Лабораторная работа № 4

Реализация обмена данными между процессами

Цель работы: Изучение набора средств коммуникации процессов в Windows NT. Получение практических навыков по использованию Win32 API для программирования механизмов межпроцессного взамиодействия

Почтовые ящики (mailslot)

Почтовые ящики обеспечивают только однонаправленные соединения. Каждый процесс, который создает почтовый ящик, является «сервером почтовых ящиков» (mailslot server). Другие процессы, называемые «клиентами почтовых ящиков» (mailslot clients), посылают сообщения серверу, записывая их в почтовый ящик. Входящие сообщения всегда дописываются в почтовый ящик и сохраняются до тех пор, пока сервер их не прочтет. Каждый процесс может одновременно быть и сервером и клиентом почтовых ящиков, создавая, таким образом, двунаправленные коммуникации между процессами.

Клиент может посылать сообщения на почтовый ящик, расположенный на том же компьютере, на компьютере в сети, или на все почтовые ящики с одним именем всем компьютерам выбранного домена. При этом широковещательное сообщение, транслируемое по домену, не может быть более 400 байт. В остальных случаях размер сообщения ограничивается только при создании почтового ящика сервером.

Почтовые ящики предлагают легкий путь для обмена короткими сообщениями, позволяя при этом вести передачу и по локальной сети, в том числе и по всему домену.

Mailslot является псевдофайлом находящимся в памяти и вы должны использовать стандартные функции для работы с файлами, чтобы получить доступ к нему. Данные в почтовом ящике могут быть в любой форме – их интерпретацией занимается прикладная программа, но их общий объем не должен превышать 64 Кб. Однако, в отличии от дисковых файлов, mailslot’ы являются временными — когда все дескрипторы почтового ящика закрыты, он и все его данные удаляются. Заметим, что все почтовые ящики являются локальными по отношению к создавшему их процессу; процесс не может создать удаленный mailslot.

Сообщения меньше, чем 425 байт, передаются с использованием дейтаграмм. Сообщения больше чем 426 байт используют передачу с установлением логического соединения на основе SMB-сеансов. Передачи с установлением соединения допускают только индивидуальную передачу от одного клиента к одному серверу. Следовательно, вы теряете возможность широковещательной трансляции сообщений от одного клиента ко многим серверам. Учтите, что Windows не поддерживает сообщения размером в 425 или 426 байт.

Когда процесс создает почтовый ящик, имя последнего должно иметь следующую форму:

\\.**mailslot**\[*path*]*name*

Например:

\\.**mailslot**\taxes\bobs_comments

\\.**mailslot**\taxes\petes_comments

\\.**mailslot**\taxes\sues_comments

Если вы хотите отправить сообщение в почтовый ящик на удаленный компьютер, то воспользуйтесь NETBIOS-именем:

\\ComputerName**mailslot**\[path]name

Если же ваша цель передать сообщение всем mailslot'ам с указанным именем внутри домена, вам понадобится NETBIOS-имя домена:

\\DomainName**mailslot**\[path]name

Или для главного домена операционной системы (домен в котором находится рабочая станция):

***mailslot**\[path]name

Клиенты и серверы, использующие почтовые ящики, при работе с ними должны пользоваться следующими функциями:

Функции серверов почтовых ящиков

Функция	**Описание**
CreateMailslot	Создает почтовый ящик и возвращает его дескриптор.
GetMailslotInfo	Извлекает максимальный размер сообщения, размер почтового ящика, размер следующего сообщения в ящике, количество сообщений и время ожидания сообщения при выполнении операции чтения.
SetMailslotInfo	Изменение таймаута при чтении из почтового ящика.

Функция	**Описание**
DuplicateHandle	Дублирование дескриптора почтового ящика.
ReadFile, ReadFileEx	Считывание сообщений из почтового ящика.
GetFileTime	Получение даты и времени создания mailslot'а.
SetFileTime	Установка даты и времени создания mailslot'а.
GetHandleInformation	Получение свойств дескриптора почтового ящика.

SetHandleInformation	Установка свойств дескриптора почтового ящика.

Функции клиентов почтовых ящиков

Функция	**Описание**
CloseHandle	Закрывает дескриптор почтового ящика для клиентского процесса.
CreateFile	Создает дескриптор почтового ящика для клиентского процесса.
DuplicateHandle	Дублирование дескриптора почтового ящика.
WriteFile, WriteFileEx	Запись сообщений в почтовый ящик.

Рассмотрим последовательно все операции, необходимые для корректной работы с почтовыми ящиками.

1. Создание почтового ящика.

Эта операция выполняется процессом сервера с использованием функции CreateMailslot:

```
HANDLE CreateMailslot(
 LPCTSTR lpName,                    // имя почтового ящика
 DWORD nMaxMessageSize,             // максимальный размер сообщения
 DWORD lReadTimeout,                // таймаут операции чтения
 LPSECURITY_ATTRIBUTES lpSecurityAttributes // опции наследования и
);                                  // безопасности
```

```
BOOL FAR PASCAL Makeslot(HWND hwnd, HDC hdc)
{
  LPSTR lpszSlotName = "\\\\.\\mailslot\\sample_mailslot";

  // Дескриптор почтового ящика "hSlot1" определен глобально.

  hSlot1 = CreateMailslot(lpszSlotName,
    0,                          // без максимального размера сообщения
    MAILSLOT_WAIT_FOREVER,      // без таймаута при операциях
    (LPSECURITY_ATTRIBUTES) NULL); // без атрибутов безопасности

  if (hSlot1 == INVALID_HANDLE_VALUE)
  {
    ErrorHandler(hwnd, "CreateMailslot"); // обработка ошибки
```

```
    return FALSE;
  }

  TextOut(hdc, 10, 10, "CreateMailslot вызвана удачно.", 26);
  return TRUE;
}
```

2. Запись сообщений в почтовый ящик.

Запись в mailslot производится аналогично записи в стандартный дисковый файл. Следующий код иллюстрирует как с помощью функций CreateFile, WriteFile и CloseHandle можно поместить сообщение в почтовый ящик.

```
LPSTR lpszMessage = "Сообщение для sample_mailslot в текущем домене.";
BOOL fResult;
HANDLE hFile;
DWORD cbWritten;

// С помощью функции CreateFile клиент открывает mailslot для записи сообщений
hFile = CreateFile("\\\\*\\mailslot\\sample_mailslot",
  GENERIC_WRITE,
  FILE_SHARE_READ,              // Требуется для записи в mailslot
  (LPSECURITY_ATTRIBUTES) NULL,
  OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL,
  (HANDLE) NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
  ErrorHandler(hwnd, "Ошибка открытия почтового ящика");
  return FALSE;
}

// Запись сообщения в почтовый ящик
fResult = WriteFile(hFile,
  lpszMessage,
  (DWORD) lstrlen(lpszMessage) + 1,  // включая признак конца строки
```

```
    &cbWritten,
    (LPOVERLAPPED) NULL);

if (!fResult)
{
    ErrorHandler(hwnd, "Ошибка при записи сообщения");
    return FALSE;
}

TextOut(hdc, 10, 10, "Сообщение отправлено успешно.", 21);

fResult = CloseHandle(hFile);

if (!fResult)
{
    ErrorHandler(hwnd, "Ошибка при закрытии дескриптора");
    return FALSE;
}

TextOut(hdc, 10, 30, "Дескриптор закрыт успешно.", 23);
return TRUE;
```

3. Чтение сообщений из почтового ящика.

Создавший почтовый ящик процесс получает право считывания сообщений из него используя дескриптор mailslot'а в вызове функции ReadFile. Следующий пример использует функцию GetMailslotInfo чтобы определить сколько сообщений находится в почтовом ящике. Если есть непрочитанные сообщения, то они отображаются в окне сообщения вместе с количеством оставшихся сообщений.

Почтовый ящик существует до тех пор, пока не вызвана функция CloseHandle на сервере или пока существует сам процесс сервера. В обоих случаях все непрочитанные сообщения удаляются из почтового ящика, уничтожаются все клиентские дескрипторы и mailslot удаляется из памяти.

Функция считывает параметры почтового ящика:

```
BOOL GetMailslotInfo(
  HANDLE hMailslot,          // дескриптор почтового ящика
```

```
  LPDWORD lpMaxMessageSize,  // максимальный размер сообщения
  LPDWORD lpNextSize,        // размер следующего непрочитанного сообщения
  LPDWORD lpMessageCount,    // количество сообщений
  LPDWORD lpReadTimeout      // таймаут операции чтения
);
```

Функция устанавливает таймаут операции чтения:

```
BOOL SetMailslotInfo(
  HANDLE hMailslot,          // дескриптор почтового ящика
  DWORD lReadTimeout         // новый таймаут операции чтения
);
```

```
BOOL WINAPI Readslot(HWND hwnd, HDC hdc)
{
   DWORD cbMessage, cMessage, cbRead;
   BOOL fResult;
   LPSTR lpszBuffer;
   CHAR achID[80];
   DWORD cAllMessages;
   HANDLE hEvent;
   OVERLAPPED ov;

   cbMessage = cMessage = cbRead = 0;

   hEvent = CreateEvent(NULL, FALSE, FALSE, "ExampleSlot");
   ov.Offset = 0;
   ov.OffsetHigh = 0;
   ov.hEvent = hEvent;

   // Дескриптор почтового ящика "hSlot1" определен глобально.

   fResult = GetMailslotInfo(hSlot1, // дескриптор mailslot'a
     (LPDWORD) NULL,              // без ограничения размера сообщения
     &cbMessage,                  // размер следующего сообщения
     &cMessage,                   // количество сообщений в ящике
```

```
    (LPDWORD) NULL);            // без таймаута чтения

if (!fResult)
{
    ErrorHandler(hwnd, "Ошибка при получении информации о почтовом ящике");
    return FALSE;
}

if (cbMessage == MAILSLOT_NO_MESSAGE)
{
    TextOut(hdc, 10, 10, "Нет непрочитанных сообщений.", 20);
    return TRUE;
}

cAllMessages = cMessage;

while (cMessage != 0)  // Считываем все сообщения
{
    // Создаем строку с номером сообщения.

    wsprintf((LPSTR) achID,
        "\nMessage #%d of %d\n", cAllMessages - cMessage + 1,
        cAllMessages);

    // Выделяем память для сообщения.

    lpszBuffer = (LPSTR) GlobalAlloc(GPTR,
        lstrlen((LPSTR) achID) + cbMessage);

    lpszBuffer[0] = '\0';

    // Считываем сообщение из почтового ящика
    fResult = ReadFile(hSlot1,
        lpszBuffer,
        cbMessage,
```

```
    &cbRead,
    &ov);

  if (!fResult)
  {
    ErrorHandler(hwnd, "Ошибка чтения сообщения");
    GlobalFree((HGLOBAL) lpszBuffer);
    return FALSE;
  }

  // Формируем строку с номером и текстом сообщения.

  lstrcat(lpszBuffer, (LPSTR) achID);

  // Выводим сообщение на экран.

  MessageBox(hwnd,
    lpszBuffer,
    "Содержимое почтового ящика",
    MB_OK);

  GlobalFree((HGLOBAL) lpszBuffer);

  fResult = GetMailslotInfo(hSlot1, // дексриптор почтового ящика
    (LPDWORD) NULL,              // размер сообщения не ограничен
    &cbMessage,                  // размер следующего сообщения
    &cMessage,                   // количество сообщения
    (LPDWORD) NULL);             // без таймаута чтения

  if (!fResult)
  {
    ErrorHandler(hwnd, "Ошибка при получении информации о mailslot'е");
    return FALSE;
  }
}
```

```
    return TRUE;
}
```

Каналы (pipe)

Существует два способа организовать двунаправленное соединение с помощью каналов: безымянные и именованные каналы.

Безымянные (или анонимные) каналы позволяют связанным процессам передавать информацию друг другу. Обычно, безымянные каналы используются для перенаправления стандартного ввода/вывода дочернего процесса так, чтобы он мог обмениваться данными с родительским процессом. Чтобы производить обмен данными в обоих направлениях, вы должны создать два безымянных канала. Родительский процесс записывает данные в первый канал, используя его дескриптор записи, в то время как дочерний процесс считывает данные из канала, используя дескриптор чтения. Аналогично, дочерний процесс записывает данные во второй канал и родительский процесс считывает из него данные. Безымянные каналы не могут быть использованы для передачи данных по сети и для обмена между несвязанными процессами.

Именованные каналы используются для передачи данных между независимыми процессами или между процессами, работающими на разных компьютерах. Обычно, процесс сервера именованных каналов создает именованный канал с известным именем или с именем, которое будет передано клиентам. Процесс клиента именованных каналов, зная имя созданного канала, открывает его на своей стороне с учетом ограничений, указанных процессом сервера. После этого между сервером и клиентом создается соединение, по которому может производиться обмен данными в обоих направлениях. В дальнейшем наибольший интерес для нас будут представлять именованные каналы.

При создании и получении доступа к существующему каналу необходимо придерживаться следующего стандарта имен каналов:

.**pipe***pipename*

Если канал находится на удаленном компьютере, то вам потребуется NETBIOS-имя компьютера:

*ComputerName***pipe***pipename*

Клиентам и серверам для работы с каналами допускается использовать функции из следующего списка:

Функция	**Описание**
CallNamedPipe	Выполняет подключение к каналу, записывает в канал сообщение, считывает из канала сообщение и затем закрывает канал.
ConnectNamedPipe	Позволяет серверу именованных каналов ожидать подключения одного или нескольких клиентских процессов к экземпляру именованного канала.
CreateNamedPipe	Создает экземпляр именованного канала и возвращает дескриптор для последующих операций с каналом.
CreatePipe	Создает безымянный канал.
DisconnectNamedPipe	Отсоединяет серверную сторону экземпляра именованного канала от клиентского процесса.
GetNamedPipeHandleState	Получает информацию о работе указанного именованного канала.
GetNamedPipeInfo	Извлекает свойства указанного именованного канала.
PeekNamedPipe	Копирует данные их именованного или безымянного канала в буфер без удаления их из канала.
SetNamedPipeHandleState	Устанавливает режим чтения и режим блокировки вызова функций (синхронный или асинхронный) для указанного именованного канала.
TransactNamedPipe	Комбинирует операции записи сообщения в канал и чтения сообщения из канала в одну сетевую транзакцию.
WaitNamedPipe	Ожидает, пока истечет время ожидания или пока экземпляр указанного именованного канала не будет доступен для подключения к нему.

Кроме того, для работы с каналами используется функция CreateFile (для подключения к каналу со стороны клиента) и функции WriteFile и ReadFile для записи и чтения данных в/из канала соответственно.

Рассмотрим пример синхронной работы с каналами (т.е. с использованием блокирующих вызовов функций работы с каналами).

1. Многопоточный сервер каналов. Синхронный режим работы.

В данном примере главный поток состоит из цикла, в котором создается экземпляр канала и ожидается подключение клиента. Когда клиент успешно подсоединяется к каналу,

сервер создает новый поток, обслуживающий этого клиента, и продолжает ожидание новых подключений.

Поток, обслуживающий каждый экземпляр канала считывает запросы из него и отправляет ответы по этому же каналу до тех пор пока не произойдет отсоединение от клиента. Когда клиент закрывает дескриптор канала, поток сервера также выполняет отсоединение, закрывает дескриптор канала и завершает свою работу.

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

VOID InstanceThread(LPVOID);
VOID GetAnswerToRequest(LPTSTR, LPTSTR, LPDWORD);

int xx = 0;

DWORD main(VOID)
{
  BOOL fConnected;
  DWORD dwThreadId;
  HANDLE hPipe, hThread;
  LPTSTR lpszPipename = "\\\\.\\pipe\\mynamedpipe";

// Главный цикл создает экземпляр именованного канала и
// затем ожидает подключение клиентов к нему. Когда происходит
// подключение, создается поток, производящий обмен данными
// с клиентом, а выполнение главного цикла продолжается.

  for (;;)
  {
    hPipe = CreateNamedPipe(
      lpszPipename,           // Имя канала
      PIPE_ACCESS_DUPLEX,     // Дуплексный доступ к каналу
      PIPE_TYPE_MESSAGE |     // Установка режима работы канала
```

```
   PIPE_READMODE_MESSAGE |   // для передачи по нему отдельных сообщений
   PIPE_WAIT,               // Синхронное выполнение операций с каналом
   PIPE_UNLIMITED_INSTANCES, // Неограниченное количество экземпляров
   BUFSIZE,                 // Размер буфера отправки
   BUFSIZE,                 // Размер буфера приема
   PIPE_TIMEOUT,            // Время ожидания клиента
   NULL);                   // Без дополнительных атрибутов безопасности

if (hPipe == INVALID_HANDLE_VALUE)
   MyErrExit("Экземпляр именованного канала не создан");

// Ждем, пока не подсоединится клиент; в случае успешного подключения,
// Функция возвращает ненулевое значение. Если функция вернет ноль,
// то GetLastError вернет значение ERROR_PIPE_CONNECTED.

fConnected = ConnectNamedPipe(hPipe, NULL) ?
  TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);

if (fConnected)
{
// Создаем поток для обслуживания клиента.
  hThread = CreateThread(
    NULL,              // без атрибутов безопасности
    0,                 // размер стека по умолчанию
    (LPTHREAD_START_ROUTINE) InstanceThread,
    (LPVOID) hPipe,    // параметр потока – дескриптор канала
    0,                 // без отложенного запуска
    &dwThreadId);      // возвращает дескриптор потока

  if (hThread == NULL)
    MyErrExit("Создание потока произошло с ошибками");
  else
    CloseHandle(hThread);

}
```

```
    else
      // Если клиент не смог подсоединиться, то уничтожаем экземпляр канала.
       CloseHandle(hPipe);
  }
  return 1;
}

// Главная функция потока, обслуживающего клиентские подключения
VOID InstanceThread(LPVOID lpvParam)
{
  CHAR chRequest[BUFSIZE];
  CHAR chReply[BUFSIZE];
  DWORD cbBytesRead, cbReplyBytes, cbWritten;
  BOOL fSuccess;
  HANDLE hPipe;

// Переданный потоку параметр интерпретируем как дескриптор канала.

  hPipe = (HANDLE) lpvParam;

  while (1)
  {
  // Считываем из канала запросы клиентов.
    fSuccess = ReadFile(
      hPipe,       // дескриптор канала
      chRequest,   // буфер для получения данных
      BUFSIZE,     // указываем размер буфера
      &cbBytesRead, // запоминаем количество считанных байт
      NULL);       // синхронный режим ввода-вывода

    // Обрабатываем запрос если он корректен
    if (! fSuccess || cbBytesRead == 0)
      break;
    GetAnswerToRequest(chRequest, chReply, &cbReplyBytes);
```

```
  // Записываем в канал результат обработки клиентского запроса.
    fSuccess = WriteFile(
      hPipe,        // дескриптор канала
      chReply,      // буфер с данными для передачи
      cbReplyBytes, // количество байт для передачи
      &cbWritten,   // запоминаем количество записанных в канал байт
      NULL);        // синхронный режим ввода-вывода

    if (! fSuccess || cbReplyBytes != cbWritten) break;
  }

// Записываем содержимое буферов в канал, чтобы позволить клиенту считать
// остаток информации перед отключением. Затем выполним отсоединение от
// канала и уничтожаем дескриптор этого экземпляра канала.

  FlushFileBuffers(hPipe);
  DisconnectNamedPipe(hPipe);
  CloseHandle(hPipe);
}
```

2. Клиент каналов. Синхронный режим работы.

В данном примере клиент открывает именованный канал с помощью функции CreateFile и устанавливает канал в режим чтения/записи сообщений с помощью функции SetNamedPipeHandleState. Затем использует функции WriteFile и ReadFile для отправки запросов серверу и чтения ответов сервера соответственно.

```
#include <windows.h>

DWORD main(int argc, char *argv[])
{
  HANDLE hPipe;
  LPVOID lpvMessage;
  CHAR chBuf[512];
  BOOL fSuccess;
  DWORD cbRead, cbWritten, dwMode;
```

```
  LPTSTR lpszPipename = "\\\\.\\pipe\\mynamedpipe";

// Пытаемся открыть именованный канал; если необходимо, то подождем.

  while (1)
  {
    hPipe = CreateFile(
      lpszPipename,   // имя канала
      GENERIC_READ |  // доступ на чтение и запись данных
      GENERIC_WRITE,
      0,              // без разделения доступа
      NULL,           // без дополнительных атрибутов безопасности
      OPEN_EXISTING,  // открываем существующий канал
      0,              // задаем атрибуты по умолчанию
      NULL);          // без файла шаблона

  // Если подключение успешно, то выходим из цикла ожидания.

    if (hPipe != INVALID_HANDLE_VALUE)
      break;

    // Если возникает ошибка отличная от ERROR_PIPE_BUSY, то прекращаем работу.

    if (GetLastError() != ERROR_PIPE_BUSY)
      MyErrExit("Не могу открыть канал");

    // Ждем 20 секунд до повторного подключения.

    if (! WaitNamedPipe(lpszPipename, 20000) )
      MyErrExit("Не могу открыть канал");
  }

// Канал подключен успешно; сменим режим работы канала на чтение/запись сообщений

  dwMode = PIPE_READMODE_MESSAGE;
```

```
  fSuccess = SetNamedPipeHandleState(
    hPipe,                    // дескриптор канала
    &dwMode,                  // новый режим работы
    NULL,                     // не устанавливаем максимальный размер
    NULL);                    // не устанавливаем максимальное время
  if (!fSuccess)
    MyErrExit("Невозможно сменить режим работы канала");

// Отправляем сообщения серверу.

  lpvMessage = (argc > 1) ? argv[1] : "default message";

  fSuccess = WriteFile(
    hPipe,                // дескриптор канала
    lpvMessage,           // сообщение
    strlen(lpvMessage) + 1, // длина сообщения
    &cbWritten,           // количество записанных в канал байт
    NULL);                // синхронный ввод/вывод
  if (! fSuccess)
    MyErrExit("Запись сообщения в канал не удалась");

  do
  {
  // Считываем сообщения за канала.

    fSuccess = ReadFile(
      hPipe,    // дескриптор канала
      chBuf,    // буфер для получения ответа
      512,      // размер буфера
      &cbRead,  // количество считанных из канала байт
      NULL);    // синхронный ввод/вывод

    if (! fSuccess && GetLastError() != ERROR_MORE_DATA)
      break;
```

```
    // Следующий код – код обработки ответа сервера.
    // В данном случае просто выводим сообщение на STDOUT

    if (! WriteFile(GetStdHandle(STD_OUTPUT_HANDLE),
      chBuf, cbRead, &cbWritten, NULL))
      break;

  } while (! fSuccess);  // если ERROR_MORE_DATA (т.е. еще остались данные),
                // то повторяем считывание из канала

  // Закрываем канал
  CloseHandle(hPipe);

  return 0;
}
```

3. Совмещенное чтение/запись данных при работе с каналом.

Транзакции в именованных каналах — это клиент–серверные коммуникации, объединяющие операции записи и чтения в одну сетевую операцию. Такие транзакции погут быть использованы только на дуплексных, ориентированных на сообщения, каналах. Совмещенное чтение/запись данных позволяет увеличить производительность канала между клиентом и удаленным сервером. Процессы могут использовать функции TransactNamedPipe и CallNamedPipe для организации транзакций.

Функция TransactNamedPipe обычно используется клиентами каналов, но при необходимости может быть использована и серверами. Следующий код показывает пример вызова данной функции клиентом.

```
fSuccess = TransactNamedPipe(
  hPipe,              // дескриптор канала
  lpszWrite,          // сообщение серверу
  strlen(lpszWrite)+1, // длина сообщения серверу
  chReadBuf,          // буфер для получения ответа
  512,                // размер буфера ответа
  &cbRead,            // количество считанных байт
  NULL);              // синхронный вызов функции
```

```
// Если возникла ошибка, то выходим из функции иначе выводим ответ сервера
// и, если необходимо, считываем оставшиеся данные из канала

if (!fSuccess && (GetLastError() != ERROR_MORE_DATA))
{
  MyErrExit("Ошибка при выполнении транзакции");
}

while(1)
{
// Направляем на STDOUT считанные из канала данные.

  if (! WriteFile(GetStdHandle(STD_OUTPUT_HANDLE),
    chReadBuf, cbRead, &cbWritten, NULL) )
    break;

// Если все операции прошли успешно, то выходим из цикла.

  if (fSuccess)
    break;

// Если в канале еще остались данные, то считываем их.

  fSuccess = ReadFile(
    hPipe,      // дескриптор канала
    chReadBuf,  // буфер для получения ответа
    512,        // размер буфера
    &cbRead,    // количество считанных байт
    NULL);      // синхронный вызов функции

// Если возникла ошибка отличная от ERROR_MORE_DATA, то прекращаем работу.

  if (! fSuccess && (GetLastError() != ERROR_MORE_DATA))
    break;
```

```
}
```

Следующий код показывает вызов функции CallNamedPipe клиентом каналов.

```
// Комбинирует операции соединения, ожидания, записи, чтения и закрытия канала.

fSuccess = CallNamedPipe(
   lpszPipename,        // дескриптор канала
   lpszWrite,           // сообщение серверу
   strlen(lpszWrite)+1, // длина сообщения серверу
   chReadBuf,           // буфер для получения ответа
   512,                 // размер буфера для получения ответа
   &cbRead,             // количество считанных байт
   20000);              // ждем 20 секунд

if (fSuccess || GetLastError() == ERROR_MORE_DATA)
{

// В случае успеха выводим данные на STDOUT.

  WriteFile(GetStdHandle(STD_OUTPUT_HANDLE),
    chReadBuf, cbRead, &cbWritten, NULL);

// Канал уже закрыт, следовательно оставшиеся в нем данные
// прочесть уже невозможно – они безвозвратно потеряны.

  if (! fSuccess)
    printf("\n...дополнительные данные в сообщении потеряны\n");
}
```

Файловые отображения и совместно используемая память

Типы памяти

На рис. 1 представлена взаимосвязь виртуального адресного пространства процесса с физической и внешней памятью.

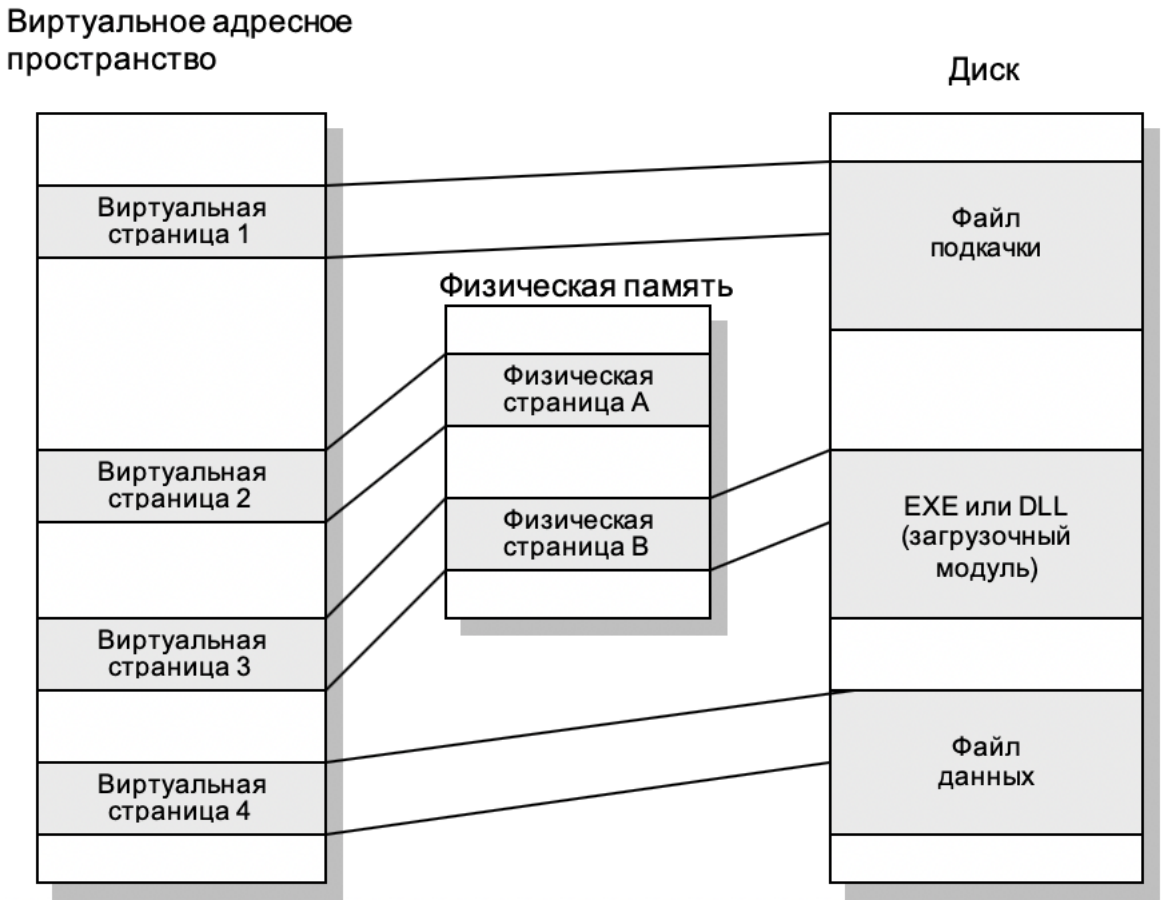

Рис. 1

Физическая память - это реальные микросхемы *RAM*, установленные в компьютере. Каждый байт физической памяти имеет физический адрес, который представляет собой число от нуля до числа на единицу меньшего, чем количество байтов физической памяти. Например, ПК с установленными 64 Мб RAM, имеет физические адреса &H00000000-&H04000000 в шестнадцатеричной системе счисления, что в десятичной системе будет 0-67 108 863.

Физическая память (в отличие от файла подкачки и виртуальной памяти) является исполняемой, то есть памятью, из которой можно читать и в которую центральный процессор может посредством системы команд записывать данные.

Виртуальная память

Виртуальная память (*virtual memory*) - это просто набор чисел, о которых говорят как о виртуальных адресах. Программист может использовать виртуальные адреса, но Windows не способна по этим адресам непосредственно обращаться к данным, поскольку такой адрес не является адресом реального физического запоминающего устройства, как в случае физических адресов и адресов файла подкачки. Для того

чтобы код с виртуальными адресами можно было выполнить, такие адреса должны быть отображены на физические адреса, по которым действительно могут храниться коды и данные. Эту операцию выполняет диспетчер виртуальной памяти (*Virtual Memory Manager - VMM*). Операционная система Windows обозначает некоторые области виртуальной памяти как области, к которым можно обратиться из программ пользовательского режима. Все остальные области указываются как зарезервированные. Какие области памяти доступны, а какие зарезервированы, зависит от версии операционной системы (Windows 9x или Windows NT).

Страничные блоки памяти

Как известно, наименьший адресуемый блок памяти - байт. Однако самым маленьким блоком памяти, которым оперирует Windows VMM, является страница памяти, называемая также страничным блоком памяти. На компьютерах с процессорами Intel объем страничного блока равен 4 Кб.

Память файла подкачки

Страничный файл, который называется также файлом подкачки, в Windows находится на жестком диске. Он используется для хранения данных и программ точно так же, как и физическая память, но его объем обычно превышает объем физической памяти. Windows использует файл подкачки (или файлы, их может быть несколько) для хранения информации, которая не помещается в RAM, производя, если нужно, обмен страниц между файлом подкачки и RAM.

Таким образом, диапазон виртуальных адресов скорее согласуется с адресами в файле подкачки, чем с адресами физической памяти. Когда такое согласование достигается, говорят, что виртуальные адреса спроецированы на файл подкачки, или являются проецируемыми на файл подкачки.

Набор виртуальных адресов может проецироваться на физическую память, файл подкачки или любой файл.

Файлы, отображаемые в память

Любой файл может быть использован для проецирования виртуальной памяти торчно так же, как для этих целей используется файл подкачки. Фактически единственное назначение файла подкачки - проецирование виртуальной памяти. Поэтому файлы, проецируемые в память подобным образом, называются отображаемыми в память. На предыдущем рисунке изображены именно такие файлы. Соответствующие виртуальные страницы являются спроецированными на файл.

Функция *CreateFileMapping* объявляется так:

function CreateFileMapping(hFile: THandle; lpFileMappingAttributes: PSecurityAttributes; flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWORD; lpName: PChar): THandle; stdcall;

function CreateFileMapping; external kernel32 name 'CreateFileMappingA';

Она создает объект «отображение файла», используя дескриптор открытого файла, и возвращает дескриптор этого объекта. Дескриптор может использоваться с функцией *MapViewOf File*, отображающей файл в виртуальную память:

function MapViewOfFile(hFileMappingObject: THandle; dwDesiredAccess: DWORD; dwFileOffsetHigh, dwFileOffsetLow, dwNumberOfBytesToMap: DWORD): Pointer; stdcall;

function MapViewOfFile; external kernel32 name 'MapViewOfFile';

Начальный адрес объекта «отображение файла» в виртуальной памяти возвращает функция *MapViewOfFile*. Можно также сказать, что представление проецируется на файл с дескриптором *hFile.*

Если параметр *hFile,* передаваемый функции *CreateFileMapping*, установлен в -1, то объект «отображение файла» (любые представления, созданные на основе этого объекта) проецируем на файл подкачки, а не на заданный файл.

Совместно используемая физическая память

О физической памяти говорят, что она совместно используется (shared), если она отображается на виртуальное адресное пространство нескольких процессов, хотя виртуальные адреса в каждом процессе могут отличаться. Следующий рисунок иллюстрирует это утверждение.

Если файл, такой как DLL, находится в совместно используемой физической памяти, то о нем можно говорить как о совместно используемом.

Одно из преимуществ файлов, отображаемых в память, заключается в том, что их легко использовать совместно. Присвоение имени объекту «отображение файла» делает возможным совместное использование файла несколькими процессами. В этом случае его содержимое отображено на совместно используемую физическую память (см. рис. 2). Возможно также совместное пользование содержимого файла подкачки с помощью механизма отображения файла.

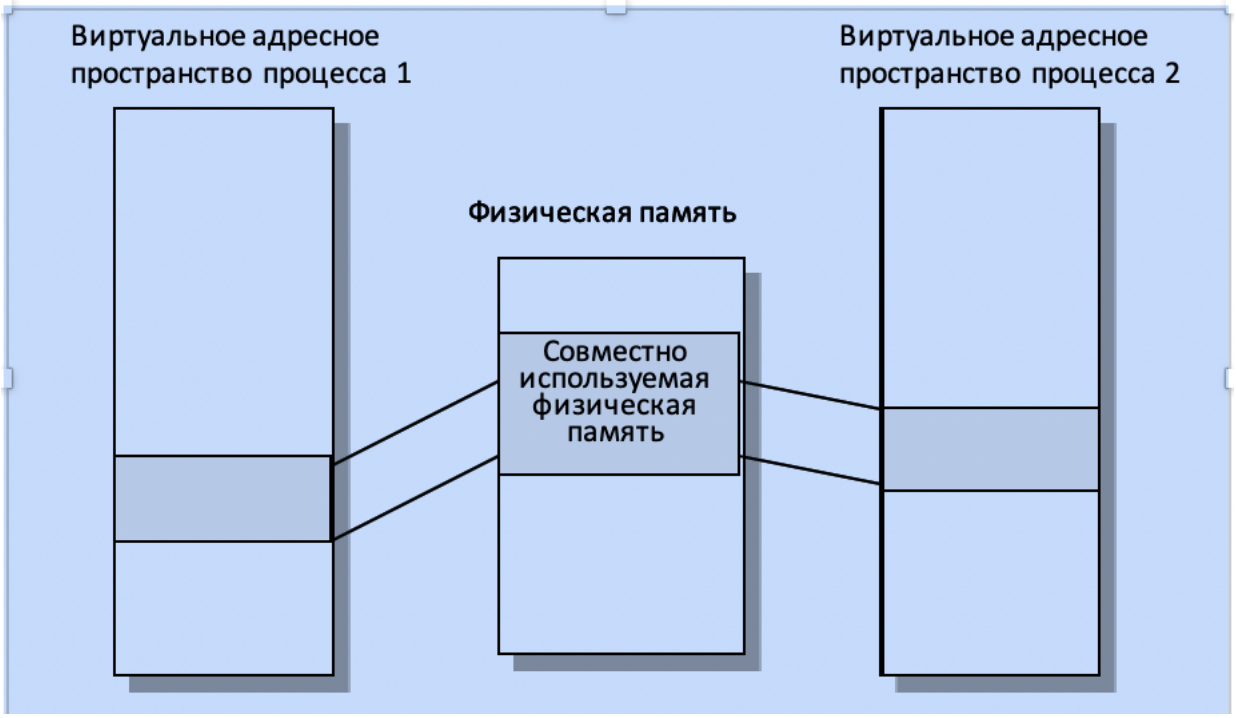

Рис. 2

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDistribution,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
 );
```

Параметры этой API- функции:

lpFileName – имя открываемого файла;

dwDesiredAccess — GENERIC_READ, чтобы разрешить чтение с устройства. GENERIC_WRITE, чтобы разрешить запись на устройство (константы можно объединить).
Ноль, чтобы разрешить только получение информации об устройстве;

dwShareMode — 0 для запрета общего доступа, FILE_SHARE_READ и/или FILE_SHARE_WRITE для разрешения общего доступа к

файлу;

lpSecurityAttributes — указатель на структуру, определяющую атрибуты безопасности файла (если они поддерживаются операционной системой);

dwCreationDistribution — одна из следующих констант: CREATE_NEW: создать файл. Если файл существует, происходит ошибка
CREATE_ALWAYS: создать файл. Предыдущий файл перезаписывается;
OPEN_EXISTING: открываемый файл должен существовать (обязательно используется для устройств)
OPEN_ALWAYS: создать файл, если он не существует
TRUNCATE_EXISTING: существующий файл усекается до нулевой длины;

dwFlagsAndAttributes Long — комбинация следующих констант:
FILE_ATTRIBUTE_ARCHIVE: установить архивный атрибут;
FILE__ATTRIBUTE_COMPRESSED: помечает файл как подлежащий сжатию или задает сжатие для файлов каталога по умолчанию;
FILE_ATTRIBUTE_NORMAL: другие атрибуты файла не заданы;
FILE_ATTRJBUTE_HIDDEN: файл или каталог является скрытым;
FILE_ATTRIBUTE_READONLY: файл доступен только для чтения;
FILE_ATTRIBUTE_SYSTEM: файл является системным;
FILE_FLAG_WRITE_THROUGH: операционная система не откладывает операции записи в файл;
FILE_FLAG_OVERLAPPED: разрешить перекрывающиеся операции с файлом;
FILE_FLAG_NO_BUFFERING: запретить промежуточную буферизацию файла. Адреса буферов должны выравниваться по границам секторов для текущего тома;
FILE_FLAG_RANDOM_ACCESS: буферизация файла оптимизируется для произвольного доступа;
FILE_FLAG_SEQUENTIAL_SCAN: буферизация файла

оптимизируется для последовательного доступа;

FILE_FLAG_DELETE_ON_CL0SE: при закрытии последнего открытого дескриптора файл удаляется. Идеально подходит для временных файлов;

В Windows NT в комбинацию также можно включить следующие флаги: SECURITY_ANONYMOUS, SECURITY_IDENTIFICATION, SECURITY_IMPERSONATION, SECURITY_DELEGATI0N,SECURITY_CONTEXT_TRACKING, SECURITY_EFFECTTVE_ONLY;

hTemplateFile — если параметр не равен нулю, он содержит дескриптор файла, с которого будут скопированы расширенные атрибуты нового файла.

Возвращаемое значение - дескриптор файла в случае успеха. INVALID_HANDLE_VALUE при ошибке. Устанавливает информацию GetLastError. Даже если функция завершилась успешно, но файл существовал и был задан флаг CREATE_ALWAYS или OPEN_ALWAYS, GetLastError возвращает ERROR_ALREADY_EXISTS.

Далее приведена структура, определяющая атрибуты безопасности файла:

```
typedef struct _SECURITY_ATTRIBUTES { // sa
   DWORD  nLength;
   LPVOID lpSecurityDescriptor;
   BOOL   bInheritHandle;
} SECURITY_ATTRIBUTES;
```

- Создание объекта файлового отображения File-mapping

```
HANDLE CreateFileMapping(
   HANDLE hFile,
   LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
   DWORD flProtect,
   DWORD dwMaximumSizeHigh,
   DWORD dwMaximumSizeLow,
   LPCTSTR lpName
```

);

Рассмотрим параметры этой API- функции:

hFile – дескриптор файла, для которого создается отображение;

lpFileMappingAttributes — SECUR1TY_ATTRIBLITES — объект безопасности, используемый при создании файлового отображения. NULL для использования стандартных атрибутов безопасности;

flProtect — одна из следующих констант:

PAGE_READONLY: созданное файловое отображение доступно только для чтения;

PAGE_READWRITE: файловое отображение доступно для чтения и записи;

PAGE_WRITECQPY: разрешается копирование при записи; Также в комбинацию могут включаться следующие константы: SEC_COMMГГ: для страниц секции выделяется физическое место в памяти или файле подкачки;

SEC_IMAGE: файл является исполняемым;

SEC_RESERVE: для страниц секции резервируется виртуальная память без фактического выделения.

dwMaximumSizeHigh — максимальный размер файлового отображения (старшие 32 бита);

dwMaximumSizeLow — максимальный размер файлового отображения (младшие 32 бита); Если параметры dwMaximumSizeHigh и dwMaximumSizeLow одновременно равны нулю, используется фактический размер файла на диске;

lpName — имя объекта файлового отображения. Если файловое отображение с заданным именем уже существует, функция открывает его.

Возвращаемое значение - дескриптор созданного объекта файлового отображения. Ноль в случае ошибки. Устанавливает информацию GetLastError. Даже если функция завершилась успешно, но возвращенный манипулятор принадлежит существующему объекту фазового отображения, GetLastError возвращает ERROR_ALREADY_EXISTS. В этом

случае размер файлового отображении определяется размером существующего объекта, а не параметрами функции.

- Функция отображает объект файлового отображения в адресное пространство текущего процесса.

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

Рассмотрим параметры этой API- функции:

hFileMappingObject – дескриптор объекта файлового отображения;

dwDesiredAccess — одна из следующих констант:

FILE_MAP_WRITE : отображение доступно для чтения и записи. Объект файлового отображения должен быть создан с флагом PAGE_READWRITE;

FILE_MAP_READ: отображение доступно только для чтения. Объект файлового отображения должен быть создан с флагом PAGE_READ или PAGE_READWRITE;

FILE_MAP_ALL_ACCESS: то же, что FILE_MAP_WRITE

HLE_MAP_COPY: копирование при записи. В Windows 95 Обьект файлового отображения должен быть создан с флагом PAGE_WRITECOPY;

dwFileOffsetHigh — старшие 32 бита смещения в файле, с которого начинается отображение;

dwFileOffsetLow — младшие 32 бита смещения в файле, с которого начинается отображение;

dwNumberOfBytesToMap — количество отображаемых байт в файле. Ноль, чтобы использовать весь объект файлового отображения;

Возвращаемое значение - начальный адрес отображения в памяти. Ноль при ошибке. Устанавливает информацию GetLastError.

Комментарии: dwOffsetLow и dwOffsetHigh должны содержать смещение с учетом гранулярности выделения памяти в системе. Другими словами, если гранулярность памяти в системе равна 64 Кбайт (выделяемые блоки выравниваются по границе 64 Кбайт), значение должно быть кратно 64 Кбайт. Чтобы получить гранулярность выделения памяти в системе, воспользуйтесь функцией GetSystemInfo. В большинстве приложений передается ноль, чтобы отображение начиналось с начала файла. Параметр lpBaseAddress также должен быть кратен значению гранулярности.

- Функция закрывает объект ядра. К числу объектов ядра относятся объекты файлов,файловых отображений, процессов, потоков, безопасности и синхронизации.

```
BOOL CloseHandle(
  HANDLE hObject        // дескриптор закрывамого объекта.
);
```

Возвращаемое значение - ненулевое значение в случае успеха, ноль при ошибке. Устанавливает информацию GetLastError.

Комментарии: Объекты ядра удаляются лишь после того, как будут закрыты все ссылки на них.

- Функция прекращает отображение объекта в адресное пространство текущего процесса.

```
BOOL UnmapViewOfFile(
  LPCVOID lpBaseAddress         // базовый адрес отображения, установленного ранее
                                функцией  MapViewOfFile
);

VOID CopyMemory (
```

```
 PVOID Destination,      // address of copy destination
 CONST VOID *Source,           // address of block to copy
 DWORD Length          // size, in bytes, of block to copy
);
```

СОДЕРЖАНИЕ ОТЧЕТА

1. Наименование лабораторной работы, ее цель.
2. Программа, выполняющая с помощью ВСЕХ перечисленных механизмов межпроцессного взаимодействия (отображение файлов, почтовые ящики, каналы) одну из следующих задач (в соответствии с № по журналу):

Таблица 1.Задание на лабораторную работу

№ варианта	Задача
1, 6, 11, 16, 21	Реализовать вычисление определителя квадратной матрицы с помощью разложения ее на определители меньшего порядка. При этом «ведущий» процесс рассылает задания «ведомым» процессам, последние выполняют вычисление определителей, а затем главный процесс вычисляет окончательный результат.
2, 7, 12, 17, 22	Реализовать нахождение обратной матрицы методом Гаусса, при этом задания по решению систем линейных уравнений распределяются поровну для каждого процесса.
3, 8, 13, 18, 23	Реализовать перемножение двух матриц с помощью нескольких процессов: каждый процесс выполняет перемножение строки первой матрицы на столбец второй (в соответствии с правилом умножения матриц). При необходимости процессам, выполняющим умножение, может быть отправлено несколько заданий.
4, 9, 14, 19, 24	Реализовать алгоритм блочной сортировки файла целых чисел. Каждый процесс, выполняющий сортировку, получает свою часть файла от ведущего процесса и сортирует его. Ведущий процесс выполняет упорядочивание уже отсортированных блоков. При необходимости ведомым процессам может быть выделено более одного задания на сортировку.
5, 10, 15, 20, 25	Реализовать обмен текстовыми сообщениями между несколькими процессами. Обеспечить возможность отправки сообщения сразу нескольким адресатам. Реализовать подтверждение приема сообщения адресатом или, в случае потери сообщения, повторную его передачу.

3. Примеры разработанных приложений (программы и результаты).