Министерство науки и высшего образования Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

А.В. ГУНЬКО

ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие

НОВОСИБИРСК
2019

УДК 004.45 (075.8)
Г 948

Рецензенты:

канд. техн. наук, доцент *В.А. Астапчук*
канд. техн. наук, доцент *Г.В. Саблина*

Гунько А.В.
Г 948 Программирование: учебно-методическое пособие / А.В. Гунько. – Новосибирск: Изд-во НГТУ, 2019. – 74 с.

ISBN 978-5-7782-3961-6

В пособии описаны методы и средства разработки многозадачного и многопоточного программного обеспечения в операционных системах семейства Windows, средства межзадачной и межпоточной коммуникации: анонимные и именованные каналы, почтовые ящики, отображаемые на память файлы, события, семафоры, взаимные исключения.

Кроме того, кратко обсуждаются средства коммуникации процессов по сети, а также особенности взаимодействия приложений и системных служб. Приводятся примеры реализации программ и даются задания для самостоятельной работы, включая варианты повышенной сложности.

Рекомендовано студентам ряда технических специальностей, связанных с разработкой многозадачного и многопоточного программного обеспечения в среде операционных систем семейства Windows.

Работа подготовлена на кафедре автоматики и утверждена Редакционно-издательским советом университета в качестве учебно-методического пособия для студентов II курса, обучающихся по направлениям 27.03.04 «Управление в технических системах» и 09.03.01 «Информатика и вычислительная техника»

УДК 004.45 (075.8)

ISBN 978-5-7782-3961-6

Тема 1

ФАЙЛОВЫЕ ОПЕРАЦИИ WinAPI

Цель работы: изучить особенности выполнения операций с файлами средствами WinAPI на языке С в операционных системах семейства Windows.

Краткие теоретические сведения

Операции открытия, чтения, записи и закрытия файлов

Прежде всего, приложение должно открыть файл при помощи функции `CreateFile`. Ниже приведен прототип этой функции:

```
HANDLE CreateFile(
   LPCTSTR lpFileName,       // адрес строки имени файла
   DWORD   dwDesiredAccess,  // режим доступа
   DWORD   dwShareMode,      // режим совместного
                             // использования файла
   LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                             // дескриптор защиты
   DWORD  dwCreationDistribution, // параметры создания
   DWORD  dwFlagsAndAttributes,   // атрибуты файла
   HANDLE hTemplateFile);         // идентификатор файла
                                     с атрибутами
```

Через параметр `lpFileName` этой функции передается адрес текстовой строки с завершающим нулевым символом, содержащей путь и имя файла, канала или любого другого именованного объекта, который необходимо открыть или создать. Допустимое количество символов при

указании путей доступа обычно ограничивается значением MAX_PATH (260).

С помощью параметра dwDesiredAccess следует указать нужный вид доступа. Если файл будет открыт только для чтения, в этом параметре необходимо указать флаг GENERIC_READ. Если необходимо выполнять над файлом операции чтения и записи, следует указать логическую комбинацию флагов GENERIC_READ и GENERIC_WRITE. В том случае, когда будет указан только флаг GENERIC_WRITE, операция чтения из файла будет запрещена.

Если файл будет использоваться одновременно несколькими процессами, через параметр dwShareMode необходимо передать режимы совместного использования файла: FILE_SHARE_READ или FILE_SHARE_WRITE.

Через параметр lpSecurityAttributes необходимо передать указатель на дескриптор защиты или значение NULL, если этот дескриптор не используется.

Параметр dwCreationDistribution определяет действия, выполняемые функцией CreateFile, если приложение пытается создать файл, который уже существует.

Параметр dwFlagsAndAttributes задает атрибуты и флаги для файла. Атрибуты являются характеристиками файла (а не открытого дескриптора) и игнорируются, если открывается существующий файл. Для создания нового файла рекомендуется атрибут FILE_ATTRIBUTE_NORMAL, который можно использовать только отдельно.

Последний параметр, hTemplateFile, предназначен для доступа к файлу шаблона с расширенными атрибутами для создаваемого файла. Этот параметр здесь не рассматривается.

В случае успешного завершения функция CreateFile возвращает идентификатор открытого файла. При ошибке возвращается значение INVALID_HANDLE_VALUE.

Для закрытия объектов любого типа, объявления недействительными их дескрипторов и освобождения системных ресурсов почти во всех случаях используется одна и та же универсальная функция:

```
BOOL CloseHandle (HANDLE hObject);
```

Единственный параметр hObject – дескриптор объекта любого типа. Возвращаемое значение: в случае успешного выполнения функции – TRUE, иначе – FALSE. Попытки закрытия недействительных дескрипторов или повторного закрытия одного и того же дескриптора приводят к исключениям.

Чтение из файла производится функцией

```
BOOL ReadFile (HANDLE hFile, LPVOID lpBuffer, DWORD
nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead,
LPOVERLAPPED lpOverlapped);
```

Здесь параметр hFile – дескриптор считываемого файла, который должен быть создан с правами доступа GENERIC_READ. Параметр lpBuffer является указателем на буфер в памяти, куда помещаются считываемые данные. Параметр nNumberOfBytesToRead – количество байтов, которые должны быть считаны из файла. Параметр lpNumberOfBytesRead – указатель на переменную, предназначенную для хранения числа байтов, которые были фактически считаны в результате вызова функции ReadFile. Этот параметр может принимать нулевое значение, если перед выполнением чтения указатель файла был позиционирован в конце файла или если во время чтения возникли ошибки, а также после чтения из именованного канала, работающего в режиме обмена сообщениями (работа с каналами описана далее), если переданное сообщение имеет нулевую длину.

Параметр lpOverlapped – указатель на структуру OVERLAPPED. Используется для организации асинхронного режима чтения (записи). Если запись выполняется синхронно, в качестве этого параметра следует указать значение NULL. Для использования асинхронного режима файл должен быть открыт функцией CreateFile с использованием флага FILE_FLAG_OVERLAPPED. Если указан этот флаг, параметр lpOverlapped не может иметь значение NULL. Он обязательно должен содержать адрес подготовленной структуры типа OVERLAPPED.

Возвращаемое функцией значение в случае успешного выполнения (которое считается таковым, даже если не был считан ни один байт из-за попытки чтения с выходом за пределы файла) – TRUE, иначе – FALSE.

Если значения дескриптора файла или иных параметров, используемых при вызове функции, оказались недействительными, возникает ошибка и функция возвращает значение FALSE. Попытка выполнения чтения в ситуациях, когда указатель файла позиционирован в конце

файла, не приводит к ошибке; вместо этого количество считанных байтов (`*lpNumberOfBytesRead`) устанавливается равным нулю.

Запись в файл производится функцией

```
BOOL WriteFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWrite, LPOVERLAPPED lpOverlapped);
```

Ее параметры аналогичны параметрам функции чтения из файла. Возвращаемое значение в случае успешного выполнения – `TRUE`, иначе – `FALSE`. Успешное выполнение записи еще не говорит о том, что данные действительно оказались записанными на диск, если только при создании файла с помощью функции `CreateFile` не был использован флаг `FILE_FLAG_WRITE_THROUGH`. Если во время вызова функции указатель файла был позиционирован в конце файла, Windows увеличит длину существующего файла.

Методические указания

1. Проект может быть реализован в среде Visual C++ или Borland C++ 5.0 и выше. В первом случае выбирается консольное приложение Win32 без дополнительных библиотек. В любом случае в программу необходимо добавить файл включения *windows.h*.

2. Проект должен предусматривать обработку исключительных ситуаций (отсутствие входных параметров, отсутствие обрабатываемого файла, ошибки создания выходного файла и записи в него).

3. Пример кода (*file.cpp*) программы доступен по адресу http://gun.cs.nstu.ru/winprog/file.cpp.

Порядок выполнения работы

1. Написать и отладить программу, получающую в аргументах командной строки имя существующего текстового файла и символ (или число), используемый для обработки файла.

2. Результатом работы программы является выходной текстовый файл с тем же именем, что и входной, но с другим типом (расширением), содержащий текст, обработанный согласно вариантам (табл. 1), возвращаемое значение – количество выполненных операций или «–1» в случае ошибки.

Варианты заданий

Таблица 1

Задание	Параметры командной строки
1. Удалить из текста заданный символ	1. Имя входного файла 2. Заданный символ
2. В конце каждой строки вставить заданный символ	1. Имя входного файла 2. Заданный символ
3. Заменить цифры на пробелы	1. Имя входного файла 2. Количество замен
4. Заменить знаки на заданный символ	1. Имя входного файла 2. Заданный символ
5. Заменить каждый пробел на два	1. Имя входного файла 2. Количество замен
6. После каждой точки вставить символ ‘\n’	1. Имя входного файла 2. Количество замен
7. Удалить из текста все пробелы	1. Имя входного файла 2. Количество замен
8. Заменить заданные символы на пробелы	1. Имя входного файла 2. Заданный символ
9. После каждого пробела вставить точку	1.Имя входного файла 2. Количество вставок
10. Заменить все пробелы первым символом текста	1. Имя входного файла 2. Максимальное количество замен
11. Во всех парах одинаковых символов второй символ заменить на пробел	1. Имя входного файла 2. Количество замен
12. Заменить на пробелы все символы, совпадающие с первым символом в строке	1. Имя входного файла 2. Количество замен
13. Заменить заданную пару букв на символы #@	1. Имя входного файла 2. Заданная пара букв
14. Заменить все цифры заданным символом	1. Имя входного файла 2. Заданный символ
15. Заменить на пробел все символы, совпадающие с последним символом в строке	1. Имя входного файла 2. Количество замен
16. Заменить все символы с кодами меньше 48 на пробелы	1. Имя входного файла 2. Количество замен
17. Заменить все символы с кодами больше 48 на пробелы	1. Имя входного файла 2. Количество замен

Окончание табл. 1

Задание	Параметры командной строки
18. Заменить каждый третий символ на пробел	1. Имя входного файла 2. Количество замен
19. Заменить все пробелы на заданный символ	1. Имя входного файла 2. Заданный символ
20. Заменить все пары одинаковых символов на пробелы	1. Имя входного файла 2. Количество замен

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинг программы.

Контрольные вопросы

1. Что такое API? На какие категории подразделяется WinAPI?
2. Перечислите принципы, лежащие в основе WinAPI.
3. Перечислите преимущества и недостатки реализации файловых операций средствами стандартной библиотеки C и WinAPI.
4. Перечислите режимы доступа и совместного использования файла в функции `CreateFile`.
5. Перечислите возможные параметры создания файла функцией `CreateFile`.
6. Перечислите возможные атрибуты и флаги файла в функции `CreateFile`.
7. Каковы особенности закрытия файла функцией `CloseHandle`?
8. Перечислите основные параметры функции `ReadFile`.
9. Каковы возвращаемое значение и результаты работы функции `ReadFile`?
10. Перечислите основные параметры функции `WriteFile`.
11. Каковы возвращаемое значение и результаты работы функции `WriteFile`?
12. Какие в WinAPI есть дополнительные функции работы с файлами?
13. Что собой представляют файлы, отображаемые на память?
14. Для чего можно применять файлы, отображаемые на память?

Тема 2

ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ (DLL) И ИХ ПРИМЕНЕНИЕ

Цель работы: изучить особенности создания и применения динамических библиотек в операционных системах семейства Windows.

Краткие теоретические сведения

Создание динамических библиотек

В 32-разрядных DLL-библиотеках операционной системы Microsoft Windows используется функция `DLLEntryPoint`, которая выполняет все необходимые задачи по инициализации библиотеки и при необходимости освобождает заказанные ранее ресурсы. Функция `DLLEntryPoint` вызывается всякий раз, когда выполняется инициализация процесса или потока, обращающихся к функциям библиотеки, а также при явной загрузке и выгрузке библиотеки функциями `LoadLibrary` и `FreeLibrary`.

Ниже приведен прототип функции `DLLEntryPoint`:

```
BOOL WINAPI DllEntryPoint(
  HINSTANCE hinstDLL,     // идентификатор модуля DLL-
                             библиотеки
   DWORD    fdwReason,    // код причины вызова функции
   LPVOID   lpvReserved); // зарезервировано
```

Через параметр `hinstDLL` функции `DLLEntryPoint` передается идентификатор модуля DLL-библиотеки, который можно использовать при обращении к ресурсам, расположенным в файле этой библиотеки.

Что же касается параметра fdwReason, то он зависит от причины, по которой произошел вызов функции DLLEntryPoint. Этот параметр может принимать следующие значения:

DLL_PROCESS_ATTACH: библиотека отображается в адресное пространство процесса в результате запуска процесса или вызова функции LoadLibrary;

DLL_THREAD_ATTACH: текущий процесс создал новый поток, после чего система вызывает функции DLLEntryPoint всех DLL-библиотек, подключенных к процессу;

DLL_THREAD_DETACH: этот код причины передается функции DLLEntryPoint, когда поток завершает свою работу нормальным (не аварийным) способом;

DLL_PROCESS_DETACH: отображение DLL-библиотеки в адресное пространство отменяется в результате нормального завершения процесса или вызова функции FreeLibrary.

Параметр lpvReserved зарезервирован.

Обработка причины вызова и выполнение необходимых действий могут быть реализованы оператором switch:

```
switch(fdwReason)
{
   // Подключение нового процесса
   case DLL_PROCESS_ATTACH:
   {
   . . . // Обработка подключения процесса
   break;
   }
   . . .
}
```

В любом случае (даже если обработка подключений не производится) функция DLLEntryPoint должна вернуть результат: return TRUE;

Экспортирование функций и глобальных переменных

Кроме функции DLLEntryPoint в 32-разрядных библиотеках операционных систем Microsoft Windows могут быть определены *экспортируемые* и *неэкспортируемые* функции. Экспортируемые функции доступны для вызова приложениям Windows. Неэкспортируемые являются локальными для DLL-библиотеки, они доступны только для

функций библиотеки. При необходимости можно экспортировать из 32-разрядных DLL-библиотек не только функции, но и глобальные переменные.

Самый простой способ сделать функцию экспортируемой – перечислить все экспортируемые функции в файле определения модуля (в среде Visual Studio он называется *Source.def*) при помощи оператора `EXPORTS`:

```
EXPORTS
   ИмяТочкиВхода [=ВнутрИмя] [@Номер] [NONAME] [CONSTANT]
   . . .
```

Здесь `ИмяТочкиВхода` задает имя, под которым экспортируемая из DLL-библиотеки функция будет доступна для вызова. Внутри DLL-библиотеки эта функция может иметь другое имя. В этом случае необходимо указать ее внутреннее имя `ВнутрИмя`. С помощью параметра `@Номер` можно задать порядковый номер экспортируемой функции. Указав флаг `NONAME` и порядковый номер, можно сделать имя экспортируемой функции невидимым. При этом экспортируемую функцию можно будет вызвать только по порядковому номеру, так как имя такой функции не попадет в таблицу экспортируемых имен DLL-библиотеки. Флаг `CONSTANT` позволяет экспортировать из DLL-библиотеки не только функции, но и данные. При этом параметр `ИмяТочкиВхода` задает имя экспортируемой глобальной переменной, определенной в DLL-библиотеке.

Использование динамических библиотек

Приложение может в любой момент времени загрузить любую DLL-библиотеку, вызвав специально предназначенную для этого функцию программного интерфейса Windows с именем `LoadLibrary`:

```
HINSTANCE WINAPI LoadLibrary(LPCSTR lpszLibFileName);
```

Параметр функции является указателем на текстовую строку, закрытую двоичным нулем. В эту строку перед вызовом функции следует записать путь к файлу DLL-библиотеки и имя этого файла.

Если файл DLL-библиотеки найден, функция `LoadLibrary` возвращает идентификатор модуля библиотеки. В противном случае возвращается значение `NULL`. При этом код ошибки можно получить при помощи функции `GetLastError`.

Функция `LoadLibrary` может быть вызвана разными приложениями для одной и той же DLL-библиотеки несколько раз. В этом случае загрузка DLL-библиотеки выполняется только один раз.

В качестве примера приведем фрагмент исходного текста приложения, загружающего DLL-библиотеку из файла *DLLDEMO.DLL*:

```
void( *TestHello)(void); // прототип импортируемой
                         // функции
HANDLE hDLL;
hDLL = LoadLibrary("DLLDEMO.DLL");
if(hDLL != NULL)
{
  TestHello = (void(*)(void))GetProcAddress(hLib,
                                          "TestHello");
if (TestHello == NULL) {
    printf("TestHello function not found");
    exit(-1);
  }
  (*TestHello)();
}
  FreeLibrary(hDLL);
}
```

Для того чтобы вызвать функцию из библиотеки, зная ее идентификатор, необходимо получить значение дальнего указателя на эту функцию, вызвав функцию `GetProcAddress`:

```
  FARPROC WINAPI GetProcAddress(HINSTANCE hLibrary,
LPCSTR lpszProcName);
```

Через параметр `hLibrary` необходимо передать функции идентификатор DLL-библиотеки, полученный ранее от функции `LoadLibrary`.

Параметр `lpszProcName` является дальним указателем на строку, содержащую имя функции или ее порядковый номер, преобразованный макрокомандой `MAKEINTRESOURCE`:

```
  lpTellMe = GetProcAddress(hLib, MAKEINTRESOURCE(8));
```

Перед тем как передать управление функции по полученному адресу, следует убедиться в том, что этот адрес не равен `NULL`:

```
  if (TestHello == NULL) {
```

Если точка входа получена, функция вызывается через указатель на нее:

```
  (*TestHello)();
```

После использования DLL-библиотека освобождается при помощи функции `FreeLibrary`:

```
void WINAPI FreeLibrary(HINSTANCE hLibrary);
```

В качестве параметра этой функции следует передать идентификатор освобождаемой библиотеки.

При освобождении DLL-библиотеки ее счетчик использования уменьшается. Если этот счетчик становится равным нулю (что происходит, когда все приложения, работавшие с библиотекой, освободили ее или завершили свою работу), DLL-библиотека выгружается из памяти.

Методические указания

1. Библиотека может быть реализована в среде Visual C++ или Borland C++ 5.0 и выше. В первом случае выбирается консольное приложение Win32 с шаблоном «Библиотека DLL». Приложение, вызывающее функцию из библиотеки, создается так же, как и в лабораторной работе 1.

2. Проект должен предусматривать обработку исключительных ситуаций (отсутствие входных параметров, отсутствие обрабатываемого файла, отсутствие файла библиотеки, отсутствие функции в библиотеке, ошибки создания выходного файла и записи в него).

3. Пример кода библиотеки (*dllmain.cpp, testfunc.cpp, Source.def*) и программы (*testdll.cpp*) доступен по адресу: http://gun.cs.nstu.ru/winprog/dll.

Порядок выполнения работы

1. Модифицировать и отладить программу из лабораторной работы 1, перенеся обработку файла в функцию.

2. Оформить функцию обработки файла как библиотечную, создать библиотеку.

3. Модифицировать функцию `main` программы п. 1 для загрузки библиотеки, вызова библиотечной функции и выгрузки библиотеки.

4. Продемонстрировать работоспособность программы и перехват ею исключений преподавателю.

Варианты заданий

См. варианты заданий лабораторной работы 1.

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинг программы.

Контрольные вопросы

1. Какова роль динамических библиотек в среде WinAPI?
2. В чем разница статической и динамической компоновки приложения с библиотекой?
3. Как функции библиотеки отображаются в адресном пространстве прикладного процесса?
4. Как происходит инициализация динамической библиотеки?
5. Как в библиотеке обеспечить экспорт функций и глобальных переменных?
6. Как загрузить и выгрузить динамическую библиотеку в прикладной программе?
7. В каких каталогах ищутся файлы динамических библиотек?
8. Как убедиться, что динамическая библиотека загружена?
9. Как импортировать из библиотеки функцию и проверить ее наличие?
10. Как вызвать функцию, импортированную из библиотеки?

Тема 3

МНОГОЗАДАЧНОЕ ПРОГРАММИРОВАНИЕ В WINDOWS

Цель работы: изучить способы и средства реализации параллельно выполняющихся процессов средствами языка С в операционных системах семейства Windows.

Краткие теоретические сведения

Рассмотрим необходимые функции WinAPI и их параметры.

Создание процесса

Для создания процесса используется функция `CreateProcess`:

```
BOOL CreateProcess
(LPCTSTR lpApplicationName,  // имя исполняемого модуля
LPTSTR lpCommandLine,        // Командная строка
LPSECURITY_ATTRIBUTES lpProcessAttributes,
          // Указатель на структуру SECURITY_ATTRIBUTES
LPSECURITY_ATTRIBUTES lpThreadAttributes,
          // Указатель на структуру SECURITY_ATTRIBUTES
BOOL bInheritHandles,      // Флаг наследования
                              текущего процесса
DWORD dwCreationFlags,     // Флаги способов создания
                              процесса
LPVOID lpEnvironment,        // Указатель на блок среды
LPCTSTR lpCurrentDirectory,  // Текущий диск или каталог
LPSTARTUPINFO lpStartupInfo, // Указатель на структуру
STARTUPINFO
```

```
LPPROCESS_INFORMATION lpProcessInformation
     // Указатель нас структуру PROCESS_INFORMATION  );
```

Здесь lpApplicationName – указатель на строку, которая заканчивается нулем и содержит имя выполняемого модуля. Этот параметр может быть NULL – тогда имя модуля должно быть в lpCommandLine самым первым элементом. Если операционная система – 16-разрядная NT, то параметр NULL стоит обязательно. Имя модуля может быть абсолютным или относительным. Если оно относительное, то будет использована информация из lpCurrentDirectory или текущий каталог;

lpCommandLine – командная строка, где передаются параметры. Она может быть NULL. Здесь можно указать и путь, и имя модуля;

lpProcessAttributes – здесь определяются атрибуты защиты для нового приложения. Если указать NULL, то система сделает это по умолчанию;

lpThreadAttributes – здесь определяются атрибуты защиты для первого потока, созданного приложением. NULL приводит к установке по умолчанию;

bInheritHandles – флаг наследования от процесса, производящего запуск. Здесь наследуются дескрипторы. Унаследованные дескрипторы имеют те же значения и права доступа, что и оригиналы;

dwCreationFlags – флаг способа создания процесса и его приоритет. Можно применять следующие флаги (перечислены не все):

- CREATE_NEW_CONSOLE – новый процесс получает новую консоль вместо того, чтобы унаследовать родительскую;
- CREATE_NEW_PROCESS_GROUP – создаваемый процесс – корневой процесс новой группы;
- CREATE_SEPARATE_WOW_VDM – запуск нового процесса в собственной Virtual DOS Machine (VDM);
- CREATE_SHARED_WOW_VDM – запуск нового процесса в разделяемой Virtual DOS Machine;
- NULL задает флаги по умолчанию.

lpEnvironment – указывает на блок среды. Если NULL, то будет использован блок среды родительского процесса. Блок среды это список переменных имя=значение в виде строк с нулевым окончанием;

lpCurrentDirectory – указывает текущий диск и каталог. Если NULL, то будет использован диск и каталог процесса родителя;

LpStartupInfo – используется для настройки свойств процесса, например расположения окон и заголовок. Структура должна быть правильно инициализирована:

```
STARTUPINFO sti;                          // структура
ZeroMemory(&sti,sizeof(STARTUPINFO)); // обнулить
sti.cb=sizeof(STARTUPINFO);           // указать размер
```

lpProcessInformation – структура PROCESS_INFORMATION с информацией о процессе. Будет заполнена Windows.

В результате выполнения функция CreateProcess вернет FALSE или TRUE. В случае успеха – TRUE. Пример использования:

```
#include <windows.h>
#include <stdio.h>
void main()
{
STARTUPINFO si[255];
PROCESS_INFORMATION pi[255];
int num=5, quant=10; //число работников, груз каждого
for (i=0;i< num;i++)
     {
          strcpy(ln,"warunit.exe");
          ln=strcat(ln," ");
          sprintf(tmp,"%d",i);
          ln=strcat(ln,tmp);
          ln=strcat(ln," ");
          sprintf(tmp,"%d", quant);
          ln=strcat(ln,tmp);
          ZeroMemory( &si[i], sizeof(si[i]) );
          si[i].cb = sizeof(si);
          ZeroMemory( &pi[i], sizeof(pi[i]) );
if( !CreateProcess( NULL, line, NULL, NULL, TRUE, NULL,
NULL, NULL, &si[i], &pi[i] ))  {printf( "CreateProcess
failed.\n" ); exit(-2);}
else { printf("Process %Lu started for #%d\n",pi[i].hPro-
cess, i);}
}
Sleep(1000);// подождать
for (i=0;i< num;i++) {
```

```
//TerminateProcess(pi[i].hProcess,NO_ERROR);
// уничтожить процесс
CloseHandle( pi[i].hProcess );
// освободить дескриптор процесса
} }
```

Для упорядоченного вывода результатов работы процессов (потоков) на экран можно использовать функцию `Sleep`:

```
VOID Sleep(
   DWORD dwMilliseconds // время ожидания в миллисекундах
);
```

Функция `Sleep` не осуществляет возврата до тех пор, пока не истечет указанное время. В течение него выполнение процесса (потока) приостанавливается, и выделения процессорного времени не происходит.

Когда поток вызывает функцию `Sleep`, задержка на заданное время относится к этому потоку. Система продолжает выполнять другие потоки этого и других процессов.

Уничтожение процесса

Для уничтожения процессов можно применять следующие функции:

```
BOOL TerminateProcess(
     HANDLE hProcess, // указатель на уничтожаемый объект
     UINT uExitCode   // код завершения
);
BOOL CloseHandle(
     HANDLE hObject   // указатель на уничтожаемый объект
);
```

Первая функция не освобождает занятые процессами ресурсы до закрытия указателей. Вторая – закрывает все открытые объектом ресурсы. Возвращаемые значения обеих функций одинаковы, `TRUE` – при успехе и `FALSE` – при неудаче. Пример применения:

```
TerminateProcess(pi[i].hProcess,NO_ERROR);
CloseHandle( pi[i].hProcess );
```

Ожидание завершения процессов (потоков)

Рассмотрим следующую функцию:

```
DWORD WaitForSingleObject (
HANDLE hHandle          // указатель на ожидаемый объект
DWORD dwMilliseconds    // время ожидания объекта
  );
```

Ожидаемым объектом может быть процесс, поток, мьютекс, семафор и другие объекты. Если указатель ожидаемого объекта уничтожен в процессе ожидания, то поведение функции не определено.

Время ожидания объекта задается в миллисекундах. Если время ожидания равно нулю, то функция проверяет состояние объекта и немедленно завершается. Если время ожидания задано константой `INFINITE`, то время ожидания бесконечно.

Возвращаемое значение может быть представлено константой `WAIT_OBJECT_0`, если ожидаемый объект сигнализировал о своем завершении, или `WAIT_TIMEOUT`, если время ожидания истекло.

```
DWORD WaitForMultipleObjects (
DWORD nCount,//количество указателей на объекты в массиве
CONST HANDLE *lpHandles, // массив указателей на ожидаемые объекты
BOOL bWaitAll,//TRUE – ждать всех, FALSE – хотя бы одного
DWORD dwMilliseconds         // время ожидания объекта
  );
```

Параметры аналогичны родственной функции `WaitForSingleObject`. Возвращаемое значение может быть представлено константой `WAIT_OBJECT_0`, если `bWaitAll=TRUE`, или значением в диапазоне от `WAIT_OBJECT_0` до `WAIT_OBJECT_0+nCount-1`, если `bWaitAll=FALSE` (номер объекта), или `WAIT_TIMEOUT`, если время ожидания истекло. Пример применения:

```
//WaitForMultipleObjects(num, hThread, TRUE, INFINITE);
for (i = 0; i < num; i++) {
printf("Process %Lu is %Lu\n",hThread[i],
WaitForSingleObject (hThread[i], INFINITE)); }
```

Получение результатов работы порожденных (*дочерних*) *процессов*

После того как процесс завершил свою работу, он может вызвать функцию `ExitProcess`, указав в качестве параметра код завершения:

```
VOID ExitProcess (UINT uExitCode)
```

Эта функция не осуществляет возврата. Она завершает вызывающий процесс и все его потоки. Выполнение оператора `return` в основной программе с использованием кода возврата равносильно вызову функции `ExitProcess`, в котором этот код возврата указан в качестве кода завершения. Другой процесс может определить код завершения, вызвав функцию `GetExitCodeProcess`:

```
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode)
```

Процесс, идентифицируемый дескриптором `hProcess`, должен обладать правами доступа `PROCESS_QUERY_INFORMATION`, `lpExitCode` указывает на переменную типа `DWORD`, вторая принимает значение кода завершения. Одним из ее возможных значений является `STILL_ACTIVE`, показывающее, что данный процесс еще не завершился.

Методические указания

1. Проект может быть реализован на Visual C++ 6.0 или в среде Borland C++ 5.0 и выше. В первом случае выбирается консольное приложение Win32 без дополнительных библиотек. В любом случае в программу необходимо добавить файл включения *windows.h.*

2. Выбор функции для ожидания завершения порожденных процессов зависит от логики работы программы, определяемой вариантом задания. Уничтожение порожденных процессов применяется лишь в тех вариантах, где это действительно необходимо.

3. Для обмена информацией между процессами рекомендуется использовать аргументы командной строки и коды завершения процессов.

4. Примеры кода дочерней (*file_new.cpp*) и родительской (*spaces_new.cpp*) программ доступны по адресу: http://gun.cs.nstu.ru/ssw/API ОС.

Порядок выполнения работы

1. В качестве программы, реализующей порожденный (дочерний) процесс, использовать программу из лабораторной работы 2.

2. Написать и отладить программу, реализующую родительский процесс, порождающий столько дочерних процессов, сколько аргументов командной строки им получено, ожидающий завершения порожденных процессов, получающий результаты выполнения порожденных процессов.

Варианты заданий

Используются варианты из лабораторной работы 1.

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинги программ.

Контрольные вопросы

1. Как осуществляется создание процессов средствами WinAPI?
2. Как получают дескрипторы и идентификаторы порожденных процессов?
3. Для чего и как применяются дескрипторы и идентификаторы порожденных процессов?
4. Чем отличаются процессы и потоки?
5. Назовите функцию ожидания завершения порожденного процесса.
6. Назовите функцию ожидания завершения порожденных процессов.
7. Назовите функции принудительного завершения порожденных процессов.
8. Как получить результат выполнения порожденного процесса?

Тема 4

МЕЖПРОЦЕССНЫЕ КОММУНИКАЦИИ В WINDOWS. КАНАЛЫ

Цель работы: изучить способы и средства обмена информацией между процессами с использованием каналов средствами языка С в операционных системах семейства Windows.

Краткие теоретические сведения

Двумя основными механизмами Windows, реализующими IPC (Interprocess Communication, межпроцессное взаимодействие), являются анонимные и именованные каналы, доступ к которым осуществляется с помощью известных функций `ReadFile` и `WriteFile`.

Анонимные каналы

Анонимные каналы Windows обеспечивают однонаправленное (полудуплексное) посимвольное межпроцессное взаимодействие между родственными процессами. Каждый канал имеет два дескриптора: дескриптор чтения (read handle) и дескриптор записи (write handle).

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, создавшему канал, необходимо вывести в него данные, которые нужны дочернему процессу. Тогда возникает вопрос о том, как передать дочернему процессу дескриптор чтения (`hRead`). Родительский процесс осуществляет это, устанавливая дескриптор стандартного ввода в структуре `STARTUPINFO` для дочерней процедуры равным `*hRead`.

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, выполняемая с использованием буфера в памяти, также будет блокирована.

Поскольку анонимные каналы обеспечивают только однонаправленное взаимодействие, то для двухстороннего взаимодействия необходимы два канала.

Для создания анонимных каналов используется функция CreatePipe, имеющая следующий прототип:

```
BOOL CreatePipe(
   PHANDLE hReadPipe,   // адрес переменной, в которую
                        // будет записан идентификатор
                        // канала  для чтения данных
   PHANDLE hWritePipe,  // адрес переменной,
                        // в которую будет записан
                        // идентификатор канала
                        // для записи данных
   LPSECURITY_ATTRIBUTES lpPipeAttributes,
              // адрес переменной  для атрибутов защиты
   DWORD nSize);      // количество байтов памяти,
                      // зарезервированной для канала
```

Канал может использоваться как для записи в него данных, так и для чтения. Поэтому при создании канала функция CreatePipe возвращает два идентификатора, записывая их по адресу, заданному в параметрах hReadPipe и hWritePipe.

Идентификатор, записанный по адресу hReadPipe, можно передавать в качестве параметра функции ReadFile для выполнения операции чтения. Идентификатор, записанный по адресу hWritePipe, передается функции WriteFile для выполнения операции записи.

Через параметр lpPipeAttributes передается адрес переменной, содержащей атрибуты защиты для создаваемого канала. В наших приложениях мы будем указывать этот параметр как NULL. В результате канал будет иметь атрибуты защиты, принятые по умолчанию.

И, наконец, параметр nSize определяет размер буфера для создаваемого канала. Если этот размер указан как нуль, будет создан буфер с размером, принятым по умолчанию. Заметим, что при необходимости система может изменить указанный вами размер буфера.

В случае успеха функция CreatePipe возвращает значение TRUE, при ошибке – FALSE. В последнем случае для уточнения причины возникновения ошибки вы можете воспользоваться функцией GetLastError.

Запись данных в канал

Запись данных в открытый канал выполняется с помощью функции WriteFile аналогично записи в обычный файл:

```
HANDLE hNamedPipe;
DWORD  cbWritten;
char   szBuf[256];
WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1, &cbWrit-
ten, NULL);
```

Через первый параметр функции WriteFile передается идентификатор реализации канала. Через второй параметр передается адрес буфера, данные из которого будут записаны в канал. Размер этого буфера указывается при помощи третьего параметра. Предпоследний параметр используется для определения количества байтов данных, действительно записанных в канал. И, наконец, последний параметр задан как NULL, поэтому запись будет выполняться в синхронном режиме.

Учтите, что если канал был создан для работы в блокирующем режиме и функция WriteFile работает синхронно (без использования вывода с перекрытием), то эта функция не вернет управление до тех пор, пока данные не будут записаны в канал.

Чтение данных из канала

Для чтения данных из канала можно воспользоваться функцией ReadFile:

```
HANDLE hNamedPipe;
DWORD  cbRead;
char   szBuf[256];
ReadFile(hNamedPipe, szBuf, 256, &cbRead, NULL);
```

Данные, прочитанные из канала hNamedPipe, будут записаны в буфер szBuf, имеющий размер 256 байт. Количество действительно прочитанных байтов данных будет сохранено функцией ReadFile в переменной cbRead. Так как последний параметр функции указан как NULL, используется синхронный режим работы без перекрытия.

Закрытие идентификатора канала

Если канал больше не нужен, процессы должны закрыть его идентификатор функцией `CloseHandle`:

```
CloseHandle(hNamedPipe);
```

Пример применения анонимных каналов

В примере представлен родительский процесс, который создает дочерний процесс и соединяет его с каналом. Родительский процесс устанавливает канал и осуществляет перенаправление стандартного ввода/вывода.

Дескрипторы каналов и потоков должны закрываться при первой же возможности. Родительский процесс должен закрыть дескриптор устройства стандартного вывода сразу же после создания дочернего процесса, чтобы тот мог распознать метку конца файла. В случае существования открытого дескриптора первого процесса второй процесс не смог бы завершиться, поскольку система не обозначила бы конец файла.

Обратите внимание на то, каким образом задается свойство наследования дескрипторов анонимного канала:

```
/* Перенаправить стандартный ввод/вывод. */
STARTUPINFO StartInfoChild;
GetStartupInfo(&StartInfoChild);
StartInfoChild.hStdInput = hReadPipe1;
                        //GetStdHandle(hReadPipe);
StartInfoChild.hStdError = GetStdHandle(STD_ERROR_HANDLE);
StartInfoChild.hStdOutput = hWritePipe2;
StartInfoChild.dwFlags = STARTF_USESTDHANDLES;
```

Дочерний процесс при запуске должен унаследовать потоки ввода/вывода:

```
CreateProcess(NULL, (LPTSTR)Command, NULL, NULL, TRUE /*
Унаследовать    дескрипторы.    */,    0,    NULL,    NULL,
&StartInfoChild, &ProcInfoChild);
```

А вот как организуется перенаправление стандартного ввода/вывода в дочернем процессе:

```
// чтение из канала
hRead= GetStdHandle(STD_INPUT_HANDLE);
ReadFile(hRead, filename, 80, &cbWritten, NULL);
// сообщение в консоль ошибок
```

```
hError= GetStdHandle(STD_ERROR_HANDLE);
WriteFile(hError, message, strlen(message),
                                    &cbWritten, NULL);
// запись в канал
hWrite= GetStdHandle(STD_OUTPUT_HANDLE);
WriteFile(hWrite, message, strlen(message) + 1, &cbWrit-
ten, NULL);
```

В программе *pipe_parent.cpp*, доступной по адресу http://gun.cs.nstu.ru/winprog/Winpipes/, представлена родительская программа, а в программе *pipe_child.cpp* – дочерняя. Запросом родителя является имя текстового файла, ответом дочернего процесса – число пробелов в указанном файле либо сообщение об ошибке его открытия.

Именованные каналы

Когда требуется, чтобы канал связи был двунаправленным, ориентированным на обмен структурированными сообщениями или доступным для нескольких клиентских процессов, следует применять именованные каналы. Кроме того, один именованный канал может иметь несколько открытых дескрипторов.

Функция `CreateNamedPipe` создает первый экземпляр именованного канала и возвращает дескриптор. При вызове этой функции указывается также максимально допустимое количество экземпляров каналов, а следовательно, и количество клиентов, одновременная поддержка которых может быть обеспечена. Как правило, создающий процесс рассматривается в качестве сервера. Клиентские процессы, которые могут выполняться и на других системах, открывают канал с помощью функции `CreateFile`.

Серверами именованных каналов могут быть только системы на основе серверных ОС; системы на базе рабочих станций могут выступать только в роли клиентов.

Прототип функции `CreateNamedPipe`:

```
HANDLE CreateNamedPipe (LPCTSTR lpName, DWORD dwOpenMode,
DWORD dwPipeMode, DWORD nMaxInstances, DWORD nOutBuffer-
Size, DWORD nInBufferSize, DWORD nDefaultTimeOut, LPSE-
CURITY ATTRIBUTES lpSecurityAttributes);
```

Параметры:

`lpName` – указатель на имя канала, который должен иметь форму `\\.\pipe\ [path]pipename`. Точка (.) обозначает локальный компьютер; т. е. создать канал на удаленном компьютере невозможно;

dwOpenMode – указывает один из следующих флагов:

- PIPE_ACCESS_DUPLEX – этот флаг эквивалентен комбинации значений GENERIC_READ и GENERIC_WRITE;
- PIPE_ACCESS_INBOUND – данные могут передаваться только в направлении от клиента к серверу; эквивалентно GENERIC_READ;
- PIPE_ACCESS_OUTBOUND – этот флаг эквивалентен GENERIC_WRITE;

dwPipeMode – имеются три пары взаимоисключающих значений:

- PIPE_TYPE_BYTE и PIPE_TYPE_MESSAGE – указывают соответственно, должны ли данные записываться в канал как поток байтов или как сообщения. Для всех экземпляров каналов с одинаковыми именами следует использовать одно и то же значение;
- PIPE_READMODE_BYTE и PIPE_READMODE_MESSAGE – указывают соответственно, должны ли данные считываться как поток байтов или как сообщения. Значение PIPE_READMODE_MESSAGE требует использования значения PIPE_TYPE_MESSAGE;
- PIPE_WAIT и PIPE_NOWAIT – определяют соответственно, будет или не будет блокироваться операция ReadFile. Рекомендуется использовать значение PIPE_WAIT;

nMaxInstances – определяет количество экземпляров каналов. При каждом вызове функции CreateNamedPipe для данного канала должно использоваться одно и то же значение. Чтобы определить значение этого параметра на основании доступных системных ресурсов, следует указать значение PIPE_UNLIMITED_INSTANCES;

nOutBufferSize и nInBufferSize – позволяют указать размеры (в байтах) выходного и входного буферов именованных каналов. Чтобы использовать размеры буферов по умолчанию, укажите значение ноль;

nDefaultTimeOut – длительность интервала ожидания по умолчанию (в миллисекундах) для функции WaitNamedPipe. В случае ошибки возвращается значение INVALID_HANDLE_VALUE;

lpSecurityAttributes – указатель на атрибуты защиты.

При первом вызове функции CreateNamedPipe происходит создание самого именованного канала. Закрытие последнего открытого дескриптора экземпляра именованного канала приводит к уничтожению этого экземпляра. Уничтожение последнего экземпляра именованного канала приводит к уничтожению самого канала, в результате чего имя канала становится вновь доступным для повторного использования.

Для подключения клиента к именованному каналу применяется функция CreateFile, при вызове которой указывается имя именованного канала. Если клиент и сервер выполняются на одном компьютере, то для указания имени канала используется форма: \\.\pipe\[path]pipename. Если сервер находится на другом компьютере, для указания имени канала используется форма: \\servername\pipe\[path]pipename. Использование точки (.) вместо имени локального компьютера в случае, когда сервер является локальным, позволяет значительно сократить время подключения.

Предусмотрены две функции, позволяющие получать информацию о состоянии каналов, и еще одна функция, позволяющая устанавливать данные состояния канала:

GetNamedPipeHandleState – возвращает для заданного открытого дескриптора информацию относительно того, работает ли канал в блокируемом или неблокируемом режиме, ориентирован ли он на работу с сообщениями или байтами, каково количество экземпляров канала и тому подобное;

GetNamedPipeInfo – определяет, принадлежит ли дескриптор экземпляру клиента или сервера, размеры буферов и прочее.

SetNamedPipeHandleState – позволяет программе устанавливать атрибуты состояния. Параметр режима (NpMode) передается не по значению, а по адресу, что может стать причиной недоразумений.

После создания именованного канала сервер может ожидать подключения клиента (осуществляемого с помощью функции CreateFile), используя для этого функцию ConnectNamedPipe, которая является серверной функцией лишь в случае серверной ОС:

```
Bool ConnectNamedPipe (HANDLE hNamedPipe, LPOVERLAPPED
lpOverlapped);
```

Если параметр lpOverlapped установлен в NULL, то функция ConnectNamedPipe осуществляет возврат сразу же после установления соединения с клиентом. В случае успешного выполнения функции возвращаемым значением является TRUE. Если же подключение клиента происходит между вызовами сервером функций CreateNamedPipe и ConnectNamedPipe, то возвращается значение FALSE, а функция GetLastError вернет значение ERROR_PIPE_CONNECTED.

После возврата из функции ConnectNamedPipe сервер может выполнять чтение запросов с помощью функции ReadFile и запись ответов посредством функции WriteFile. Наконец, сервер должен вызвать

функцию DisconnectNamedPipe, чтобы освободить дескриптор экземпляра канала для соединения с другим клиентом.

Функция WaitNamedPipe, используется клиентами для синхронизации соединений с сервером. Функция осуществляет успешный возврат, когда на сервере имеется незавершенный вызов функции ConnectNamedPipe, указывающий на наличие доступного экземпляра именованного канала. Используя WaitNamedPipe, клиент может убедиться в том, что сервер готов к образованию соединения, после чего можно вызывать функцию CreateFile. Вызов клиентом функции CreateFile может завершиться ошибкой, если в это же время другой клиент открывает экземпляр именованного канала или дескриптор экземпляра закрывается сервером. При этом неудачного завершения вызванной сервером функции ConnectNamedPipe не произойдет. Заметьте, что для функции WaitNamedPipe предусмотрен интервал ожидания, который (если он указан) отменяет значение интервала ожидания, заданного при вызове серверной функции CreateNamedPipe.

Последовательность операций, выполняемых сервером: сервер создает соединение с клиентом, взаимодействует с клиентом до тех пор, пока тот не разорвет единение (вынуждая функцию ReadFile вернуть значение FALSE), разрывает соединение на стороне сервера, образует соединение с другим клиентом:

```
hNp = CreateNamedPipe ("\\\\. \\pipe\\my_pipe",    ...);
while   ( /*Цикл до завершения работы сервера.*/)
{
ConnectNamedPipe (hNp, NULL);
while (ReadFile (hNp, Request, ...) {
WriteFile (hNp, Response, ...);  }
DisconnectNamedPipe (hNp); }
CloseHandle (hNp);
```

Последовательность операций, выполняемых клиентом:

```
WaitNamedPipe("\\\\ServerName\\pipe\\my_pipe",
                                  NMPWAIT_WAIT_FOREVER);
hNp = CreateFile("\\\\ServerName\\pipe\\my_pipe",...);
while ()/*Цикл, пока не прекратятся запросы.*/
{ WriteFile (hNp, Request,...);
ReadFile  (hNp, Response); }
CloseHandle (hNp);
```

В программе *namedpipeclient.cpp*, доступной по адресу http://gun.cs.nstu.ru/ssw/Winpipes/, представлен клиент, а в программе *namedpipeserver.cpp* – сервер. Запросом клиента является имя текстового файла, ответом сервера – число пробелов в указанном файле или сообщение об ошибке его открытия.

Методические указания

1. Проект может быть реализован на Visual C++ 6.0 или в среде Borland C++ 5.0 и выше. В первом случае выбирается консольное приложение Win32 без дополнительных библиотек. В любом случае в программу необходимо добавить файл включения *windows.h*.

2. Для обмена данными между родственными процессами рекомендуется использовать анонимные каналы. Использование именованных каналов соответствует заданию повышенной сложности.

Порядок выполнения работы

1. Модифицировать и отладить программу из лабораторной работы 2, реализующую порожденный процесс – клиентское приложение, для приема параметров не через аргументы командной строки, а через канал.

2. Модифицировать и отладить программу из лабораторной работы 3, реализующую родительский процесс, вызывающий и отслеживающий состояние порожденных процессов – клиентов, передающий им параметры и получающий результаты выполнения порожденных процессов через канал.

Варианты заданий

Используются варианты из лабораторной работы 1. Применение именованных каналов соответствует заданию повышенной сложности.

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинги программ.

Контрольные вопросы

1. Что такое каналы? В чем отличие неименованных и именованных каналов?
2. Какие каналы и когда можно применять для коммуникации процессов в локальной сети?
3. Каков порядок работы с анонимными каналами?
4. Какова последовательность операций родительского процесса с каналом?
5. Какова последовательность операций дочернего процесса с каналом?
6. Порядок работы с именованными каналами.
7. Какова последовательность операций сервера с каналом?
8. Какова последовательность операций клиента с каналом?
9. Как оценить состояние канала и получить о нем информацию?
10. Как определить размер сообщения в канале?

Тема 5

МЕЖПРОЦЕССНЫЕ КОММУНИКАЦИИ В WINDOWS. ПОЧТОВЫЕ ЯЩИКИ

Цель работы: изучить способы и средства обмена информацией между процессами с использованием почтовых ящиков (каналов mailslot) средствами языка С в операционных системах семейства Windows.

Краткие теоретические сведения

Как и именованные каналы, почтовые ящики (mailslots) Windows являются объектами IPC и снабжаются именами, которые могут быть использованы для обеспечения взаимодействия между независимыми процессами. В отличие от каналов почтовые ящики представляют собой широковещательный механизм, основанный на дейтаграммах.

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 сервера попытка открытия почтового ящика завершится ошибкой 2 («не удается найти указанный файл»).

Сообщение клиента может быть прочитано всеми серверами, создавшими почтовый ящик с этим именем; все серверы получают одно и

то же сообщение. В вызове функции CreateFile клиент может уточнить имя почтового ящика, указав конкретный сервер или любой сервер домена или рабочей группы.

Создание почтового ящика

Для создания почтового ящика серверы (программы считывания) вызывают функцию CreateMailslot:

```
HANDLE CreateMailslot(LPCTSTR lpName, DWORD cbMaxMsg,
DWORD dwReadTimeout, LPSECURITY_ATTRIBUTES lpsa)
```

Здесь lpName – указатель на строку с именем почтового ящика, которая должна иметь следующий вид: \\.\mailslot\[путь]*имя*. Имя должно быть уникальным. Точка (.) указывает на то, что почтовый ящик создается на локальном компьютере;

cbMaxMsg – максимальный размер сообщения (в байтах), которые может записывать клиент. Значению ноль соответствует отсутствие ограничений;

dwReadTimeOut – длительность интервала ожидания (в миллисекундах) для операции чтения. Значению ноль соответствует немедленный возврат, а значению MAILSLOT_WAIT_FOREVER – неопределенный период ожидания (который может длиться сколь угодно долго).

Параметр lpSecurityAttributes задает адрес структуры защиты, по умолчанию задается как NULL.

При ошибке функцией CreateMailslot возвращается значение INVALID_HANDLE_VALUE. Код ошибки можно определить при помощи функции GetLastError.

Открытие канала Mailslot

Прежде чем приступить к работе с каналом Mailslot, клиентский процесс должен его открыть. Для выполнения этой операции следует использовать функцию CreateFile, например, так:

```
LPSTR   lpszMailslotName = "\\\\.\\mailslot\\$Mailslot-
Name$";
hMailslot = CreateFile( lpszMailslotName, GENERIC_WRITE,
  FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
```

Здесь в качестве первого параметра функции CreateFile передается имя канала на текущей рабочей станции в сети. В качестве второго

параметра функции передается константа GENERIC_WRITE. Эта константа определяет, что над открываемым каналом будет выполняться операция записи. Клиентский процесс может только посылать сообщения в канал Mailslot.

Третий параметр указан как FILE_SHARE_READ, так как сервер может читать сообщения, посылаемые одновременно несколькими клиентскими процессами.

Константа OPEN_EXISTING используется потому, что функция CreateFile открывает существующий канал, а не создает новый.

Запись сообщений в канал Mailslot

Запись сообщений в канал Mailslot выполняет клиентский процесс, вызывая для этого функцию WriteFile:

```
HANDLE hMailslot;
char   szBuf[512];
DWORD  cbWritten;
WriteFile(hMailslot, szBuf, strlen(szBuf) + 1,
&cbWritten, NULL);
```

В качестве первого параметра этой функции необходимо передать идентификатор канала Mailslot, полученный от функции CreateFile. Второй параметр определяет адрес буфера с сообщением, третий – размер сообщения. Если сообщения передаются в виде текстовой строки, закрытой двоичным нулем, то для определения длины сообщения используется функция strlen.

Чтение сообщений из канала Mailslot

Серверный процесс может читать сообщения из созданного им канала Mailslot при помощи функции ReadFile, как это показано ниже:

```
HANDLE hMailslot;
char   szBuf[512];
DWORD  cbRead;
ReadFile(hMailslot, szBuf, 512, &cbRead, NULL);
```

Через первый параметр функции ReadFile передается идентификатор созданного ранее канала Mailslot, полученный от функции CreateMailslot. Второй и третий параметры задают соответственно адрес буфера для сообщения и его размер.

Заметим, что перед выполнением операции чтения следует проверить состояние канала Mailslot. Если в нем нет сообщений, то функцию ReadFile вызывать не следует. Для проверки состояния канала необходимо воспользоваться функцией GetMailslotInfo, описанной ниже.

Определение состояния канала Mailslot

Прототип функции GetMailslotInfo:

```
BOOL GetMailslotInfo(
  HANDLE  hMailslot,           // идентификатор канала
                               // Mailslot
  LPDWORD lpMaxMessageSize,    // адрес максимального
                               // размера сообщения
  LPDWORD lpNextSize,        // адрес размера следующего
                               // сообщения
  LPDWORD lpMessageCount,      // адрес количества
                               // сообщений
  LPDWORD lpReadTimeout);      // адрес времени ожидания
```

Через параметр hMailslot функции передается идентификатор канала Mailslot, состояние которого необходимо определить. В переменную, адрес которой передается через параметр lpMaxMessageSize, после возвращения из функции GetMailslotInfo будет записан максимальный размер сообщения (из нескольких). В переменную, адрес которой указан через параметр lpNextSize, записывается размер следующего сообщения, если оно есть в канале. Если же в канале больше нет сообщений, в эту переменную будет записана константа MAILSLOT_NO_MESSAGE. С помощью параметра lpMessageCount можно определить количество сообщений, записанных в канал клиентскими процессами. Если это количество равно нулю, то не следует вызывать функцию ReadFile для чтения несуществующего сообщения. В переменную, адрес которой задается в параметре lpReadTimeout, записывается текущее время ожидания, установленное для канала (в миллисекундах).

В случае успешного завершения функция GetMailslotInfo возвращает значение TRUE, а при ошибке – FALSE. Код ошибки можно получить, вызвав функцию GetLastError.

Изменение состояния канала Mailslot

С помощью функции `SetMailslotInfo` серверный процесс может изменить время ожидания для канала Mailslot уже после его создания. Прототип функции `SetMailslotInfo`:

```
BOOL SetMailslotInfo(
HANDLE hMailslot,      // идентификатор канала Mailslot
DWORD  dwReadTimeout); // время ожидания
```

Через параметр `hMailslot` передается идентификатор канала Mailslot, для которого нужно изменить время ожидания. Новое значение времени ожидания в миллисекундах задается через параметр `dwReadTimeout`. Можно указать здесь константы «0» или `MAILSLOT_WAIT_FOREVER`. В первом случае функции, работающие с каналом, вернут управление немедленно, во втором – будут находиться в состоянии ожидания до тех пор, пока не завершится выполняемая операция.

В программе *mslotclient.cpp*, доступной по адресу http://gun.cs.nstu.ru/winprog/mailslots/, представлен клиент, а в программе *mslotserver.cpp* – сервер. Запросом клиента является имя текстового файла, ответом сервера – число пробелов в указанном файле или сообщение об ошибке его открытия.

Методические указания

1. Проект может быть реализован на Visual C++ 6.0 или в среде Borland C++ 5.0 и выше. В первом случае выбирается консольное приложение Win32 без дополнительных библиотек. В любом случае в программу необходимо добавить файл включения *windows.h*.

2. Для двустороннего обмена данными между процессами каждый из них должен быть и клиентом, и сервером почтовых ящиков.

Порядок выполнения работы

1. Модифицировать и отладить родительскую программу из лабораторной работы 4, реализующую клиентское приложение, передающее входные данные не в канал, а в почтовый ящик.

2. Модифицировать и отладить дочернюю программу из лабораторной работы 4, реализующую серверный процесс, ожидающий подключения клиентов, реализующий обработку полученных данных по варианту и возвращающий результаты выполнения клиентским процессам через средства межзадачных коммуникаций.

Варианты заданий

Используются варианты из лабораторной работы 1.

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинги программ.

Контрольные вопросы

1. Что такое почтовые ящики? В чем их отличие от каналов?
2. Какие имена почтовых ящиков можно применять для коммуникации процессов в локальной сети?
3. Какова последовательность операций сервера с почтовым ящиком?
4. Какова последовательность операций клиента с почтовым ящиком?
5. Каков порядок работы с почтовыми ящиками при двунаправленном обмене информацией?
6. Как оценить состояние почтового ящика и получить о нем информацию?
7. Как и какое состояние почтового ящика можно изменить?
8. Как определить количество и размеры сообщений в почтовом ящике?

Тема 6

МЕЖПРОЦЕССНЫЕ КОММУНИКАЦИИ В WINDOWS. СОБЫТИЯ И СЕМАФОРЫ

Цель работы: изучить способы и средства обмена информацией между процессами с использованием отображаемых на память файлов и их синхронизации с помощью событий, мьютексов или семафоров средствами языка С в операционных системах семейства Windows.

Краткие теоретические сведения

Как и почтовые ящики, события, мьютексы и семафоры Windows являются объектами IPC и снабжаются именами, которые могут быть использованы для обеспечения синхронизации между неродственными процессами. В отличие от каналов и почтовых ящиков события и семафоры представляют собой средство синхронизации, а не обмена данными. Для обмена данными используются файлы, отображаемые на память.

Файлы, отображаемые на память

Методика использования файлов, отображенных на память, для передачи данных между процессами заключается в следующем.

Один из процессов создает такой файл с помощью функции `CreateFileMapping`, задавая при этом имя отображения:

```
HANDLE CreateFileMapping(
   HANDLE hFile,           // идентификатор
                           // отображаемого файла
   LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
                           // дескриптор защиты
```

```
  DWORD flProtect,          // защита
                            // для отображаемого файла
DWORD dwMaximumSizeHigh,    // размер файла (старшее
                            // слово)
DWORD dwMaximumSizeLow,     // размер файла (младшее
                            // слово)
LPCTSTR lpName);            // имя отображенного файла
```

Через параметр `hFile` этой функции нужно передать идентификатор файла, для которого будет выполняться отображение в память, или значение *0xFFFFFFFF*. В первом случае функция `CreateFileMapping` отобразит заданный файл в память, а во втором – создаст отображение с использованием файла виртуальной памяти. Отображение с использованием файла виртуальной памяти удобно для организации передачи данных между процессами.

Заметим, что если функция `CreateFile` завершится с ошибкой и эта ошибка не будет обработана приложением, функция `CreateFileMapping` получит через параметр `hFile` значение `INVALID_HANDLE_VALUE`, численно равное *0xFFFFFFFF*. В этом случае она вместо того, чтобы выполнить отображение файла в память, создаст отображение с использованием файла виртуальной памяти.

Параметр `lpFileMappingAttributes` задает адрес дескриптора защиты. В большинстве случаев для этого параметра вы можете указать значение `NULL`.

Параметр `flProtect` задает защиту для создаваемого отображения файла: `PAGE_READONLY`, `PAGE_READWRITE` или `PAGE_WRITECOPY`.

С помощью параметров `dwMaximumSizeHigh` и `dwMaximumSizeLow` необходимо указать 64-разрядный размер файла. Заметим, что можно указать нулевые значения для обоих этих параметров. В этом случае предполагается, что размер файла изменяться не будет.

Через параметр `lpName` можно указать имя отображения, которое будет доступно *всем работающим одновременно приложениям*, в виде текстовой строки, закрытой двоичным нулем и не содержащей символов “\”.

Так как *имя отображения глобально*, возможно возникновение ситуации, когда процесс пытается создать отображение с уже существующим именем. В этом случае функция `CreateFileMapping` возвращает идентификатор существующего отображения. Такую ситуацию можно определить с помощью функции `GetLastError`, вызвав ее сразу после

функции CreateFileMapping. Функция GetLastError при этом вернет значение ERROR_ALREADY_EXISTS.

Получив от функции CreateFileMapping идентификатор объекта-отображения, необходимо выполнить само отображение, вызвав для этого функцию MapViewOfFile:

```
LPVOID MapViewOfFile(
HANDLE hFileMappingObject,  // идентификатор отображения
   DWORD dwDesiredAccess,   // режим доступа
   DWORD dwFileOffsetHigh,  // смещение в файле
                            // (старшее слово)
   DWORD dwFileOffsetLow,   // смещение в файле
                            // (младшее слово)
   DWORD dwNumberOfBytesToMap);// количество отображаемых
                               // байтов
```

Функция MapViewOfFile создает окно размером dwNumberOfBytesToMap байтов, которое смещено относительно начала файла на количество байтов, заданное параметрами dwFileOffsetHigh и dwFileOffsetLow. Если задать значение параметра dwNumberOfBytesToMap равным нулю, будет выполнено отображение всего файла.

Параметр dwDesiredAccess определяет требуемый режим доступа к отображению, т. е. режимы доступа для страниц виртуальной памяти, используемых для отображения. Для этого параметра вы можете указать одно из следующих значений: FILE_MAP_WRITE, FILE_MAP_READ, FILE_MAP_ALL_ACCESS и FILE_MAP_COPY. Последнее обеспечит доступ для копирования при записи. Для этого при создании отображения необходимо указать атрибут PAGE_WRITECOPY.

Функция вернет адрес начала отображенной области памяти. При ошибке возвращается значение NULL.

Другие процессы могут воспользоваться именем отображения, указанным в последнем аргументе функции CreateFileMapping, открыв созданный ранее файл в виртуальной памяти с помощью функции OpenFileMapping:

```
HANDLE OpenFileMapping(
   DWORD  dwDesiredAccess, // режим доступа
   BOOL   bInheritHandle,  // флаг наследования
   LPCTSTR lpName);       // адрес имени отображения файла
```

Через параметр lpName этой функции следует передать имя открываемого отображения. Имя должно быть задано точно так же, как при создании отображения функцией CreateFileMapping. Параметр dwDesiredAccess определяет требуемый режим доступа к отображению и указывается точно так же, как и для описанной выше функции MapViewOfFile. Параметр bInheritHandle определяет возможность наследования идентификатора отображения. Если он равен TRUE, порожденные процессы могут наследовать идентификатор, если FALSE – то нет.

С помощью отображения, выполненного функцией MapViewOfFile, оба процесса могут получить указатели на область памяти, для которой выполнено отображение, и эти указатели будут ссылаться на одни и те же страницы виртуальной памяти. Обмениваясь данными через эту область (выполняя операции чтения/записи по адресу), процессы должны обеспечить синхронизацию своей работы, например с помощью событий, мьютексов или семафоров (в зависимости от логики процесса обмена данными). Если отображение файла на память больше не нужно, его следует отменить с помощью функции UnmapViewOfFile:

```
BOOL UnmapViewOfFile(LPVOID lpBaseAddress);
```

Через единственный параметр этой функции необходимо передать адрес области отображения, полученный от функции MapViewOfFile.

В случае успеха функция возвращает значение TRUE. При этом гарантируется, что все измененные страницы оперативной памяти, расположенные в отменяемой области отображения, будут записаны на диск в отображаемый файл. При ошибке функция возвращает значение FALSE.

События

События – самая примитивная разновидность объектов ядра. События используются для того, чтобы сигнализировать другим процессам/потокам, например, о появлении нового сообщения. Важной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько процессов/потоков.

Схема использования событий достаточно проста. Один из процессов создает объект-событие, вызывая для этого функцию CreateEvent:

```
HANDLE CreateEvent(
   LPSECURITY_ATTRIBUTES lpsa,    // атрибут безопасности
   BOOL bManualReset,             // тип события
   BOOL bInitialState,            // начальное состояние
   LPTCSTR lpEventName);          // адрес глобального
                                  // имени события
```

Чтобы создать событие, сбрасываемое вручную, необходимо установить значение параметра `bManualReset` равным `True`. Точно так же, чтобы сделать начальное состояние события сигнальным, установите равным `True` значение параметра `bInitialState`.

При этом событие имеет имя, доступное всем активным процессам. В случае успешного завершения функция `CreateEvent` возвращает идентификатор события, которым нужно будет пользоваться при выполнении всех операций над объектом-событием. При ошибке возвращается значение `NULL`.

Вызывая функции `WaitForSingleObject` или `WaitForMultipleObjects`, процесс может выполнять ожидание момента, когда событие перейдет в отмеченное состояние.

Другой поток, принадлежащий тому же самому или другому процессу, может получить идентификатор события по его имени, например с помощью функции `OpenEvent`:

```
HANDLE OpenEvent(
   DWORD fdwAccess,  // флаги доступа
   BOOL fInhent,  // флаг наследования
   LPCTSTR lpEventName);  // адрес глобального имени события
```

Флаги доступа, передаваемые через параметр `fdwAccess`, определяют требуемый уровень доступа к объекту-событию. Этот параметр может быть комбинацией следующих значений: `EVENT_ALL_ACCESS`, `EVENT_MODIFY_STATE` (событие можно использовать только для функций `SetEvent` и `ResetEvent`), `SYNCHRONIZE` (событие можно использовать в любых функциях ожидания события). Параметр `fInherit` определяет возможность наследования полученного идентификатора. Если этот параметр равен `TRUE`, идентификатор может наследоваться дочерними процессами. Если же он равен `FALSE`, наследование не допускается.

Далее, пользуясь функциями `SetEvent(HANDLE hEvent)`, `ResetEvent(HANDLE hEvent)` или `PulseEvent(HANDLE hEvent)`, этот процесс может изменить состояние события, соответственно устанавливая его в отмеченное состояние, сбрасывая, устанавливая в отмеченное состояние с последующим сбросом события в неотмеченное состояние.

Семафоры

В отличие от других объектов IPC семафоры позволяют обеспечить доступ к ресурсу для заранее определенного, ограниченного приложением количества задач. Все остальные задачи, пытающиеся получить доступ сверх установленного лимита, будут переведены при этом в состояние ожидания до тех пор, пока какая-либо задача, получившая доступ к ресурсу раньше, не освободит ресурс, связанный с данным семафором. С каждым семафором связывается счетчик, начальное и максимальные значения которого задаются при создании семафора. Значение этого счетчика уменьшается, когда задача вызывает для семафора функцию `WaitForSingleObject` или `WaitForMultipleObjects`, и увеличивается при вызове другой, специально предназначенной для этого функции.

Так же как и другие объекты IPC, семафор может находиться в отмеченном или неотмеченном состоянии. Если значение счетчика семафора равно нулю, он находится в неотмеченном состоянии. Если же это значение больше нуля, семафор переходит в отмеченное состояние.

Для создания семафора приложение должно вызвать функцию `CreateSemaphore`:

```
HANDLE CreateSemaphore(
   LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
                                        // атрибуты защиты
   LONG lInitialCount,  // начальное значение счетчика
                        // семафора
   LONG lMaximumCount,  // максимальное значение счетчика
                        // семафора
   LPCTSTR  lpName);    // адрес строки с именем семафора
```

В качестве атрибутов защиты можно передать значение `NULL`. Через параметры `lInitialCount` и `lMaximumCount` передаются соответственно начальное и максимальное значения счетчика, связанного с создаваемым семафором. Начальное значение счетчика `lInitialCount`

должно быть больше или равно нулю и не должно превосходить максимальное значение счетчика, передаваемое через параметр lMaximumCount.

Имя семафора указывается аналогично имени рассмотренного ранее объекта-события с помощью параметра lpName.

В случае удачного создания семафора функция CreateSemaphore возвращает его идентификатор. В случае возникновения ошибки возвращается значение NULL, при этом код ошибки можно узнать при помощи функции GetLastError.

Когда необходимо синхронизовать задачи разных процессов, следует определить имя семафора. При этом один процесс создает семафор с помощью функции CreateSemaphore, а второй открывает его, получая идентификатор для уже существующего семафора, функцией OpenSemaphore:

```
HANDLE OpenSemaphore(
   DWORD   fdwAccess,             // требуемый доступ
   BOOL    fInherit,              // флаг наследования
   LPCTSTR lpszSemaphoreName );   // адрес имени семафора
```

Флаги доступа, передаваемые через параметр fdwAccess, определяют требуемый уровень доступа к семафору. Этот параметр может быть комбинацией следующих значений: SEMAPHORE_ALL_ACCESS, SEMAPHORE_MODIFY_STATE (семафор можно использовать для функции ReleaseSemaphore), SYNCHRONIZE (семафор можно использовать в любых функциях ожидания).

Параметр fInherit определяет возможность наследования полученного идентификатора. Если этот параметр равен TRUE, идентификатор может наследоваться дочерними процессами. Если же он равен FALSE, наследование не допускается.

Через параметр lpszSemaphoreName необходимо передать функции адрес символьной строки, содержащей имя семафора.

Если семафор открыт успешно, функция OpenSemaphore возвращает его идентификатор. При ошибке возвращается значение NULL. Код ошибки можно определить при помощи функции GetLastError.

Для увеличения значения счетчика семафора на значение, указанное в параметре cReleaseCount, приложение должно использовать функцию ReleaseSemaphore:

```
BOOL ReleaseSemaphore(
   HANDLE hSemaphore,          // идентификатор семафора
   LONG   cReleaseCount,       // значение инкремента
   LPLONG lplPreviousCount);  // адрес переменной для записи
              // предыдущего значения счетчика семафора
```

Заметим, что через параметр `cReleaseCount` можно передавать только положительное значение, большее нуля. При этом если в результате увеличения новое значение счетчика должно будет превысить максимальное значение, заданное при создании семафора, функция `ReleaseSemaphore` возвращает признак ошибки и не изменяет значение счетчика.

Для уменьшения значения семафора задача вызывает функции ожидания, такие как `WaitForSingleObject` или `WaitForMultipleObjects`. Если задача вызывает несколько раз функцию ожидания для одного и того же семафора, содержимое его счетчика каждый раз будет уменьшаться.

Для уничтожения семафора необходимо передать его идентификатор функции `CloseHandle`. Заметим, что при завершении процесса все созданные им семафоры уничтожаются автоматически.

В программе *mfe_client.cpp*, доступной по адресу http://gun.cs.nstu.ru/winprog/mapping/, представлено клиентское приложение, а в программе *mfe_server.cpp* – серверное. Запросом клиента является имя текстового файла, ответом сервера – число пробелов в указанном файле или сообщение об ошибке его открытия. Обмен данными производится через виртуальный файл, отображаемый на память. Синхронизация клиентского и серверного процессов осуществляется с помощью событий.

В программе *mfs_client.cpp*, доступной по адресу http://gun.cs.nstu.ru/ssw/Mapping/, представлено клиентское приложение, а в программе *mfs_server.cpp* – серверное. Запросом клиента, как и в предыдущем случае, является имя текстового файла, ответом сервера – число пробелов в указанном файле или сообщение об ошибке его открытия. Обмен данными производится через виртуальный файл, отображаемый на память. Синхронизация клиентского и серверного процессов осуществляется с помощью семафоров.

Методические указания

1. Проект может быть реализован на Visual C++ 6.0 или в среде Borland C++ 5.0 и выше. В первом случае выбирается консольное приложение Win32 без дополнительных библиотек. В любом случае в программу необходимо добавить файл включения *windows.h*.

2. Для двустороннего обмена данными между процессами используются отображаемые на память файлы, а для их синхронизации – события, мьютексы или семафоры.

Порядок выполнения работы

1. Модифицировать и отладить программу из лабораторной работы 5, реализующую клиентское приложение.

2. Модифицировать и отладить программу из лабораторной работы 5, реализующую серверный процесс, ожидающий подключения клиентов, реализующий бизнес-логику и возвращающий результаты выполнения клиентским процессам через средства межзадачных коммуникаций.

Варианты заданий

Используются варианты из лабораторной работы 1. Использование мьютексов или семафоров соответствует заданию повышенной сложности.

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинги программ.

Контрольные вопросы

1. Перечислите функции, необходимые для работы с реальным файлом, отображаемым на память.
2. Что такое события? В чем их отличие от других средств IPC?
3. Чем отличаются события, сбрасываемые вручную и автоматически?
4. Какие существуют флаги доступа к объекту-событию?
5. В каких случаях необходимо выполнять сброс события функцией `ResetEvent`?
6. Когда необходим вызов функции `PulseEvent`?
7. Как и когда необходимо уничтожать объекты-события?
8. Что такое семафоры? В чем их отличие от других средств IPC?
9. Как задать начальное и максимальное значение счетчика, связанного с создаваемым семафором?
10. Какие существуют флаги доступа к объекту-семафору?
11. Когда необходим вызов функции `ReleaseSemaphore`?
12. Как и когда необходимо уничтожать объекты-семафоры?

Тема 7

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В WINDOWS

Цель работы: изучить способы и средства реализации параллельно выполняющихся потоков средствами языка С в операционных системах семейства Windows и обеспечить их синхронизацию с помощью критических секций и мьютексов.

Краткие теоретические сведения

Создание потока

Создать поток можно с помощью функции `CreateThread`:

```
HANDLE CreateThread(
   LPSECURITY_ATTRIBUTES lpThreadAttributes
   DWORD dwStackSize,
   LPTHREAD_START_ROUTINE lpStartAddress,
   LPVOID lpParameter,
   DWORD dwCreationFlags,
   LPDWORD lpThreadId    );
```

Здесь lpSecurityAttributes – атрибут защиты, обычно устанавливается в нуль, чтобы использовать заданный по умолчанию;

`dwStackSize` – размер стека. Каждый поток имеет собственный стек;

`lpStartAddress` – адрес памяти, где стартует поток. Он должен быть равен адресу функции (адрес функции – ее имя);

`lpParameter` – параметр, который передается функции нового потока;

`dwCreationFlags` – переменная флагов, позволяющая управлять запуском потока (активный, приостановленный и т. д.);

`lpThreadId` – переменная, в которую загружается идентификатор нового потока.

Пример применения:

```
for (i=0;i<num;i++)  {
hThread[i]=CreateThread(NULL, //атрибутов
                              // безопасности нет
0,                       // размер стека – по умолчанию
(LPTHREAD_START_ROUTINE) unit, // функция потока
(LPVOID)i,             //аргумент функции потока
0,               // флаг создания – по умолчанию
&IDThread);      //возвращаемый идентификатор
                 // созданного потока
if (hThread[i] == NULL)
printf("Ошибка создания потока #%d\n", i);
else
printf("Указатель %Lu потока#%d\n",hThread[i], i); }
```

Уничтожение потока

Уничтожить поток можно с помощью следующей функции:

```
BOOL WINAPI TerminateThread(
   HANDLE hThread, // указатель на уничтожаемый объект
   DWORD dwExitCode // код завершения
   );
```

Ожидание завершения потоков

Поток может дожидаться завершения выполнения другого потока. При вызове функций ожидания (`WaitForSingleObject` и `WaitForMultipleObjects`) следует использовать дескрипторы потоков.

Допустимое количество объектов, одновременно ожидаемых функцией `WaitForMultipleObjects`, ограничено значением `MAXIMUM WAIT OBJECTS` (64), но при большом количестве потоков можно воспользоваться серией вызовов функций ожидания.

Функция ожидания дожидается, пока объект, указанный дескриптором, не перейдет в *сигнальное* состояние. Поток переводится в сигнальное состояние при помощи функций `ExitThread` и `TerminateThread`. Функция `ExitProcess` переводит в сигнальное состояние как сам процесс, так и все его потоки.

Синхронизация потоков

Windows предоставляет четыре объекта, предназначенных для синхронизации потоков и процессов. Три из них – события, семафоры и мьютексы – объекты ядра, имеющие дескрипторы. Первые два рассмотрены для синхронизации процессов, а мьютексы и четвертый объект – критические участки кода (локальный для приложения) рассмотрены ниже.

Критические секции

Объект критического участка кода – это участок программного кода, который каждый раз должен выполняться только одним потоком; параллельное выполнение этого участка несколькими потоками может приводить к непредсказуемым или неверным результатам.

Объекты `CRITICAL_SECTION` (CS) можно инициализировать и удалять, но они не имеют дескрипторов и не могут совместно использоваться другими процессами. Объекты должны объявляться как переменные типа `CRITICAL_SECTION`. Потоки входят в объекты CS и покидают их, но выполнение кода отдельного объекта CS каждый раз разрешено только одному потоку. Вместе с тем один и тот же поток может входить в несколько отдельных объектов CS и покидать их, если они расположены в разных местах программы.

Для инициализации и удаления переменной типа `CRITICAL_SECTION` используются следующие функции:

```
VOID InitializeCriticalSection (
LPCRITICAL SECTION lpCriticalSection);
VOID DeleteCriticalSection (
LPCRITICAL_SECTION lpCriticalSection);
```

Функция `EnterCriticalSection` блокирует поток, если на данном критическом участке кода присутствует другой поток. Ожидающий поток разблокируется после того как другой поток выполнит функцию `LeaveCriticalSection`. Говорят, что поток *получил права владения* объектом CS, если произошел возврат из функции `EnterCriticalSection`, тогда как для уступки прав владения используется функция `LeaveCriticalSection`:

```
VOID EnterCriticalSection (
LPCRITICAL_SECTION lpCriticalSection);
VOID LeaveCriticalSection (
LPCRITICAL_SECTION lpCriticalSection);
```

Поток, владеющий объектом CS, может повторно войти в этот же CS без его блокирования; таким образом, объекты CS являются *рекурсивными*. Поддерживается счетчик вхождений в объект CS, и поэтому поток должен покинуть CS столько раз, сколько было вхождений в него, чтобы разблокировать этот объект для других потоков. Выход из объекта CS, которым данный поток не владеет, может привести к непредсказуемым результатам, включая блокирование самого потока.

Для возврата из функции `EnterCriticalSection` нет конечного интервала ожидания; другие потоки будут блокированы на неопределенное время, пока поток, владеющий объектом CS, не покинет его. Используя функцию `TryEnterCriticalSection`, можно тестировать (опросить) CS, чтобы проверить, не владеет ли им другой поток:

```
   BOOL TryEnterCriticalSection (LPCRITICAL_SECTION
lpCriticalSection);
```

Возврат функцией `TryEnterCriticalSection` значения `True` означает, что вызывающий поток приобрел права владения критическим участком кода, тогда как возврат значения `False` говорит о том, что данный критический участок кода уже принадлежит другому потоку.

Одним из наиболее распространенных способов применения объектов CS является обеспечение доступа потоков к разделяемым глобальным переменным. Рассмотрим пример:

```
CRITICAL_SECTION csl;
volatile DWORD N = 0, M;
/* N - глобальная переменная */
InitializeCriticalSection (&csl);
EnterCriticalSection (&csl);
if (N < N_MAX) { M = N; M += 1; N = M; }
LeaveCriticalSection (&csl);
DeleteCriticalSection (&csl);
```

Мьютексы

Объект взаимного исключения (mutual exception), или мьютекс (mutex), обеспечивает более универсальную функциональность по сравнению с другими объектами. Поскольку мьютексы могут иметь имена и дескрипторы, их можно использовать также для синхронизации потоков, принадлежащих различным процессам.

Поток приобретает права владения мьютексом (блокирует мьютекс) путем вызова функции ожидания (WaitForSingleObject или WaitForMultipleObjects) по отношению к дескриптору мьютекса и уступает эти права посредством вызова функции ReleaseMutex.

При работе с мьютексами в потоках используются функции CreateMutex и ReleaseMutex. Рассмотрим их подробнее:

```
HANDLE CreateMutex (
LPSECURITY_ATTRIBUTES lpsa, // атрибут безопасности
BOOL bInitialOwner, // начальное владение мьютексом
LPCTSTR lpMutexName); // имя мьютекса
```

Если bInitialOwner равно True, то вызывающий поток немедленно приобретает права владения новым мьютексом. Эта атомарная операция предотвращает приобретение прав владения мьютексом другими потоками, прежде чем это сделает поток, создающий мьютекс. Флаг не оказывает никакого действия, если мьютекс уже существует;

lpMutexName – указатель на строку, содержащую имя мьютекса; в отличие от файлов имена мьютексов чувствительны к регистру. Если этот параметр равен NULL, то мьютекс создается без имени. Длина имен объектов не может превышать 260 символов.

Возвращаемое значение имеет тип HANDLE; значение NULL указывает на неудачное завершение функции.

Возможно возникновение такой ситуации, когда приложение пытается создать мьютекс с именем, которое уже используется в системе. В этом случае функция CreateMutex вернет идентификатор существующего объекта, а функция GetLastError, вызванная сразу после вызова функции CreateMutex, вернет значение ERROR_ALREADY_EXISTS.

Функция ReleaseMutex освобождает мьютекс. Если мьютекс не принадлежит потоку, функция завершается с ошибкой:

```
BOOL ReleaseMutex (HANDLE hMutex);
```

Мьютекс, владевший которым поток завершился, не освободив его, называют покинутым. На то, что дескриптор представляет собой покинутый мьютекс, указывает возврат функцией WaitForSingleObject значения WAIT_ABANDONED_0 или использование значения WAIT_ABANDONED_0 в качестве базового значения функцией WaitForMultipleObject. Покинутый мьютекс переходит в сигнальное состояние (освобождается).

Пример применения критических секций для синхронизации потоков – программа *csthreads.cpp* из папки «*WinThreads*» сайта дисциплины.

Пример применения мьютексов для синхронизации потоков – программа *mthreads.cpp* из папки «*WinThreads*» сайта дисциплины.

Методические указания

1. Проект может быть реализован на Visual C++ 6.0 или в среде Borland C++ 5.0 и выше. В первом случае выбирается консольное приложение Win32 без дополнительных библиотек. В любом случае в программу необходимо добавить файл включения *windows.h*.

2. Выбор функции для ожидания завершения порожденных потоков зависит от логики работы программы, определяемой вариантом задания.

3. Для обмена информацией между потоками рекомендуется использовать параметры функций, выполняемых в потоках, возвращаемые значения функций потоков, а для их синхронизации – критические секции или взаимные исключения (мьютексы).

Порядок выполнения работы

1. Написать и отладить функцию, реализующую порожденный поток, при необходимости обеспечить синхронизацию потоков через критические секции или мьютексы. Применение мьютексов соответствует задаче повышенной сложности.

2. Написать и отладить программу, реализующую родительский процесс, вызывающий и отслеживающий состояние порожденных потоков (функций) (ждущий их завершения или уничтожающий их, в зависимости от варианта), получающий результаты выполнения порожденных потоков.

Варианты заданий

Используются варианты из лабораторной работы 1. Использование мьютексов соответствует заданию повышенной сложности.

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинги программ.

Контрольные вопросы

1. Чем отличаются процессы и потоки?
2. Назовите функции ожидания завершения порожденных потоков.
3. Назовите функции завершения порожденных потоков.
4. Как получить результат выполнения порожденного потока?
5. Какие средства синхронизации потоков есть в Windows?
6. Каковы особенности критической секции как объекта синхронизации потоков?
7. Назовите функции работы с критическими секциями.
8. Каковы особенности мьютекса как объекта синхронизации потоков?
9. Назовите функции работы с мьютексами.
10. Какие мьютексы называются покинутыми?

Тема 8

СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ В WINDOWS

Цель работы: изучить способы и средства сетевого взаимодействия процессов средствами языка C++ в операционных системах семейства Windows.

Краткие теоретические сведения

Сокеты (sockets) представляют собой высокоуровневый унифицированный интерфейс взаимодействия с телекоммуникационными протоколами. 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.

ОС Windows 9x/NT и выше поддерживают оба вида сокетов, однако в силу того, что синхронные сокеты программируются более просто, чем асинхронные, последние не получили большого распространения. Далее обсуждаются исключительно синхронные сокеты.

Сокеты независимо от вида делятся на два типа:

- потоковые;
- дейтаграммные.

Потоковые сокеты работают с установкой соединения, обеспечивая надежную идентификацию обеих сторон и гарантируют целостность и успешность доставки данных.

Дейтаграммные сокеты работают без установки соединения и не обеспечивают ни идентификации отправителя, ни контроля успешности доставки данных, зато они заметно быстрее потоковых.

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. Дейтаграммные сокеты опираются на протокол UDP, а потоковые – на TCP.

Первый шаг – подготовка к работе. Перед началом использования функций библиотеки Winsock ее необходимо подготовить к работе вызовом функции `int WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData)`, передав в старшем байте слова `wVersionRequested` номер требуемой версии, а в младшем – номер подверсии.

Аргумент `lpWSAData` должен указывать на структуру `WSADATA`, в которую при успешной инициализации будет занесена информация о производителе библиотеки. Никакого особенного интереса она не представляет, и прикладное приложение может ее игнорировать. Если инициализация проваливается, функция возвращает ненулевое значение.

Второй шаг – создание объекта «сокет». Это осуществляется функцией `socket (int af, int type, int protocol)`. Первый слева аргумент указывает на семейство используемых протоколов. Для интернет-приложений он должен иметь значение `AF_INET`. Следующий аргумент задает тип создаваемого сокета – потоковый (`SOCK_STREAM`) или дейтаграммный (`SOCK_DGRAM`). Последний аргумент уточняет, какой транспортный протокол следует использовать. Нулевое значение соответствует выбору по умолчанию: TCP – для потоковых сокетов и UDP – для дейтаграммных. В большинстве случаев нет никакого смысла задавать протокол вручную и обычно полагаются на автоматический выбор по умолчанию.

Если функция завершилась успешно, она возвращает дескриптор сокета, в противном случае – `INVALID_SOCKET`.

Дальнейшие шаги зависят от того, является приложение сервером или клиентом. Ниже эти два случая будут описаны раздельно.

Клиент: шаг третий – для установки соединения с удаленным узлом потоковый сокет должен вызвать функцию `int connect (SOCKET s, const struct sockaddr FAR* name, int namelen)`. Датаграммные сокеты работают без установки соединения, поэтому обычно не обращаются к функции `connect`.

Первый слева аргумент – дескриптор сокета, возвращенный функцией `socket`; второй – указатель на структуру `sockaddr`, содержащую в себе адрес и порт удаленного узла, с которым устанавливается соединение. Последний аргумент сообщает функции размер структуры `sockaddr`.

После вызова `connect` система предпринимает попытку установить соединение с указанным узлом. Если по каким-то причинам это сделать не удастся (адрес задан неправильно, узел не существует или компьютер находится не в сети), функция возвратит ненулевое значение.

Сервер: шаг третий – прежде чем сервер сможет использовать сокет, он должен связать его с локальным адресом. Локальный, как, впрочем, и любой другой адрес Интернета, состоит из IP-адреса узла и номера порта. Если сервер имеет несколько IP-адресов, то сокет может быть связан как со всеми ними сразу (для этого вместо IP-адреса следует указать константу `INADDR_ANY`, равную нулю), так и с каким-то конкретным одним.

Связывание осуществляется вызовом функции `int bind (SOCKET s, const struct sockaddr FAR* name, int namelen)`. Первым слева аргументом передается дескриптор сокета, возвращенный функцией `socket`, за ним следуют указатель на структуру `sockaddr` и ее длина.

Клиент также должен связывать сокет с локальным адресом перед его использованием, однако за него это делает функция `connect`, ассоциируя сокет с одним из портов, наугад выбранных из диапазона 1024–5000. Сервер же должен занимать заранее определенный порт (например, 21 для FTP, 80 для WEB и т. д.). Поэтому ему приходится осуществлять связывание «вручную».

При успешном выполнении функция возвращает нулевое значение и ненулевое в противном случае.

Сервер: шаг четвертый – выполнив связывание, потоковый сервер переходит в режим ожидания подключений, вызывая функцию `int listen (SOCKET s, int backlog)`, где `s` – дескриптор сокета, а `backlog` – максимально допустимый размер очереди сообщений.

Размер очереди ограничивает количество одновременно обрабатываемых соединений, поэтому к его выбору следует подходить внимательнее. Если очередь полностью заполнена, очередной клиент при попытке установить соединение получит отказ (TCP пакет с установленным флагом RST). В то же время максимально разумное количество

подключений определяется производительностью сервера, объемом оперативной памяти и т. д.

Датаграммные серверы не вызывают функцию `listen`, так как работают без установки соединения и сразу же после выполнения связывания могут вызывать **`recvfrom`** для чтения входящих сообщений, минуя четвертый и пятый шаги.

Сервер: шаг пятый – извлечение запросов на соединение из очереди осуществляется функцией `accept (SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen)`, которая автоматически создает новый сокет, выполняет связывание и возвращает его дескриптор, а в структуру `sockaddr` заносит сведения о подключившемся клиенте (IP-адрес и порт). Если в момент вызова `accept` очередь пуста, функция не возвращает управление до тех пор, пока с сервером не будет установлено хотя бы одно соединение. В случае возникновения ошибки функция возвращает отрицательное значение.

Для параллельной работы с несколькими клиентами следует сразу же после извлечения запроса из очереди порождать новый поток (процесс), передавая ему дескриптор созданного функцией `accept` сокета, затем вновь извлекать из очереди очередной запрос и т. д. В противном случае, пока не завершит работу один клиент, север не сможет обслуживать всех остальных.

Обмен данными

После того как соединение установлено, потоковые сокеты могут обмениваться с удаленным узлом данными, вызывая функции `int send (SOCKET s, const char FAR * buf, int len,int flags)` и `int recv (SOCKET s, char FAR* buf, int len, int flags)` для посылки и приема данных соответственно.

Функция `send` возвращает управление сразу же после ее выполнения независимо от того, получила ли принимающая сторона наши данные или нет. При успешном завершении функция возвращает количество передаваемых (не переданных!) данных, т. е. успешное завершение еще не свидетельствует об успешной доставке. В общем случае протокол TCP гарантирует успешную доставку данных получателю, но лишь при условии, что соединение не будет преждевременно разорвано. Если связь прервется до окончания пересылки, данные останутся не переданными, но вызывающий код не получит об этом никакого уведомления. А ошибка возвращается лишь в том случае, если соединение разорвано до вызова функции `send`.

Функция же recv возвращает управление только после того, как получит хотя бы один байт. Точнее говоря, она ожидает прихода целой дейтаграммы. Дейтаграмма – это совокупность одного или нескольких IP-пакетов, посланных вызовом send. Упрощенно говоря, каждый вызов recv за один раз получает столько байтов, сколько их было послано функцией send. При этом подразумевается, что функции recv предоставлен буфер достаточных размеров, в противном случае – ее придется вызвать несколько раз. Однако при всех последующих обращениях данные будут браться из локального буфера, а не приниматься из сети, так как TCP-провайдер не может получить «кусочек» дейтаграммы, а только всю ее целиком.

Работой обеих функций можно управлять с помощью флагов, передаваемых в одной переменной типа int третьим слева аргументом. Эта переменная может принимать одно из двух значений: MSG_PEEK и MSG_OOB.

Флаг MSG_PEEK заставляет функцию recv просматривать данные вместо их чтения. Флаг MSG_OOB предназначен для передачи и приема срочных (Out Of Band) данных. Срочные данные не имеют преимущества перед другими при пересылке по сети, а всего лишь позволяют оторвать клиента от нормальной обработки потока обычных данных и сообщить ему «срочную» информацию. Если данные передавались функцией send с установленным флагом MSG_OOB, для их чтения флаг MSG_OOB функции recv также должен быть установлен.

Еще существует флаг MSG_DONTROUTE, предписывающий передавать данные без маршрутизации, но он не поддерживается Winsock, поэтому здесь не рассматривается.

Дейтаграммный сокет также может пользоваться функциями send и recv, если предварительно вызовет connect (см. «Клиент: шаг третий»), но у него есть и свои, «персональные», функции: int sendto (SOCKET s, const char FAR * buf, int len,int flags, const struct sockaddr FAR * to, int tolen) и int recvfrom (SOCKET s, char FAR* buf, int len, int flags, struct sockaddr FAR* from, int FAR* fromlen).

Они очень похожи на send и recv, – разница лишь в том, что sendto и recvfrom требуют явного указания адреса узла, принимающего или передающего данные. Вызов recvfrom не требует предварительного задания адреса передающего узла – функция принимает все

пакеты, приходящие на заданный UDP-порт со всех IP-адресов и портов. Напротив, отвечать отправителю следует на тот же самый порт, откуда пришло сообщение. Поскольку функция `recvfrom` запоминает IP-адрес и номер порта клиента после получения от него сообщения, программисту нужно передать в `sendto` тот же самый указатель на структуру `sockaddr`, который был ранее передан функции `recvfrom`, получившей сообщение от клиента.

Напоминаем, что транспортный протокол UDP, на который опираются дейтаграммные сокеты, не гарантирует успешной доставки сообщений и эта задача ложится на плечи самого разработчика. Решить ее можно, например, посылкой клиентом подтверждения об успешности получения данных. Лучше же не использовать дейтаграммные сокеты на ненадежных каналах.

Во всем остальном обе пары функций полностью идентичны и работают с флагами `MSG_PEEK` и `MSG_OOB`.

Все четыре функции при возникновении ошибки возвращают значение `SOCKET_ERROR == -1`.

Шаг шестой – для закрытия соединения и уничтожения сокета предназначена функция `int closesocket (SOCKET s)`, которая в случае удачного завершения операции возвращает нулевое значение.

Перед выходом из программы необходимо вызвать функцию `int WSACleanup (void)` для деинициализации библиотеки Winsock и освобождения используемых этим приложением ресурсов.

Внимание! Завершение процесса функцией `ExitProcess` автоматически не освобождает ресурсы сокетов!

Примеры применения сокетов, реализующие клиент-серверную систему с использованием протоколов TCP и UDP, доступны по адресу http://gun.cs.nstu.ru/ssw/Winsockets.

Методические указания

1. Проект может быть реализован на Visual C или в среде Borland C++ 5.0 и выше. В первом случае выбирается консольное приложение Win32 без дополнительных библиотек. Если в программе используются функции WinAPI, то файл включения *winsock2.h*. указывается до *windows.h.*

2. Для работы с библиотекой Winsock 2.x в исходный тест программы необходимо включить директиву "`#include <winsock2.h>`", а в свойствах проекта (компоновщик, командная строка) указать "`ws2_32.lib`" либо указать библиотеку в заголовке программы: `#pragma comment (lib,"Ws2_32.lib")`.

3. Выбор протокола передачи данных определяется вариантом задания.

4. Для нормального запуска серверного приложения необходимо выбрать свободный порт, используя утилиту Netstat.

5. Для нормального выполнения сетевых соединений требуется соответствующая настройка межсетевых экранов.

Порядок выполнения работы

1. Написать и отладить программу, реализующую сервер, получающий запросы от клиентов (аналог сервера в лабораторных работах 5, 6) и выполняющий вычисления (действия) в соответствии с вариантом.

2. Написать и отладить программу, реализующую клиентский процесс (аналог клиента в лабораторных работах 5, 6), передающий серверу исходные данные (имя файла, параметры замены) и ожидающий от сервера результатов.

Варианты заданий

Для обработки файлов сервером используются варианты из лабораторной работы № 1. Для передачи данных (имен файлов и параметров обработки) для нечетных вариантов используется протокол TCP, для четных – UDP.

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинги программ.

Контрольные вопросы

1. Что такое сокет?
2. Какие типы сокетов вы знаете?
3. Какие виды сокетов вы знаете?
4. Какие этапы работы с сокетами в клиентской части приложения вы знаете?
5. Какие этапы работы с сокетами в серверной части приложения вы знаете?
6. Какие функции реализуют отправку и получение сообщений в дейтаграммном режиме?
7. Какие функции реализуют отправку и получение сообщений в потоковом режиме?
8. Что такое сырые сокеты?
9. Для чего используются сырые сокеты?
10. Какие библиотеки для работы с сокетами вы знаете и как их следует подключать?

Тема 9

СИСТЕМНЫЕ СЛУЖБЫ В WINDOWS

Цель работы: изучить особенности написания и отладки системных служб в операционных системах семейства Windows и обеспечить взаимодействие с ними клиентских приложений.

Краткие теоретические сведения

При запуске операционной системы Microsoft Windows автоматически стартует специальный процесс, называемый процессом управления сервисами (Service Control Manager, SCM). В программном интерфейсе WinAPI имеются функции, с помощью которых приложения и сервисы могут управлять работой сервисов, обращаясь к процессу управления сервисами.

Преобразование консольного приложения в службу (сервис) Windows осуществляется в три этапа, после выполнения которых программа переходит под управление SCM.

1. Создание новой точки входа `main()`, которая регистрирует службу в SCM, предоставляя точки входа и имена логических служб.

2. Преобразование прежней функции точки входа `main()` в функцию `ServiceMain()`, которая регистрирует обработчик управляющих команд службы и информирует SCM о своем состоянии. Остальная часть кода, по существу, сохраняет прежний вид, хотя и может быть дополнена командами регистрации событий. Имя `ServiceMain()` является заменителем имени логической службы, причем логических служб может быть несколько.

3. Написание функции обработчика управляющих команд службы, которая должна предпринимать определенные действия в ответ на команды, поступающие от SCM.

Функция main сервисного процесса

Задачей новой функции `main()`, которая вызывается SCM, является регистрация службы в SCM и запуск диспетчера службы (service control dispatcher). Для этого необходимо вызвать функцию `StartServiceControlDispatcher()`, передав ей имя (имена) и точку (точки) входа одной или нескольких логических служб.

Эта функция принимает единственный аргумент `lpServiceStartTable`, являющийся адресом массива элементов `SERVICE_TABLE_ENTRY`, каждый из которых представляет имя и точку входа логической службы. Конец массива обозначается двумя последовательными значениями `NULL`.

Тип `SERVICE_TABLE_ENTRY` и соответствующий указатель определены следующим образом:

```
typedef struct SERVICE_TABLE_ENTRY
{
   LPTSTR lpServiceName;
   LPSERVICE_MAIN_FUNCTION lpServiceProc;
}  SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;
```

В поле `lpServiceName` записывается указатель на текстовую строку имени сервиса, а в поле `lpServiceProc` – указатель на функцию – точку входа сервиса.

Функция возвращает значение `TRUE`, если регистрация службы прошла успешно.

Точка входа сервиса

Точка входа сервиса – это функция, адрес которой записывается в поле `lpServiceProc` массива структур `SERVICE_TABLE_ENTRY`. Имя функции может быть любым, а прототип должен быть таким, как показанный ниже:

```
   void    WINAPI    ServiceMain(DWORD    dwArgc,    LPSTR
*lpszArgv);
```

Точка входа сервиса вызывается при запуске сервиса функцией `StartService` (эта функция рассмотрена далее). Через параметр

dwArgc передается счетчик аргументов, а через параметр lpszArgv – указатель на массив строк параметров. В качестве первого параметра всегда передается имя сервиса. Остальные параметры можно задать при запуске сервиса функцией StartService.

Несмотря на то что функция ServiceMain – адаптированный вариант функции main с ее параметрами, представляющими количество аргументов и содержащую их строку, между ними имеется одно незначительное отличие: функция службы должна быть объявлена с типом void, а не иметь возвращаемое значение типа int, как в случае обычной функции main.

Функция точки входа сервиса должна зарегистрировать функцию обработки команд и выполнить инициализацию сервиса.

Первая задача решается с помощью функции RegisterServiceCtrlHandler, прототип которой приведен ниже:

```
SERVICE_STATUS_HANDLE RegisterServiceCtrlHandler(
   LPCTSTR lpszServiceName,           // имя сервиса
   LPHANDLER_FUNCTION lpHandlerProc); // адрес функции
                                      // обработки команд
```

Через первый параметр этой функции необходимо передать адрес текстовой строки имени сервиса, а через второй – адрес функции обработки команд (функция обработки команд будет рассмотрена ниже).

Функция RegisterServiceCtrlHandler в случае успешного завершения возвращает идентификатор состояния сервиса. При ошибке возвращается нулевое значение.

Заметим, что регистрация функции обработки команд должна быть выполнена немедленно в самом начале работы функции-точки входа сервиса.

Обработчик службы должен устанавливать состояние службы при каждом вызове, даже если ее состояние не менялось.

Настройка состояния службы

В процессе инициализации функция точки входа сервиса выполняет действия, зависящие от назначения сервиса. Необходимо, однако, помнить, что без принятия дополнительных мер инициализация должна выполняться не дольше одной секунды. В противном случае перед началом инициализации функция точки входа сервиса должна сообщить процессу управления сервисами, что данный сервис находится

в состоянии ожидания запуска. Это можно сделать с помощью функции SetServiceStatus, которая будет описана позже. Перед началом инициализации вы должны сообщить процессу управления сервисами, что сервис находится в состоянии SERVICE_START_PENDING.

После завершения инициализации функция точки входа сервиса должна указать процессу управления сервисами, что процесс запущен и находится в состоянии SERVICE_RUNNING.

Функция обработки команд

Как следует из названия, функция обработки команд, зарегистрированная функцией RegisterServiceCtrlHandler, обрабатывает команды, передаваемые сервису операционной системой, другими сервисами или приложениями. Эта функция может иметь любое имя и выглядит следующим образом:

```
void WINAPI ServiceControl(DWORD dwControlCode)
{
  switch(dwControlCode)
  {
    case SERVICE_CONTROL_STOP:
    {
      ss.dwCurrentState = SERVICE_STOP_PENDING;
      ReportStatus(ss.dwCurrentState, NOERROR, 0);
      // Выполняем остановку сервиса, вызывая функцию,
      // которая выполняет все необходимые действия
      // ServiceStop();
      ReportStatus(SERVICE_STOPPED, NOERROR, 0);
      break;
    }
    case SERVICE_CONTROL_INTERROGATE:
    {
      ReportStatus(ss.dwCurrentState, NOERROR, 0);
      break;
    }
    default:
    {
      ReportStatus(ss.dwCurrentState, NOERROR, 0);
      break;
    }
  }
}
```

В приведенном выше фрагменте кода для сообщения процессу управления сервисами текущего состояния сервиса вызывается функция ReportStatus, которая должна вызвать функцию SetServiceStatus:

```
BOOL SetServiceStatus(
   SERVICE_STATUS_HANDLE sshServiceStatus,
   // идентификатор состояния сервиса
   LPSERVICE_STATUS lpssServiceStatus);
   // адрес структуры, содержащей состояние сервиса
```

Через параметр sshServiceStatus функции SetServiceStatus необходимо передать идентификатор состояния сервиса, полученный от функции RegisterServiceCtrlHandler.

В параметре lpssServiceStatus необходимо передать адрес предварительно заполненной структуры типа SERVICE_STATUS:

```
typedef struct _SERVICE_STATUS
{
DWORD dwServiceType;        // тип сервиса
DWORD dwCurrentState;       // текущее состояние сервиса
DWORD dwControlsAccepted;   // обрабатываемые команды
DWORD dwWin32ExitCode;      // код ошибки при запуске
                            // и остановке
DWORD dwServiceSpecificExitCode; // специфический код
                                 // ошибки
DWORD dwCheckPoint;
 // контрольная точка при выполнении длительных операций
DWORD dwWaitHint;           // время ожидания
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

Для определения текущего состояния сервиса можно использовать функцию QueryServiceStatus:

```
BOOL QueryServiceStatus(
SC_HANDLE schService,         // идентификатор сервиса
LPSERVICE_STATUS lpssServiceStatus);
// адрес структуры SERVICE_STATUS
```

Установка и удаление сервиса

Установка и удаление сервиса производятся при помощи утилиты `sc`, запускаемой от имени администратора системы. Установка производиться следующей командой:

```
sc create SampleService binpath= c:\SampleService.exe
```

Удаление сервиса:

```
sc delete SampleService
```

Запуск и остановка службы может производиться также утилитой `sc`:

```
sc start SampleService
sc stop SampleService
```

либо утилитой `net`, либо через оснастку *services.msc* консоли управления MMC.exe. По умолчанию служба запускается от имени системной учетной записи (LocalSystem).

Примеры исходных текстов системных служб доступны по адресу http://gun.cs.nstu.ru/winprog/Services.

Методические указания

1. Проект может быть реализован на Visual C или в среде Borland C++ 5.0 и выше. Тип проекта – консольное приложение Win32 без дополнительных библиотек. В программу необходимо добавить файл включения *windows.h*.

2. Проект выполняется в операционной системе, в которой имеются права администратора (для установки и запуска службы).

3. Для контроля объектов ядра ОС, создаваемых службой, рекомендуется использовать утилиту WinObj или WinObjEx (см. http://gun. cs.nstu. ru/winprog/Services).

4. Варианты, использующие именованные каналы и сокеты, соответствуют заданиям повышенной сложности.

Порядок выполнения работы

1. Написать и отладить системную службу, содержащую функцию регистрации в текстовом файле ее событий (запуск, останов, ошибки выполнения, обмен данными с клиентом), реализующую серверный

процесс. Методы обмена данными с клиентским приложением и средства их синхронизации зависят от варианта.

2. Установить службу в операционной системе с помощью утилиты `sc` (требуются права администратора ОС), запустить и убедиться в ее работоспособности (проверив файл ее событий).

3. Написать и отладить программу, реализующую клиентское приложение, убедиться в корректном взаимодействии приложения со службой.

Варианты заданий

1. Варианты представлены в табл. 2, где № – номер студента по списку группы, способ обмена данными:

1) именованные каналы;
2) почтовые ящики;
3) отображаемые на память файлы;
4) сокеты TCP;
5) сокеты UDP.

2. Способы синхронизации процессов:

1) события;
2) семафоры;
3) мьютексы.

3. Варианты обработки данных взять из табл. 1 (лабораторная работа 1).

Таблица 2

Варианты заданий РГР

№	Способ обмена данными	Синхронизация	Обработка данных
1	1	–	1
2	2	–	2
3	3	1	3
4	3	2	4
5	3	3	5
6	4	–	6
7	5	–	7
8	1	–	8

Окончание табл. 2

№	Способ обмена данными	Синхронизация	Обработка данных
9	2	–	9
10	3	1	10
11	3	2	11
12	3	3	12
13	4	–	13
14	5	–	14
15	1	–	15
16	2	–	16
17	3	1	17
18	3	2	18
19	3	3	19
20	4	–	20
21	5	–	1
22	1	–	1
23	2	–	2
24	3	1	3
25	3	2	4
26	3	3	5
27	4	–	6
28	5	–	7

Содержание отчета

1. Цель работы.
2. Вариант задания.
3. Листинги программ.

Контрольные вопросы

1. Чем отличаются процессы и службы?
2. Управление службой средствами операционной системы.
3. Назначение и содержание функции `main` сервисного процесса.
4. Что такое точка входа в службу? Каковы ее особенности?
5. В чем заключается настройка состояния службы?
6. Как зарегистрировать функцию обработки команд службы?
7. Каково назначение зарегистрированной в службе функции обработки команд? Какие функции она вызывает?
8. Как установить состояние службы? Какие ее состояния возможны?
9. Как установить и удалить службу?
10. Какова область видимости и права доступа к объектам, созданным службой?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. *Гунько А.В.* Программирование (в среде Windows): учебное пособие. – Новосибирск: Изд-во НГТУ, 2019. – 128 с.

2. *Харт Дж.М*. Системное программирование в среде Windows: Вильямс, 2005. – 592 с.

3. *Фролов А*., *Фролов Г.* Программирование для Windows NT. Том 27, часть 2. – М.: Диалог-МИФИ, 1996. – 272 с.

4. *Кили Д.* Winsock [Электронный ресурс] Winsock. – Режим доступа: http://msdn.microsoft.com/ru-ru/library/dd335942.aspx.

Гунько А.В. Программирование (в среде Windows) [Электронный ресурс]. – Режим доступа: http://gun.cs.nstu.ru/winprog.

ОГЛАВЛЕНИЕ

Гунько Андрей Васильевич

ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие

Редактор *М.О. Мокшанова*
Выпускающий редактор *И.П. Бровановa*
Корректор *И.Е. Семенова*
Дизайн обложки *А.В. Ладыжская*
Компьютерная верстка *Л.А. Веселовская*

Налоговая льгота – Общероссийский классификатор продукции
Издание соответствует коду 95 3000 ОК 005-93 (ОКП)

Подписано в печать 01.10.2019. Формат 60 × 84 1/16. Бумага офсетная. Тираж 50 экз.
Уч.-изд. л. 4,41. Печ. л. 4,75. Изд. № 73. Заказ № 1335. Цена договорная

Отпечатано в типографии
Новосибирского государственного технического университета
630073, г. Новосибирск, пр. К. Маркса, 20