

Отчет по расчетной работе № 2
по предмету «Системное программное обеспечение»

ОБРАБОТКА ИСКЛЮЧЕНИЙ В ОС WINDOWS

Работу выполнил студент гр. 53501/3 _____ Мартынов С. А.

Работу принял преподаватель _____ Душутина Е. В.

Оглавление

Постановка задачи	2
Анонимные каналы	4
Именованные каналы	9
Почтовые ящики	11
Shared memory	13
Сокеты	18
Порты завершения	20
Сигналы	21
Заключение	22

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.

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.

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).

Исходный код всех представленных листингов доступен по адресу
https://github.com/SemenMartynov/SPbPU_SystemProgramming.

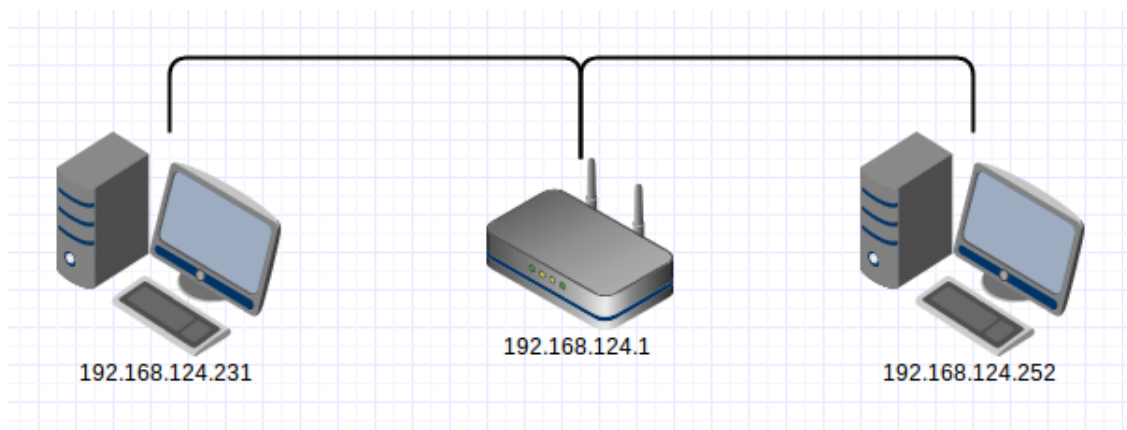


Рис. 1: Топология сети.

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:

```
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 заблокирована.

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

В листинге 1 Представлен сервер, а в листинге 2 клиент для работы с анонимными каналами Windows. В программе используется передача дескрипторов через наследование. Рисунок 2 показывает результат работы.

Листинг 1: Сервер анонимных каналов

```

1 #include <windows.h>
2 #include <stdio.h>
3 int main(int argc, char* argv[])
4 {
5     //дескрипторы канала для передачи от сервера клиенту
6     HANDLE hReadPipeFromServToClient, hWritePipeFromServToClient;
7     //дескрипторы канала для передачи от сервера клиенту
8     HANDLE hReadPipeFromClientToServ, hWritePipeFromClientToServ;
9
10    //чтобы сделать дескрипторы наследуемыми
11    //создаем канал для передачи от сервера клиенту, сразу делаем дескрипторы
        наследуемыми
12    SECURITY_ATTRIBUTES PipeSA = { sizeof(SEcurity_ATTRIBUTES), NULL, TRUE };
13    if (CreatePipe(&hReadPipeFromServToClient, &hWritePipeFromServToClient, &
        PipeSA, 0) == 0)
14    {
15        printf("impossible to create anonymous pipe from serv to client\n");
16        getchar();
17        return 1000;
18    }
19    //создаем канал для передачи от клиента серверу, сразу делаем дескрипторы
        наследуемыми
20    if (CreatePipe(&hReadPipeFromClientToServ, &hWritePipeFromClientToServ, &
        PipeSA, 0) == 0)
21    {
22        printf("impossible to create anonymous pipe from client to serv\n");
23        getchar();
24        return 1001;
25    }
26
27    PROCESS_INFORMATION processInfo_Client; // информация о процессе-клиенте
28    STARTUPINFO startupInfo_Client;
29    //структура, которая описывает внешний вид основного
30    //окна и содержит дескрипторы стандартных устройств нового процесса, испол
        ьзуем для установки
31    //процесс-клиент будет иметь те же параметры запуска, что и сервер, за иск
        лчением
32    //дескрипторов ввода, вывода и ошибок
33    GetStartupInfo(&startupInfo_Client);
34    startupInfo_Client.hStdInput = hReadPipeFromServToClient;
35    //устанавливаем поток ввода
36    startupInfo_Client.hStdOutput = hWritePipeFromClientToServ;
37    //установим поток вывода
38    startupInfo_Client.hStdError = GetStdHandle(STD_ERROR_HANDLE);

```

```

39 //установим поток ошибок
40 startupInfo_Client.dwFlags = STARTF_USESTDHANDLES; //устанавливаем наследо
    вание
41 //создаем процесс клиента
42 CreateProcess(NULL, L"..\\..\\anonymousPipesSonProc\\Release\\
    anonymousPipesSonProc.exe", NULL,
43     NULL, TRUE, CREATE_NEW_CONSOLE, NULL, NULL,
44     &startupInfo_Client, &processInfo_Client);
45 CloseHandle(processInfo_Client.hThread); //закрываем дескрипторы созданног
    о процесса и его
46 //потока
47 CloseHandle(processInfo_Client.hProcess);
48 //закрываем ненужные дескрипторы каналов, которые не использует сервер
49 CloseHandle(hReadPipeFromServToClient);
50 CloseHandle(hWritePipeFromClientToServ);
51 #define BUF_SIZE 100
52 //размер буфера для сообщений
53 BYTE buf[BUF_SIZE];
54 //буфер приема/передачи
55 DWORD readbytes, writebytes; //число прочитанных/переданных байт
56 for (int i = 0; i < 10; i++)
57 {
58     //читаем данные из канала от клиента
59     if (!ReadFile(hReadPipeFromClientToServ, buf, BUF_SIZE, &readbytes, NULL
        ))
60     {
61         printf("impossible to use readfile\n GetLastError= %d\n", GetLastError
            ());
62         getchar();
63         return 10000;
64     }
65     printf("get from client: \"%s\"\n", buf);
66     if (!WriteFile(hWritePipeFromServToClient, buf, readbytes, &writebytes,
        NULL))
67     {
68         printf("impossible to use writefile\n GetLastError= %d\n",
            GetLastError());
69         getchar();
70         return 10001;
71     }
72     //пишем данные в канал клиенту
73 }
74 //закрываем HANDLE каналов
75 CloseHandle(hReadPipeFromClientToServ);
76 CloseHandle(hWritePipeFromServToClient);

```

```

77 printf("server ended work\n Press any key");
78 getchar();
79 return 0;
80 }

```

Листинг 2: Клиент анонимных каналов

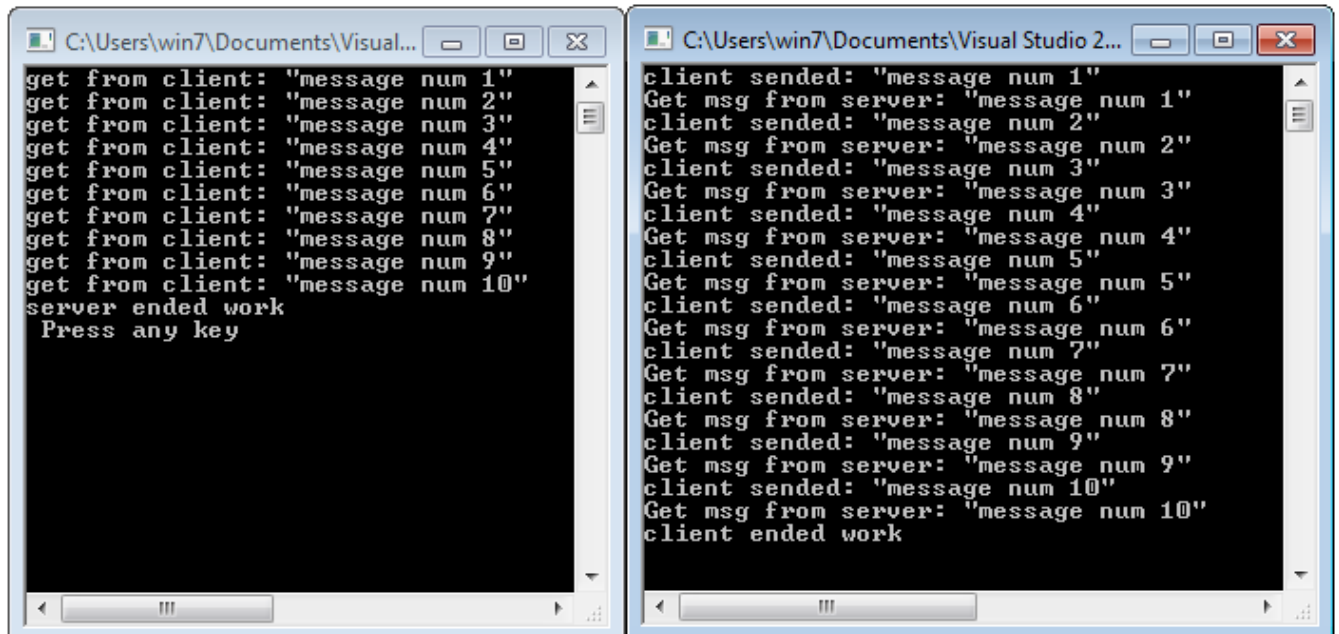
```

1 #include <stdio.h>
2 #include <Windows.h>
3 int main(int argc, char* argv[])
4 {
5     //строка для передачи
6     char strtosend[100];
7     //буфер приема
8     char getbuf[100];
9     //число переданных и принятых байт
10    DWORD bytessended, bytesreaded;
11
12    for (int i = 0; i < 10; i++)
13    {
14        //формирование строки для передачи
15        bytessended = sprintf_s(strtosend, "message num %d", i + 1);
16        strtosend[bytessended] = 0;
17        fprintf(stderr, "client sended: \"%s\"\n", strtosend);
18        if (!WriteFile(GetStdHandle(STD_OUTPUT_HANDLE), strtosend, bytessended +
19            1, &bytesreaded, NULL)
20        {
21            fprintf(stderr, "Error with writeFile\n Wait 5 sec GetLastError=%d\n",
22                GetLastError());
23            Sleep(5000);
24            return 1000;
25        }
26        if (!ReadFile(GetStdHandle(STD_INPUT_HANDLE), getbuf, 100, &bytesreaded,
27            NULL))
28            //проем ответа от сервера
29        {
30            fprintf(stderr, "Error with readFile\n Wait 5 sec GetLastError=%d\n",
31                GetLastError());
32            Sleep(5000);
33            return 1001;
34        }
35        fprintf(stderr, "Get msg from server: \"%s\"\n", getbuf);

```



```
33 }  
34 fprintf(stderr, "client ended work\n Wait 5 sec");  
35 Sleep(5000);  
36 return 0;  
37 }
```



```
C:\Users\win7\Documents\Visual...  
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"  
server ended work  
Press any key  
  
C:\Users\win7\Documents\Visual Studio 2...  
client sended: "message num 1"  
Get msg from server: "message num 1"  
client sended: "message num 2"  
Get msg from server: "message num 2"  
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"  
client ended work
```

Рис. 2: Работа с анонимными каналами.

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:

- 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 двунаправленным, ориентированным на обмен сообщениями или доступным для нескольких

клиентских процессов, следует применять именованные каналы. Попытки реализации последующих примеров с использованием анонимных каналов натолкнулись бы на значительные трудности.

Листинг 3 содержит код, реализующий сервер именованного канала. В листинге 4 - код клиента. Рисунок 3 - результат работы одного сервера с несколькими клиентами. Сервер не содержит разделяемых ресурсов, следовательно в средствах синхронизации потоков необходимости нет.

Листинг 3: Клиент именованного каналов

```
1 //http://msdn.microsoft.com/ru-ru/windows/desktop/aa365785%28v=vs.85%29
2
3 #include <windows.h>
4 #include <stdio.h>
5 #include <conio.h>
6 #include <tchar.h>
7 #include <strsafe.h>
8
9 #define BUFSIZE 512
10
11 int _tmain(int argc, TCHAR *argv[])
12 {
13     _tprintf(TEXT("Client is started!\n\n"));
14
15     HANDLE hPipe = INVALID_HANDLE_VALUE; // Идентификатор канала
16     LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\$$$MyPipe$$"); // Имя создаваемого
        канала Pipe
17     TCHAR chBuf[BUFSIZE]; // Буфер для передачи данных через канал
18     DWORD readbytes, writebytes; // Число байт прочитанных и переданных
19
20     _tprintf(TEXT("Try to use WaitNamedPipe...\n"));
21     // Пытаемся открыть именованный канал, если надо - ожидаем его освобождени
        я
22     while (1)
23     {
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 if (GetLastError() != ERROR_PIPE_BUSY)
41 {
42     _tprintf(TEXT("Could not open pipe. GLE=%d\n"), GetLastError());
43     _getch();
44     return -1;
45 }
46
47 // Если все каналы заняты, ждём 20 секунд
48 if (!WaitNamedPipe(lpszPipename, 20000))
49 {
50     _tprintf(TEXT("Could not open pipe: 20 second wait timed out. "));
51     _getch();
52     return -1;
53 }
54 }
55
56 // Выводим сообщение о создании канала
57 _tprintf(TEXT("Successfully connected!\n\nInput message...\n"));
58 // Цикл обмена данными с серверным процессом
59 while (1)
60 {
61     // Выводим приглашение для ввода команды
62     _tprintf(TEXT("cmd>"));
63     // Вводим текстовую строку
64     _fgetts(chBuf, BUFSIZE, stdin);
65     // Передаем введенную строку серверному процессу в качестве команды
66     if (!WriteFile(hPipe, chBuf, (lstrlen(chBuf) + 1)*sizeof(TCHAR), &
        writebytes, NULL))
67     {
68         _tprintf(TEXT("connection refused\n"));
69         break;
70     }
71     // Получаем эту же команду обратно от сервера
72     if (ReadFile(hPipe, chBuf, BUFSIZE*sizeof(TCHAR), &readbytes, NULL))
73         _tprintf(TEXT("Received from server: %s\n"), chBuf);
74     // Если произошла ошибка, выводим ее код и завершаем работу приложения
75     else {
76         _tprintf(TEXT("ReadFile: Error %ld\n"), GetLastError());
77         _getch();

```

```

78     break;
79 }
80 // В ответ на команду "exit" завершаем цикл обмена данными с серверным п
    роцессом
81 if (!_tcsncmp(chBuf, L"exit", 4))
82     break;
83 }
84
85 // Закрываем идентификатор канала
86 CloseHandle(hPipe);
87
88 _tprintf(TEXT("Press ENTER to terminate connection and exit\n"));
89 _getch();
90 return 0;
91 }

```

Листинг 4: Клиент именованного каналов

```

1 //http://msdn.microsoft.com/ru-ru/windows/desktop/aa365785%28v=vs.85%29
2
3 #include <windows.h>
4 #include <stdio.h>
5 #include <conio.h>
6 #include <tchar.h>
7 #include <strsafe.h>
8
9 #define BUFSIZE 512
10
11 int _tmain(int argc, TCHAR *argv[])
12 {
13     _tprintf(TEXT("Client is started!\n\n"));
14
15     HANDLE hPipe = INVALID_HANDLE_VALUE; // Идентификатор канала
16     LPTSTR lpszPipename = TEXT("\\\\.\\pipe\\\\"$MyPipe$$$"); // Имя создаваемого
        канала Pipe
17     TCHAR chBuf[BUFSIZE]; // Буфер для передачи данных через канал
18     DWORD readbytes, writebytes; // Число байт прочитанных и переданных
19
20     _tprintf(TEXT("Try to use WaitNamedPipe...\n"));
21     // Пытаемся открыть именованный канал, если надо - ожидаем его освобождени
        я
22     while (1)
23     {

```

```

24 // Создаем канал с процессом-сервером:
25 hPipe = CreateFile(
26     lpzPipeName, // 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 if (GetLastError() != ERROR_PIPE_BUSY)
41 {
42     _tprintf(TEXT("Could not open pipe. GLE=%d\n"), GetLastError());
43     _getch();
44     return -1;
45 }
46
47 // Если все каналы заняты, ждём 20 секунд
48 if (!WaitNamedPipe(lpzPipeName, 20000))
49 {
50     _tprintf(TEXT("Could not open pipe: 20 second wait timed out.));
51     _getch();
52     return -1;
53 }
54 }
55
56 // Выводим сообщение о создании канала
57 _tprintf(TEXT("Successfully connected!\n\nInput message...\n"));
58 // Цикл обмена данными с серверным процессом
59 while (1)
60 {
61     // Выводим приглашение для ввода команды
62     _tprintf(TEXT("cmd>"));
63     // Вводим текстовую строку
64     _fgetts(chBuf, BUFSIZE, stdin);
65     // Передаем введенную строку серверному процессу в качестве команды
66     if (!WriteFile(hPipe, chBuf, (lstrlen(chBuf) + 1)*sizeof(TCHAR), &
67         writebytes, NULL))

```

```

68     _tprintf(TEXT("connection refused\n"));
69     break;
70 }
71 // Получаем эту же команду обратно от сервера
72 if (ReadFile(hPipe, chBuf, BUFSIZE*sizeof(TCHAR), &readbytes, NULL))
73     _tprintf(TEXT("Received from server: %s\n"), chBuf);
74 // Если произошла ошибка, выводим ее код и завершаем работу приложения
75 else {
76     _tprintf(TEXT("ReadFile: Error %ld\n"), GetLastError());
77     _getch();
78     break;
79 }
80 // В ответ на команду "exit" завершаем цикл обмена данными с серверным п
    процессом
81 if (!_tcsncmp(chBuf, L"exit", 4))
82     break;
83 }
84
85 // Закрываем идентификатор канала
86 CloseHandle(hPipe);
87
88 _tprintf(TEXT("Press ENTER to terminate connection and exit\n"));
89 _getch();
90 return 0;
91 }

```

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. В листинге 5 содержится отрывок кода клиента с самыми значительными изменениями. Результаты работы внешне не отличаются от работы в локальном варианте.

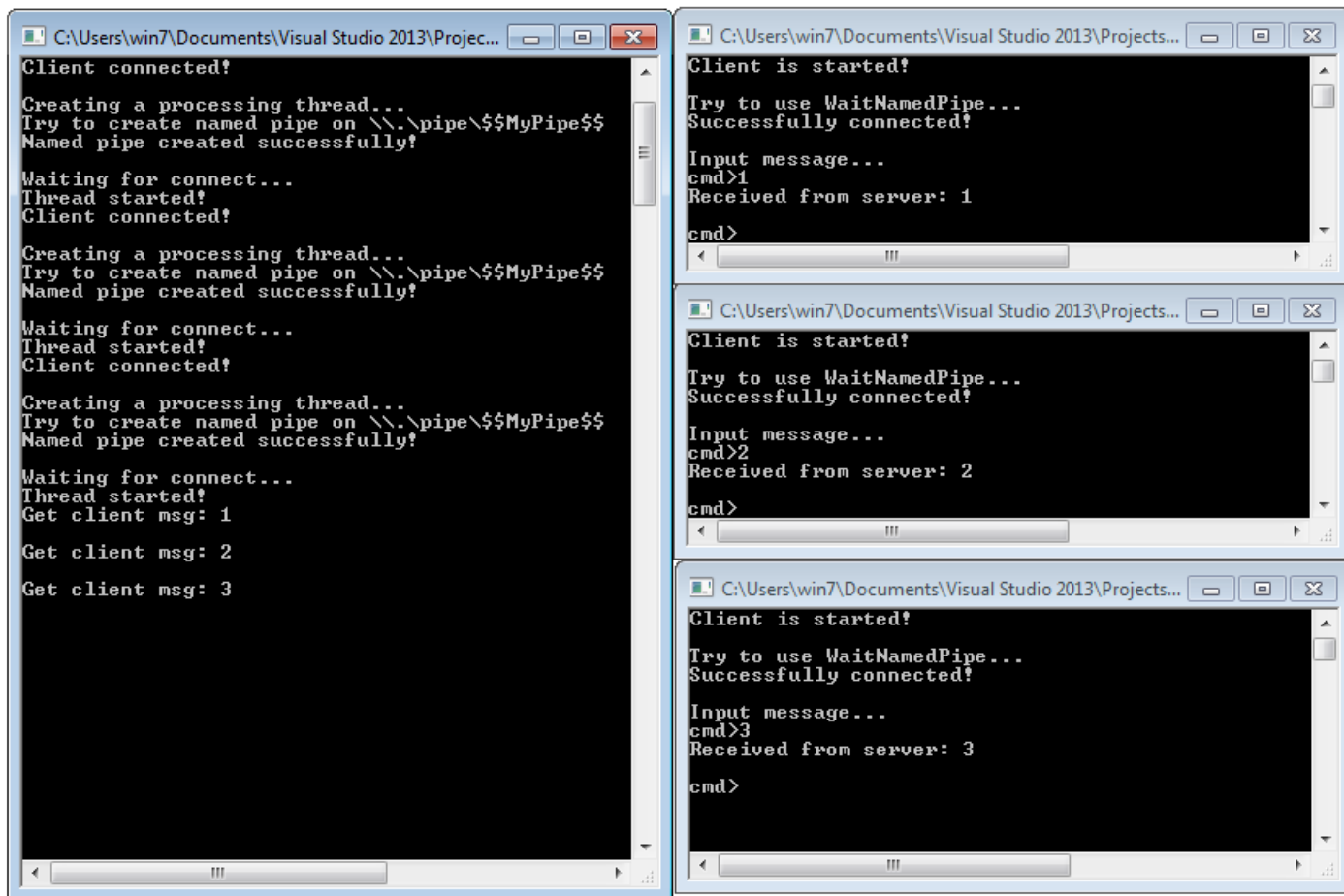


Рис. 3: Работа с нескольких клиентов с одним сервером по именованному каналу.

Почтовые ящики

Как и именованные каналы, почтовые ящики (mailslots) Windows снабжаются именами, которые могут быть использованы для обеспечения взаимодействия между независимыми каналами. 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.

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.

Листинг 6 и 7 демонстрируют реализацию приложения, иллюстрирующую обмен информацией почтовыми слотами. В процессе экспериментов было протестировано локальное, сетевое взаимодействие. Для широковещательной передач сообщений, адрес заменялся символом звездочки (*).

Листинг 6. Реализация серверной части почтового ящика.

Листинг 7. Реализация клиентской части почтового ящика.

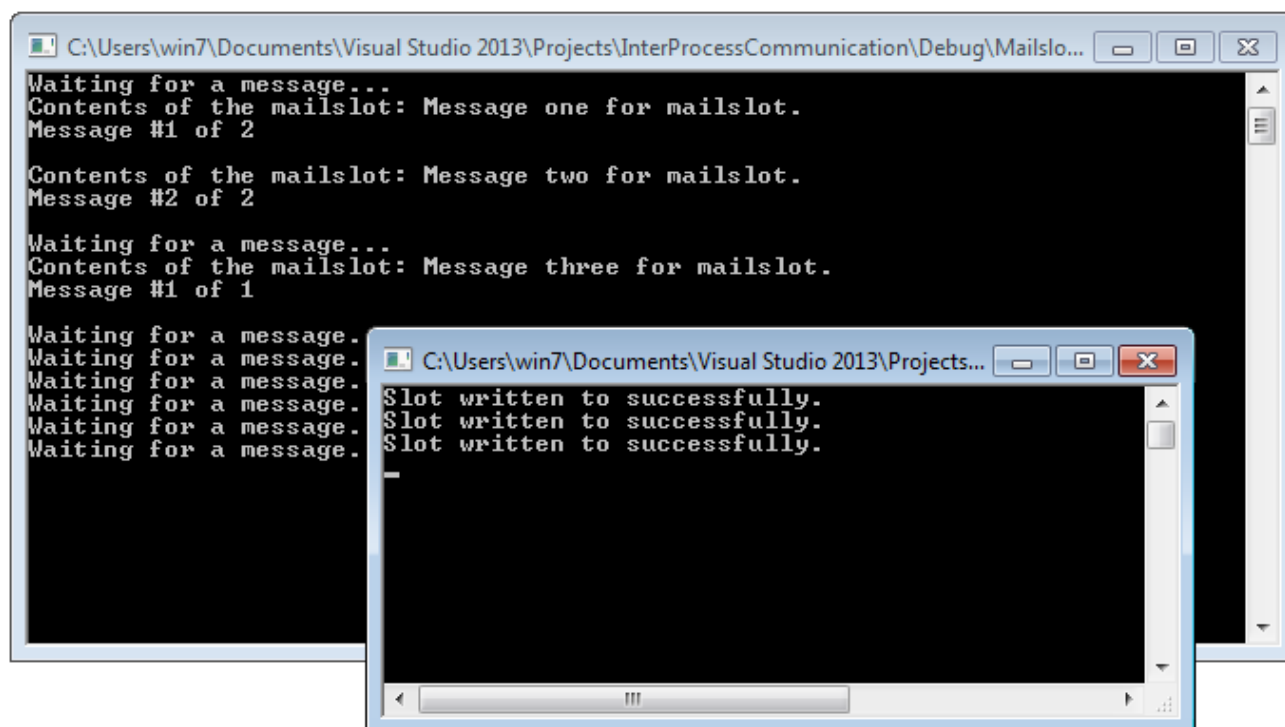


Рис. 4: Работа почтовыми ящиками.

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. Получаемая область памяти будет общей между процессами. Работая с этой памятью, потоки обязательно должны согласовывать свои действия с помощью объектов синхронизации.

В листинге 8 и 9 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 5 показывает результат такого взаимодействия.

Листинг 8. Программа, генерирующая случайные числа в разделяемую память.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define BUF_SIZE 256
TCHAR szName[] = TEXT("MyFileMappingObject");
TCHAR szMsg[] = TEXT("Message from first process");
HANDLE mutex;
void main()
{
    HANDLE hMapFile;
    LPCTSTR pBuf;
    mutex = CreateMutex(NULL, false, TEXT("SyncMutex"));
    // create a memory, wicth two proccess will be working
    hMapFile = CreateFileMapping(
        // использование файла подкачки
        INVALID_HANDLE_VALUE,
```

```

    // защита по умолчанию
    NULL,
    // доступ к чтению/записи
    PAGE_READWRITE,
    // макс. размер объекта
    0,
    // размер буфера
    BUF_SIZE,
    // имя отраженного в памяти объекта
    szName);

if (hMapFile == NULL || hMapFile == INVALID_HANDLE_VALUE)
{
    printf("Не может создать отраженный в памяти объект (%d).\n",
        GetLastError());
    return;
}
pBuf = (LPTSTR)MapViewOfFile(
    //дескриптор проецируемого в памяти объекта
    hMapFile,
    // разрешение чтения/записи(режим доступа)
    FILE_MAP_ALL_ACCESS,
    //Старшее слово смещения файла, где начинается отображение
    0,
    //Младшее слово смещения файла, где начинается отображение
    0,
    //Число отображаемых байтов файла
    BUF_SIZE);

if (pBuf == NULL)
{
    printf("Представление проецированного файла невозможно (%d).\n",
        GetLastError());
    return;
}

int i = 0;

```

```

while (true)
{
    i = rand();
    itoa(i, (char *)szMsg, 10);
    WaitForSingleObject(mutex, INFINITE);
    CopyMemory((PVOID)pBuf, szMsg, sizeof(szMsg));
    printf("write message: %s\n", (char *)pBuf);
    //необходимо только для отладки - для удобства представления и анализа
    Sleep(1000);
    //результатов
    ReleaseMutex(mutex);
}
// освобождение памяти и закрытие описателя handle
UnmapViewOfFile(pBuf);
CloseHandle(hMapFile);
CloseHandle(mutex);
}

```

Листинг 9. Программа, читающая случайные числа из разделяемой памяти.

```

#include <windows.h>
#include <stdio.h>
#include <conio.h>
#define BUF_SIZE 256
#define TIME 15
// number of reading operation in this process
TCHAR szName[] = TEXT("MyFileMappingObject");
HANDLE mutex;
void main()
{
    HANDLE hMapFile;
    LPCTSTR pBuf;
    mutex = OpenMutex(
        // request full access
        MUTEX_ALL_ACCESS,

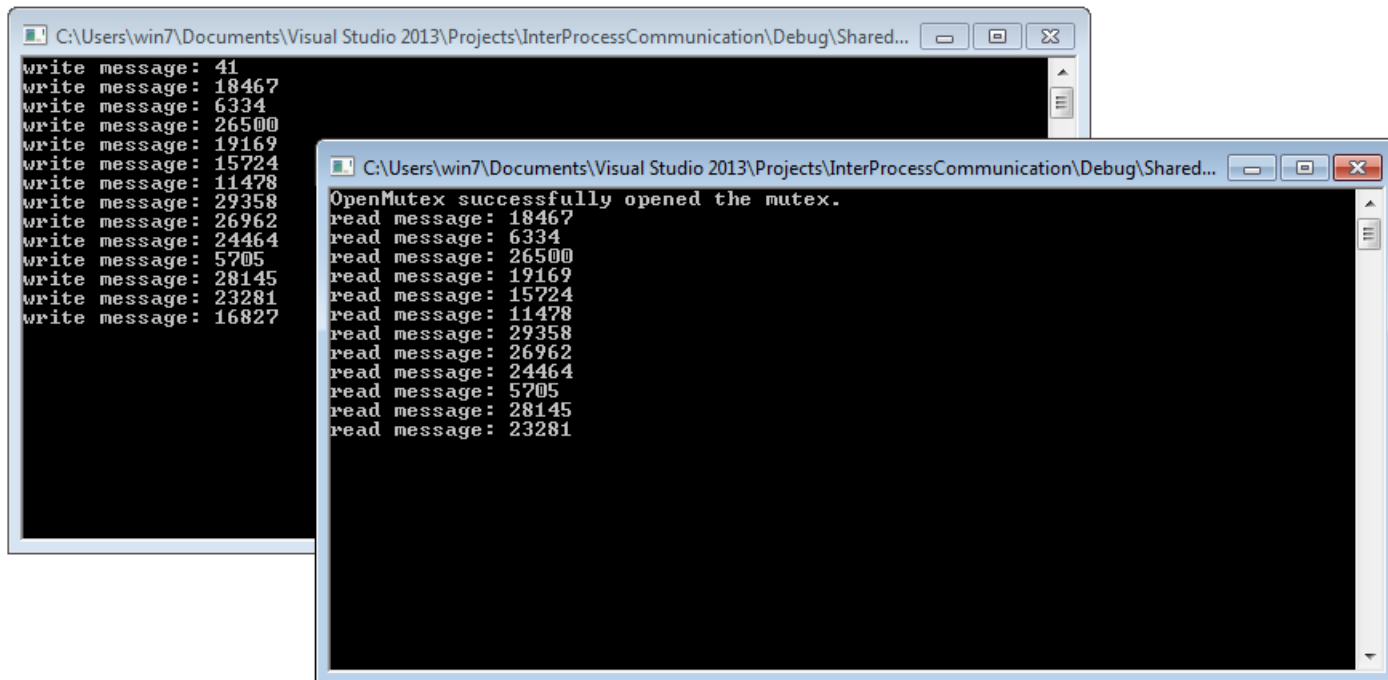
```

```

        // handle not inheritable
        FALSE,
        // object name
        TEXT("SyncMutex"));
if (mutex == NULL)
    printf("OpenMutex error: %d\n", GetLastError());
else printf("OpenMutex successfully opened the mutex.\n");
hMapFile = OpenFileMapping(
    // доступ к чтению/записи
    FILE_MAP_ALL_ACCESS,
    // имя не наследуется
    FALSE,
    // имя "проецируемого " объекта
    szName);
if (hMapFile == NULL)
{
    printf("Невозможно открыть объект проекция файла (%d).\n", GetLastError());
    return;
}
pBuf = (LPTSTR)MapViewOfFile(hMapFile,
    // дескриптор "проецируемого" объекта
    FILE_MAP_ALL_ACCESS, // разрешение чтения/записи
    0,
    0,
    BUF_SIZE);
if (pBuf == NULL)
{
    printf("Представление проецированного файла (%d) невозможно .\n",
        GetLastError());
    return;
}
for (int i = 0; i < TIME; i++)
{
    WaitForSingleObject(mutex, INFINITE);
    printf("read message: %s\n", (char *)pBuf);
    ReleaseMutex(mutex);
}

```

```
UnmapViewOfFile(pBuf);  
CloseHandle(hMapFile);  
}
```



The image shows two overlapping console windows from Visual Studio. The background window displays a list of memory addresses written to a shared memory buffer. The foreground window displays the same addresses read from the shared memory buffer, along with a confirmation message for the mutex operation.

```
C:\Users\win7\Documents\Visual Studio 2013\Projects\InterProcessCommunication\Debug\Shared...  
write message: 41  
write message: 18467  
write message: 6334  
write message: 26500  
write message: 19169  
write message: 15724  
write message: 11478  
write message: 29358  
write message: 26962  
write message: 24464  
write message: 5705  
write message: 28145  
write message: 23281  
write message: 16827  
  
C:\Users\win7\Documents\Visual Studio 2013\Projects\InterProcessCommunication\Debug\Shared...  
OpenMutex successfully opened the mutex.  
read message: 18467  
read message: 6334  
read message: 26500  
read message: 19169  
read message: 15724  
read message: 11478  
read message: 29358  
read message: 26962  
read message: 24464  
read message: 5705  
read message: 28145  
read message: 23281
```

Рис. 5: Работа с разделяемой памятью.

Сокеты

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

- 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 (глава 14), 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 10 и 11. Рисунок 6 показывает работу программ, из этих листингов.

Листинг 10. Сервер для работы с Win-сокетами.

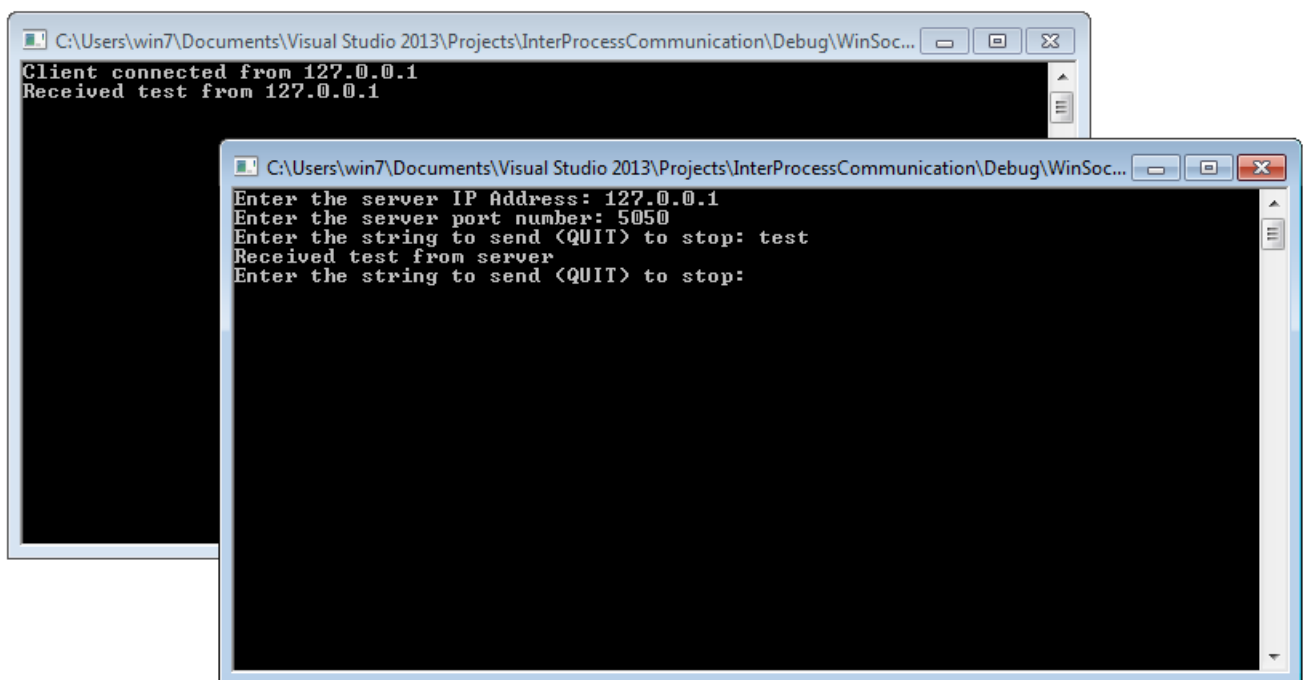


Рис. 6: Работа с сокетами.

Порты завершения

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

- 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 12 показывает реализацию порта завершения. Для работы с ним использовался клиент из предыдущего примера. Визуальной разницы нет, но она драматически ощущается при большом количестве клиентов.

Листинг 12. Порт завершения.

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 13 по сигналу меняется цвет консоли.

Листинг 13. Сигналы 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.

Отдельно выделяются только сигналы, которые значительно уступают подобному механизму из мира linux.

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.