Управление процессами

Процесс (process) представляет собой объект, обладающий собственным независимым виртуальным адресным пространством, в котором могут размещаться код и данные, защищенные от других процессов. В свою очередь, внутри каждого процесса могут независимо выполняться один или несколько *потоков* (threads). Поток, выполняющийся внутри процесса, может сам создавать новые потоки и новые независимые процессы, а также управлять взаимодействием объектов между собой и их синхронизацией.

Создавая процессы и управляя ими, приложения могут организовывать параллельное выполнение нескольких задач, обеспечивающих обработку файлов, проведение вычислений или связь с другими системами в сети. Допускается даже использование нескольких процессоров с целью ускорения обработки данных.

В этой главе объясняются основы управления процессами и вводятся; простейшие операции синхронизации, которые будут использоваться на протяжении оставшейся части книги.

Процессы и потоки Windows

Внутри каждого процесса могут выполняться один или несколько потоков, и именно поток является базовой единицей выполнения в Windows. Выполнение потоков планируется системой на основе обычных факторов: наличие таких ресурсов, как CPU и физическая память, приоритеты, равнодоступность ресурсов и так далее. Начиная с версии NT4, в Windows поддерживается симметричная многопроцессорная обработка (Symmetric Multiprocessing, SMP), позволяющая распределять выполнение потоков между отдельными процессорами, установленными в системе.

С точки зрения программиста каждому процессу принадлежат ресурсы, представленные следующими компонентами:

- Один или несколько потоков.
- Виртуальное адресное пространство, отличное от адресных пространств других процессов, если не считать областей памяти, распределенных явным образом для совместного использования (разделения) несколькими процессами. Заметьте, что разделяемые отображенные файлы совместно используют физическую память, тогда как разделяющие их процессы используют различные виртуальные адресные пространства.
- Один или несколько сегментов кода, включая код DLL.
- Один или несколько сегментов данных, содержащих глобальные переменные.
- Строки, содержащие информацию об окружении, например, информацию о текущем пути доступа к файлам.
- Куча процесса.
- Различного рода ресурсы, например, дескрипторы открытых файлов и другие кучи.

Поток разделяет вместе с процессом код, глобальные переменные, строки окружения и другие ресурсы. Каждый поток планируется независимо от других и располагает следующими элементами:

• Стек, используемый для вызова процедур, прерываний и обработчиков исключений, а также хранения автоматических переменных.

• Локальные области хранения потока (Thread Local Storage, SLT) — массивы указателей, используя которые каждый поток может создавать собственную уникальную информационную среду.

• Аргумент в стеке, получаемый от создающего потока, который обычно является уникальным для каждого потока.

• Структура контекста, поддерживаемая ядром системы и содержащая значения машинных регистров.

На рис. 6.1 показан процесс с несколькими потоками. Рисунок является схематическим, поэтому на нем не указаны фактические адреса памяти и не соблюдены масштабы.

В данной главе показано, как работать с процессами, состоящими из единственного потока.

Примечание

Рисунок 6.1 является высокоуровневым с точки зрения программиста представлением процесса. В действительности эта картина должна быть дополнена множеством технических деталей и особенностями реализации. Более подробную информацию заинтересованные читатели могут найти в книге Соломона (Solomon) и Руссиновича (Russinovich) *Inside Windows 2000*.

Процессы UNIX сопоставимы с процессами Windows, имеющими единственный поток.

Реализации UNIX недавно пополнились потоками POSIX Pthreads, которые в настоящее время используются почти повсеместно. В [40] потоки не обсуждаются; все рассмотрение основано на процессах.

Наверное, можно было бы даже не напоминать о том, что понятие потоков не является новым, и их различные реализации предлагаются поставщиками уже на протяжении целого ряда лет. Однако потоки Pthreads являются самым распространенным стандартом, в то время как коммерческие реализации потоков являются устаревшими.

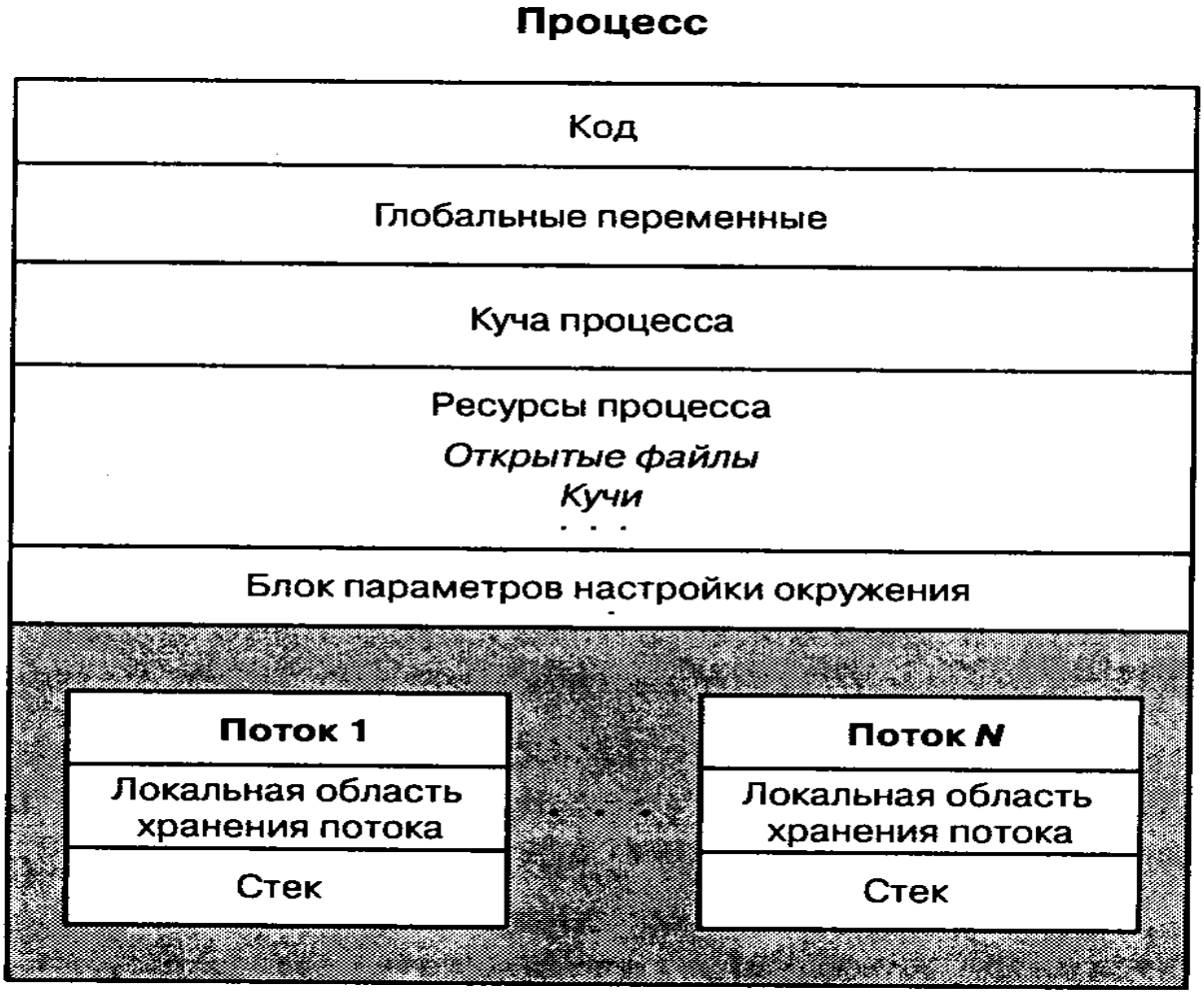

Рис. 6.1. Процесс и его потоки

Создание процесса

Одной из важнейших функций Windows, обеспечивающих управление процессами, является функция CreateProcess, которая создает новый процесс с единственным потоком. При вызове этой функции требуется указать имя файла исполняемой программы.

Обычно принято говорить о *процессах-предках,* или *родительских процессах* (parent processes), и *процессах-потомках,* или *дочерних процессах* (child processes), однако между процессами Windows эти отношения фактически не поддерживаются. Использование данной терминология является просто удобным способом выражения того факта, что один процесс порождается другим.

Гибкие и мощные возможности функции CreateProcess обеспечиваются ее десятью параметрами. На первых порах для упрощения работы целесообразно использовать значения параметров, заданные по умолчанию. Точно так же, как и в случае функции CreateFile, имеет смысл подробно рассмотреть каждый из параметров функции CreateProcess.

Благодаря этому изучить другие аналогичные функции вам будет гораздо легче.

Прежде всего, заметьте, что возвращаемое значение функции не является дескриптором типа HANDLE; вместо этого функция возвращает два отдельных дескриптора, по одному для процесса и потока, передавая их в структуре, которая указывается при вызове функции. Эти дескрипторы относятся к создаваемому функцией CreateProcess новому процессу и его *основного* (primary) потока. Во избежание утечки ресурсов в процессе работы с примерами программ тщательно следите за своевременным закрытием обоих дескрипторов, когда они вам больше не нужны; забывчивость в отношении закрытия дескрипторов потоков является одной из самых распространенных ошибок. Закрытие дескриптора потока не приводит к прекращению ее выполнения; функция CloseHandle лишь удаляет ссылку на поток внутри процесса, вызвавшего функцию CreateProcess.

BOOL CreateProcess (lpApplicationName, LPTSTR lpCommandLine, LPSECURITY_ATTRIBUTES lpsaProcess, LPSECURITY_ATTRIBUTES lpsaThread, BOOL bInheritHandles, DWORD dwCreationFlags, LPVOID lpEnvironment, LPCTSTR lpCurDir, LPSTARTUPINFO lpStartupInfo, LPPROCESS_INFORMATION lpProcInfo)

Возвращаемое значение: в случае успешного создания процесса и потока — TRUE, иначе — FALSE.

Параметры

Некоторые параметры потребуют дальнейшего подробного обсуждения в следующих разделах, тогда как смысл многих других станет для вас более понятным при рассмотрении примеров программ.

lpApplicationName и lpCommandLine (последний указатель имеет тип LPTSTR, а не LPCTSTR) — используются вместе для указания исполняемой программы и аргументов командной строки, о чем говорится в следующем разделе.

lpsaProcess и lpsaThread — указатели на структуры атрибутов защиты процесса и потока. Значениям NULL соответствует использование атрибутов защиты, заданных по умолчанию, и именно эти значения будут использоваться нами вплоть до главы 15, посвященной рассмотрению средств безопасности Windows.

bInheritHandles — показывает, наследует ли новый процесс наследуемые открытые дескрипторы (файлов, отображений файлов и так далее) из вызывающего процесса. Наследуемые дескрипторы имеют те же атрибуты, что и исходные, и их обсуждение будет продолжено в одном из следующих разделов.

dwCreationFlags — может объединять в себе несколько флаговых значений, включая следующие:

• CREATE_SUSPENDED — указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функция ResumeThread.

• DETACHED_PROCESS и CREATE_NEW_CONSOLE — взаимоисключающие значения, которые не должны устанавливаться оба одновременно. Первый флаг означает создание нового процесса, у которого консоль отсутствует, а второй — процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса.

• Create_New_Process_Group — указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (Ctrl-C или Ctrl-break). Обработчики управляющих сигналов консоли описывались в главе 4, а их применение было продемонстрировано в программе 4.5. Упомянутые группы процессов в некотором отношении аналогичны группам процессов UNIX и рассматриваются далее в этой главе.

Некоторые из флагов управляют приоритетами потоков нового процесса. О возможных значениях этих флагов более подробно говорится в главе 7. Пока же нам будет достаточно использовать приоритет родительского процесса (этот режим устанавливается по умолчанию) или указывать значение NORMAL_PRIORITY_CLASS.

lpEnvironment — указывает на блок параметров настройки окружения нового процесса. Если задано значение NULL, то новый процесс будет использовать значения параметров окружения родительского процесса. Блок параметров содержит строки, в которых заданы пары "имя-значение", определяющие, например, пути доступа к файлам.

lpCurDir — указатель на строку, содержащую путь к текущему каталогу нового процесса. Если задано значение NULL, то в качестве текущего каталога будет использоваться рабочий каталог родительского процесса.

lpStartupInfo — указатель на структуру, которая описывает внешний вид основного окна и содержит дескрипторы стандартных устройств нового процесса. Используйте соответствующую информацию из родительского процесса, которую можно получить при помощи функции GetStartupInfo. Можно поступить и по-другому, обнулив структуру STARTUPINFO перед вызовом функции CreateProcess. Для указания стандартных устройств ввода, вывода информации и вывода сообщений об ошибках следует определить значения полей дескрипторов стандартных устройств (hStdInput, hStdOutput и hStdError) в структуре STARTUPINFO. Чтобы эти значения не игнорировались, следует задать для другого элемента этой же структуры, а именно, элемента dwFlags, значение STARTF_USESTDHANDLES и определить все дескрипторы, которые потребуются дочернему процессу. Убедитесь в том, что эти дескрипторы являются наследуемыми и что при вызове функции CreateProcess значение параметра bInheritHandles

установлено равным TRUE. Более подробная информация по этому вопросу, сопровождаемая соответствующим примером, приводится в разделе "Наследуемые дескрипторы".

lpProInfo — указатель на структуру, в которую будут помещены возвращаемые функцией значения дескрипторов и глобальных идентификаторов процесса и потока. Структура PROCESS_INFORMATION, о которой идет речь, имеет следующий вид:

```
typedef struct PROCESS_INFORMATION {
 HANDLE hProcess;
 HANDLE hThread;
 DWORD dwProcessId;
 DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Зачем процессам и потокам нужны еще и дескрипторы, если они снабжаются глобальными идентификаторами (ID)? Глобальные идентификаторы остаются уникальными для данного объекта на протяжении всего времени его существования и во всех процессах, тогда дескрипторов процесса может быть несколько и каждый из которых может характеризоваться собственным набором атрибутов, например определенными разрешениями доступа. В силу указанных причин одним функциям управления процессами требуется предоставлять идентификаторы процессов, а другим — дескрипторы. Кроме того, необходимость в дескрипторах процессов возникает при использовании универсальных функций, которые требуют указания дескрипторов. В качестве примера можно привести функции ожидания, обсуждаемые далее, которые обеспечивают отслеживание переходов объектов различного типа, в том числе и процессов, указываемых с помощью дескрипторов, в определенные состояния. Точно так же, как и дескрипторы файлов, дескрипторы процессов и потоков должны закрываться сразу же после того, как необходимость в них отпала.

Примечание

Новый процесс получает информацию об окружении, рабочем каталоге и иную информацию в результате вызова функции CreateProcess. По завершении этого вызова любые изменения характеристик родительского процесса никак не отразятся на дочернем процессе. Так, после вызова функции CreateProcess рабочий каталог родительского процесса может измениться, но на дочерний процесс это не окажет никакого влияния, если только он сам не сменит рабочий каталог. Оба процесса полностью независимы друг от друга.

Модели процесса в UNIX и Windows значительно отличаются друг от друга. Прежде всего, в Windows отсутствует эквивалент UNIX-функции fork, создающей копию родительского процесса, включая его пространство данных, кучу и стек. В Windows трудно добиться точной эмуляции fork, но как ни расценивать последствия этого ограничения, остается фактом, что

проблемы с использованием функции fork существуют и в многопоточных системах UNIX, поскольку любые попытки создания точной реплики многопоточной системы с копиями всех потоков и объектов синхронизации, особенно в случае SMP-систем, приводят к возникновению множества трудностей. Поэтому в действительности функция fork вообще плохо подходит для многопоточных систем.

В то же время, функция CreateProcess аналогична обычной для UNIX цепочке последовательных вызовов функций fork и execl (или одной из пяти остальных функций exec). В отличие от Windows пути доступа в UNIX определяются исключительно переменной среды PATH.

Как ранее уже отмечалось, отношения "предок-потомок" между процессами в Windows не поддерживаются. Так, выполнение дочернего процесса будет продолжаться даже после того, как завершится родительский процесс. Кроме того, в Windows отсутствуют группы процессов. Существует, однако, ограниченная форма группы процессов, в которой все процессы получают управляющие события консоли.

Процессы Windows идентифицируются как дескрипторами, так и идентификаторами процессов, тогда как в UNIX дескрипторы процессов отсутствуют.

Указание исполняемого модуля и командной строки

Для указания имени файла исполняемого модуля используются как параметр lpApplicationName, так и параметр lpCommandLine. При этом действуют следующие правила:

• Указатель lpApplicationName, если его значение не равно NULL, указывает на строку, содержащую имя файла исполняемого модуля. Если имя модуля содержит пробелы, его следует заключить в кавычки. Более детальное описание приводится ниже.

• Если же значение указателя lpApplicationName равно NULL, то имя модуля определяется первой из лексем, переданных параметром lpCommandLine.

Обычно задается только параметр lpCommandLine, в то время как параметр lpApplicationName полагается равным NULL. Тем не менее, ниже приведены более подробные правила, которые определяют порядок использования этих двух параметров.

• Параметр lpApplicationName, если он не равен NULL, определяет исполняемый модуль. В строке, на которую указывает этот указатель, задайте полный путь доступа и имя файла или же ограничьтесь только именем файла, и тогда будут использоваться текущие диск и каталог; дополнительный поиск при этом производиться не будет. В имя файла включите расширение, например, .EXE или .BAT.

• Если значение параметра lpApplicationName равно NULL, то именем исполняемого модуля является первая из разделенных пробельными символами лексем, переданных параметром lpCommandLine. Если полный

путь доступа не указан, то поиск файла осуществляется в следующем порядке:

1. Каталог модуля текущего процесса.
2. Текущий каталог.
3. Системный каталог Windows, информацию о котором можно получить с помощью функции GetSystemDirectory.
4. Каталог Windows, возвращаемый функцией GetWindowsDirectory.
5. Каталоги, перечисленные в переменной окружения PATH.

Новый процесс может получить командную строку посредством обычного argv-механизма или путем вызова функции GetCommandLine для получения командной строки в виде одиночной строки символов.

Заметьте, что командная строка не является строковой константой. Это согласуется с тем, что параметры argv главной программы не являются константами. Программа может модифицировать свои аргументы, хотя для внесения любых изменений рекомендуется использовать копию строки аргументов.

Вовсе не обязательно, чтобы новый процесс создавался с тем же определением UNICODE, что и родительский процесс. Возможны любые комбинации. Использование _tmain, как обсуждалось в главе 2, облегчает разработку программного кода, который сможет работать как с символами Unicode, так и с символами ASCII.

Наследуемые дескрипторы

Часто бывает так, что дочернему процессу требуется доступ к объекту, к которому можно обратиться через дескриптор, определенный в родительском процессе, и если этот дескриптор — наследуемый, то дочерний процесс может получить копию открытого дескриптора родительского процесса. Часто именно так обеспечивается возможность использования дескрипторов стандартного ввода и вывода дочерним процессом. Преобразование дескриптора в наследуемый, чтобы дочерний процесс мог получить и использовать его копию, требует выполнения нескольких шагов.

Флаг bInheritHandles, который можно указать при вызове функции CreateProcess, определяет, будет ли дочерний процесс наследовать копии наследуемых дескрипторов открытых файлов, процессов и так далее. Этот флаг можно рассматривать как главный переключатель, действующий в отношении всех дескрипторов.

Кроме того, чтобы сделать наследуемым любой отдельный дескриптор, также требуется предпринимать специальные действия, поскольку дескрипторы не становятся таковыми по умолчанию. Создать наследуемый дескриптор можно либо путем использования структуры SECURITY_ATTRIBUTES в момент создания дескриптора, либо путем копирования существующего дескриптора.

В структуре SECURITY_ATTRIBUTES присутствует флаг bInheritHandle, значение которого должно быть установлено равным TRUE.

Не забывайте также о том, что элемент nLength должен инициализироваться следующим значением:

sizeof(SECURITY_ATTRIBUTES)

Приведенный ниже фрагмент кода иллюстрирует создание наследуемых файловых или иных дескрипторов в типичных случаях. В этом примере дескриптор защиты в структуре атрибутов защиты установлен в NULL; подробнее об использовании дескрипторов защиты говорится в главе 15.

```
HANDLE h1, h2, h3;
SECURITY_ATTRIBUTES sa = { sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
…
h1 = CreateFile(…, &sa, …); /* Наследуемый. */
h2 = CreateFile(…, NULL, …); /* Ненаследуемый. */
h3 = CreateFile(…, &sa, …); /* Наследуемый. Возможно повторное использование структуры sa. */
```

Однако дочернему процессу значение наследуемого дескриптора пока еще не известно, и поэтому родительский процесс должен передать это значение дочернему процессу либо через механизм межпроцессного взаимодействия (Interprocess Communication, IPC), либо путем назначения дескриптора стандартному устройству ввода/вывода в структуре STARTUPINFO, как это делается в первом из примеров, приведенных в данной главе (программа 6.1), а также в ряде примеров в остальной части книги. Обычно последний метод является более предпочтительным, так как он позволяет перенаправить ввод/вывод стандартным способом без внесения каких-либо изменений в дочернюю программу.

В случае дескрипторов, которые не являются дескрипторами файлов или не используются для перенаправления ввода/вывода, применим другой способ, в соответствии с которым дескриптор преобразуется в текстовый формат и помещается в командную строку или переменную окружения. Такой подход можно использовать лишь в том случае, если дескриптор является наследуемым, поскольку и родительский, и дочерний процессы используют для идентификации дескриптора одно и то же значение. Один из способов реализации этого подхода предлагается в упражнении 6.2, а соответствующее решение приводится на Web-сайте книги.

Унаследованные дескрипторы представляют собой отдельные экземпляры. Поэтому родительский и дочерний процессы могут получить доступ к одному и тому же файлу, используя различные указатели файлов. Более того, каждый из обоих процессов может и должен самостоятельно закрывать принадлежащий ему дескриптор.

На рис. 6.2 показан пример двух процессов с двумя различными таблицами дескрипторов, в которых с одним и тем же файлом или иным объектом связаны два различных дескриптора. Процесс 1 является родительским, процесс 2 — дочерним. Если принадлежащий дочернему процессу дескриптор был унаследован им, как это имеет место в случае

дескрипторов 1 и 3, то значения дескрипторов в обоих процессах будут одинаковыми.

Однако подобные дескрипторы могут иметь и различные значения. Так, на файл D указывают два дескриптора, причем процесс 2 получил дескриптор за счет вызова функции CreateFile, а не путем наследования. Наконец, возможны ситуации, когда один из процессов имеет дескриптор объекта, а второй — не имеет, что наблюдается для файлов B и E. Так происходит в тех случаях, когда дескриптор создается дочерним процессом или дублируется из одного процесса в другой, о чем говорится в разделе "Дублирование дескрипторов".

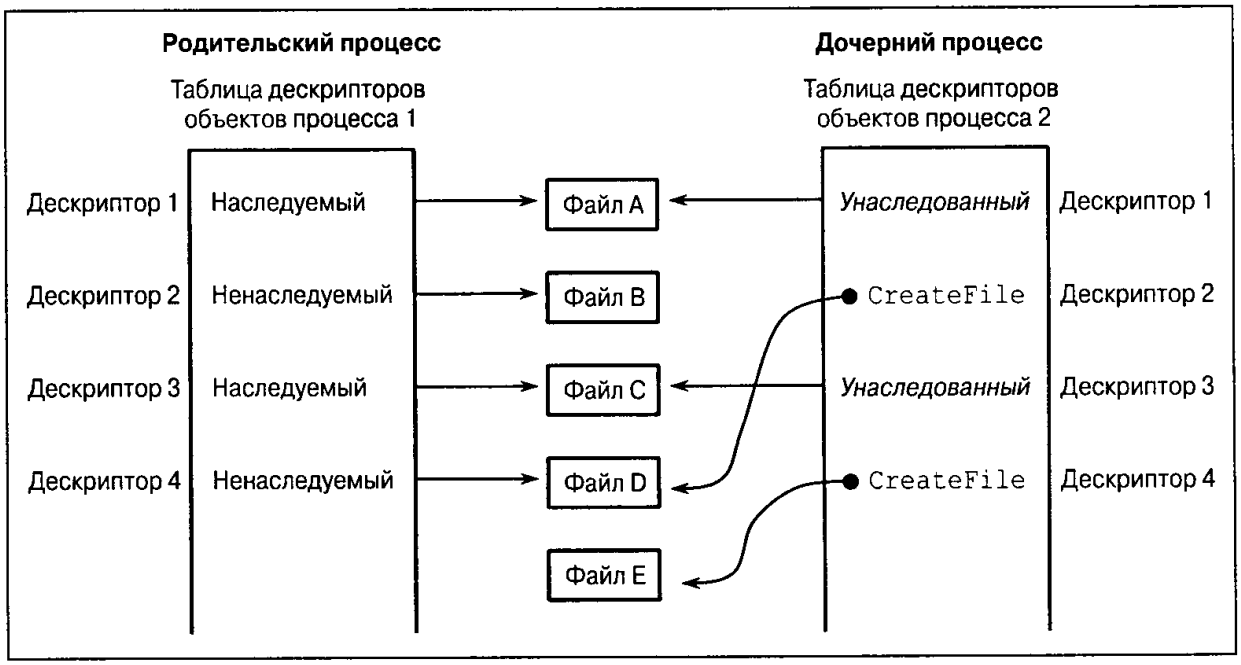

Рис. 6.2. Таблицы дескрипторов объектов для двух процессов

Счетчики дескрипторов процессов

Распространенной ошибкой программистов является пренебрежение закрытием дескрипторов после того, как необходимость в них отпала; это может стать причиной утечки ресурсов, что, в свою очередь, может приводить к снижению производительности или сбоям в программе и даже влиять на другие процессы. В версии NT 5.1 добавлена новая функция, позволяющая определить количество открытых дескрипторов, принадлежащих указанному процессу. Таким способом вы можете контролировать как собственный, так и другие процессы.

Приведенное ниже определение упомянутой функции не нуждается в отдельных пояснениях:

BOOL GetProcessHandleCount (HANDLE hProcess, PDWORD pdwHandleCount)

Идентификаторы процессов

Процесс может получить идентификатор и дескриптор нового дочернего процесса из структуры PROCESS_INFORMATION. Разумеется, закрытие дескриптора дочернего процесса не приводит к уничтожению

самого процесса; становится невозможным лишь доступ к нему со стороны родительского процесса. Для получения идентификационной информации о текущем процессе служат две функции.

HANDLE GetCurrentProcess(VOID)

DWORD GetCurrentProcessId(VOID)

В действительности функция GetCurrentProcess возвращает *псевдодескриптор* (pseudohandle), который не является наследуемым. Это значение может использоваться вызывающим процессом всякий раз, когда ему требуется его собственный дескриптор. Реальный дескриптор процесса создается на основе идентификатора (ID) процесса, включая и тот, который возвращается функцией GetCurrentProcessID, путем использования функции OpenProcess. Как и в случае любого разделяемого объекта, при отсутствии надлежащих разрешений доступа попытка открытия объекта процесса окажется неуспешной.

HANDLE OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwProcessId)

Возвращаемое значение: в случае успешного завершения — дескриптор процесса, иначе — NULL.

Параметры

dwDesiredAccess — определяет права доступа к процессу. Некоторые из возможных значений этого параметра перечислены ниже.

• SYNCHRONIZE — разрешается использование дескриптора процесса в функциях ожидания завершения процесса, которые описываются далее.

• PROCESS_ALL_ACCESS — устанавливаются все флаги доступа к процессу.

• PROCESS_TERMINATE — делает возможным завершение процесса с использованием функции TerminateProcess.

• PROCESS_QUERY_INFORMATION — разрешает использование дескриптора процесса в функциях GetExitCodeProcess и GetPriorityClass для получения информации о процессе.

bInheritHandle — позволяет указать, является ли новый дескриптор наследуемым. Параметр dwProcessID является идентификатором процесса, запрашивающего дескриптор.

Наконец, выполняющийся процесс может определить полный путь доступа к файлу исполняемого модуля, который использовался для его запуска, с помощью функций GetModuleFileName или GetModuleFileNameEx, при вызове которых значение параметра hModule должно устанавливаться равным NULL. При вызове этой функции из DLL будет возвращено имя файла DLL, а не .EXE-файла, который использует эту библиотеку DLL.

Дублирование дескрипторов

Родительскому и дочернему процессам может требоваться различный доступ к объекту, идентифицируемому дескриптором, который наследует дочерний процесс. Кроме того, процессу вместо псевдодескриптора, получаемого с помощью функции GetModuleFileName или GetModuleFileNameEx, может потребоваться реальный, наследуемый дескриптор, который мог бы использоваться дочерним процессом. Родительский процесс может обеспечить это, создав копию дескриптора с желаемыми разрешениями доступа и свойствами наследования. Функция, позволяющая создавать копии дескрипторов, имеет следующий вид:

BOOL DuplicateHandle(HANDLE hSourceProcessHandle, HANDLE hSourceHandle, HANDLE hTargetProcessHandle, LPHANDLE lphTargetHandle, DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwOptions)

По завершении выполнения функции указатель lphTargetHandle будет указывать на копию исходного дескриптора, hSourceHandle. hSourceHandle является дескриптором дублируемого объекта в процессе, указанном дескриптором hSourceProcessHandle, и должен иметь права доступа PROCESS_DUP_HANDLE; если указанного дескриптора в исходном процессе не существует, функция DuplicateHandle завершается ошибкой. Новый дескриптор, на который указывает указатель lphTargetHandle, является действительным в целевом процессе, hTargetProcessHandle. Обратите внимание на то, что в нашем рассмотрении фигурировали три процесса, включая вызывающий. Часто в роли вызывающего процесса выступает целевой или исходный процесс, и тогда соответствующий дескриптор получают с помощью функции GetCurrentProcess. Заметьте также, что процесс может создать дескриптор в другом процессе; если вы это делаете, то вам потребуется механизм, с помощью которого можно было бы передать в другой процесс идентификационные данные нового дескриптора.

Функция DuplicateHandle может применяться к дескрипторам любого типа.

Если действие параметра dwDesiredAccess не отменяется флагом DUPLICATE_SAME_ACCESS параметра dwOptions, то у него может быть много возможных значений (для получения более подробных сведений обратитесь к библиотеке MSDN оперативного справочного руководства).

Параметр dwOptions может содержать любую комбинацию указанных ниже двух флагов.

• DUPLICATE_CLOSE_SOURCE — вызывает закрытие исходного дескриптора.

• DUPLICATE_SAME_ACCESS — вынуждает игнорировать параметр dwDesiredAccess.

Напоминание

Ядро Windows поддерживает счетчики ссылок для всех объектов; этот счетчик представляет количество различных дескрипторов, ссылающихся на данный объект. В то же время, приложения не имеют доступа к этому счетчику. Любой объект не может быть уничтожен до тех

пор, пока не будет закрыт его последний дескриптор, а счетчик ссылок не примет нулевое значение. Унаследованные и продублированные дескрипторы считаются отличными от исходных и также учитываются в счетчике ссылок. Наследуемые дескрипторы используются в программе 6.1 далее в этой главе. В то же время, дескрипторы, переданные из одного процесса в другой посредством той или иной формы механизма IPC, не считаются независимыми, и поэтому если один процесс закрывает такой дескриптор, то другие процессы использовать его не могут. Подобной методикой пользуются редко, однако в упражнении 6.2 вам предлагается передать значение унаследованного дескриптора из одного процесса в другой, используя механизм IPC.

Далее вы узнаете о том, как определить, завершено ли выполнение процесса.

Завершение и прекращение выполнения процесса

После того как процесс завершил свою работу, он, или, точнее, выполняющийся в этом процессе поток, может вызвать функцию ExitProcess, указав в качестве параметра кодом завершения (exit code):

VOID ExitProcess(UINT uExitCode)

Эта функция не осуществляет возврата. Вместо этого она завершает вызывающий процесс и все его потоки. Обработчики завершения игнорируются, но делаются все необходимые вызовы точек входа DllMain (см. главу 5) с кодом отключения от библиотеки. Код завершения связывается с процессом. 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 (см. описание функции OpenProcess, которая нами уже обсуждалась). lpExitCode указывает на переменную типа DWORD, которая принимает значение кода завершения. Одним из ее возможных значений является STILL_ACTIVE, означающее, что данный процесс еще не завершился.

Наконец, один процесс может прекратить выполнение другого процесса, если у дескриптора завершаемого процесса имеются права доступа PROCESS_TERMINATE. При вызове функции завершения процесса указывается код завершения.

BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode)

Предостережение

Прежде чем завершить выполнение процесса, убедитесь в том, что все ресурсы, которые он мог разделять с другими процессами, освобождены. В частности, должны быть корректно обработаны ресурсы синхронизации, о

которых говорится в главе 8 (мьютексы, семафоры и события). В этом отношении могут оказаться полезными SEH (глава 4), а вызов функции ExitProcess может быть осуществлен из обработчика. В то же время, при вызове функции ExitProcess обработчики __finally и __except *не* выполняются, поэтому в идее завершения выполнения изнутри программы нет ничего хорошего. Особенно рискованно применять функцию TerminateProcess, поскольку у завершаемого процесса в этом случае отсутствует возможность выполнить свои SEH или вызвать функции DllMain связанных с ним библиотек DLL. Ограниченной альтернативой являются обработчики управляющих сигналов консоли, обеспечивающие возможность передачи сигнала одним процессом другому, который после этого может корректно организовать свое завершение.

Программа 6.3 иллюстрирует применение методики, обеспечивающей взаимодействие между процессами. В этом примере один процесс посылает другому процессу запрос завершения выполнения, получив который второй процесс сможет аккуратно завершить свою работу.

Процессы UNIX имеют свои идентификаторы, pid, которые сопоставимы с идентификаторами процессов Windows. Функция getpid аналогична функции GetCurrentProcessID, но эквивалентов функциям getppid и getgpid в Windows не находится ввиду отсутствия предков процессов и групп процессов.

И, наоборот, в UNIX отсутствуют дескрипторы процессов, и поэтому в ней нет функций, которые можно было бы сравнить с функциями GetCurrentProcess или OpenProcess.

В UNIX допускается использование дескрипторов (descriptors) открытых файлов после вызова функции exec, если для дескриптора файла не был установлен флаг close-on-exec. Это правило применимо только к дескрипторам файлов, которые, в силу вышесказанного, можно сравнить с наследуемыми дескрипторами (handles) файлов Windows.

Функция UNIX exit, которая фактически является функцией библиотеки C, аналогична функции ExitProcess; чтобы прекратить выполнение другого процесса ему следует послать сигнал SIGKILL.

Ожидание завершения процесса

Простейшим, но наряду с этим и обладающим наиболее ограниченными возможностями, методом синхронизации с другим процессом является ожидание его завершения. Представленные ниже стандартные функции ожидания Windows обладают рядом интересных свойств.

• Функции ожидания могут работать с самыми различными типами объектов; дескрипторы процессов являются лишь самым первым из рассматриваемых нами примеров применения этих функций.

• Эти функции могут ожидать завершения одного процесса, первого из нескольких указанных процессов или всех процессов, образующих группу.

• Существует возможность устанавливать конечный интервал ожидания (time-out).

Обе рассмотренных ниже функции ожидают перехода объекта синхронизации в *сигнальное* состояние. Например, система переводит процесс в сигнальное состояние, когда он завершается или его выполнение прекращается извне. Функциями ожидания, которые мы будем впоследствии неоднократно использовать, являются следующие функции:

DWORD WaitForSingleObject(HANDLE hObject, DWORD dwMilliseconds)

DWORD WaitForMultipleObjects(DWORD nCount, CONST HANDLE *lpHandles, BOOL fWaitAll, DWORD dwMilliseconds)

Возвращаемое значение: указывает причину завершения ожидания или, в случае ошибки, равно 0xFFFFFFFF (для получения более подробной информации используйте функцию GetLastError).

В аргументах этих функций указывается либо дескриптор одиночного процесса (hObject), либо дескрипторы ряда отдельных объектов, хранящиеся в массиве, на который указывает указатель lpHandles. Значение параметра nCount, определяющего размер массива, не должно превышать значение MAXIMUM_WAIT_OBJECTS (определено равным 64 в файле WINNT.H).

dwMilliseconds — число миллисекунд интервала ожидания. Если значение этого параметра равно 0, то возврат из функции осуществляется сразу же после проверки состояния указанного объекта, что позволяет программе опрашивать процессы для определения их состояния завершения. Если же значение этого параметра равно INFINITE, то ожидание длится до тех пор, пока ожидаемый процесс не завершится.

fWaitAll — параметр второй функции, указывающий (если его значение равно TRUE) на необходимость ожидания завершения всех процессов, а не только одного.

Возможными возвращаемыми значениями этой функции в случае ее успешного завершения являются следующие:

• WAIT_OBJECT_0 — означает, что указанный объект перешел в сигнальное состояние (в случае функции WaitForSingleObject) или что одновременно все nCount объектов перешли в сигнальное состояние (в специальном случае функции WaitForMultipleObject, когда значение параметра fWaitAll равно TRUE).

• WAIT_OBJECT_0+n, где $0 \leq n < nCount$ — вычтите значение WAIT_OBJECT_0 из возвращенного значения, чтобы определить, выполнение какого именно процесса завершилось, когда ожидается завершение выполнения любого из группы процессов. Если в сигнальное состояние перешли несколько объектов, то возвращается наименьшее из возможных значений. WAIT_ABANDONED является возможным базовым значением в случае использования дескрипторов мьютексов; см. главу 8.

• WAIT_TIMEOUT — указывает на то, что в течение отведенного периода ожидания сигнализируемый объект (объекты) не смогли удовлетворить условию ожидания.

• WAIT_FAILED — означает неудачное завершение функции, вызванное, например, тем, что у дескриптора отсутствовали права доступа SYNCHRONIZE.

• WAIT_ABANDONED_0 — это значение невозможно в случае процессов и рассматривается в главе 8 при рассмотрении мьютексов.

Для определения кода завершения процесса используется функция GetExitCodeProcess, описанная в предыдущем разделе.

Блоки и строки окружения

Схема, представленная на рис. 6.1, включает блок окружения процесса. Блок окружения (environment block) процесса содержит последовательность строк вида:

Имя = Значение

Каждая строка окружения (environment string), будучи символьной строкой, заканчивается нулевым символом, а весь блок строк в целом завершается дополнительным нулевым символом. Одним из примеров широко используемых переменных среды является переменная PATH.

Чтобы передать информацию об окружении из родительского процесса в дочерний, параметр lpEnvironment при вызове функции CreateProcess следует установить равным NULL. В свою очередь, любой процесс может запросить или изменить свои переменные окружения или добавить новые в блок окружения.

Для получения, а также создания новых или изменения существующих переменных окружения используются следующие функции:

DWORD GetEnvironmentVariable(LPCTSTR lpName, LPTSTR lpValue, DWORD cchValue)

BOOL SetEnvironmentVariable(LPCTSTR lpName, LPCTSTR lpValue)

lpName — указатель на строку, содержащую имя переменной окружения. После определения переменной окружения она добавляется в блок окружения при условии, что такая переменная ранее не существовала, а определяемое значение не равно NULL. Если же определяемое значение равно NULL, то переменная удаляется из блока. Строка значения не может содержать символы "=".

В случае успешного завершения функция GetEnvironmentVariable возвращает длину строки значения переменной окружения, иначе — 0. Если размер буфера lpValue, указанный значением параметра cchValue, оказался недостаточно большим, то возвращаемое значение равно количеству символов, которое фактически требуется для сохранения значения переменной. Вспомните, что аналогичный механизм используется и в функции GetCurrentDirectory (глава 2).

Защита процесса

Обычно функция CreateProcess предоставляет права доступа к процессу на уровне PROCESS_ALL_ACCESS. Однако имеется возможность определения детализированных прав доступа, из которых в качестве примера

можно назвать права доступа PROCESS_QUERY_INFORMATION, CREATE_PROCESS, PROCESS_TERMINATE, PROCESS_SET_INFORMATION, DUPLICATE_HANDLE и CREATETHREAD. В частности, с учетом возможных рисков, которые могут подстерегать вас в случае принудительного завершения выполняющихся процессов, на что мы уже неоднократно обращали ваше внимание, может оказаться полезным ограничить предоставление прав доступа к процессам на уровне PROCESS_TERMINATE для родительского процесса. Подробнее об атрибутах защиты процессов и других объектов говорится в главе 15.

В UNIX для ожидания завершения процессов используются функции wait и waitpid, однако отсутствует понятие интервала ожидания, хотя функция waitpid может опрашивать процессы (существует возможность ее вызова без блокировки). Эти функции способны ожидать лишь завершения дочерних процессов, и эквивалентных им функций, применимых к ряду процессов, не существует, хотя и возможно ожидание завершения всех процессов, относящихся к одной группе. Кроме того, имеется одно незначительное отличие, заключающееся в том, что функции wait и waitpid возвращают код завершения сами, в результате чего отпадает необходимость в вызове отдельной функции, эквивалентной функции GetExitCodeProcess.

Строки окружения, аналогичные строкам окружения Windows, поддерживаются и в UNIX. Функция getenv (входящая в библиотеку C) имеет те же самые функциональные возможности, что и функция GetEnvironmentVariable, но программист сам должен заботиться о необходимом размере буфера. Функции putenv, setenv и unsetenv обеспечивают различные способы добавления, изменения и удаления переменных окружения и их значений, предлагая функциональность, аналогичную функциональности SetEnvironmentVariable.

Пример: параллельный поиск указанного текстового шаблона

Настало время посмотреть на процессы Windows в действии. Приведенная ниже в качестве примера программа grepMP создает процессы для поиска указанного текстового шаблона в файлах, по одному процессу на каждый файл. Эта программа моделирует UNIX-утилиту grep, хотя используемая нами методика применима к любой программе, которая полагается на стандартный вывод. Рассматривайте программу поиска как "черный ящик" и считайте, что она является просто исполняемой программой, выполнение которой должно контролироваться родительским процессом.

Командная строка программы имеет следующий вид:

grepMP *шаблон* F1 F2 … FN

Программа 6.1 выполняет следующие виды обработки:

• Для поиска указанного шаблона в каждом из входных файлов, от F1 до FN, используется отдельный процесс, запускающий один и тот же исполняемый модуль. Для каждого процесса программа создает командную строку такого вида: grep *шаблон* FK.

• Полю hStdOut структуры STARTUPINFO нового процесса присваивается значение дескриптора временного файла, который определяется как наследуемый.

• Программа организует ожидание завершения всех процессов поиска, используя для этого функцию WaitForMultipleObjects.

• По завершении всех процессов поиска осуществляется поочередный вывод результатов (временных файлов). Вывод временного файла осуществляет процесс, выполняющий утилиту cat (программа 2.3).

• Возможности функции WaitForMultipleObjects ограничиваются лишь максимально допустимым количеством дескрипторов, которое устанавливается значением MAXIMUM_WAIT_OBJECTS (64), поэтому она вызывается многократно.

• Для определения успешности попытки нахождения данным процессом заданного шаблона программа использует код завершения процесса grep.

Порядок обработки файлов программой 6.1 иллюстрируется на рис. 6.3.

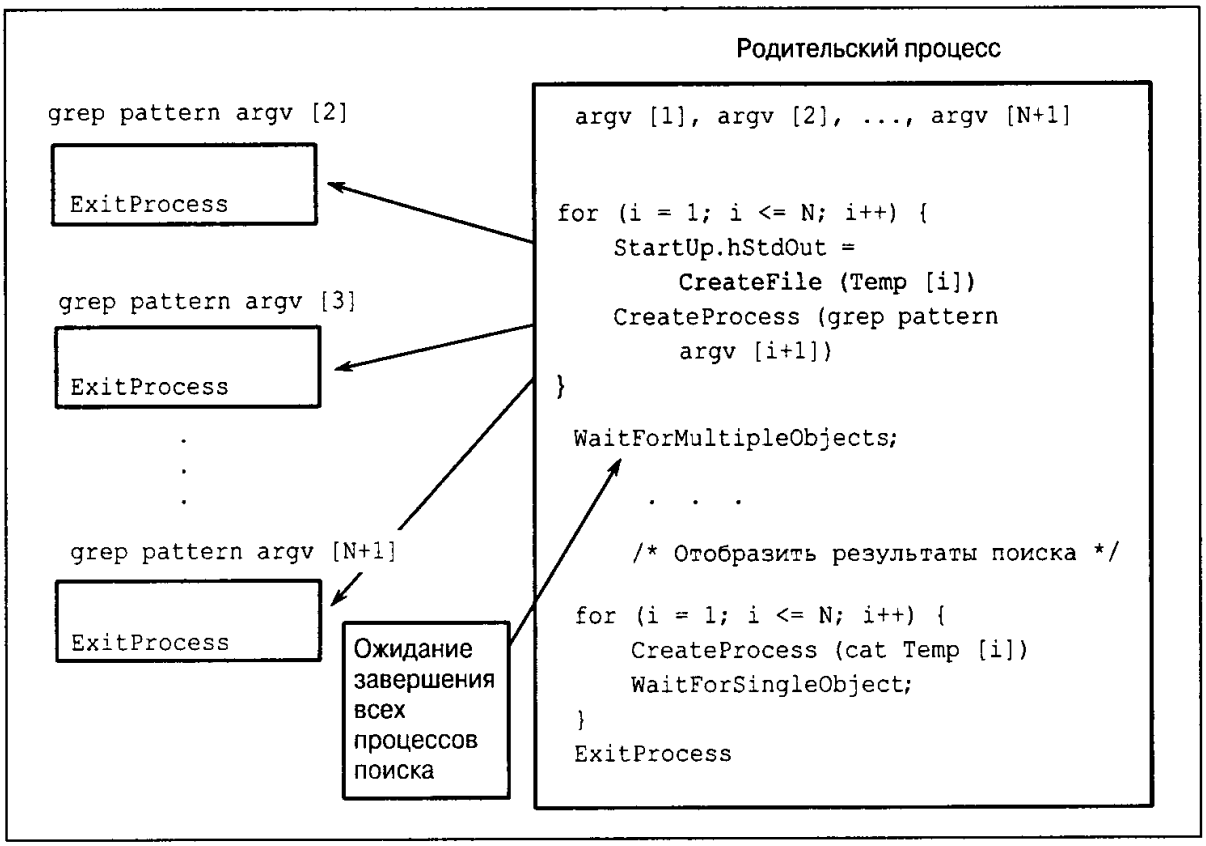

Рис. 6.3. Поиск текстового шаблона в файлах с использованием нескольких процессов

Программа 6.1. grepMP: выполнение параллельного поиска текстового шаблона

```
/* Глава 6. grepMP. */
/* Версия команды grep, использующая несколько процессов. */
#include "EvryThng.h"

int _tmain(DWORD argc, LPTSTR argv[])
/* Для выполнения поиска в каждом из файлов, указанных в командной
строке, создается отдельный процесс. Каждому процессу предоставляется
временный файл в текущем каталоге, в котором сохраняются результаты. */
{
 HANDLE hTempFile;
 SECURITY_ATTRIBUTES StdOutSA = /* Атрибуты защиты для
наследуемого дескриптора. */
  {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
 TCHAR CommandLine[MAX_PATH + 100];
 STARTUPINFO StartUpSearch, Startup;
 PROCESS_INFORMATION ProcessInfo;
 DWORD iProc, ExCode;
 HANDLE *hProc; /* Указатель на массив дескрипторов процессов. */
 typedef struct {TCHAR TempFile[MAX_PATH];} PROCFILE;
 PROCFILE *ProcFile; /* Указатель на массив имен временных файлов. */
 GetStartupInfo(&StartUpSearch);
 GetStartupInfo(&StartUp);
 ProcFile = malloc((argc – 2) * sizeof(PROCFILE));
 hProc = malloc((argc – 2) * sizeof(HANDLE));
 /* Создать для каждого файла отдельный процесс "grep". */
 for (iProc = 0; iProc < argc – 2; iProc++) {
  _stprintf(CommandLine, _T("%s%s %s"), _T("grep "), argv[1], argv[iProc +
2]);
  GetTempFileName(_T("."), _T("gtm"), 0, ProcFile[iProc].TempFile); /* Для
хранения результатов поиска.*/
  hTempFile = /* Этот дескриптор является наследуемым */
   CreateFile(ProcFile[iProc].TempFile, GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, &StdOutSA,
CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
  StartUpSearch.dwFlags = STARTF_USESTDHANDLES;
  StartUpSearch.hStdOutput = hTempFile;
  StartUpSearch.hStdError = hTempFile;
  StartUpSearch.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
  /* Создать процесс для выполнения командной строки. */
  CreateProcess(NULL, CommandLine, NULL, NULL, TRUE, 0, NULL,
NULL, &StartUpSearch, &ProcessInfo);
  /* Закрыть ненужные дескрипторы. */
```

```
    CloseHandle(hTempFile);
    CloseHandle(ProcessInfo.hThread);
    hProc[iProc] = ProcessInfo.hProcess;
   }
  /* Выполнить все процессы и дождаться завершения каждого из них. */
  for (iProc = 0; iProc < argc – 2; iProc += MAXIMUM_WAIT_OBJECTS)
WaitForMultipleObjects( /* Разрешить использование достаточно большого
количества процессов */
    min(MAXIMUM_WAIT_OBJECTS, argc – 2 – iProc), &hProc [iProc], TRUE,
INFINITE);
  /* Переслать результирующие файлы на стандартный вывод с
использованием утилиты cat */
  for (iProc = 0; iProc < argc – 2; iProc++) {
   if (GetExitCodeProcess(hProc[iProc], &ExCode) && ExCode==0) {
    /* Обнаружен шаблон — Вывести результаты. */
    if (argc > 3) _tprintf(_T("%s:\n"), argv [iProc + 2]);
    fflush(stdout); /* Использование стандартного вывода несколькими
процессами. */
    _stprintf(CommandLine, _T("%s%s"), _T("cat "), ProcFile[iProc].TempFile);
    CreateProcess(NULL, CommandLine, NULL, NULL, TRUE, 0, NULL,
NULL, &StartUp, &ProcessInfo);
    WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
    CloseHandle(ProcessInfo.hProcess);
    CloseHandle(ProcessInfo.hThread);
   }
   CloseHandle(hProc [iProc]);
   DeleteFile(ProcFile[iProc].TempFile);
  }
  free(ProcFile);
  free(hProc);
  return 0;
 }
```

Процессы в многопроцессорной среде

В программе 6.1 процессы и их основные (и только эти) потоки выполняются практически полностью независимо друг от друга. Единственная зависимость между ними проявляется лишь в конце выполнения родительского процесса, поскольку он ожидает завершения выполнения каждого из них, чтобы перейти к последовательной обработке выходных файлов. Поэтому в SMP-системах планировщик Windows может и будет обеспечивать параллельное выполнение потоков процесса на нескольких независимых процессорах. В результате этого производительность, если оценивать ее по времени выполнения всей

программы, значительно повышается, причем для этого с вашей стороны не требуется предпринимать никаких действий.

Типичные результаты тестирования производительности приведены в приложении B. Ввиду выполнения программой ряда вспомогательных операций, а также необходимости последовательного вывода результатов, зависимость производительности от количества процессоров не является линейной. Тем не менее, улучшение производительности налицо, и это автоматически обеспечивается организацией программы, которая предусматривает передачу выполнения независимых вычислительных задач независимым процессам.

Вместе с тем, существует возможность привязки процессов к определенным процессорам, что позволяет всегда быть уверенным в том, что другие процессоры остаются свободными и их можно использовать для решения каких-либо иных, критических задач. Это достигается за счет применения маски родства процессора (processor affinity mask) (см. главу 9) в объекте задачи. Объекты задач (job objects) описываются в одном из следующих разделов настоящей главы.

Наконец, внутри процесса можно создавать независимые потоки, и для этих потоков также будет спланировано выполнение с использованием отдельных процессоров SMP для каждого из них. Связь между использованием потоков и показателями производительности обсуждается в главе 7.

Временные характеристики процесса

Воспользовавшись функцией GetProcessTimes, можно получить различные временные характеристики процесса, а именно: истекшее время (elapsed time), время, затраченное ядром (kernel time), и пользовательское время (user time).

BOOL GetProcessTimes(HANDLE hProcess, LPFILETIME lpCreationTime, LPFILETIME lpExitTime, LPFILETIME lpKernelTime, LPFILETIME lpUserTime)

Дескриптор процесса может ссылаться как на процесс, который продолжает выполняться, так и на процесс, выполнение которого прекратилось. Вычитая время создания процесса (creation time) из времени завершения процесса (exit time), мы получаем истекшее время, как показано в следующем примере. Тип данных FILETIME является 64-битовым; для вычисления указанной разности объедините переменную этого типа с переменной тип LARGE_INTEGER в структуру типа union. Ранее преобразование и отображение отметок времени файлов было продемонстрировано в главе 3 на примере программы lsw.

Функция GetThreadTimes аналогична только что описанной, но требует указания дескриптора потока в качестве параметра. Управлению потоками посвящена глава 7.

Пример: временные характеристики процессов

Наш следующий пример (программа 6.2) представляет собой команду timep (от *time print* — вывод временных параметров), аналогичную UNIX-команде time (поскольку команда time поддерживается процессором командной строки, мы должны использовать для нашей команды другое имя). Программа позволяет вывести все три временные характеристики, однако в Windows 9x будет доступно лишь истекшее время процесса.

Одним из возможных применений этой команды является сравнительный анализ времени выполнения и эффективности различных версий функций копирования и преобразования файлов из ASCII в Unicode, реализованных в предыдущих главах.

В данной программе используется функция Windows GetCommandLine, которая возвращает целую командную строку, а не отдельные строки из массива argv.

Кроме того, программа использует вспомогательную функцию SkipArg, которая просматривает командную строку и устанавливает в ней указатель в позицию, непосредственно следующую за именем исполняемого файла. Листинг функции SkipArg приведен в приложении А.

Для определения версии ОС в программе 6.2 используется функция GetVer-sionEx. В операционных системах Windows 9x и Windows CE доступным будет лишь истекшее время процесса. Программный код для этих систем представлен с той целью, чтобы показать, что в некоторых случаях работоспособность программ, по крайней мере — с частичным сохранением их функциональности, удается обеспечивать для целого диапазона различных версий Windows.

Программа 6.2. timep: временные характеристики процессов

```
/* Глава 6. timep. */
#include "EvryThng.h"

int _tmain(int argc, LPTSTR argv[]) {
 STARTUPINFO Startup;
 PROCESS_INFORMATION ProcInfo;
 union { /* Эта структура используется для выполнения арифметических операций с участием временных параметров. */
  LONGLONG li;
  FILETIME ft;
 } CreateTime, ExitTime, ElapsedTime;
 FILETIME KernelTime, UserTime;
 SYSTEMTIME ElTiSys, KeTiSys, UsTiSys, StartTimeSys, ExitTimeSys;
 LPTSTR targv = SkipArg(GetCommandLine());
 OSVERSIONINFO OSVer;
 BOOL IsNT;
 HANDLE hProc;
```

```
OSVer.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
GetVersionEx(&OSVer);
IsNT = (OSVer.dwPlatformId == VER_PLATFORM_WIN32_NT);
/* NT (все версии) возвращает VER_PLATFORM_WIN32_NT. */
GetStartupInfo(&StartUp);
GetSystemTime(&StartTimeSys);
/* Выполнить командную строку; дождаться завершения процесса. */
CreateProcess (NULL, targv, NULL, NULL, TRUE, NORMAL_PRIORITY_CLASS, NULL, NULL, &StartUp, &ProcInfo);
/* Убедиться в наличии ВСЕХ НЕОБХОДИМЫХ прав доступа к процессу. */
DuplicateHandle(GetCurrentProcess(), ProcInfo.hProcess, GetCurrentProcess(), &hProc, PROCESS_QUERY_INFORMATION | SYNCHRONIZE, FALSE, 0);
WaitForSingleObject(hProc, INFINITE);
GetSystemTime (&ExitTimeSys);
if (IsNT) { /* Windows NT. Для процесса вычисляется истекшее время, время выполнения в режиме ядра и время выполнения в пользовательском режиме. */
 GetProcessTimes(hProc, &CreateTime.ft, &ExitTime.ft, &KernelTime, &UserTime);
 ElapsedTime.li = ExitTime.li – CreateTime.li;
 FileTimeToSystemTime(&ElapsedTime.ft, &ElTiSys);
 FileTimeToSystemTime(&KernelTime, &KeTiSys);
 FileTimeToSystemTime(&UserTime, &UsTiSys);
 _tprintf(_T("Истекшее время: %02d:%02d:%02d:%03d\n"), ElTiSys.wHour, ElTiSys.wMinute, ElTiSys.wSecond, ElTiSys.wMilliseconds);
 _tprintf(_T("Пользовательское время: %02d:%02d:%02d:%03d\n"), UsTiSys.wHour, UsTiSys.wMinute, UsTiSys.wSecond, UsTiSys.wMilliseconds);
 _tprintf(_T("Системное время: %02d:%02d:%02d:%03d\n"), KeTiSys.wHour, KeTiSys.wMinute, KeTiSys.wSecond, KeTiSys.wMilliseconds);
} else {
 /* Windows 9x и CE. Вычисляется лишь истекшее время. */
 …
}
CloseHandle(ProcInfo.hThread);
CloseHandle(ProcInfo.hProcess);
CloseHandle(hProc);
return 0;
}
```

Использование команды timep

Теперь мы можем воспользоваться командой timep для анализа производительности различных вариантов программ копирования файлов и их преобразования из ASCII в Unicode, таких, например, как утилиты atou (программа 2.4) и sortMP (программа 5.5). Некоторые из полученных результатов и краткий их анализ представлены в приложении B.

Обратите внимание, что для таких программ, как grepMP, тестирование предоставляет системное и пользовательское время только для родительских процессов. Объекты задач, описанные в конце настоящей главы, позволяют собрать информацию, касающуюся группы процессов. Как показано в приложении B, в случае SMP-систем производительность может повышаться за счет того, что отдельные процессы, вернее, потоки, выполняются на различных процессорах. Выигрыш в производительности возможен и в тех случаях, когда файлы располагаются на различных физических дисках.

Генерация управляющих событий консоли

Прерывание выполнения процесса извне может порождать проблемы, поскольку это лишает процесс возможности произвести необходимую завершающую обработку данных и очистку ресурсов. Воспользоваться SEH в данном случае нельзя ввиду того, что не существует общего метода, который позволял бы одному процессу возбуждать исключения в другом[25]. В то же время, с учетом некоторых ограничений, механизм управляющих событий консоли делает возможной передачу одним процессом другому управляющих сигналов, или событий, консоли. В программе 4.5 было продемонстрировано, как установить обработчик для перехвата сигналов и организовать генерацию исключений этим обработчиком. В указанном примере сигнал генерировался по приказу пользователя средствами пользовательского интерфейса.

Таким образом, вполне можно добиться того, чтобы один процесс генерировал сигнал, соответствующий определенному событию, в другом указанном процессе или группе процессов. Вспомните флаг CREATE_NEW_PROCESS_GROUP функции CreateProcess. Если этот флаг установлен, то идентификатор нового процесса идентифицирует группу процессов и является корневым (root) процессом данной группы. Все новые процессы, создаваемые данным родительским процессом, будут автоматически попадать в эту группу до тех пор, пока при вызове функции CreateProcess не будет использован флаг CREATE_NEW_PROCESS_GROUP. Сгруппированные процессы аналогичны группам процессов в UNIX.

Процесс может генерировать события CTRL_C_EVENT или CTRL_BREAK_EVENT в указанной группе процессов, идентифицируя ее с помощью идентификатора корневого процесса. Консоль целевых процессов должна совпадать с консолью процесса, генерирующего событие. В частности, вызывающий процесс не может быть создан с использованием собственной консоли (посредством флагов CREATE_NEW_CONSOLE или DETACHED_PROCESS).

BOOL GenerateConsoleCtrlEvent(DWORD dwCtrlEvent, DWORD dwProcessGroup)

Тогда значением первого параметра должно быть либо CTRL_C_EVENT, либо CTRL_BREAK_EVENT. Второй параметр идентифицирует группу процессов.

Пример: простое управление задачами

Оболочки UNIX предоставляют команды, позволяющие выполнять процессы в фоновом режиме и получать их текущее состояние. В этом разделе разрабатывается простой "процессор задач" ("job shell") с аналогичным набором команд, перечень которых приводится ниже.

• jobbg — использует остальную часть командной строки в качестве командной строки для нового процесса, или *задачи* (job), однако возврат из команды осуществляется немедленно, без ожидания завершения нового процесса. По желанию пользователя новый процесс может либо получить собственную консоль, либо выполняться как *отсоединенный* (detached) процесс, то есть как процесс, связь с которым не поддерживается. Этот подход аналогичен запуску команд UNIX с указанием опции & в конце команды.

• jobs — выводит список текущих активных задач, снабжая каждую из задач порядковым номером и идентификатором процесса. Эта команда аналогична одноименной команде UNIX.

• kill — прекращает выполнение задачи. В данной реализации используется функция TerminateProcess, которая, как ранее уже отмечалось, не обеспечивает корректного завершения задачи, сопровождающегося "уборкой мусора". Доступна также опция, позволяющая передавать управляющие сигналы консоли.

Создать дополнительные команды, позволяющие приостанавливать существующие задачи или переводить их в фоновый режим, вам будет несложно.

Поскольку выполнение оболочки, которая поддерживает список задач, может быть прекращено, она использует специфический для каждого пользователя разделяемый файл, в котором содержатся идентификаторы процессов, команды и другая необходимая информация. Благодаря этому перезапуск оболочки никак не отразится на списке задач. В одном из упражнений вам предлагается применять для хранения этой информации не временный файл, а реестр.

Реализация программы наталкивается на определенные проблемы, связанные с параллельным выполнением задач. Некоторые процессы, запущенные из командных строк различных оболочек, могут одновременно пытаться управлять задачами. Чтобы справиться с этим, функции управления задачами используют блокировки (глава 3) в файле списка задач, в результате чего пользователь может активизировать управление задачами из различных оболочек или процессов.

В полном варианте программы, находящемся на Web-сайте книги, содержится ряд дополнительных возможностей, не представленных в

приводимых листингах, например, возможность получения входных данных для командной строки из файла. Программа JobDhell послужит основой для создания более общего "процессора служб" ("service processor") в главе 13 (программа 13.3). Службы NT являются фоновыми процессами, обычно — серверами, управление которыми осуществляется командами запуска, остановки, приостановки, а также другими командами.

Создание фоновых задач

Программа 6.3 реализует процессор задач, в котором пользователю предлагается ввести одну из трех возможных команд для их дальнейшего выполнения программой. В этой программе используется набор функций управления задачами, представленный программами 6.4, 6.5 и 6.6.

Программа 6.3. JobShell: создание, вывод списка и прекращение выполнения фоновых задач

```
/* Глава 6. */
/* JobShell.c – команды управления задачами:
   jobbg — Выполнить задачу в фоновом режиме.
   jobs — Вывести список всех фоновых задач.
   kill — Прекратить выполнение указанной задачи из семейства задач.
    Существует опция, позволяющая генерировать управляющие сигналы консоли. */
#include "EvryThng.h"
#include "JobMgt.h"

int _tmain(int argc, LPTSTR argv[]) {
 BOOL Exit = FALSE;
 TCHAR Command[MAX_COMMAND_LINE + 10], *pc;
 DWORD i, LocArgc; /* Локальный параметр argc. */
 TCHAR argstr[MAX_ARG][MAX_COMMAND_LINE];
 LPTSTR pArgs[MAX_ARG];
 for (i = 0; i < MAX_ARG; i++) pArgs[i] = argstr[i];
 /* Вывести подсказку пользователю, считать команду и выполнить ее. */
 _tprintf(_T("Управление задачами Windows\n"));
 while (!Exit) {
  _tprintf(_T("%s"), _T("JM$"));
  _fgetts(Command, MAX_COMMAND_LINE, stdin);
  pc = strchr(Command, '\n');
  *pc = '\0';
  /* Выполнить синтаксический разбор входных данных с целью получения командной строки для новой задачи. */
  GetArgs(Command, &LocArgc, pArgs); /* См. Приложение А. */
  CharLower(argstr[0]);
  if(_tcscmp(argstr[0], _T("jobbg")) == 0) {
   Jobbg(LocArgc, pArgs, Command);
  } else if(_tcscmp(argstr[0], _T("jobs")) == 0) {
   Jobs(LocArgc, pArgs, Command);
```

```
    } else if(_tcscmp(argstr[0], _T("kill")) == 0) {
     Kill(LocArgc, pArgs, Command);
    } else if(_tcscmp(argstr[0], _T("quit")) == 0) {
     Exit = TRUE;
    } else _tprintf(_T("Такой команды не существует. Повторите ввод\n"));
   }
   return 0;
  }

  /* jobbg [параметры] командная строка [Параметры являются взаимоисключающими]
   –c: Предоставить консоль новому процессу.
   -d: Отсоединить новый процесс без предоставления ему консоли.
   Если параметры не заданы, процесс разделяет консоль с jobbg. */
  int Jobbg(int argc, LPTSTR argv[], LPTSTR Command) {
   DWORD fCreate;
   LONG JobNo;
   BOOL Flags[2];
   STARTUPINFO Startup;
   PROCESS_INFORMATION ProcessInfo;
   LPTSTR targv = SkipArg(Command);
   GetStartupInfo(&StartUp);
   Options(argc, argv, _T("cd"), &Flags[0], &Flags[1], NULL);
   /* Пропустить также поле параметра, если он присутствует. */
   if (argv[1][0] == '-') targv = SkipArg(targv);
   fCreate = Flags[0] ? CREATE_NEW_CONSOLE : Flags [1] ? DETACHED_PROCESS : 0;
   /* Создать приостановленную задачу/поток. Возобновить выполнение после ввода номера задачи. */
   CreateProcess(NULL, targv, NULL, NULL, TRUE, fCreate | CREATE_SUSPENDED | CREATE_NEW_PROCESS_GROUP, NULL, NULL, &StartUp, &ProcessInfo);
   /* Создать номер задачи и ввести ID и дескриптор процесса в "базу данных" задачи. */
   JobNo = GetJobNumber(&ProcessInfo, targv); /* См. "JobMgt.h" */
   if (JobNo >= 0) ResumeThread(ProcessInfo.hThread);
   else {
    TerminateProcess(ProcessInfo.hProcess, 3);
    CloseHandle(ProcessInfo.hProcess);
    ReportError(_T("Ошибка: Не хватает места в списке задач."), 0, FALSE);
    return 5;
   }
   CloseHandle(ProcessInfo.hThread);
   CloseHandle(ProcessInfo.hProcess);
   _tprintf(_T(" [%d] %d\n"), JobNo, ProcessInfo.dwProcessId);
```

```
 return 0;
}

/* jobs: вывод списка всех выполняющихся и остановленных задач. */
int Jobs(int argc, LPTSTR argv[], LPTSTR Command) {
 if (!DisplayJobs ()) return 1; /*См. описание функций управления задачами*/
 return 0;
}

/* kill  [параметры] Номер задачи (JobNumber)
 –b: Генерировать Ctrl-Break.
 –c: Генерировать Ctrl-C.
 В противном случае прекратить выполнение процесса. */
int Kill(int  argc, LPTSTR argv[], LPTSTR Command) {
 DWORD ProcessId, JobNumber, iJobNo;
 HANDLE hProcess;
 BOOL CntrlC, CntrlB, Killed;
 iJobNo = Options(argc, argv, _T("bc"), &CntrlB, &CntrlC, NULL);
 /* Найти ID процесса, связанного с данной задачей. */
 JobNumber = _ttoi(argv [iJobNo]);
 ProcessId = FindProcessId(JobNumber); /* См. описание функций управления задачами. */
 hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, ProcessId);
 if (hProcess == NULL) { /* ID процесса может не использоваться. */
  ReportError(_T("Выполнение процесса уже прекращено.\n"), 0, FALSE);
  return 2;
 }
 if (CntrlB) GenerateConsoleCtrlEvent(CTRL_BREAK_EVENT, ProcessId);
 else if (CntrlC) GenerateConsoleCtrlEvent(CTRL_C_EVENT, ProcessId);
 else TerminateProcess(hProcess, JM_EXIT_CODE);
 WaitForSingleObject(hProcess, 5000);
 CloseHandle(hProcess);
 _tprintf(T("Задача [%d] прекращена или приостановлена \n"), JobNumber);
 return 0;
}
```

Обратите внимание на то, как команда jobbg создает процесс в приостановленном состоянии, а затем вызывает функцию управления задачами Get JobNumber (программа 6.4) для получения номера задачи, а также регистрации задачи и процесса, который с ней связан. Если в силу каких-либо причин задача не может быть зарегистрирована, выполнение данного процесса немедленно прекращается. Обычно' генерируется корректный номер задачи, после чего выполнение основного потока возобновляется, и он может продолжать выполнение.

Получение номера задачи

Следующие три программы представляют три отдельные функции управления задачами. Все эти функции включены в единый файл JobMgt.c, содержащий все исходные тексты.

Первая из них, программа 6.4, представляет функцию Get JobNumber. Обратите внимание на использование блокирования файлов, а также обработчиков завершения, осуществляющих разблокирование файлов. Эта методика обеспечивает защиту от исключений и непреднамеренного обхода вызова функции разблокирования файлов. Переходы такого рода могут быть случайно вставлены в процессе сопровождения кода, даже если исходная программа корректна. Обратите также внимание на блокирование попыток записи за пределами конца файла в тех случаях, когда файл должен быть расширен за счет добавления новой записи.

Программа 6.4. JobMgt: создание информации о новой задаче

```
/* Вспомогательная функция управления задачами. */
#include "EvryThng.h"
#include "JobMgt.h" /* Листинг приведен в приложении А. */
void GetJobMgtFileName (LPTSTR);

LONG GetJobNumber(PROCESS_INFORMATION *pProcessInfo, LPCTSTR
Command)
/* Создать номер задачи для нового процесса и ввести информацию о
новом процессе в базу данных задачи. */
{
 HANDLE hJobData, hProcess;
 JM_JOB JobRecord;
 DWORD JobNumber = 0, nXfer, ExitCode, FsLow, FsHigh;
 TCHAR JobMgtFileName[MAX_PATH];
 OVERLAPPED RegionStart;
 if (!GetJobMgtFileName(JobMgtFileName))  return –1;
 /* Предоставление результата в виде строки "\tmp\UserName.JobMgt" */
 hJobData = CreateFile(JobMgtFileName, GENERIC_READ |
GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
 if (hJobData == INVALID_HANDLE_VALUE)  return –1;
 /* Блокировать весь файл плюс одну возможную запись для получения
исключительного доступа. */
 RegionStart.Offset = 0;
 RegionStart.OffsetHigh = 0;
 RegionStart.hEvent = (HANDLE)0;
 FsLow = GetFileSize(hJobData,  &FsHigh);
 LockFileEx(hJobData, LOCKFILE_EXCLUSIVE_LOCK, 0, FsLow +
SJM_JOB, 0, &RegionStart);
 __try {
  /* Чтение записи для нахождения пустого сегмента. */
```

```
      while(ReadFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL) &&
(nXfer > 0)) {
     if (JobRecord.ProcessId == 0) break;
          hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
JobRecord.ProcessId);
     if (hProcess == NULL) break;
        if (GetExitCodeProcess(hProcess, &ExitCode) && (ExitCode !=
STILL_ACTIVE)) break;
      JobNumber++;
     }
      /* Либо найден пустой сегмент, либо мы находимся в конце файла и
должны создать новый сегмент. */
     if (nXfer != 0) /* Не конец файла. Резервировать. */
      SetFilePointer(hJobData, –(LONG)SJM_JOB, NULL, FILE_CURRENT);
     JobRecord.ProcessId = pProcessInfo->dwProcessId;
     _tcsnccpy(JobRecord.CommandLine, Command, MAX_PATH);
     WriteFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
    } /* Конец try-блока. */
    __finally {
     UnlockFileEx(hJobData, 0, FsLow + SJM_JOB, 0, &RegionStart);
     CloseHandle(hJobData);
    }
    return JobNumber + 1;
   }
```

Вывод списка фоновых задач

Программа 6.5 реализует функцию управления задачами DisplayJobs.

Программа 6.5. JobMgt: отображение списка активных задач

```
   BOOL DisplayJobs(void)
   /* Просмотреть файл базы данных, сообщить статус задачи. */
   {
    HANDLE hJobData, hProcess;
    JM_JOB JobRecord;
    DWORD JobNumber = 0, nXfer, ExitCode, FsLow, FsHigh;
    TCHAR JobMgtFileName[MAX_PATH];
    OVERLAPPED RegionStart;
    GetJobMgtFileName(JobMgtFileName);
    hJobData = CreateFile(JobMgtFileName, GENERIC_READ |
GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    RegionStart.Offset = 0;
    RegionStart.OffsetHigh = 0;
    RegionStart.hEvent = (HANDLE)0;
    FsLow = GetFileSize(hJobData, &FsHigh);
    LockFileEx(hJobData, LOCKFILE_EXCLUSIVE_LOCK, 0, FsLow, FsHigh,
&RegionStart);
```

```
  __try {
    while(ReadFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL) &&
(nXfer > 0)) {
    JobNumber++;
    if (JobRecord.ProcessId == 0) continue;
        hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
JobRecord.ProcessId);
    if (hProcess != NULL) GetExitCodeProcess(hProcess, &ExitCode);
    _tprintf(_T(" [%d] "), JobNumber);
    if (hProcess == NULL) _tprintf(_T(" Готово"));
    else if (ExitCode != STILL_ACTIVE) _tprintf(_T("+ Готово"));
    else _tprintf(_T(" "));
    _tprintf(_T(" %s\n"), JobRecord.CommandLine);
    /* Удалить процессы, которые в системе уже не присутствуют. */
    if (hProcess == NULL) {
     /* Зарезервировать одну запись. */
     SetFilePointer(hJobData, –(LONG)nXfer, NULL, FILE_CURRENT);
     JobRecord.ProcessId = 0;
     WriteFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
    }
   } /* Конец цикла while. */
  } /* Конец __try-блока. */
  __finally {
   UnlockFileEx(hJobData, 0, FsLow, FsHigh, &RegionStart);
   CloseHandle(hJobData);
  }
  return TRUE;
 }
```

Поиск задачи в файле списка задач

Программа 6.6 представляет последнюю функцию управления задачами, FindProcessID, которая получает идентификатор процесса, соответствующего задаче с указанным номером. В свою очередь, идентификатор процесса может использоваться вызывающей программой для получения дескриптора и другой информации о состоянии процесса.

Программа 6.6. JobMgt: получение идентификатора процесса по номеру задачи

```
 DWORD FindProcessId(DWORD JobNumber)
 /* Получить ID процесса для задачи с указанным номером. */
 {
  HANDLE hJobData;
  JM_JOB JobRecord;
  DWORD nXfer;
  TCHAR JobMgtFileName[MAX_PATH];
  OVERLAPPED RegionStart;
  /* Открыть файл управления задачами. */
```

```
GetJobMgtFileName(JobMgtFileName);
hJobData = CreateFile(JobMgtFileName, GENERIC_READ,
FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
if (hJobData == INVALID_HANDLE_VALUE) return 0;
/* Перейти к позиции записи, соответствующей указанному номеру задачи.
 * В полной версии программы обеспечивается принадлежность номера
задачи (JobNumber) допустимому диапазону значений. */
SetFilePointer(hJobData, SJM_JOB * (JobNumber – 1), NULL, FILE_BEGIN);
/* Блокировка и чтение записи. */
RegionStart.Offset = SJM_JOB * (JobNumber – 1);
RegionStart.OffsetHigh =0; /* Предполагаем, что файл "короткий". */
RegionStart.hEvent = (HANDLE)0;
LockFileEx(hJobData, 0, 0, SJM_JOB,0, &RegionStart);
ReadFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
UnlockFileEx(hJobData, 0, SJM_JOB, 0, &RegionStart);
CloseHandle(hJobData);
return JobRecord.ProcessId;
}
```

Объекты задач

Процессы можно объединять в объекты задач (job objects), что позволяет управлять процессами как группой, устанавливать лимиты ресурсов для всех процессов, входящих в объект задачи, и вести учетную информацию. Объекты задач были впервые введены в Windows 2000 и теперь поддерживаются во всех системах NT5.

Первым шагом является создание пустого объекта задачи с помощью функции CreateObject, которая принимает два аргумента, имя и атрибуты защиты, и возвращает дескриптор объекта задачи. Существует также функция OpenJobObject, которую можно применять к именованным объектам задач. Для уничтожения объектов используется функция CloseHandle.

Функция AssignProcessToJobObject просто добавляет процесс с указанным дескриптором в объект задачи; она принимает только два параметра. Процесс может принадлежать только одной задаче, поэтому в тех случаях, когда процесс, связанный с указанным дескриптором, уже является элементом какого-либо задания, функция AssignProcessToJobObject завершается с ошибкой. Добавляемый в задачу процесс наследует значения всех ограничений, связанных с задачей, и добавляет в задачу свою учетную информацию, например использованное процессорное время.

По умолчанию новый дочерний процесс, созданный функцией CreateProcess, также принадлежит задаче, если только в аргументе dwCreationFlags при вызове функции CreateProcess не был задан флаг CREATE_BREAKWAY_FROM_JOB. В предусмотренном по умолчанию случае попытки назначения дочернего процесса задаче при помощи функции AssignProcessToJobObject приводят к ее сбойному завершению.

Наконец, для установления управляющих лимитов процессов, входящих в задачу, используется функция SetInformationJobObject.

BOOL SetInformationJobObject(HANDLE hJob, JOBOBJECTINFOCLASS JobObjectInformationClass, LPVOID lpJobObjectInformation, DWORD cbJobObjectInformationLength)

• hJob — дескриптор существующего объекта задачи.

• JobObjectInformationClass — указывает информационный класс устанавливаемых ограничений. Всего существует пять возможных значений; одним из них является значение JobObjectBasicLimitInformation, используемое для указания такой информации, как ограничения общего времени и времени, приходящегося на один процесс, ограничения размеров рабочего набора (working set)[26], ограничения на количество активных процессов, приоритет и родство процессоров (в SMP-системах родственными называются процессоры, которые могут использоваться потоками в процессах задач).

• lpJobObjectInformation — указывает на фактическую информацию, необходимую для предыдущего параметра. Для каждого информационного класса существует своя структура.

• JOBOBJECT_BASIC_ACCOUNTING_INFORMATION — позволяет получить суммарные временные характеристики (пользовательское, системное и истекшее время) процессов, входящих в задачу.

• Значением последнего параметра является размер предыдущей структуры.

Функция QueryJobInformationObject позволяет получить значения текущих ограничений. Другие информационные классы устанавливают ограничения в отношении пользовательского интерфейса, портов завершения ввода/вывода (см. главу 14), атрибутов защиты, а также завершения задачи.

Резюме

Windows предоставляет простой механизм управления процессами и синхронизацией их выполнения. Приведенные примеры продемонстрировали способы управления параллельным выполнением нескольких процессов, а также получения информации о временных характеристиках каждого процесса. Отношения "предок-потомок" между процессами в Windows не поддерживаются, так что в необходимых случаях управление этой информацией возлагается на программиста.

В следующих главах

В следующей главе описываются потоки, являющиеся независимыми единицами выполнения внутри процесса. В некоторых отношениях управление потоками аналогично управлению процессами; все, что связано с кодами завершения, прекращением выполнения и ожиданием завершения, применимо и к потокам. Чтобы продемонстрировать эту аналогию, самый первый из рассматриваемых в главе 7 примеров является переделанным вариантом программы grepMP (программа 6.1), который приспособлен для работы с потоками.

Глава 8 ознакомит вас с методами синхронизации, которые могут быть использованы для координации выполнения потоков, принадлежащих одному и тому же или различным процессам.

Упражнения

6.1. Расширьте возможности программы 6.1 (grepMP) таким образом, чтобы она принимала также параметры командной строки, а не только текстовый шаблон.

6.2. Вместо того чтобы передавать дочернему процессу имя временного файла, как это делается в программе 6.1, преобразуйте наследуемый дескриптор файла к типу DWORD (для типа HANDLE требуется 4 байта), а затем в строку символов. Передайте эту строку дочернему процессу в командной строке. В свою очередь, дочерний процесс должен осуществить обратное преобразование строки символов в значение дескриптора файла, который будет использован для вывода. Эту методику иллюстрируют программы catHA.c и grepHA.c, доступные на Web-сайте книги.

6.3. Программа 6.1 ожидает завершения всех процессов и лишь после этого выводит результаты. При этом возможность определения того, в каком именно порядке завершились процессы внутри программы, отсутствует. Модифицируйте программу таким образом, чтобы она определяла очередность завершения процессов. *Подсказка.* Измените вызов функции WaitForMultipleObjects таким образом, чтобы возврат из нее осуществлялся после завершения каждого отдельного процесса. Другой возможностью является сортировка времени завершения процессов.

6.4. В программе 6.1 временные файлы должны удаляться явным образом. Возможно ли использование флага FILE_FLAG_DELETE_ON_CLOSE при создании временных файлов таким образом, чтобы избавиться от необходимости удаления указанных файлов?

6.5. Определите, заметны ли какие-либо преимущества программы grepMP в отношении производительности (по сравнению с простой последовательной обработкой) в случае SMP-систем, если такая возможность у вас имеется, или при размещении файлов на отдельных или сетевых дисках. Частичные результаты соответствующих тестов приведены в приложении В.

6.6. Можете ли вы предложить способ, возможно, связанный с использованием объектов задач, для определения времени, затраченного на выполнение операций в пользовательском режиме и в режиме ядра? Использование объектов задач может потребовать внесения изменений в программу grepMP.

6.7. Улучшите функцию grepMP (программа 6.5) таким образом, чтобы она сообщала код завершения для каждой завершенной задачи. Кроме того, организуйте вывод временных характеристик (истекшего времени, времени работы в режиме ядра и времени работы в пользовательском режиме) суммарно для всех процессов.

6.8. У функций управления задачами есть один трудно устранимый недостаток. Предположим, что задача уничтожена и что главная программа повторно использует идентификатор процесса данного задания еще до того,

как этот идентификатор будет удален из файла управления задачами. Вместе с тем, ранее этот идентификатор мог быть использован функцией OpenProcess для создания дескриптора какого-либо процесса, хотя теперь этот же идентификатор ссылается на совершенно другой процесс. Чтобы устранить возможность появления проблем подобного рода, требуется создать вспомогательный процесс, в котором будут храниться копии дескрипторов каждого созданного процесса, что позволит избежать повторного использования идентификаторов. Другая возможная методика заключается в сохранении времени запуска процесса в файле управления задачами. Это время должно совпадать со временем запуска процесса, полученного с использованием идентификатора. *Примечание.* Идентификаторы процессов быстро исчерпываются, и поэтому вероятность их повторного использования очень велика. В UNIX для получения идентификаторов новых процессов применяются последовательно увеличиваемые значения 32-битового счетчика, так что идентификаторы могут повторяться только после исчерпания этих значений, что происходит очень редко. В отличие от этого, в программах Windows никогда нельзя полагаться на то, что идентификатор процесса не будет повторно использован.

6.9. Измените программу JobShell таким образом, чтобы информация сохранялась в реестре, а не во временном файле.

6.10. Измените программу JobShell таким образом, чтобы процессы связывались с объектом задачи. Наложите временные и другого рода ограничения на объекты задач, предоставив пользователю возможность ввода числовых значений некоторых из этих ограничений.

6.11. Улучшите программу JobShell таким образом, чтобы команда jobs обеспечивала подсчет числа дескрипторов, используемых каждой из задач. *Подсказка.* Воспользуйтесь функцией GetProcessHandleCount, для которой требуется NT 5.1.

6.12. Создайте проект Version (находится на Web-сайте), использующий программу verison.c. Попытайтесь произвести пробные запуски этой программы под управлением как можно большего числа различных версий Windows, к которым у вас имеется доступ, включая Windows 9x и NT 4.0, если это возможно. Каковы старшие и младшие номера версий для этих систем, полученные вами, и какую дополнительную информацию о версиях вам удалось получить?