Усовершенствованные средства для работы с файлами и каталогами и знакомство с реестром

Файловые системы обеспечивают не только простую последовательную обработку файлов; кроме этого, они должны предоставлять возможности прямого доступа к файлам и блокирования файлов, а также предлагать средства для управления каталогами и атрибутами файлов. В данной главе, которая начинается с обсуждения прямого доступа к файлам, требуемого для обслуживания баз данных, обработки файлов и решения целого ряда других задач, демонстрируются методы непосредственного доступа к данным, находящимся в произвольном месте файла, которые обеспечиваются файловыми указателями. Для этого, в частности, нам надо будет обсудить использование 64-битовых указателей Windows, поскольку файловая система NTFS способна поддерживать файлы гигантских размеров.

Далее будут рассмотрены методы просмотра каталогов, рассказано о том, что такое атрибуты файлов, такие, например, как метки времени, атрибуты прав доступа или размер файла, и показано, как управлять атрибутами и интерпретировать их. Наконец, вы ознакомитесь с тем, как использовать блокирование файлов с целью предотвращения попыток изменения их содержимого одновременно несколькими процессами.

Завершает данную главу рассмотрение реестра Windows — централизованной базы данных, хранящей информация о конфигурации системы, которой могут пользоваться как приложения, так и сама операционная система. Приведенный в конце главы пример программы показывает, что функции, с помощью которых осуществляется доступ к реестру, и структура соответствующих программ напоминают те, которые применяются для управления файлами и каталогами, что и послужило причиной включения этой темы в данную главу.

64-битовая файловая система

Win32 и Win64, работающие с NTFS, поддерживают 64-битовую адресацию в файлах, и поэтому допустимыми являются файлы размером до 2^{64} байт.

В 32-разрядных файловых системах, характеризующихся наличием 2^{32} – байтового предела, допустимый размер файлов ограничивается величиной 4 Гбайт ($4x10^9$ байт). Для некоторых приложений, включая крупные базы данных и мультимедийные системы, это ограничение носит серьезный характер, что вынуждает современные ОС обеспечивать поддержку файлов гораздо больших размеров. Файлы, размеры которых превышают 4 Гбайт, иногда называют *гигантскими* (huge).

Вполне очевидно, что многим приложениям гигантские файлы никогда не понадобятся, так что большинству программистов на протяжении ближайших нескольких лет возможностей 32-битовой файловой адресации будет вполне достаточно. Однако, с учетом темпов технической модернизации и увеличения емкости дисков[14], улучшения их стоимостных

показателей и повышения уровня требований со стороны приложений, целесообразно уже с самого начала работы над новым проектом предусмотреть возможность использования 64-битовых адресов.

Несмотря на возможность использования 64-битовой адресации файлов и поддержку гигантских файлов, интерфейс Win32, в силу его привязки к 32-битовой адресации памяти, остается API 32-битовой ОС, так что для работы с 64-битовыми адресами памяти нам потребуется интерфейс Win64.

Указатели файлов

В Windows аналогично тому, как это предусмотрено в UNIX, библиотеке C и почти любой другой ОС, для каждого дескриптора открытого файла поддерживается *указатель файла* (file pointer), отмечающий позицию текущего байта в данном файле. Именно эта позиция служит отправной точкой для последующей передачи данных при выполнении очередной операции WriteFile или ReadFile, что сопровождается увеличением значения указателя файла на соответствующее количество переданных байт. При открытии файла путем вызова функции CreateFile указатель файла принимает нулевое значение, отмечающее начало файла, которое изменяется по мере чтения или записи каждого очередного байта. Ключевую роль в обеспечении возможности прямого доступа к данным, хранящимся в файле, играет функция SetFilePointer, позволяющая устанавливать значения указателя файла.

Функция SetFilePointer является первой из функций, на примере которых мы познакомимся с обработкой 64-битовых указателей файлов NTFS. Методы, основанные на этой функции, не всегда удобны в применении, и поэтому функцию SetFilePointer проще всего использовать в случае небольших файлов.

DWORD SetFilePointer(HANDLE hFile, LONG lDistanceToMove, PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod)

Возвращаемое значение: младшее двойное слово (DWORD, беззнаковое) нового значения указателя файла. Старшая часть значения этого указателя помещается в двойное слово, на которое указывает указатель lpDistanceToMoveHigh (если он отличен от NULL). В случае неудачного завершения функция возвращает значение 0xFFFFFFFF.

Параметры

hFile — дескриптор файла, который должен быть создан с правами доступа по чтению или по записи (или с правами доступа одновременно обоих типов).

lDistanceToMove — 32-битовое число типа LONG *со знаком,* указывающее величину смещения, на которое должен быть перемещен указатель файла, или число типа LONG *без знака,* указывающее позицию, в которую должен

быть перемещен указатель файла, в зависимости от значения параметра dwMoveMethod.

lpDistanceToMoveHigh — указатель на старшую часть 64-битового смещения, на которое должен быть перемещен указатель файла. Если значение этого параметра задано равным NULL, то функция может применяться только к файлам, размер которых не превышает $2^{32}-2$ (в байтах). Этот же параметр используется для получения старшей части возвращаемого функцией значения указателя файла.[15] Младшую часть указателя файла возвращает сама функция.

dwMoveMethod — этот параметр устанавливает один из трех возможных режимов перемещения указателя файла.

• FILE_BEGIN — указатель файла позиционируется относительно начала файла, причем параметр DistanceToMove интерпретируется как беззнаковое число.

• FILE_CURRENT — указатель файла перемещается в сторону больших или меньших значений относительно текущей позиции, причем параметр DistanceToMove интерпретируется как число со знаком. Положительным значениям соответствует перемещение указателя файла в сторону больших значений.

• FILE_END — указатель файла перемещается в сторону больших или меньших значений относительно позиции конца файла.

Эту функцию можно использовать для получения размера файла, задав нулевое смещение указателя от позиции конца файла.

Описанный метод представления 64-битовых указателей файлов становится причиной некоторых затруднений, поскольку возвращенное функцией значение может представлять как действительную позицию указателя файла, так и код ошибки. Рассмотрите, например, случай, когда фактической позиции указателя соответствует значение $2^{32}-1$ (то есть, 0xFFFFFFFF), а при вызове функции указывается ненулевое значение старшей части перемещения указателя файла. Чтобы определить, представляет ли значение, возвращенное функцией SetFilePointer, действительную позицию указателя файла или же код ошибки, следует вызвать функцию **GetLastError**, возвращаемым значением которой в случае неудачного завершения не может быть NO_ERROR. Из этих рассуждений становится ясно, почему размеры файлов не могут превышать значения $2^{32}-2$, если при вызове функции SetFilePointer старшая часть указателя файла опускается.

Дополнительную неразбериху привносит тот факт, что старшая и младшая компоненты указателя файла отделены друг от друга и обрабатываются по-разному. Младшая часть определяется через передачу параметра по значению и равна возвращаемому значению функции, тогда как для старшей части применяется передача параметра по ссылке, и этот параметр используется как в качестве входного, так и выходного.

К счастью, 32-битовой адресации вам будет вполне достаточно для большинства задач программирования. Тем не менее, приведенные примеры

программ рассчитаны на перспективу и используют, "как и положено", 64-битовую арифметику.

64-битовая арифметика

Арифметика 64-битовых указателей файлов не так уж сложна, и для ее реализации в примерах программ используется принятый в Microsoft C 64-битовый тип данных LARGE_INTEGER, объединяющий в одном типе данных union величину типа LONGLONG (носящую название QuadPart) и две 32-битовые величины (LowPart типа DWORD и, HighPart типа LONG). Тип данных LONGLONG поддерживает все арифметические операции. Существует также соответствующий тип данных без знака ULONGLONG.

Аналогами функции SetFilePointer являются функции lseek (UNIX) и fseek (библиотека C). В обеих упомянутых системах выполнение операций чтения или записи также сопровождается перемещением указателя файла.

Указание позиции файла с помощью структуры OVERLAPPED

Для указания позиции в файле Windows предоставляет еще один способ, не требующий использования функции SetFilePointer. Вспомните, что последним параметром в обеих функциях ReadFile и WriteFile является адрес структуры перекрытия OVERLAPPED, который в предыдущих примерах всегда полагался равным NULL. В структуру перекрытия входят элементы Offset и OffsetHigh. Устанавливая соответствующие значения элементов структуры OVERLAPPED, вы можете добиться того, чтобы выполнение операций ввода/вывода начиналось с указанной позиции. В отличие от указателя файла, значение которого изменяется, соответствуя позиции, следующей за последним переданным байтом, значения элементов структуры OVERLAPPED остаются неизменными. Элементом этой структуры является также дескриптор hEvent, значение которого должно устанавливаться равным NULL.

Предостережение

Хотя в рассмотренном примере и используется структура OVERLAPPED, здесь не идет речь о перекрывающемся вводе/выводе, который обсудим позже.

Использование структуры OVERLAPPED оказывается особенно удобным в тех случаях, когда требуется обновить запись в файле, что иллюстрирует приведенный ниже фрагмент программного кода; в противном случае вы должны были бы перед каждым вызовом функций ReadFile и WriteFile отдельно вызывать функцию SetFilePointer. Последним из пяти полей структуры OVERLAPPED является поле hEvent, как это видно из оператора инициализации. Для хранения вычисленного значения позиции в файле используется переменная FilePos типа LARGE_INTEGER.

```
OVERLAPPED ov = { 0, 0, 0, 0, NULL };
```

```
RECORD r; /* Хотя определение этой структуры не приведено, в ней имеется поле RefCount. */
LONGLONG n;
LARGE_INTEGER FilePos;
DWORD nRead, nWrite;
…
/* Обновить счетчик, чтобы он соответствовал n-й записи. */
FilePos.QuadPart = n * sizeof(RECORD);
ov.Offset = FilePos.LowPart;
ov.OffsetHigh = FilePos.HighPart;
ReadFile(hFile,  r, sizeof(RECORD), &nRead, &ov);
r.RefCount++; /* Обновить запись. */
WriteFile(hFile,  r, sizeof(RECORD), &nWrite, &ov);
```

Если дескриптор файла был создан за счет вызова функции CreateFile с установленным флагом FILE_FLAG_NO_BUFFERING, то как смещение позиции в файле, так и размер записи (количество байт) должны быть кратными размеру сектора диска. Соответствующую информацию относительно физического диска, включая информацию о размере сектора, возвращает функция GetDiskFreeSpace.

Структуры OVERLAPPED будут вновь использованы далее в этой главе для указания областей блокирования файлов.

Определение размера файла

Размер файла можно получить, используя значение указателя файла, возвращаемое функцией SetFilePointer, если при вызове этой функции задать количество байтов, на которое должен быть перемещен указатель файла, равным 0. Для этой же цели можно воспользоваться также функцией GetFileSize.

DWORD GetFileSize(HANDLE hFile, LPDWORD lpFileSizeHigh)

Возвращаемое значение: младшая компонента размера файла. Значение 0xFFFFFFFF указывает на возможную ошибку; для проверки наличия ошибок следует использовать функцию GetLastError.

Обратите внимание, что для возвращения размера файла используется, по сути, тот же способ, что и для возвращения фактического указателя файла функцией SetFilePointer.

Функции GetFileSize и GetFileSizeEx (возвращающая 64-битовое значение размера файла в одном элементе данных) требуют указания дескриптора, открытого для файла. Для определения размера файла можно применять также имя файла. Функция **GetCompressedFileSize** возвращает размер сжатого файла, тогда как функция FindFirstFile, которая обсуждается в разделе "Атрибуты файлов и управление каталогами" далее в этой главе, предоставляет точный размер именованного файла.

Установка размера файла, инициализация файла и разреженные файлы

Функция **SetEndOfFile** позволяет переустановить размер файла, используя текущее значение указателя файла для определения его размера. Возможно как расширение, так и усечение файла. В случае расширения файла содержимое области расширения не определено. Файл будет фактически потреблять выделенные квоты дискового и пользовательского пространств, если только не является разреженным. Файлы можно сжимать с целью уменьшения объема занимаемого ими пространства. Этот вопрос исследуется в упражнении 3.1.

Функция SetEndOfFile устанавливает физический конец файла. Прежде чем выполнять эту операцию, на которую может уйти довольно длительное время, необходимое для записи данных файла с целью его заполнения, можно установить также логический конец файла, используя для этого функцию **SetValidFileData**. Эта функция определяет ту часть файла, которая, в соответствии с вашими предположениями, в настоящий момент содержит достоверные данные, благодаря чему вы сможете сэкономить время при установке физического конца файла. Часть файла, заключенная между его логическим и физическим концами, называется *хвостовиком* (tail) и может быть сокращена путем записи оставшихся данных после логического конца файла или в результате дополнительного вызова функции SetValidFileData

.

В случае разреженных файлов (sparse files), появившихся в Windows 2000, дисковое пространство расходуется лишь по мере записи данных. Администратор может назначать, какие файлы, каталоги или тома должны быть разреженными. Кроме того, можно назначить существующий файл в качестве разреженного с помощью функции DeviceIoControl, если установить при ее вызове флаг FSCTL_SET_SPARSE. Ситуацию, в которой удобно использовать разреженные файлы, иллюстрирует программа 3.1. К разреженным файлам функция SetValidFileData неприменима.

Файлы FAT нулями автоматически *не* инициализируются. Согласно документации Microsoft содержимое вновь созданных файлов не определено, что подтверждается экспериментами. Поэтому, если для корректной работы требуется инициализация файлов, приложения должны это делать самостоятельно путем вызова функции WriteFile. Файлы NTFS будут инициализированы, поскольку уровень безопасности C2, обеспечиваемый Windows, требует, чтобы чтение содержимого удаленных файлов было невозможным.

Обратите внимание, что кроме функции SetEndOfFile существуют и другие способы расширения размера файла. Так, можно расширить файл, используя ряд последовательных операций записи, хотя при этом существует риск увеличения степени фрагментации файла; размещение на диске файлов в виде непрерывных блоков большого размера функция SetEndOfFile отдает на откуп операционной системе.

Пример: обновление записей, находящихся в произвольном месте файла

Программа RecordAccess (программа 3.1) обеспечивает поддержку файлов фиксированного размера, состоящих из записей фиксированного размера. В заголовке файла хранится количество непустых записей, содержащихся в файле, а также емкость файла. Пользователю предоставляется возможность выполнять в интерактивном режиме чтение, запись (обновление) и удаление записей, каждая из которых содержит метки времени, текстовую строку и счетчик, показывающий, сколько раз запись изменялась. В качестве несложного и реалистичного расширения возможностей программы можно было бы добавить в структуру записи ключ и определять местоположение записей в файле путем применения хэш-функции к значениям ключа.

Программа демонстрирует позиционирование указателя файла перед заданной записью, а также выполнение 64-битовых арифметических операций с использованием данных типа LARGE_INTEGER Microsoft C. Чтобы проиллюстрировать логику работы указателей файла, в программу включен код, проверяющий наличие ошибок. Программа в целом иллюстрирует применение файловых указателей и множественных структур OVERLAPPED, а также обновление файлов с использованием 64-битовых файловых указателей.

Общее количество записей в файле указывается в командной строке; при большом количестве записей размеры создаваемых файлов могут быть очень большими и даже гигантскими, поскольку длина одной записи составляет примерно 300 байт. После выполнения нескольких экспериментов вы убедитесь, что большие файлы должны быть разреженным; в противном случае необходимо размещать и инициализировать на диске весь файл целиком, в результате чего может существенно увеличиться время обработки файла и занимаемое им место на диске. Хотя в листинге программы 3.1 это и не отражено, в программе предусмотрен участок кода, обеспечивающий создание разреженных файлов, если в этом возникает необходимость; в некоторых системах, например Windows XP Home, этот код правильно работать не сможет.

На Web-сайте книги предоставляются три дополнительные программы, родственные этой: tail.c — другой пример реализации произвольного доступа к файлу, getn.c — упрощенная версия программы RecordAccess, обеспечивающая лишь чтение записей, и atouMT (включена в программы для главы 14, находящиеся на Web-сайте, однако не включена в программы, приведенные в книге), также иллюстрирующая прямой доступ к файлам.

Программа 3.1. RecordAccess

```
/* Глава 3. RecordAccess. */
/* Использование: RecordAccess имя файла [количество записей]
```

Количество записей (nrec) можно не указывать, если файл с указанным именем уже существует. Если количество записей (nrec) задано, создается

файл с указанным именем (если файл с таким именем существует, он уничтожается). При большом количестве записей (nrec) файлы рекомендуется создавать как разреженные. */

/* Программа иллюстрирует:

1. Произвольный доступ к файлам.

2. Арифметику данных типа LARGE_INTEGER и использование 64-битовых указателей файла.

3. Обновление записей на месте.

4. Запись в файл нулей во время инициализации (требует использования файловой системы NTFS).

*/

```
#include "EvryThng.h"
#define STRING_SIZE 256
typedef struct _RECORD { /* Структура записи в файле */
 DWORD ReferenceCount; /* 0 означает пустую запись. */
 SYSTEMTIME RecordCreationTime;
 SYSTEMTIME RecordLastReferenceTime;
 SYSTEMTIME RecordUpdateTime;
 TCHAR DataString[STRING_SIZE];
} RECORD;
typedef struct _HEADER { /* Дескриптор заголовка файла */
 DWORD NumRecords;
 DWORD NumNonEmptyRecords;
} HEADER;

int _tmain(int argc, LPTSTR argv[]) {
 HANDLE hFile;
 LARGE_INTEGER CurPtr;
 DWORD FPos, OpenOption, nXfer, RecNo;
 RECORD Record;
 TCHAR String[STRING_SIZE], Command, Extra;
 OVERLAPPED ov = {0, 0, 0, 0, NULL}, ovZero = {0, 0, 0, 0, NULL};
 HEADER Header = {0, 0};
 SYSTEMTIME CurrentTime;
 BOOLEAN HeaderChange, RecordChange;
 OpenOption = (argc == 2) ? OPEN_EXISTING : CREATE_ALWAYS;
 hFile = CreateFile(argv[1], GENERIC_READ | GENERIC_WRITE, 0, NULL,
OpenOption, FILE_ATTRIBUTE_NORMAL, NULL);
 if (argc >= 3) { /* Записать заголовок и заранее установить размер нового
файла */
  Header.NumRecords = atoi(argv[2]);
  WriteFile(hFile, &Header, sizeof(Header), &nXfer, &ovZero);
  CurPtr.QuadPart = sizeof(RECORD)*atoi(argv[2])+sizeof(HEADER);
```

```
    FPos = SetFilePointer(hFile, CurPtr.LowPart, &CurPtr.HighPart,
FILE_BEGIN);
    if (FPos == 0xFFFFFFFF && GetLastError() != NO_ERROR)
ReportError(_T("Ошибка указателя."), 4, TRUE);
    SetEndOfFile(hFile);
   }
   /* Считать заголовок файла: определить количество записей и количество
непустых записей. */
   ReadFile(hFile, &Header, sizeof(HEADER), &nXfer, &ovZero);
   /* Предложить пользователю считать или записать запись с определенным
номером. */
   while(TRUE) {
    HeaderChange = FALSE;
    RecordChange = FALSE;
    _tprintf(_T("Введите r(ead)/w(rite)/d(elete)/q Запись#\n"));
    _tscanf(_T("%c" "%d" "%c"), &Command, &RecNo, &Extra );
    if (Command == 'q') break;
    CurPtr.QuadPart = RecNo * sizeof(RECORD) + sizeof(HEADER);
    ov.Offset = CurPtr.LowPart;
    ov.OffsetHigh = CurPtr.HighPart;
    ReadFile(hFile, &Record, sizeof(RECORD), &nXfer, &ov);
    GetSystemTime(&CurrentTime); /* Обновить поля даты и времени в записи.
*/
    Record.RecordLastRefernceTime = CurrentTime;
    if (Command == 'r' || Command == 'd') { /*Вывести содержимое записи.*/
     if (Record.ReferenceCount == 0) {
      _tprintf(_T("Запись номер %d – пустая.\n"), RecNo);
      continue;
     } else {
       _tprintf(_T("Запись номер %d. Значение счетчика: %d \n"), RecNo,
Record.ReferenceCount);
      _tprintf(_T("Данные: %s\n"), Record.DataString);
      /* Упражнение: вывести метки времени. См. следующий пример. */
      RecordChange = TRUE;
     }
     if (Command == 'd') { /* Удалить запись. */
      Record.ReferenceCount = 0;
      Header.NumNonEmptyRecords--;
      HeaderChange = TRUE;
      RecordChange = TRUE;
     }
    } else if (Command == 'w') { /* Записать данные. Впервые? */
     _tprintf(_T("Введите новую строку для записи.\n"));
     _getts(String);
     if (Record.ReferenceCount == 0) {
```

```
      Record.RecordCreationTime = CurrentTime;
      Header.NumNonEmptyRecords++;
      HeaderChange = TRUE;
     }
     Record.RecordUpdateTime = CurrentTime;
     Record.ReferenceCount++;
     _tcsncpy(Record.DataString, String, STRING_SIZE-1);
     RecordChange = TRUE;
    } else {
     _tprintf(_T("Допустимые команды: r, w и d. Повторите ввод.\n"));
    }
    /* Обновить запись на месте, если ее содержимое изменилось. */
     if (RecordChange) WriteFile(hFile, &Record, sizeof(RECORD), &nXfer,
&ov);
    /* При необходимости обновить количество непустых записей. */
       if (HeaderChange) WriteFile(hFile, &Header, sizeof(Header), &nXfer,
&ovZero);
   }
   _tprintf(_T("Вычисленное количество непустых записей: %d\n"),
Header.NumNonEmptyRecords);
   CloseHandle(hFile);
   return 0;
  }
```

Атрибуты файлов и управление каталогами

Существует возможность просмотра указанного каталога с целью поиска файлов и других каталогов, имена которых соответствуют заданному шаблону, одновременно с получением атрибутов файлов. Для выполнения поиска требуется дескриптор поиска (search handle), получаемый с помощью функции FindFirstFile. Для нахождения файлов, имена которых удовлетворяют заданным условиям, используется функция FindNextFile, а для прекращения поиска — функция FindClose.

HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpffd)

Возвращаемое значение: дескриптор поиска. Значение INVALID_HANDLE_VALUE указывает на неудачное завершение функции.

В процессе поиска имен, соответствующих искомому, функция FindFirstFile проверяет имена не только файлов, но и подкаталогов. Возвращенное функцией значение дескриптора типа HANDLE используется для продолжения поиска.

Параметры

lpFileName — указатель на строку, содержащую имя каталога или полное имя файла, при указании которых можно использовать метасимволы (? и *). Если необходимо осуществить поиск конкретного файла, метасимволы опускаются.

lpffd — указатель на структуру WIN32_FIND_DATA, которая принимает информацию о первом найденном файле или каталоге, который удовлетворяет критерию поиска, если таковой был найден.

Структура WIN32_FIND_DATA определяется следующим образом:

```
typedef struct_WIN32_FIND_DATA {
 DWORD dwFileAttributes;
 FILETIME ftCreationTime;
 FILETIME ftLastAccessTime;
 FILETIME ftLastWriteTime;
 DWORD nFileSizeHigh;
 DWORD nFileSizeLow;
 DWORD dwReserved0;
 DWORD dwReserved1;
 TCHAR cFileName[MAX_PATH];
 TCHAR cAlternateFileName[14];
 } WIN32_FIND_DATA;
```

Параметр dwFileAttributes можно тестировать на присутствие значений, описанных при рассмотрении функции CreateFile, а также некоторых других значений, например, FILE_ATTRIBUTE_SPARSE_FILE или FILE_ATTRIBUTE_ENCRYTPED, которые не устанавливаются функцией CreateFile. Описание меток времени трех типов (время создания, время последнего обращения и время последнего изменения) приведено в одном из следующих разделов. Названия полей размера файла (nFileSizeHigh и nFileSizeLow) говорят сами за себя. cFileName — это не полное имя файла, содержащее путь доступа, а само имя файла. cAlternateFileName — имя файла в формате DOS 8.3 (включая точку); эта информация редко используется и может понадобиться лишь для того, чтобы определить, каким будет имя файла в файловой системе FAT16.

Во многих случаях требуется просматривать каталог с целью поиска файлов, имена которых соответствуют некоторому шаблону, содержащему метасимволы ? и *. Для этого следует использовать дескриптор поиска, полученный из функции FindFirstFile, в котором содержится информация об искомом имени, и вызвать функцию FindNextFile.

BOOL FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA lpffd)

Функция FindNextFile возвращает значение FALSE, если аргументы недействительны или если не удается найти файл, удовлетворяющий критерию поиска, причем последнему случаю соответствует возвращаемое значение функции GetLastError, равное ERROR_NO_MORE_FILES.

После того как поиск завершен, дескриптор поиска должен быть закрыт. Функцию CloseHandle для этой цели использовать нельзя. Это редкий пример нарушения правила, согласно которому функция CloseHandle применима к

дескрипторам любого типа; в данном случае закрытие дескриптора поиска подобным способом приведет к генерации исключения. Вместо этого необходимо использовать следующую функцию:

BOOL FindClose(HANDLE hFindFile)

Функция **GetFileInformationByHandle** позволяет получить информацию о конкретном файле, на который указывает открытый дескриптор файла. Она также возвращает поле nNumberOfLinks, в котором содержится количество жестких ссылок на файл, установленных функцией CreateHardLink.

Описанный метод расширения метасимволов необходим даже в программах, запускаемых на выполнение из командной строки DOS, поскольку оболочка DOS не расширяет метасимволы.

Полные имена файлов

Полное имя файла можно получить, используя функцию GetFullPathName. Функция GetShortPathName возвращает имя файла в формате DOS 8.3, в предположении, что данный том поддерживает короткие имена файлов.

В NT 5.1 была введена функция SetFileShortName, позволяющая изменить существующее сокращенное имя файла или каталога. Иногда это оказывается удобным, поскольку интерпретация сокращенных имен файлов часто вызывает затруднения.

Другие методы определения атрибутов файлов и каталогов

Функции FindFirstFile и FindNextFile позволяют получить следующую информацию, связанную с атрибутами файла: флаги атрибутов, метки времени трех типов и размер файла. Существуют также другие аналогичные функции, одна из которых предназначена для задания атрибутов, причем эти функции могут работать непосредственно с открытыми дескрипторами файлов, не требуя просмотра каталогов или указания имен файлов. Три из этих функций, а именно, GetFileSize, GetFileSizeEx и SetEndOfFile, были описаны ранее в этой главе.

Для получения других атрибутов используются отдельные функции. Например, чтобы получить метки времени открытого файла, следует вызвать функцию GetFileTime.

BOOL GetFileTime(HANDLE hFile, LPFILETIME lpftCreation, LPFILETIME lpftLastAccess, LPFILETIME lpftLastWrite)

Указанные здесь и в структуре WIN32_FIND_DATA метки времени представляют собой 64-битовые целые числа без знака, которые выражают величину временного интервала, вычисленную относительно условного начала отсчета (1 января 1601 года) и преобразованную во время UTC (Universal Coordinated Time — всеобщее скоординированное время)[16], в 100-наносекундных единицах времени (10^7 единиц в 1 секунде). Для работы с

этими временными параметрами предусмотрено несколько удобных функций.

• Функция FileTimeToSystemTime (здесь не описывается; см. справочную систему Windows и программу 3.2) разбивает метки времени файла на отдельные блоки, соответствующие естественным единицам измерения, от годов до секунд и миллисекунд. Эти блоки удобно, например, использовать при выводе временных атрибутов файлов на экран или принтер.

• Функция SystemTimeToFileTime обращает этот процесс, преобразуя время, выраженное в естественных единицах, в метки времени файла.

• Функция CompareFileTime сравнивает метки времени двух файлов и в случае успешного завершения возвращает значение, зависящее от того, меньше (-1), равно (0) или больше (+1) значение метки времени первого файла по сравнению со значением метки времени второго файла.

• Для изменения меток времени служит функция SetFileTime; метки времени, не подлежащие изменению, при вызове функции указываются равными 0. NTFS поддерживает все три типа меток времени файлов, но FAT дает точные результаты только для меток времени последнего обращения.

• Функции FileTimeToLocalFileTime и LocalFileTimeToFileTime преобразуют значения меток времени, соответственно, от всеобщего скоординированного времени UTC к местному времени и наоборот.

Функция GetFileType, которая здесь подробно не описывается, позволяет различать файлы трех типов: дисковые, символьные (к ним, по сути, относятся такие устройства, как принтеры и консоли) и каналы (см. главу 11). Как и в большинстве других случаев, файл, характеристику которого необходимо определить, задается дескриптором.

Функция GetFileAttributes принимает в качестве аргумента имя файла или каталога, а всю информацию об атрибутах передает через свое возвращаемое значение dwFileAttributes.

DWORD GetFileAttributes(LPCTSTR lpFileName)

Возвращаемое значение: в случае успешного завершения — атрибуты файла, иначе — 0xFFFFFFFF.

Для определения атрибутов можно воспользоваться логическим сравнением возвращаемого значения функции с соответствующими масками значений атрибутов. Некоторые атрибуты, например атрибут временного файла, изначально устанавливаются функцией CreateFile. В качестве примера можно привести следующие атрибуты:

• FILE_ATTRIBUTE_DIRECTORY
• FILE_ATTRIBUTE_NORMAL
• FILE_ATTRIBUTE_READONLY
• FILE_ATTRIBUTE_TEMPORARY

Для изменения атрибутов именованных файлов служит функция SetFileAttributes.

В UNIX трем вышеописанным функциям Find соответствуют функции opendir, readdir и closedir. Функция stat предоставляет размер файла и значения меток времени, а также информацию о его индивидуальном или групповом владельце, необходимую для защиты файлов в UNIX. Разновидностями этой функции являются функции fstat и lstat. Эти функции позволяют также получать информацию о типе файла. Метки времени файла в UNIX устанавливаются с помощью функции utime. Эквивалента атрибута временного файла в UNIX не существует.

Именование временных файлов

Следующая функция создает имена для временных файлов. Файл может находиться в любом заданном каталоге, и его имя должно быть уникальным.

Функция GetTempFileName предоставляет уникальное имя файла с расширением .tmp, используя указанный путь доступа, и при необходимости создает файл. Эта функция широко используется в ряде следующих примеров (программа 6.1, программа 7.1 и другие).

UINT GetTempFileName(LPCTSTR lpPathName, LPCTSTR lpPrefixString, UINT uUnique, LPTSTR lpTempFileName)

Возвращаемое значение: уникальное числовое значение, используемое для создания имени файла. Этим значением будет значение параметра uUnique, если при вызове функции оно было задано ненулевым. В случае неудачного завершения функции возвращаемое значение равно нулю.

Параметры

lpPathName — каталог, в котором размещается временный файл. Типичным значением этого параметра является строка ".", указывающая на текущий каталог. В других случаях можно воспользоваться функцией Windows GetTempPath, которая предоставляет имя каталога, используемого для хранения временных файлов, но нами здесь не рассматривается.

lpPrefixString — префикс, используемый в имени временного файла. Допускаются лишь 8-битовые символы ASCII. Значение параметра uUnique обычно устанавливается равным нулю, чтобы функция самостоятельно сгенерировала уникальный четырехразрядный префикс и использовала его в имени создаваемого файла. При ненулевом значении этого параметра файл не создается, так что это необходимо сделать отдельно при помощи функции CreateFile, возможно — с использованием флага FILE_FLAG_DELETE_ON_CLOSE.

lpTempFileName — указатель на буфер, предназначенный для хранения имени временного файла. Размер буфера, выраженный в байтах, должен быть не менее MAXPATH. Результирующее полное имя файла получается объединением строк, соответствующих пути доступа к файлу, префикса, четырехразрядного шестнадцатеричного числа и суффикса .tmp.

Точки монтирования

NT 5.0 разрешает монтирование (или подключение) одной файловой системы в точке монтирования, находящейся в другой файловой системе. Обычно управление точками монтирования является прерогативой администратора системы, но эти же задачи можно решать и программным путем.

Функция SetVolumeMountPoint монтирует диск (второй аргумент) в точке монтирования, указанной первым аргументом. Например, вызов

SetVolumeMountPoint("C:\\mycd\\, "D:\\");

монтирует диск D: (которому в персональных системах часто соответствует привод компакт-диска) в каталоге mycd (точка монтирования), находящемуся на диске C:. Обратите внимание на то, что обозначения всех путей доступа заканчиваются символами обратной косой черты. Тогда после применения этой функции пути доступа C:\mycd\memos\book.doc будет соответствовать путь доступа D:\memos\book.doc.

Одну и ту же точку монтирования можно использовать для подключения нескольких файловых систем. Для размонтирования файловых систем служит функция DeleteMountPoint.

Функция GetVolumePathName возвращает корневую точку монтирования абсолютного или относительного пути доступа или имени файла. В свою очередь, функция GetVolumeNameForVolumeMountPoint предоставляет имя тома, например, C:\, соответствующего точке монтирования.

Пример: вывод списка атрибутов файла

Настало время увидеть функции управления файлами и каталогами в действии. Программа 3.2 представляет собой ограниченную версию команды UNIX ls, предназначенной для вывода содержимого каталогов, которая позволяет вывести дату и время последнего изменения файла и размер файла, хотя данная версия отображает лишь младшую часть размера файла.

Программа просматривает каталог для поиска файлов, соответствующих шаблону поиска. Для каждого найденного файла программа отображает имя файла и, если был задан параметр –1, то и его атрибуты. Данная программа иллюстрирует принцип построения многих, хотя и далеко не всех, функций Windows, предназначенных для работы с каталогами.

Значительная часть кода программы 3.2 отвечает за обход дерева каталогов. Заметьте, что каждый каталог проходится дважды: при первом проходе обрабатываются файлы, а при втором — подкаталоги, чем обеспечивается поддержка параметра рекурсивного обхода каталогов (-R).

В том виде, как она представлена ниже, программа 3.2 будет корректно выполняться в том случае, если при ее вызове используются относительные полные имена файлов, например:

lsW –R include*.h

Вместе с тем, в результате указания абсолютного полного имени файла, например:

lsW –R C:\Projects\ls\Debug*.obj

правильная работа программы будет нарушена, поскольку в ней самым существенным образом используется привязка каталогов к текущему каталогу. Завершенное решение (доступное на Web-сайте) анализирует абсолютные полные пути доступа к файлам и поэтому обеспечивает правильное выполнение программы и для второй команды.

Программа 3.2. lsw: вывод списка файлов и обход дерева каталогов

```
/* Глава 3. lsW — команда вывода списка файлов */
/* lsW [параметры] [файлы] */
#include "EvryThng.h"
BOOL TraverseDirectory(LPCTSTR, DWORD, LPBOOL);
DWORD FileType(LPWIN32_FIND_DATA);
BOOL ProcessItem(LPWIN32_FIND_DATA, DWORD, LPBOOL);

int _tmain(int  argc, LPTSTR argv[]) {
 BOOL Flags [MAX_OPTIONS], ok = TRUE;
 TCHAR PathName [MAX_PATH +1], CurrPath [MAX_PATH + 1];
 LPTSTR pSlash, pFileName;
 int i, FileIndex;
 FileIndex = Options(argc, argv, _T("R1"), &Flags[0], &Flags[1], NULL);
 I* "Разобрать" шаблон поиска на "родительскую часть" и имя файла. */
 GetCurrentDirectory(MAX_PATH, CurrPath); /* Сохранить текущий путь доступа. */
 if (argc < FileIndex +1) /* Путь доступа не указан. Использовать текущий каталог. */
  ok = TraverseDirectory(_T("*"), MAX_OPTIONS, Flags);
 else for (i = FileIndex; i < argc; i++) {
  /* Обработать все пути, указанные в командной строке. */
  ok = TraverseDirectory(pFileName,  MAX_OPTIONS, Flags) && ok;
  SetCurrentDirectory(CurrPath);
  /* Восстановить каталог. */
 }
 return ok ? 0 : 1;
}

static BOOL TraverseDirectory(LPCTSTR PathName, DWORD NumFlags, LPBOOL Flags)
/* Обход дерева каталогов; выполнить функцию ProcessItem для каждого случая совпадения. */
/* PathName: относительное или абсолютное имя просматриваемого каталога.*/
```

```
{
 HANDLE SearchHandle;
 WIN32_FIND_DATA FindData;
 BOOL Recursive = Flags[0];
 DWORD FType, iPass;
 TCHAR CurrPath[MAX_PATH + 1];
 GetCurrentDirectory(MAX_PATH, CurrPath);
 for (iPass = 1; iPass <= 2; iPass++) {
  /* Проход 1: вывод списка файлов. */
  /* Проход 2: обход дерева каталогов (если задана опция –R). */
  SearchHandle = FindFirstFile(PathName, &FindData);
  do {
   FType = FileType(&FindData);
   /* Файл или каталог? */
   if (iPass == 1) /* Вывести имя и атрибуты файла. */
    ProcessItem(&FindData, MAX_OPTIONS, Flags);
   if (FType == TYPE_DIR && iPass == 2 && Recursive) {
    /* Обработать подкаталог. */
    _tprintf(_T ("\n%s\\%s:"), CurrPath, FindData.cFileName);
    /* Подготовка к обходу каталога. */
    SetCurrentDirectory(FindData.cFileName);
    TraverseDirectory(_T("*"), NumFlags, Flags);
    /* Рекурсивный вызов. */
    SetCurrentDirectory(_T(".."));
   }
  } while (FindNextFile(SearchHandle, &FindData));
  FindClose (SearchHandle);
 }
 return TRUE;
}

static BOOL ProcessItem(LPWIN32_FIND_DATA pFileData, DWORD
NumFlags, LPBOOL Flags)
 /* Выводит список атрибутов файла или каталога. */
 {
  const TCHAR FileTypeChar[] = {' ', 'd'};
  DWORD FType = FileType(pFileData);
  BOOL Long = Flags[1];
  SYSTEMTIME LastWrite;
  if (FType != TYPE_FILE && FType != TYPE_DIR) return FALSE;
  _tprintf(_T ("\n"));
  if (Long) { /* Указан ли в командной строке параметр "-1"? */
   _tprintf(_T("%c"), FileTypeChar[FType – 1]);
   _tprintf(_T("%10d"), pFileData->nFileSizeLow);
   FileTimeToSystemTime(&(pFileData->ftLastWriteTime), &LastWrite);
```

```
      _tprintf(_T(" %02d/%02d/%04d %02d:%02d:%02d"), LastWrite.wMonth,
LastWrite.wDay, LastWrite.wYear, LastWrite.wHour, LastWrite.wMinute,
LastWrite.wSecond);
    }
    _tprintf(_T(" %s"), pFileData->cFileName);
    return TRUE;
   }

   static DWORD FileType(LPWIN32_FIND_DATA pFileData)
   /* Поддерживаемые типы файлов – TYPE_FILE: файл; TYPE_DIR: каталог;
TYPE_DOT: каталоги . или .. */
   {
    BOOL IsDir;
    DWORD FType;
    FType = TYPE_FILE;
    IsDir = (pFileData->dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) !=
0;
    if (IsDir) if (lstrcmp(pFileData->cFileName, _T(".")) == 0 || lstrcmp(pFileData-
>cFileName, _T("..")) == 0) FType = TYPE_DOT;
    else FType = TYPE_DIR;
    return FType;
   }
```

Пример: установка меток времени файла

Программа 3.3 реализует UNIX-команду touch, предназначенную для изменения кода защиты файлов и обновления меток времени до текущих значений системного времени. В упражнении 3.11 от вас требуется расширить возможности функции touch таким образом, чтобы новые значения меток времени можно было указывать в параметрах командной строки.

Программа 3.3. touch: установка меток даты и времени файла

```
   /* Глава 3. команда touch. */
   /* touch [параметры] [файлы] */
   #include "EvryThng.h"

   int _tmain(int argc, LPTSTR argv[]) {
    SYSTEMTIME SysTime;
    FILETIME NewFileTime;
    LPFILETIME pAccessTime = NULL, pModifyTime = NULL;
    HANDLE hFile;
    BOOL Flags[MAX_OPTIONS], SetAccessTime, SetModTime, CreateNew;
    DWORD CreateFlag;
```

```
    int i, FileIndex;
    FileIndex = Options(argc, argv, _T("amc"), &Flags[0], &Flags[1], &Flags[2],
NULL);
    SetAccessTime = !Flags[0];
    SetModTime = !Flags[1];
    CreateNew = !Flags[2];
    CreateFlag = CreateNew ? OPEN_ALWAYS : OPEN_EXISTING;
    for (i = FileIndex; i < argc; i++) {
     hFile = CreateFile(argv[i], GENERIC_READ | GENERIC_WRITE, 0, NULL,
CreateFlag, FILE_ATTRIBUTE_NORMAL, NULL);
     GetSystemTime(&SysTime);
     SystemTimeToFileTime(&SysTime, &NewFileTime);
     if (SetAccessTime) pAccessTime = &NewFileTime;
     if (SetModTime) pModifyTime = &NewFileTime;
     SetFileTime(hFile, NULL, pAccessTime, pModifyTime);
     CloseHandle(hFile);
    }
    return 0;
   }
```

Стратегии обработки файлов

Уже на ранних стадиях любого проекта разработки приложения или подготовки его к переносу на другую платформу приходится принимать решение относительно того, должна ли осуществляться обработка файлов с использованием функций библиотеки C или функций Windows. Характер этого решения не относится к категории "или-или", поскольку при соблюдении определенных мер предосторожности смешанное применение функций возможно даже по отношению к одному и тому же файлу.

Библиотека C обладает рядом явных преимуществ, среди которых можно выделить следующие:

• Полученный программный код легко переносится на другие системы.

• Наличие удобных функций для работы с символами и строками, не имеющих прямых аналогов среди функций Windows, упрощает обработку строк.

• Функции библиотеки C обычно проще в использовании по сравнению с функциями Windows.

• Функции, ориентированные на обработку символьных строк и потоков, легко преобразовать к форме, допускающей указание обобщенных символов при их вызове, хотя преимущества переносимости при этом будут утеряны.

• Как показано в главе 7, функции библиотеки C способны работать и в средах с многопоточной поддержкой.

Тем не менее, использование библиотеки C налагает некоторые ограничения. В пользу этого утверждения можно привести перечисленные ниже соображения:

• Средства библиотеки C не обеспечивают управление каталогами и обход дерева каталогов и в большинстве случаев не позволяют получать или устанавливать атрибутов файлов.

• В функции fseek, входящей в библиотеку C, используются 32-битовые указатели файла, и поэтому, несмотря на возможность последовательного считывания гигантских файлов, установка произвольной позиции в таком файле, как это требуется, например, в программе 3.1, оказывается невозможной.

• Библиотека C не предоставляет такие развитые возможности, как защита файлов, отображение файлов, блокирование файлов, асинхронный ввод/вывод и взаимодействие между процессами. Вместе с тем, как показано в приложении B, использование некоторых из этих возможностей в ряде случаев может обеспечивать существенное улучшение показателей производительности программ.

Альтернативным вариантом является перенос существующего UNIX-кода с привлечением библиотеки совместимости (compatibility library). Microsoft C предоставляет ограниченную библиотеку совместимости, включающую многие, хотя и далеко не все, функции UNIX. К числу функций UNIX, входящих в состав библиотеки Microsoft, относятся функции ввода/вывода, однако большинство функций управления процессами, не говоря о многих других функциях, в ней отсутствуют. В именах функций-аналогов присутствует префикс в виде символа подчеркивания, например, _read, _write, _stat и так далее.

Решения относительно смешанного использования функций библиотеки C, библиотеки совместимости и Win32/64 API должны приниматься на основании требований проекта. Многие из преимуществ функций Windows будут продемонстрированы в следующих главах, а для ознакомления с количественными данными, характеризующими производительность, которые пригодятся вам в тех случаях, когда этот фактор становится решающим, вы можете обратиться к приложению B.

Блокирование файлов

В системах, допускающих одновременное выполнение нескольких процессов, особую актуальность приобретает проблема координации и синхронизации доступа к разделяемым (совместно используемым) объектам, например файлам.

В Windows имеется возможность блокировать файлы (целиком или частично) таким образом, что никакой другой процесс (выполняющаяся программа) не сможет получить доступ к заблокированному участку файла. Блокирование файла может оставлять другим приложениям возможность доступа только для чтения (разделяемый доступ) или же закрывать им доступ к файлу как для записи, так и для чтения (монопольный доступ). Что немаловажно, владельцем блокировки является блокирующий процесс. Любая попытка получения доступа к части файла (с помощью функций

ReadFile или WriteFile) в нарушение существующей блокировки закончится неудачей, поскольку блокировки носят обязательный характер на уровне процесса. Любая попытка получения несовместимой блокировки также завершится неудачей, даже если процесс уже владеет данной блокировкой. Блокирование файлов является ограниченной разновидностью синхронизации параллельно выполняющихся процессов и потоков; обсуждение синхронизации с использованием гораздо более общей терминологии начнется в главе 8.

Для блокирования файлов предусмотрены две функции. Более общей из них является функция LockFileEx, менее общей — LockFile, которую можно использовать и в Windows 9x.

Функция LockFileEx относится к классу функций расширенного (extended) ввода/вывода, поэтому для указания 64-битовой позиции в файле и границ области файла, подлежащей блокированию, необходимо использовать структуру OVERLAPPED, которая ранее уже применялась при указании позиции в файле для функций ReadFile и WriteFile.

BOOL LockFileEx(HANDLE hFile, DWORD dwFlags, DWORD dwReserved, DWORD nNumberOfBytesToLockLow, DWORD nNumberOfBytesToLockHigh, LPOVERLAPPED lpOverlapped)

Функция LockFileEx блокирует участок открытого файла либо для разделяемого доступа (разрешающего доступ одновременно нескольким приложениям в режиме чтения), либо для монопольного доступа (разрешающего доступ только одному приложению в режиме чтения/записи).

Параметры

hFile — дескриптор открытого файла. Дескриптор должен быть создан либо с правами доступа GENERIC_READ, либо с правами доступа GENERIC_READ и GENERIC_WRITE.

dwFlags — определяет вид блокировки файла, а также режим ожидания доступности затребованной блокировки. Этот параметр определяется комбинацией следующих значений:

LOCKFILE_EXCLUSIVE_LOCK — запрос монопольной блокировки в режиме чтения/записи. Если это значение не задано, запрашивается разделяемая блокировка (только чтение).

LOCKFILE_FAIL_IMMEDIATELY — задает режим немедленного возврата функции с возвращаемым значением равным FALSE, если приобрести блокировку не удается. Если это значение не задано, функция переходит в режим ожидания.

dwReserved — значение этого параметра должно устанавливаться равным 0. Следующие два параметра определяют соответственно младшие и старшие 32-битовые значения размера блокируемого участка файла (в байтах).

lpOverlapped — указатель на структуру данных OVERLAPPED, содержащую информацию о начале блокируемого участка. В этой структуре необходимо устанавливать значения трех элементов (остальные элементы

игнорируются), первые два из которых определяют смещение начала блокируемого участка от начала файла.

• DWORD Offset (используется именно такое имя параметра, а не OffsetLow).

• DWORD OffsetHigh.

• HANDLE hEvent должен задаваться равным 0.

Чтобы разблокировать файл, следует вызвать функцию UnlockFileEx, все параметры которой, за исключением dwFlags, совпадают с параметрами предыдущей функции:

BOOL UnlockFileEx(HANDLE hFile, DWORD dwReserved, DWORD nNumberOfBytesToLockLow, DWORD nNumberOfBytesToLockHigh, LPOVERLAPPED lpOverlapped)

Используя блокирование файлов, вы должны принимать во внимание следующие обстоятельства:

• Границы области разблокирования должны в точности совпадать с границами ранее заблокированной области. Не допускается, например, объединение двух ранее заблокированных областей или разблокирование части заблокированной области. Любая попытка разблокирования области, не совпадающей в точности с одной из существующих заблокированных областей, будет неудачной. В этом случае функция вернет значение FALSE, а в выведенном системой сообщении об ошибке будет указано, что данная область блокирования не существует.

• Вновь создаваемая и существующие области блокирования в файле не могут перекрываться, если это приводит к возникновению конфликтной ситуации.

• Возможно блокирование участка, границы которого выходят за пределы файла. Такая операция может оказаться полезной в случае расширения файла процессом или потоком.

• Блокировки не наследуются вновь создаваемыми процессами.

Обычно операции чтения и записи выполняются путем вызова функций ReadFile и WriteFile или их расширенных версий ReadFileEx и WriteFileEx. Для диагностики ошибок, возникающих в процессе выполнения операций ввода/вывода, следует вызывать функцию GetLastError.

Одна из разновидностей операций ввода/вывода с участием файлов предполагает использование отображения файлов. Обнаружение конфликтов блокировки на этапе обращения к памяти не производится; такая проверка осуществляется во время вызова функции MapViewOfFile. Указанная функция делает часть файла доступной для процесса, вследствие чего проверка наличия блокировок на этом этапе является необходимой.

Разновидностью функции LockFileEx с ограниченной сферой применимости является функция LockFile, вызов которой, скорее, лишь уведомляет о намерении осуществить блокировку. Функция LockFile предоставляет блокирующему процессу только монопольный доступ, а возврат из функции происходит сразу же. Таким образом, функция LockFile

не блокируется. Проверить, предоставлена блокировка или нет, можно путем тестирования возвращаемого функцией значения.

Снятие блокировок

Каждый успешный вызов функции LockFileEx должен сопровождаться последующим вызовом функции UnlockFileEx (то же самое касается и пары функций LockFile и UnlockFile). Если программа не позаботится о снятии блокировки или будет удерживать ее в течение большего, чем это необходимо, времени, другие программы либо вовсе не смогут работать, либо будут вынуждены простаивать. Поэтому уже на стадии проектирования и реализации программ необходимо очень тщательно следить за тем, чтобы снятие блокировки осуществлялось сразу же после того, как необходимость в ней отпала, а логика работы программ не позволяла оставлять невыполненными необходимые операции разблокирования файлов.

Одним из способов, гарантирующих своевременное разблокирования файлов, является использование дескрипторов завершения (termination handlers).

Следствия принятой логики блокирования файлов

Несмотря на всю естественность логики блокирования файлов, представленной в таблицах 3.1 и 3.2, последствия ее применения могут оказаться для вас неожиданными и вызвать на первый взгляд необъяснимые изменения в поведении программы. Некоторые возможные примеры этого приводятся ниже.

• Предположим, что процессы A и B периодически приобретают разделяемые блокировки файла, а процесс C блокируется при попытке получения монопольной блокировки того же файла после того, как процесс A стал владельцем собственной разделяемой блокировки. В этих условиях процесс B может получить свою разделяемую блокировку, но процесс C будет оставаться блокированным даже после того, как процесс A снимет свою блокировку файла. Процесс C будет оставаться блокированным до тех пор, пока все процессы не снимут свои блокировки, даже если они были получены уже тогда, когда процесс C пребывал в блокированном состоянии. Согласно этому сценарию процесс C может оставаться блокированным сколь угодно долго, тогда как другие процессы сохраняют возможность управления своими разделяемыми блокировками.

• Предположим, что процесс A стал владельцем разделяемой блокировки файла, а процесс B пытается осуществить считывание файла без предварительного приобретения разделяемой блокировки. В этой ситуации чтение может быть успешно осуществлено даже несмотря на то, что процесс, выполняющий чтение, не владеет ни одной блокировкой данного файла, поскольку операция чтения не вступает в конфликт с существующей разделяемой блокировкой.

• Все, о чем говорилось выше, относится не только к блокировке файла в целом, но и к блокировке отдельного его участка.

• Процессы чтения и записи вполне могут успешно завершить часть своего запроса, прежде чем возникнет конфликт с существующей блокировкой. В этом случае функции чтения и записи возвратят значения FALSE, а значение счетчика переданных байтов окажется меньше затребованного.

Использование блокирования файлов

Рассмотрение примеров блокирования файлов мы отложим до главы 6, в которой обсуждается управление процессами. В программах 4.2, 6.4, 6.5 и 6.6 блокирование файлов используется для обеспечения того, чтобы в каждый момент времени изменять файл мог только один процесс.

В UNIX блокирование файлов является *уведомляющим* (advisory); выполнение процесса ввода/вывода может продолжаться даже в том случае, если попытка получения блокировки оказалась неудачной (логика, отраженная в табл. 3.1, действует и в этом случае). Это обеспечивает в UNIX возможность блокирования файлов взаимодействующими процессами, но любой другой процесс может нарушить описанный протокол.

Для получения уведомляющей блокировки используются параметры, указываемые при вызове функции fcntl. Допустимыми командами (второй параметр) являются F_SETLK, F_SETLKW и F_GETLK. Информация о типе блокировки (F_RDLCK, F_WRLCK или F_UNLCK) и блокируемой области содержится в дополнительной структуре данных.

Помимо этого, в некоторых UNIX-системах доступна *обязательная* (mandatory) блокировка, обеспечиваемая путем определения групповых полномочий для файла с помощью команды chmode.

Блокирование файлов в UNIX имеет много особенностей. Например, блокировки наследуются при выполнении вызова функции exec.

Блокирование файлов библиотекой C не поддерживается, но в Visual C++ обеспечивается поддержка нестандартных расширений механизма блокирования.

Реестр

Реестр — это централизованная иерархическая база данных, хранящая информацию о параметрах конфигурации операционной системы и установленных приложений. Доступ к реестру осуществляется через *разделы,* или *ключи, реестра* (registry keys), играющие ту же роль, что и каталоги в файловой системе. Раздел может содержать подразделы или пары "имя-значение", в которых между именем и значением существует примерно та же взаимосвязь, что и между именами файлов и их содержимым.

Пользователь или системный администратор может просматривать и изменять содержимое реестра, пользуясь редактором реестра, для запуска которого необходимо выполнить команду REGEDIT. Реестром можно

управлять также из программ, используя функции API реестра, описанные в данном разделе.

Примечание

Программирование реестра обсуждается в данной главе по той причине, что решаемая при этом задача весьма напоминает обработку файлов, а также потому, что оно играет важную роль в некоторых, хотя и не во всех, приложениях. Соответствующий пример будет получен путем несложного изменения программы lsW. Вместе с тем, данный раздел вполне мог бы стать небольшой отдельной главой. Поэтому читатели, для которых программирование реестра не представляет непосредственного интереса, могут пропустить этот раздел, чтобы вернуться к нему впоследствии, если это окажется необходимым.

В парах "имя-значение" реестра хранится следующая информация:

- Номер версии операционной системы, номер сборки и информация о зарегистрированном пользователе.
- Аналогичная информация обо всех приложения, которые были надлежащим образом установлены в системе.
- Информация о типе процессоров в системе и их количестве, системной памяти и тому подобное.
- Специфическая для каждого отдельного пользователя системы информация, включая данные относительно основного каталога пользователя и предпочтительных пользовательских настройках приложений.
- Информация, необходимая для системы безопасности, включая имена учетных записей пользователей.
- Информация об установленных службах (глава 13).
- Список соответствий между расширениями имен файлов и ассоциированными с ними исполняемыми программами. Именно эти соответствия используются системой после того, как пользователь щелкнет на пиктограмме какого-либо файла. Например, щелчок на файле с расширением .doc может приводить к запуску текстового редактора Microsoft Word.
- Отображения сетевых адресов на имена, используемые локальным компьютером.

В операционной системе UNIX аналогичная информация хранится в каталоге /etc и файлах, находящихся в основном каталоге пользователя. В Windows 3.1 для этих целей использовались .INI-файлы. Реестр обеспечивает единообразное централизованное хранение всей информации подобного рода. Кроме того, используя средства защиты, описанные в главе 15, можно обеспечить безопасность реестра.

API управления реестром описывается ниже, однако подробное рассмотрение содержимого и смысла различных записей, образующих реестр, выходит за рамки данной книги. Рассмотрение принципов реализации реестра, включая организацию хранения и извлечения хранящихся в реестре данных, выходит за рамки данной книги; для более

глубокого изучения этих вопросов обратитесь к списку дополнительной литературы, приведенному в конце главы.

Ключи реестра

Есть аналогия между разделами реестра и каталогами файловой системы. Каждый раздел может содержать другие разделы или последовательности пар "имя-значение". В то время как доступ к файловой системе реализуется посредством указания путей доступа, доступ к реестру осуществляется через его разделы. Существует несколько предопределенных разделов, которые играют роль точек входа в реестр.

- HKEY_LOCAL_MACHINE. В этом разделе хранится информация об оборудовании локального компьютера и установленном на нем программном обеспечении. Информация об установленном программном обеспечении обычно создается в подразделах (subkeys) в виде: SOFTWARE\НазваниеКомпании\НазваниеПродукта\Версия.
- HKEY_USERS. В этом разделе хранится информация о настройке пользовательских конфигураций.
- HKEY_CURRENT_CONFIG. В этом разделе хранятся текущие настройки таких параметров, как разрешение дисплея или гарнитура шрифта.
- HKEY_CLASSES_ROOT. В этом разделе содержатся подчиненные записи, устанавливающие соответствие между именами файлов и классами, а также приложениями, используемыми оболочкой для доступа к объектам, имена которых имеют определенные расширения. В этот раздел также входят все подразделы, необходимые для функционирования модели компонентных объектов (Component Object Model — COM), разработанной компанией Microsoft.
- HKEY_CURRENT_USER. В этом разделе хранится информация, определяемая пользователем, в том числе информация о переменных среды, принтерах и предпочтительных для вошедшего в систему пользователя конфигурационных параметрах приложений.

Управление системным реестром

Функции управления реестром позволяют запрашивать и изменять данные, относящиеся к парам "имя-значение", а также создавать новые подразделы и новые пары "имя-значение". Как для указания существующих разделов, так и для создания новых используются дескрипторы типа HKEY.[17] Нужные значения необходимо вводить; тип значения можно выбрать из нескольких готовых вариантов, соответствующих, например, строкам, двойным словам или расширяемым (expandable) строкам, параметры которых могут быть заменены переменными окружения.

Управление подразделами реестра

Первая из рассматриваемых нами функций, RegOpenKeyEx, предназначена для открытия подразделов системного реестра. Начав с одного из предопределенных зарезервированных дескрипторов, вы можете получить дескриптор любого из его подразделов, совершая обход дерева разделов.

LONG RegOpenKeyEx(HKEY hKey, LPCTSTR lpSubKey, DWORD ulOptions, REGSAM samDesired, PHKEY phkResult)

Параметры

hKey — указатель на текущий открытый раздел реестра или значение дескриптора одного из предопределенных зарезервированных разделов. pHkResult — указатель на переменную типа HKEY, получающую значение дескриптора вновь открываемого раздела.

lpSubKey — указатель на строку с именем подраздела. Именем подраздела может быть путь, например: Microsoft\WindowsNT\CurrentVersion. Значению NULL соответствует открытие новой копии раздела для hKey. Значение параметра ulOptions должно быть равным 0.

samDesired — маска доступа, описывающая уровень защиты нового раздела. К числу возможных значений относятся значения KEY_ALL_ACCESS, KEY_WRITE, KEY_QUERY_VALUE и KEY_ENUMARATE_SUBKEYS.

В случае успешного завершения функции возвращается значение ERROR_SUCCESS. Возврат любого другого значения указывает на ошибку. Для закрытия дескриптора открытого раздела используется функция RegCloseKey, которая в качестве своего единственного параметра принимает дескриптор.

Для получения имен подразделов любого заданного раздела следует воспользоваться функцией RegEnumKeyEx.

Для получения пар "имя-значение" используются две взаимно дополняющих функции: RegEnumValue и RegQueryValueEx.[18] Функция RegSetValueEx сохраняет данные различного типа в поле значения открытого раздела реестра. Описания перечисленных функций, применение которых будет проиллюстрировано примером, содержатся в этом и следующем разделах книги.

Функция RegEnumKeyEx перечисляет подразделы открытого раздела системного реестра во многом аналогично тому, как функции FindFirstFile и FindNextFile перечисляют содержимое каталогов. Эта функция извлекает имя подраздела, строку с именем класса подраздела, а также дату и время последнего изменения.

LONG RegEnumValue(HKEY hKey, DWORD dwIndex, LPTSTR lpValueName, LPDWORD lpcbValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)

Значение параметра dwIndex должно устанавливаться равным 0 при первом вызове функции и увеличиваться на единицу при каждом последующем вызове. Название раздела и его размер, а также строка с именем класса и ее

размер, возвращаются обычным способом. В случае успешного завершения функция возвращает значение ERROR_SUCCESS, иначе — код ошибки.

Можно также создавать новые разделы, используя для этого функцию RegCreateKeyEx. Разделам системного реестра можно присваивать атрибуты защиты точно так же, как каталогам и файлам.

LONG RegCreateKeyEx(HKEY hKey, LPCTSTR lpSubKey, DWORD Reserved, LPTSTR lpClass, DWORD dwOptions, REGSAM samDesired, LPSECURITY_ATTRIBUTES lpSecurityAttributes, PHKEY phkResult, LPDWORD lpdwDisposition)

Параметры

lpSubKey — указатель на строку, содержащую имя нового подраздела, создаваемого в разделе, на который указывает дескриптор hKey.

lpClass — указатель на строку, содержащую имя класса, или объектный тип, раздела, описывающее данные, представляемые разделом. Одними из многочисленных возможных значений являются REG_SZ (строка, завершающаяся нулевым символом) и REG_DWORD (двойное слово).

Параметр dwOptions может иметь значение 0 или одно из двух взаимоисключающих значений — REG_OPTION_VOLATILE или REG_OPTION_NON_VOLATILE. Постоянно хранимая (nonvolatile) информация системного реестра сохраняется в файле на диске и не теряется после перезапуска системы. При этом временные (volatile) разделы системного реестра, хранящиеся в оперативной памяти, не будут восстановлены.

Параметр samDesired имеет тот же смысл, что и в случае функции RegOpenKeyEx.

Параметр lpSecurityAttributes может принимать значение NULL или указывать атрибуты защиты. Опции прав доступа к разделу могут выбираться из того же набора значений, что и в случае параметра samDesired.

lpdwDisposition — указатель на переменную типа DWORD, значение которой говорит о том, существовал ли раздел ранее (REG_OPENED_EXISTING_KEY) или он только создается (REG_CREATED_NEW_KEY).

Для удаления раздела используется функция RegDeleteKey. Двумя ее параметрами являются дескриптор открытого раздела и имя подраздела.

Управление значениями

Для перечисления значений параметров открытого раздела реестра используется функция RegEnumValue. Значение параметра dwIndex должно устанавливаться равным 0 при первом вызове функции и увеличиваться на единицу при каждом последующем вызове. После возврата из функции вы получаете строку, содержащую имя перечисляемого параметра, а также

размер данных. Кроме того, вы получаете значение перечисляемого параметра и его тип.

LONG RegEnumValue(HKEY hKey, DWORD dwIndex, LPTSTR lpValueName, LPDWORD lpcbValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData)

Фактическое значение параметра возвращается в буфере, на который указывает указатель lpData. Размер результата содержится в переменной, на которую указывает указатель lpcbData. Тип данных, содержащийся в переменной, на которую указывает указатель lpType, может быть самым различным, включая REG_BINARY, REG_DWORD, REG_SZ (строка) и REG_EXPAND_SZ (расширяемая строка с параметрами, которые заменяются переменными окружения). Полный список типов данных системного реестра можно найти в оперативной справочной системе.

Чтобы определить, все ли параметры перечислены, следует проверить возвращаемое значение функции. После успешного нахождения действительного параметра оно должно быть равным ERROR_SUCCESS.

Функция RegQueryValueEx ведет себя аналогичным образом, за исключением того, что требует указания имени перечисляемого параметра, а не его индекса. Эту функцию можно использовать в тех случаях, когда известны имена параметров. Если же имена параметров неизвестны, следует использовать функцию RegEnumValueEx.

Для установки значения параметра в открытом разделе служит функция RegSetValueEx, которой необходимо предоставить имя параметра, тип значения и фактические данные, образующие значение.

LONG RegSetValueEx(HKEY hKey, LPCTSTR lpValueName, DWORD Reserved, DWORD dwType, CONST BYTE * lpData, CONST cbData)

Наконец, для удаления именованных значений используется функция RegDeleteValue.

Пример: вывод списка разделов и содержимого реестра

Программа lsReq (программа 3.4), является видоизменением lsW (программа 3.2, предназначенная для вывода списка файлов и каталогов) и обрабатывает не каталоги и файлы, а разделы и пары "имя-значение" системного реестра.

Программа 3.4. lsReq: вывод списка разделов и содержимого системного реестра

```
/* Глава 3. lsReg: Команда вывода содержимого реестра. Адаптированная версия программы 3.2. */
/* lsReg [параметры] подраздел */
#include "EvryThng.h"
BOOL TraverseRegistry(HKEY,  LPTSTR, LPTSTR, LPBOOL);
BOOL DisplayPair(LPTSTR, DWORD, LPBYTE, DWORD, LPBOOL);
```

```
BOOL DisplaySubKey (LPTSTR, LPTSTR, PFILETIME, LPBOOL);

int _tmain(int argc, LPTSTR argv[]) {
 BOOL Flags[2], ok = TRUE;
 TCHAR KeyName[MAX_PATH + 1];
 LPTSTR pScan;
 DWORD i, KeyIndex;
 HKEY hKey, hNextKey;
 /* Таблица предопределенных имен и дескрипторов разделов. */
 LPTSTR PreDefKeyNames[] = {
  _T("HKEY_LOCAL_MACHINE"), _T("HKEY_CLASSES_ROOT"),
  _T("HKEY CURRENT USER"), _T ("HKEY CURRENT CONFIG"), NULL
 };
 HKEY PreDefKeys[] = {
            HKEY_LOCAL_MACHINE, HKEY_CLASSES_ROOT,
HKEY_CURRENT_USER, HKEY_CURRENT_CONFIG
 };
 KeyIndex = Options(argc, argv, _T("Rl"), &Flags[0], &Flags[1], NULL);
 /* "Разобрать" шаблон поиска на "раздел" и "подраздел". */
 /* Воссоздать раздел. */
 pScan = argv[KeyIndex];
 for (i = 0; *pScan != _T('\\') && *pScan != _T('\0'); pScan++, i++) KeyName [i]
= *pScan;
 KeyName[i] = _T('\0');
 if (*pScan == _T('\\')) pScan++;
 /* Преобразовать предопределенное имя раздела в соответствующий
HKEY.*/
 for (i = 0; PreDefKeyNames [i] != NULL && _tcscmp(PreDefKeyNames[i],
KeyName) != 0; i++);
 hKey = PreDefKeys[i];
 RegOpenKeyEx(hKey, pScan, 0, KEY_READ, &hNextKey);
 hKey = hNextKey;
 ok = TraverseRegistry(hKey, argv[KeyIndex], NULL, Flags);
 return ok ? 0 : 1;
}

BOOL TraverseRegistry(HKEY hKey, LPTSTR FullKeyName, LPTSTR
SubKey, LPBOOL Flags)
 /*Совершить обход разделов и подразделов реестра, если задан параметр –
R.*/
 {
 HKEY hSubK;
 BOOL Recursive = Flags[0];
 LONG Result;
 DWORD ValType, Index, NumSubKs, SubKNameLen, ValNameLen, ValLen;
```

```
   DWORD MaxSubKLen, NumVals, MaxValNameLen, MaxValLen;
   FILETIME LastWriteTime;
   LPTSTR SubKName, ValName;
   LPBYTE Val;
   TCHAR FullSubKName[MAX_PATH + 1];
   /* Открыть дескриптор раздела. */
   RegOpenKeyEx(hKey, SubKey, 0, KEY_READ, &hSubK);
   /* Определить максимальный размер информации относительно раздела и
распределить память. */
   RegQueryInfoKey(hSubK, NULL, NULL, NULL, &NumSubKs,
&MaxSubKLen, NULL, &NumVals, &MaxValNameLen, &MaxValLen, NULL,
&LastWriteTime);
   SubKName = malloc (MaxSubKLen+1); /* Размер без учета завершающего
нулевого символа. */
   ValName = malloc(MaxValNameLen+1); /* Учесть нулевой символ. */
   Val = malloc(MaxValLen); /* Размер в байтах. */
   /* Первый проход: пары "имя-значение". */
   for (Index = 0; Index < NumVals; Index++) {
    ValNameLen = MaxValNameLen + 1; /* Устанавливается каждый раз! */
    ValLen = MaxValLen + 1;
     RegEnumValue(hSubK, Index, ValName, &ValNameLen, NULL, &ValType,
Val, &ValLen);
    DisplayPair(ValName, ValType, Val, ValLen, Flags);
   }
   /* Второй проход: подразделы. */
   for (Index = 0; Index < NumSubKs; Index++) {
    SubKNameLen = MaxSubKLen + 1;
     RegEnumKeyEx(hSubK, Index, SubKName, &SubKNameLen, NULL, NULL,
NULL, &LastWriteTime);
    DisplaySubKey(FullKName, SubKName, &LastWriteTime, Flags);
    if (Recursive) {
     _stprintf(FullSubKName, _T("%s\\%s"), FullKName, SubKName);
     TraverseRegistry(hSubK, FullSubKName, SubKName, Flags);
    }
   }
   _tprintf(_T("\n"));
   free(SubKName);
   free(ValName);
   free(Val);
   RegCloseKey(hSubK);
   return TRUE;
  }

  BOOL DisplayPair(LPTSTR ValueName, DWORD ValueType, LPBYTE
Value, DWORD ValueLen, LPBOOL Flags)
```

```
  /* Функция, отображающая пары "имя-значение". */
  {
   LPBYTE pV = Value;
   DWORD i;
   _tprintf(_T("\nValue: %s = "), ValueName);
   switch (ValueType) {
   case REG_FULL_RESOURCE_DESCRIPTOR: /* 9: описание оборудования.
*/
   case REG_BINARY: /* 3: Любые двоичные данные. */
    for (i = 0; i < ValueLen; i++, pV++) _tprintf (_T (" %x"), *pV);
    break;
   case REG_DWORD: /* 4: 32-битовое число. */
    _tprintf(_T ("%x"), (DWORD)*Value);
    break;
   case REG_MULTI SZ: /*7: массив строк, завершающихся нулевым
символом.*/
   case REG_SZ: /* 1: строка, завершающаяся нулевым символом. */
    _tprintf(_T("%s"), (LPTSTR)Value);
    break;
   /* … Несколько других типов … */
   }
   return TRUE;
  }

  BOOL DisplaySubKey(LPTSTR KeyName, LPTSTR SubKeyName,
PFILETIME pLastWrite, LPBOOL Flags) {
   BOOL Long = Flags[1];
   SYSTEMTIME SysLastWrite;
   _tprintf(_T("\nSubkey: %s"), KeyName);
   if (_tcslen(SubKeyName) > 0) _tprintf (_T ("\\%s "), SubKeyName);
   if (Long) {
    FileTimeToSystemTime(pLastWrite, &SysLastWrite);
    _tprintf(_T("%02d/%02d/%04d %02d:%02d:%02d"), SysLastWrite.wMonth,
SysLastWrite.wDay, SysLastWrite.wYear, SysLastWrite.wHour,
SysLastWrite.wMinute, SysLastWrite.wSecond);
   }
   return TRUE;
  }
```

Резюме

В главах 2 и 3 описаны все наиболее важные базовые функции, необходимые для работы с файлами, каталогами и консольным вводом/выводом. Использование этих функций для построения типичных приложений иллюстрировали многочисленные примеры. Как показывает

последний из примеров, между управлением системным реестром и управлением файловой системой имеется много общего.

В последующих главах будут рассмотрены такие усовершенствованные методы ввода/вывода, как асинхронные операции ввода/вывода и отображение файлов. Этих средств будет достаточно для того, чтобы воспроизвести в Windows почти любой из обычных видов обработки файлов, доступных при использовании UNIX или библиотечных функций C.

Упражнения

3.1. Используя функции GetDiskFreeSpace и GetDiskFreeSpaceEx, определите, насколько разреженным оказывается файловое пространство, распределяемое различными версиями операционной системы Windows. Например, создайте новый файл, установите для указателя файла большое значение, задайте размер файла и исследуйте наличие свободного пространства на жестком диске при помощи функции GetDiskFreeSpace. Эту же функцию Windows можно использовать для определения того, чтобы определить, каким образом сконфигурирован диск в терминах секторов и кластеров. Определите, инициализируется ли выделенное для вновь созданного файла дисковое пространство. Решение в виде исходного текста функции FreeSpace.c доступно на Web-сайте книги. Сравните результаты, полученные для столь различных систем, как Windows NT и Windows 9x. Представляет интерес также исследование вопроса о том, как сделать файл разреженным.

3.2. Что произойдет, если длину файла задать такой, чтобы его размер превышал объем диска? Обеспечивает ли Windows изящный выход из функции в случае ее неудачного завершения?

3.3. Измените предоставляемую на Web-сайте программу tail.c таким образом, чтобы в ней можно было обойтись без применения функции SetFilePointer; воспользуйтесь для этого структурой OVERLAPPED.

3.4. Исследуйте значение поля "количество ссылок" (nNumberOfLinks), полученное с использованием функции GetFileInformationByHandle. Всегда ли оно равно 1? Различаются ли ответы на этот вопрос для файловых систем NTFS и FAT? Не включают ли значения счетчиков ссылок жесткие ссылки и ссылки из родительских каталогов и подкаталогов, как это имеет место в UNIX? Открывает ли Windows каталог как файл для получения дескриптора, прежде чем использовать эту функцию? Что можно сказать о ярлыках, поддерживаемых пользовательским интерфейсом?

3.5. В программе 3.2 поиск текущего и родительских каталогов осуществляется с использованием имен "." и "..". Что произойдет в случае, если файлы с такими именами действительно существуют? Могут ли файлы иметь такие имена?

3.6. Значения какого времени выводятся в программе 3.2 — местного или UCT? При необходимости измените программу таким образом, чтобы выводимые значения соответствовали местному времени.

3.7. Усовершенствуйте программу 3.2 таким образом, чтобы в выводимый список включались также текущий (".") и родительский ("..") каталоги (завершенная программа находится на Web-сайте). Кроме того, добавьте опции, позволяющие наряду с датой и временем последнего изменения отображать дату и время создания файла, а также дату и время последнего доступа к нему.

3.8. Напишите программу, которая реализует команду rm, позволяющую удалять файлы, изменив для этого функцию ProcessItem в программе 3.2. Решение доступно на Web-сайте.

3.9. Усовершенствуйте команду cp из главы 2, предназначенную для копирования файлов, таким образом, чтобы она позволяла копировать файлы в указанный каталог. Дополнительно предусмотрите опцию рекурсивного копирования файлов (параметр –r) и опцию сохранения вместе с копией также времени последнего изменения файла (параметр –p). Для реализации опции рекурсивного копирования файлов вам потребуется создать новые каталоги.

3.10. Напишите программу mv, которая реализует одноименную команду UNIX, позволяющую переместить целиком любой каталог. При этом имеет существенное значение, осуществляется ли перемещение файла или каталога на другой диск или они остаются на прежнем диске. В случае смены диска используйте операцию копирования файлов, в противном случае используйте команды MoveFile или MoveFileEx.

3.11. Усовершенствуйте программу 3.3 (touch) таким образом, чтобы новое время создания файла можно было указывать в командной строке. Команда UNIX допускает (по выбору) указание метки времени после обычных параметров, но перед именами файлов. Метки даты и времени имеют формат MMddhhmm [yy], где MM — месяцы, dd — дни, hh — часы, mm — минуты, yy — года. Двух цифр для обозначения года нам будет недостаточно, поэтому предусмотрите для указания года четыре разряда.

3.12. Программа 3.1 рассчитана на работу с большими файловыми системами NTFS. Если на вашем жестком диске имеется достаточно много свободного места, протестируйте работу этой программы на файлах гигантских размеров (свыше 4 Гбайт). Проверьте, насколько корректно работает 64-битовая арифметика. Выполнять это упражнение на сетевом диске без предварительного разрешения администратора сети не рекомендуется. Завершив работу над этим упражнением, не забудьте удалить тестовый файл.

3.13. Напишите программу, которая блокирует заданный файл и удерживает его в блокированном состоянии в течение длительного времени (вероятно, вызахотите воспользоваться функцией Sleep). Воспользовавшись любым текстовым редактором, попытайтесь получить доступ к файлу (используйте текстовый файл) в период действия блокировки. Что при этом

происходит? Заблокирован ли файл должным образом? Вы также можете написать программу, предлагающую пользователю задать блокировку для тестового файла. Чтобы проверить, срабатывает ли блокировка описанным образом, запустите на выполнение два экземпляра программы в разных окнах. Решение этого упражнения содержится в файле TestLock.c, находящемся на Web-сайте.

3.14. Исследуйте представление временных характеристик файла Windows в формате данных FILETIME. В этом формате используются 64-битовые счетчики, выражающие в 100-наносекундных единицах длительность истекшего периода времени, отсчитываемого от 1 января 1601 года. Когда исчерпаются показания этого счетчика? Какова максимально допустимая дата для временных характеристик файлов UNIX?

3.15. Напишите интерактивную утилиту, в которой пользователю предлагается ввести имя раздела реестра и имя значения реестра. Отобразите текущее значение и предложите пользователю указать новое.

3.16. В этой главе, как и в большинстве других глав книги, описываются наиболее важные функции. Однако во многих случаях вам могут оказаться полезными и другие функции. На страницах оперативного справочного руководства для каждой функции приведены ссылки на родственные функции. Ознакомьтесь с некоторыми из них, такими, например, как FindFirstFileEx, ReplaceFile, SearchPath или WriteFileGather. Некоторые функции доступны не во всех версиях NT5.