETION_CONTEXT;
```

```
typedef struct _FILE_COMPLETION_INFORMATION {  
    HANDLE IoCompletionHandle;  
    ULONG CompletionKey; } FILE_COMPLETION_INFORMATION, *PFILE_COMPLETION_INFORMATION
```

После завершения асинхронной операции ввода/вывода для ассоциированного файла диспетчер ввода/вывода создает пакет запроса из структуры `OVERLAPPED` и ключа завершения и помещает его в очередь с помощью вызова `KeInsertQueue`. Когда поток вызывает функцию `GetQueuedCompletionStatus`, на самом деле вызывается функция `NtRemoveIoCompletion`. `NtRemoveIoCompletion` проверяет параметры и вызывает функцию

KeRemoveQueue, которая блокирует поток, если в очереди отсутствуют запросы, или поле CurrentCount структуры KQUEUE больше или равно MaximumCount. Если запросы есть, и число активных потоков меньше максимального, KeRemoveQueue удаляет вызвавший ее поток из очереди ожидающих потоков и увеличивает число активных потоков на 1. При занесении потока в очередь ожидающих потоков поле Queue структуры KTHREAD устанавливается равным адресу порта завершения. Когда запрос помещается в порт завершения функцией PostQueuedCompletionStatus, на самом деле вызывается функция NtSetIoCompletion, которая после проверки параметров и преобразования хендла порта в указатель, вызывает KeInsertQueue.

Листинги 21 и 22 содержат протокол работы программы. Стоит обратить внимание, что в отличие от предыдущего примера, когда клиент ожидал данные сразу после отправки, здесь клиент получает ответ асинхронно.

Листинг 21: Протокол работы серверного модуля программы работы с портом завершения

```
1 [14/2/2015 14:36:25] C:\Users\win7\Documents\Visual Studio 2013\Projects\
   InterProcessCommunication\Debug\CompletionPortServer.exe is starting.
2 [14/2/2015 14:36:25] Windows Socket environment ready
3 [14/2/2015 14:36:25] Io Completion Port created
4 [14/2/2015 14:36:26] Ready for connection
5 [14/2/2015 14:36:54] Socket 136 connected
6
7 [14/2/2015 14:36:54] Get task from 136:
8 [14/2/2015 14:37:1] Get task from 136: Hi!
9 [14/2/2015 14:37:6] Get task from 136: Salut!
10 [14/2/2015 14:37:10] Get task from 136: 1
11 [14/2/2015 14:37:12] Get task from 136: 2
12 [14/2/2015 14:37:13] Get task from 136: 3
```

Листинг 22: Протокол работы клиентского модуля программы работы с портом завершения

```
1 [14/2/2015 14:36:25] C:\Users\win7\Documents\Visual Studio 2013\Projects\
   InterProcessCommunication\Debug\WinSockClient.exe is starting.
2 [14/2/2015 14:36:54] Windows Socket environment ready
3 [14/2/2015 14:36:54] Client socket created
4 [14/2/2015 14:36:54] Connect
5 [14/2/2015 14:37:1] Received ... from server
6 [14/2/2015 14:37:6] Received Hi! from server
7 [14/2/2015 14:37:10] Received Salut! from server
8 [14/2/2015 14:37:12] Received 1 from server
9 [14/2/2015 14:37:13] Received 2 from server
10 [14/2/2015 14:39:1] Shutting down.
```

Рисунок 13 показывает работу программы, а на рисунке 14 видны несколько процессов серверного модуля (в клиентском модуле никаких изменений, по сравнению с обычными сокетами нет).

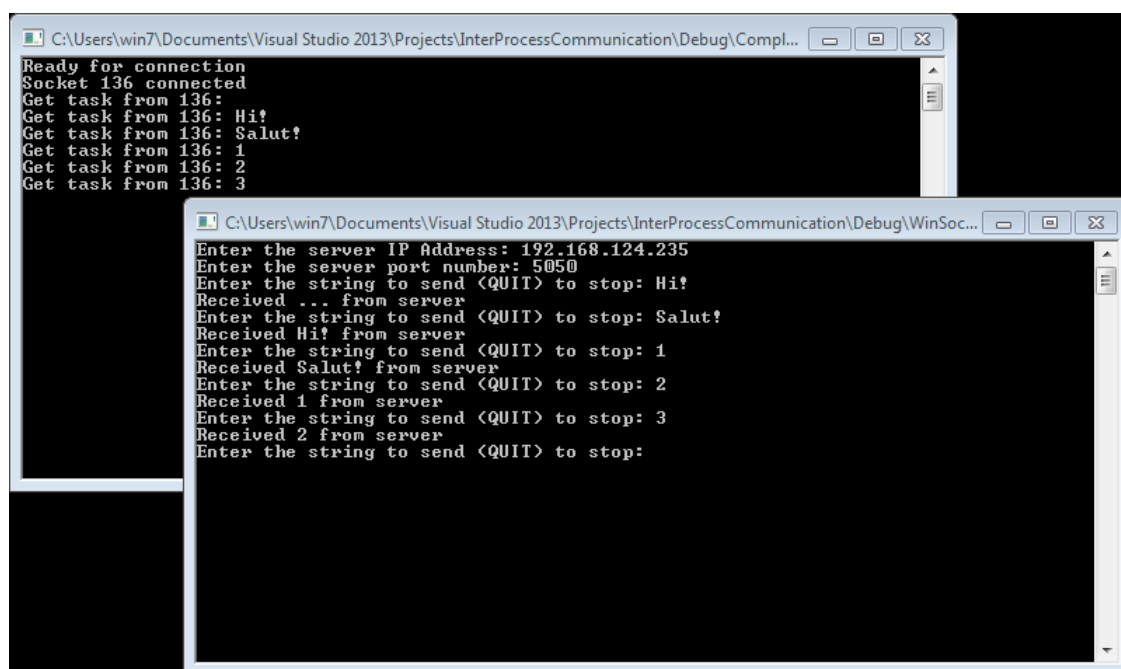


Рис. 13: Программа работы с портом завершения

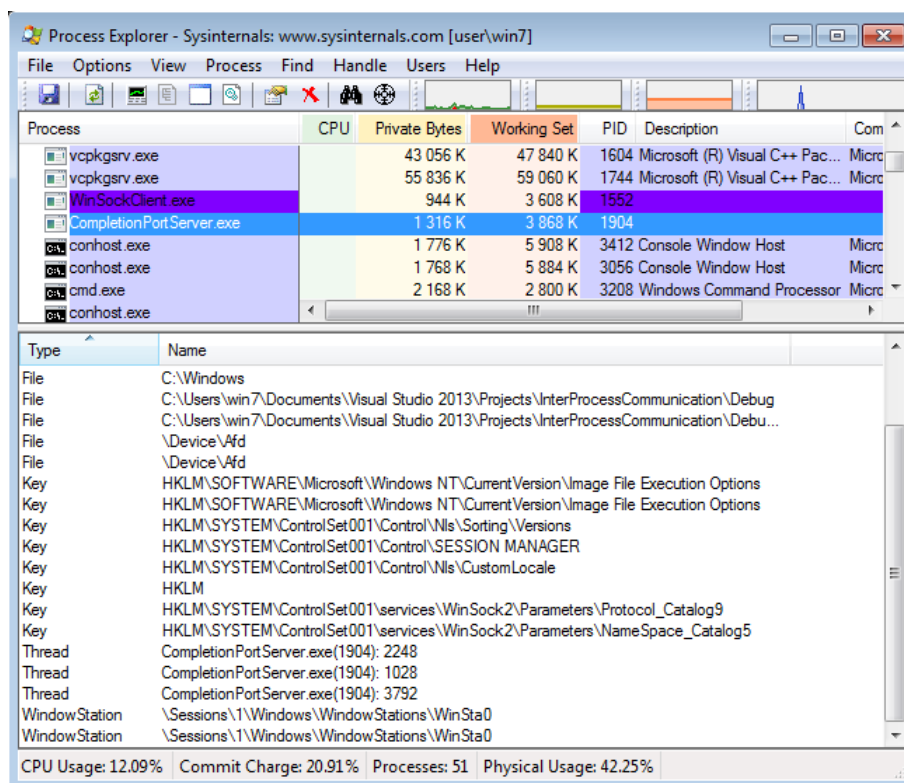


Рис. 14: Процессы серверного модуля

# Сигналы

В отличие от Linux, сигналы в Windows имеют сильно усеченные возможности. Наиболее сложной задаче при работе с сигналами было придумать, что можно с ними сделать. В листинге 23 по сигналу меняется цвет консоли, это видно на рисунке 15.

Листинг 23: Сигналы в Windows  
(src/InterProcessCommunication/Signals/main.cpp)

```
1 #include <windows.h>
2 #include <stdio.h>
3 #include <iostream>
4 #include "logger.h"
5
6 BOOL CtrlHandler(DWORD fdwCtrlType)
7 {
8     HANDLE hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
9     if (hStdout == INVALID_HANDLE_VALUE)
10    {
11        std::cout << "Error while getting input handle" << std::endl;
12        writelog(_T("Error while getting input handle"));
13        return EXIT_FAILURE;
14    }
15
16    switch (fdwCtrlType) //тип сигнала
17    {
18        // Handle the CTRL-C signal.
19        case CTRL_C_EVENT:
20            SetConsoleTextAttribute(hStdout, FOREGROUND_RED | BACKGROUND_BLUE |
21                                   FOREGROUND_INTENSITY);
22            std::cout << "Ctrl-C event\n\n" << std::endl;
23            writelog(_T("Ctrl-C event"));
24            SetConsoleTextAttribute(hStdout, FOREGROUND_RED | FOREGROUND_GREEN |
25                                   FOREGROUND_BLUE);
26            return(TRUE);
27            // CTRL-CLOSE: confirm that the user wants to exit.
28        case CTRL_CLOSE_EVENT:
```

```

27     SetConsoleTextAttribute(hStdout, FOREGROUND_RED | BACKGROUND_BLUE |
        FOREGROUND_INTENSITY);
28     std::cout << "Ctrl-Close event\n\n" << std::endl;
29     writelog(_T("Ctrl-Close event"));
30     SetConsoleTextAttribute(hStdout, FOREGROUND_RED | FOREGROUND_GREEN |
        FOREGROUND_BLUE);
31     return(TRUE);
32     // Pass other signals to the next handler.
33 case CTRL_BREAK_EVENT:
34     SetConsoleTextAttribute(hStdout, FOREGROUND_RED | BACKGROUND_BLUE |
        FOREGROUND_INTENSITY);
35     std::cout << "Ctrl-Break event\n\n" << std::endl;
36     writelog(_T("Ctrl-Break event"));
37     SetConsoleTextAttribute(hStdout, FOREGROUND_RED | FOREGROUND_GREEN |
        FOREGROUND_BLUE);
38     return FALSE;
39 case CTRL_LOGOFF_EVENT:
40     SetConsoleTextAttribute(hStdout, FOREGROUND_RED | BACKGROUND_BLUE |
        FOREGROUND_INTENSITY);
41     std::cout << "Ctrl-Logoff event\n\n" << std::endl;
42     writelog(_T("Ctrl-Logoff event"));
43     SetConsoleTextAttribute(hStdout, FOREGROUND_RED | FOREGROUND_GREEN |
        FOREGROUND_BLUE);
44     return FALSE;
45 case CTRL_SHUTDOWN_EVENT:
46     SetConsoleTextAttribute(hStdout, FOREGROUND_RED | BACKGROUND_BLUE |
        FOREGROUND_INTENSITY);
47     std::cout << "Ctrl-Shutdown event\n\n" << std::endl;
48     writelog(_T("Ctrl-Shutdown event"));
49     SetConsoleTextAttribute(hStdout, FOREGROUND_RED | FOREGROUND_GREEN |
        FOREGROUND_BLUE);
50     return FALSE;
51 default:
52     return FALSE;
53 }
54 }
55 int _tmain(int argc, _TCHAR* argv[]) {
56     //Init log
57     initlog(argv[0]);
58
59     if (SetConsoleCtrlHandler((PHANDLER_ROUTINE)CtrlHandler, TRUE)) {
60         std::cout << "The Control Handler is installed." << std::endl
61             << "-- Now try pressing Ctrl+C or Ctrl+Break, or" << std::endl
62             << "    try logging off or closing the console..." << std::endl << std
                ::endl

```

```

63         << "...waiting in a loop for events..." << std::endl << std::endl;
64     while (1){}
65 }
66 else {
67     std::cout << "ERROR: Could not set control handler" << std::endl;
68     writelog(_T("ERROR: Could not set control handler"));
69     return EXIT_FAILURE;
70 }
71 closelog();
72 return EXIT_SUCCESS;
73 }

```

```

C:\Windows\system32\cmd.exe - Signals.exe
C:\Users\win7\Documents\Visual Studio 2013\Projects\InterProcessCommunication\De
bug>Signals.exe
The Control Handler is installed.
-- Now try pressing Ctrl+C or Ctrl+Break, or
   try logging off or closing the console...

<...waiting in a loop for events...>
Ctrl-C event

```

Рис. 15: Работа с сигналами в Windows



# Заключение

В данной работе были рассмотрены основные механизмы межпроцессорного взаимодействия, от самых простых, типа анонимных каналов, до самых сложных, таких как сокеты и порты завершения. Каждый механизм имеет свою нишу для использования.

1. Анонимные каналы – достаточно слабый инструмент. Представляют собой полудуплексное средство потоковой передачи байтов между родственными процессами. Они функционируют в пределах локальной вычислительной системы и хорошо подходят для перенаправления выходного потока одной программы на вход другой. Минусом является то, что использование этого средства в некоторых случаях приводит к невозможности использования стандартных механизмов ввода/вывода (т.к. они уже заняты каналом).
2. Именованные каналы – являясь объектами ядра ОС Windows, они позволяют организовать межпроцессный обмен не только в изолированной вычислительной системе, но и в локальной сети. Они обеспечивают дуплексную связь и позволяют использовать как потоковую модель, так и модель, ориентированную на сообщения. Обмен данными может быть синхронным и асинхронным.
3. Почтовые ящики – хотя этот механизм обеспечивают одностороннее взаимодействие процессов, любой процесс может выступать одновременно как в роли сервера, так и в роли клиента, вследствие чего возможно двустороннее межпроцессное взаимодействие (путем создания по крайней мере двух почтовых ящиков). Сообщения хранятся в почтовом ящике до тех пор, пока сервер их не прочтет.
4. Shared memory – очень быстрый и очень опасный механизм взаимодействия. Общая память должна быть защищена каким-то механизмом типа блокировки, в противном случае один процесс может испортить то, что перед этим записал другой.
5. Сокеты – представляет собой независимый от протокола интерфейс, который даёт возможность использовать преимущества базовых протоколов. Сокет Windows Sockets 2 представляет собой дескриптор, который может быть использован и как

дескриптор файла в стандартных файловых функциях ввода и вывода. Приложение, использующее сокеты Windows, может взаимодействовать с сокетами других ОС.

6. Порты завершения – обеспечивают эффективную обработку асинхронных запросов ввода/вывода на многопроцессорных системах за счёт использования очередей и пула потоков обработки.
7. Сигналы – самый слабый инструмент, которому сложно придумать практическое применение. Механизм сигналов в Windows значительно уступает аналогичному механизму из мира \*nix.

Наиболее интересным средством взаимодействия оказался сокет. Он не имеет больших отличий от классического сокета Беркли, что упрощает его изучение. Работа в асинхронном режиме (порты завершения) оказывает драматическое влияние на скорость работы системы, и должна применяться в высоко нагруженных системах.

# Литература

1. Душутина Е.В. Межпроцессные взаимодействия в операционных системах. Учебное пособие – СПб.: 2014 – 135 стр.
2. Джонсон М. Харт. Системное программирование в среде Windows – М.: Издательский дом "Вильямс 2005 – 587 стр.
3. Александр Шаргин. Программирование сокетов в Linux – <http://rsdn.ru/article/unix/sockets.xml>
4. Бердников Алексей: Использование портов завершения для клиент-серверных приложений. – <http://lexlunnyi.blogspot.ru/2013/08/blog-post.html>
5. MSDN: CreateIoCompletionPort function – <https://msdn.microsoft.com/en-us/aa363862.aspx>
6. MSDN: GetQueuedCompletionStatus function – <https://msdn.microsoft.com/aa364986.aspx>