Потоки и планирование выполнения

Основной единицей выполнения в Windows является поток, и одновременно несколько потоков могут выполняться в рамках одного процесса, разделяя его адресное пространство и другие ресурсы. В главе 6 процессы ограничивались только одним потоком, однако существует много ситуаций, в которых возможность использования более одного потока была бы весьма желательной. Настоящая глава посвящена описанию потоков и иллюстрации областей их применения. Глава 8 продолжает эту тему описанием объектов синхронизации и анализом как положительных, так и отрицательных аспектов использования потоков, в то время как в главе 9 исследуется проблема повышения производительности и компромиссные способы ее решения. В главе 10 описываются методы и модели программирования объектов синхронизации, позволяющие значительно упростить проектирование и разработку надежных многопоточных программ. Эти методы применяются далее во всей оставшейся части книги.

Завершается настоящая глава кратким обсуждением облегченных потоков, посредством которых можно создавать отдельные задачи в контексте потоков. Ввиду того, что облегченные потоки используются довольно редко, можно предположить, что многие читатели предпочтут пропустить этот материал при первом чтении.

Обзор потоков

Поток (thread) — это независимая единица выполнения в контексте процесса. Программист, разрабатывающий многопоточную программу, должен организовать выполнение потоков таким образом, чтобы это позволило упростить программу и воспользоваться предоставляемыми самим хост-компьютером возможностями распараллеливания задач.

При традиционном подходе программы выполняются в виде единственного потока. Несмотря на возможность организации параллельного выполнения нескольких процессов, что было продемонстрировано на ряде примеров в главе 6, и даже их взаимодействия между собой посредством таких механизмов, как разделение памяти или каналы (глава 11), однопоточные процессы имеют ряд недостатков.

• Переключение между выполняющимися процессами потребляет заметную долю временных и других ресурсов ОС, а в случаях, аналогичных многопроцессному поиску (grepMP, программа 6.1), все процессы заняты выполнением одной и той же программы. Организация параллельной обработки файла с помощью потоков в контексте единственного процесса позволяет снизить общие накладные расходы системы.

• Не считая случаев разделения памяти, между процессами существует лишь слабая взаимосвязь, а организация разделения ресурсов, например, открытых файлов, вызывает затруднения.

• С использованием только однопоточных процессов трудно организовать простое и эффективное управление несколькими параллельно выполняющимися задачами, взаимодействующими между собой, в таких, например, случаях, как ожидание и обработка пользовательского ввода, ожидание ввода из файла или сети и выполнение вычислений.

• Тесно связанные с выполнением операций ввода/вывода программы, подобные рассмотренной в главе 2 программе преобразования файлов из ASCII в Unicode (atou, программа 2.4), вынуждены ограничиваться простой моделью "чтение-изменение-запись". При обработке последовательностей файлов гораздо эффективнее инициализировать выполнение как можно большего числа операций чтения. Windows NT предлагает дополнительные возможности перекрывающегося асинхронного ввода/вывода (глава 14), однако потоки позволяют добиться того же эффекта с меньшими усилиями.

• В SMP-системах планировщик Windows распределяет выполнение отдельных потоков между различными процессорами, что во многих случаях приводит к повышению производительности.

В этой главе обсуждаются потоки и способы управления ими. Использование потоков рассматривается на примере задач параллельного поиска и многопоточной сортировки содержимого файлов. Эти две задачи позволяют сопоставить применение потоков в операциях ввода/вывода и в операциях, связанных с выполнением интенсивных вычислений. Кроме того, в этой главе представлен общий обзор планирования выполнения процессов и потоков в Windows.

Перспективы и проблемы

Согласно принятой в этой и последующих главах точке зрения использование потоков не только позволяет упростить проектирование и реализацию некоторых программ, но и (при условии соблюдения нескольких элементарных правил и следования определенным моделям программирования) обеспечивает повышение производительности и надежности программ, а также делает более понятной их структуру и облегчает их обслуживание. Функции управления потоками весьма напоминают функции управления процессами, так что, например, наряду с функцией GetProcessExitCode существует также функция GetThreadExitCode.

Указанная точка зрения не является общепринятой. Многие авторы и разработчики программного обеспечения обращают внимание на всевозможные риски и проблемы, которые возникают в случае использования потоков, и отдают предпочтение использованию нескольких процессов, когда требуется параллелизм операций. К числу проблем упомянутого рода относятся следующие:

• Поскольку потоки разделяют общую память и другие ресурсы, принадлежащие одному процессу, существует вероятность того, что один поток может случайно изменить данные, относящиеся к другому потоку.

• При определенных обстоятельствах вместо улучшения производительности может наблюдаться ее резкое ухудшение.

• Разделение потоками общей памяти и других ресурсов в контексте одного процесса может стать причиной нарушения условий состязаний между процессами и вызывать блокирование некоторых из них.

Некоторых проблем, с которыми действительно приходится сталкиваться, можно избежать, тщательно проектируя и программируя соответствующие задачи, тогда как природа других проблем обусловлена самим параллелизмом, независимо от того, реализуется он путем разбиения процессов на потоки, использованием нескольких процессов или применением специальных методов, например, методов асинхронного ввода/вывода, предоставляемых Windows.

Основные сведения о потоках

В предыдущей главе на рис. 6.1 было показано, каким образом обеспечивается существование потоков в среде процесса. Использование потоков на примере многопоточного сервера, способного обрабатывать запросы одновременно нескольких клиентов, иллюстрирует рис. 7.1; каждому клиенту отведен поток. Эта модель будет реализована в главе 11.

Потоки, принадлежащие одному процессу, разделяют общие данные и код, поэтому очень важно, чтобы каждый поток имел также собственную область памяти, относящуюся только к нему. В Windows удовлетворение этого требования обеспечивается несколькими способами.

• У каждого потока имеется собственный стек, который она использует при вызове функций и обработке некоторых данных.

• При создании потока вызывающий процесс может передать ему аргумент (Arg на рис. 7.1), который обычно является указателем. На практике этот аргумент помещается в стек потока.

• Каждый поток может распределять индексы собственных локальных областей хранения (Thread Local Storage, TLS), а также считывать и устанавливать значения TLS. TLS, описанные далее, предоставляют в распоряжение потоков небольшие массивы данных, и каждый из потоков может обращаться к собственной TLS. Одним из преимуществ TLS является то, что они обеспечивают защиту данных, принадлежащих одному потоку, от воздействия со стороны других потоков.

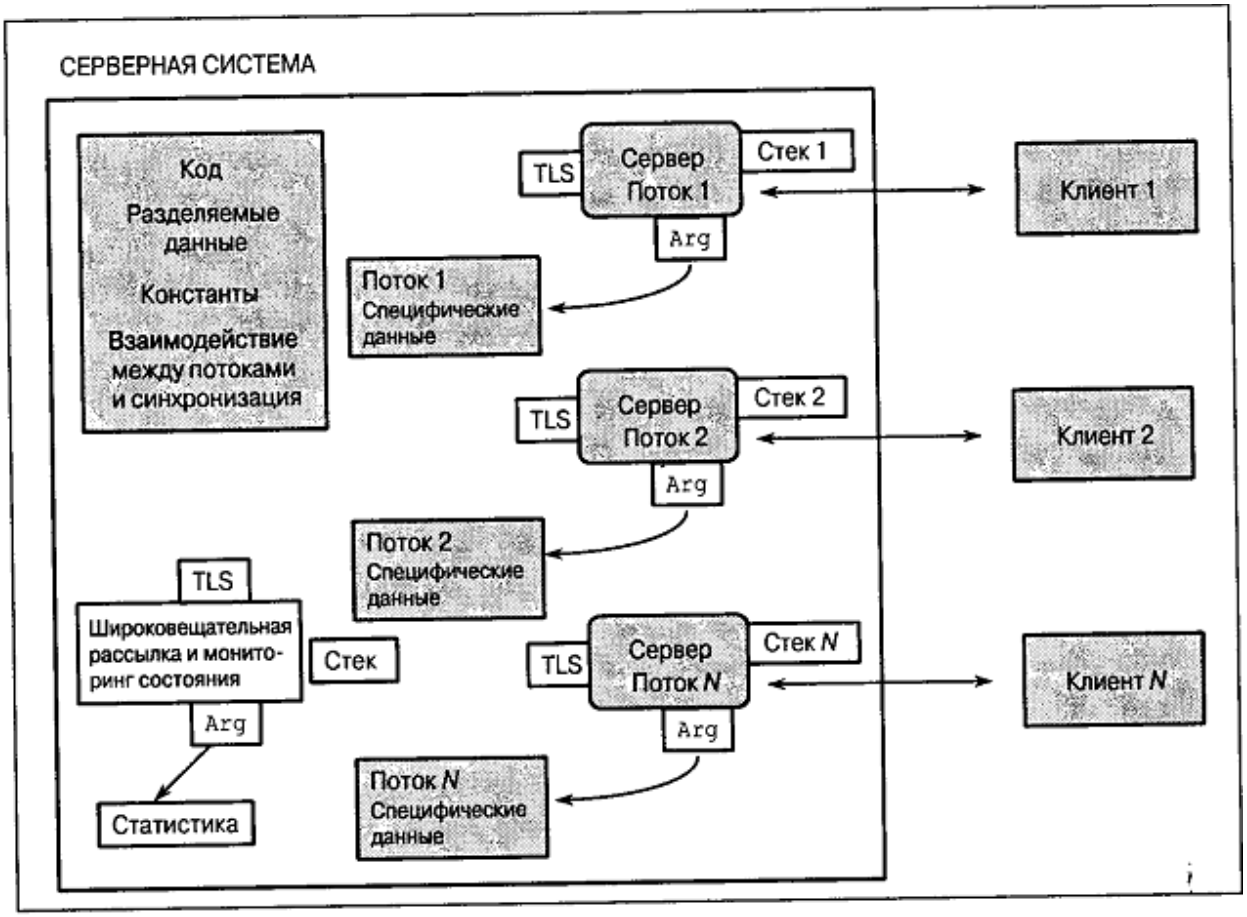

Рис. 7.1. Потоки в среде сервера

Аргумент потока и TLS могут использоваться для указания произвольной структуры данных. Применительно к представленному на рис. 7.1 примеру сервера эта структура может содержать текущий запрос и отклик потока на этот запрос, а также предоставлять рабочую память для других целей.

В случае SMP-систем Windows обеспечивает параллельное выполнение различных потоков, в том числе и принадлежащих одному и тому же процессу, на разных процессорах. Правильное использование этой возможности позволяет повысить производительность, однако, как будет показано в двух следующих главах, в результате непродуманных действий без заранее определенной стратегии использования нескольких процессоров производительность SMP-систем может даже ухудшиться по сравнению с однопроцессорными системами.

Управление потоками

Вероятно, вы не будете удивлены, узнав о том, что у потоков, как и у любого другого объекта Windows, имеются дескрипторы и что для создания потоков, выполняющихся в адресном пространстве вызывающего процесса, предусмотрен системный вызов CreateThread. Как и в случае процессов, мы

будем говорить иногда о "родительских" и "дочерних" потоках, хотя ОС не делает в этом отношении никаких различий. Системный вызов CreateThread предъявляет ряд специфических требований:

• Укажите начальный адрес потока в коде процесса.

• Укажите размер стека, и необходимое пространство стека будет выделено из виртуального адресного пространства процесса. Размер стека по умолчанию равен размеру стека основного потока (обычно 1 Мбайт). Первоначально для стека отводится одна страница (см. главу 5). Новые страницы стека выделяются по мере надобности до тех пор, пока стек не достигнет своего максимального размера, поэтому не сможет больше расти.

• Задайте указатель на аргумент, передаваемый потоку. Этот аргумент может быть чем угодно и должен интерпретироваться самим потоком.

• Функция возвращает значение идентификатора (ID) и дескриптор потока. В случае ошибки возвращаемое значение равно NULL.

HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpsa, DWORD dwStackSize, LPTHREAD_START_ROUTINE lpStartAddr, LPVOID lpThreadParm, DWORD dwCreationFlags, LPDWORD lpThreadId)

Параметры

lpsa — указатель на уже хорошо знакомую структуру атрибутов защиты.

dwStackSize — размер стека нового потока в байтах. Значению 0 этого параметра соответствует размер стека по умолчанию, равный размеру стека основного потока.

lpStartAddr — указатель на функцию (принадлежащую контексту процесса), которая должна выполняться. Эта функция принимает единственный аргумент в виде указателя и возвращает 32-битовый код завершения. Этот аргумент может интерпретироваться потоком либо как переменная типа DWORD, либо как указатель. Функция потока (ThreadFunc) имеет следующую сигнатуру:

DWORD WINAPI *ThreadFunc*(LPVOID)

lpThreadParm — указатель, передаваемый потоку в качестве аргумента, который обычно интерпретируется потоком как указатель на структуру аргумента.

dwCreationFlags — если значение этого параметра установлено равным 0, то поток запускается сразу же после вызова функции CreateThread. Установка значения CREATE_SUSPENDED приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности путем вызова функции ResumeThread.

lpThreadId — указатель на переменную типа DWORD, которая получает идентификатор нового потока.

Любой поток процесса может сам завершить свое выполнение, вызвав функцию ExitThread, однако более обычным способом самостоятельного завершения потока является возврата из функции потока с использованием кода завершения в качестве возвращаемого значения. По завершении

выполнения потока память, занимаемая ее стеком, освобождается. В случае если поток был создан в библиотеке DLL, будет вызвана соответствующая точка входа DllMain (глава 4) с указанием флага DLL_THREAD_DETACH в качестве "причины" этого вызова.

VOID ExitThread(DWORD dwExitCode)

Когда завершается выполнение последнего потока, завершается и выполнение самого процесса.

Выполнение потока также может быть завершено другим потоком с помощью функции TerminateThread, однако освобождения ресурсов потока при этом не происходит, обработчики завершения не выполняются и уведомления библиотекам DLL не посылаются. Лучше всего, когда поток сам завершает свое выполнение; применять для этого функцию TerminateThread крайне нежелательно. Функции TerminateThread присущи те же недостатки, что и функции TerminateProcess.

Поток, выполнение которого было завершено (напомним, что обычно поток должен самостоятельно завершать свое выполнение), продолжает существовать до тех пор, пока посредством функции CloseHandle не будет закрыт ее последний дескриптор. Любой другой поток, возможно и такой, который ожидает завершения другого потока, может получить код завершения потока.

BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode)

lpExitCode — будет содержать код завершения потока, указывающий на его состояние. Если поток еще не завершен, значение этой переменной будет равно STILL_ACTIVE.

Идентификация потоков

Функции, используемые для получения идентификаторов (ID) и дескрипторов потоков, напоминают те, которые используются для аналогичных целей в случае процессов.

• GetCurrentThread — возвращает ненаследуемый псевдодескриптор вызывающего потока.

• GetCurrentThreadId — позволяет получить идентификатор потока, а не его дескриптор.

• GetThreadId — позволяет получить идентификатор потока, если известен его дескриптор; эта функция требует использования Windows Server 2003.

• OpenThread — создает дескриптор потока по известному идентификатору.

В программе JobShell (программа 6.3) нам очень пригодилась функция OpenProcess, и функция OpenThread может применяться для аналогичных целей.

Дополнительные функции управления потоками

Несмотря на то что функций управления потоками, которые мы выше обсуждали, вполне достаточно для большинства случаев, в том числе и для примеров, приведенных в этой книге, в Windows XP и Windows Server 2003 были введены две дополнительные функции. Их краткие описания представлены ниже.

1. Функция **GetProcessIdOfThread**, требующая использования Windows Server 2003, позволяет получать идентификатор процесса, которому принадлежит поток, по известному дескриптору потока. Вы могли бы задействовать эту функцию в программах, предназначенных для управления потоками, принадлежащими другим процессам, или взаимодействия с такими потоками. Если необходимо получить дескриптор процесса, применяйте для этого функцию OpenProcess.

2. Функция **GetThreadIOPendingFlag** позволяет определить, имеются ли у потока, на который указывает дескриптор, необслуженные запросы ввода/вывода. Например, поток мог быть заблокирован во время выполнения операции ReadFile. В качестве результата возвращается состояние потока во время выполнения данной функции; фактическое состояние может в любой момент измениться, если целевой поток завершает или начинает выполнение операции. Эта функция требует использования NT 5.1 и поэтому доступна лишь в Windows XP или Windows Server 2003.

Приостановка и возобновление выполнения потока

Для каждого потока поддерживается *счетчик приостановок* (suspend count), и выполнение потока может быть продолжено лишь в том случае, если значение этого счетчика равно 0. Поток может увеличивать или уменьшать значение счетчика приостановок другого потока с помощью функций SuspendThread и ResumeThread. Вспомните, что поток можно создать в приостановленном состоянии со счетчиком приостановок равным 1.

DWORD ResumeThread(HANDLE hThread)
DWORD SuspendThread(HANDLE hThread)

В случае успешного выполнения обе функции возвращают предыдущее значение счетчика приостановок, иначе — 0xFFFFFFFF.

Ожидание завершения потока

Поток может дожидаться завершения выполнения другого потока точно так же, как потоки могут дожидаться завершения процесса, что обсуждалось в главе 6. В этом случае при вызове функций ожидания (WaitForSingleObject и WaitForMultipleObjects) вместо дескрипторов процессов следует использовать дескрипторы потоков. Заметьте, что не все дескрипторы в массиве, передаваемом функции WaitForMultipleObjects, должны быть обязательно одного и того же типа; например, в одном вызове могут быть одновременно указаны дескрипторы потоков, процессов и других объектов.

Допустимое количество объектов, одновременно ожидаемых функцией WaitForMultipleObjects, ограничено значением MAXIMUM_WAIT_OBJECTS (64), но при большом количестве потоков можно воспользоваться серией вызовов функций ожидания. Эта техника уже была продемонстрирована в программе 6.1; программы, приведенные в книге, ожидают завершения выполнения одиночных объектов, но на Web-сайте приведены полные решения.

Функция ожидания дожидается, пока объект, указанный дескриптором, не перейдет в *сигнальное* состояние. В случае потоков объект потока переводится в сигнальное состояние при помощи функций ExitThread и TerminateThread, что приводит к освобождению всех других потоков, дожидающихся перехода данного объекта в сигнальное состояние, включая и те потоки, которые могли оставаться в состоянии ожидания и впоследствии, после того, как поток завершится. Дескриптор потока, перешедший в сигнальное состояние, не выходит из этого состояния. То же самое остается справедливым и по отношению к дескрипторам процессов, но не относится к дескрипторам некоторых других объектов, например, мьютексов и событий (описываются в следующей главе).

Заметьте, что дожидаться перехода в сигнальное состояние одного и того же объекта могут одновременно несколько потоков. Аналогично, функция ExitProcess переводит в сигнальное состояние как сам процесс, так и все его потоки.

Удаленные потоки

Функция **CreateRemoteThread** позволяет создавать потоки, выполняющиеся в другом процессе. По сравнению с функцией CreateThread в ней имеется один дополнительный параметр для указания дескриптора процесса, а адрес функции, задающий начальный адрес нового потока, должен находиться в адресном пространстве целевого процесса. Использование функции CreateRemoteThread относится к числу интересных, однако рискованных способов непосредственного воздействия одним процессом на другой, и может пригодиться, например, при написании отладчиков.

У функции CreateRemoteThread есть одно очень интересное применение. Вместо того чтобы вызывать функцию TerminateProcess, управляющий процесс может создать поток, выполняющийся в другом процессе, который и организует корректное завершение этого процесса. Однако в главе 10 демонстрируется более безопасный метод, позволяющий одному потоку завершить другой с использованием асинхронного вызова процедур.

Понятие о потоках твердо упрочилось во многих ОС, и исторически так сложилось, что многие поставщики и пользователи UNIX предоставляли собственные частные варианты их реализации. Были разработаны некоторые библиотеки, обеспечивающие многопоточную поддержку вне ядра. В настоящее время стандартом в этой области являются потоки POSIX Pthreads. Потоки Pthreads включены в частные варианты реализации UNIX и

Linux и иногда считаются частью UNIX. Соответствующие системные вызовы отличаются от обычных системных вызовов UNIX наличием в именах префикса pthread. Потоки Pthreads поддерживаются также некоторыми другими системами, отличными от UNIX, такими, например, как Open VMS.

Системный вызов pthread_create эквивалентен вызову CreateThread, а системный вызов pthread_exit — вызову ExitThread. Для организации ожидания одним потоком завершения другого применяется системный вызов pthread_join. Потоки Pthreads предоставляют очень полезную функцию pthread_cancel, гарантирующую, в отличие от функции TerminateThread, выполнение обработчиков завершения и уничтожение ненужных дескрипторов. Возможность уничтожения потоков была бы в Windows крайне желательной, но в главе 10 представлен метод, обеспечивающий получение такого же эффекта.

Использование библиотеки C в потоках

В большинстве программ требуется библиотека C, хотя бы для того, чтобы обеспечить выполнение операций над строками. Исторически так сложилось, что библиотека C была рассчитана на применение в однопоточных процессах, поэтому для хранения промежуточных результатов многие функции используют области глобальной памяти. Подобные библиотеки, в которых отсутствует многопоточная поддержка, не являются безопасными (thread-safe) с точки зрения одновременного выполнения нескольких потоков, поскольку, например, одновременно две независимые потоки могут пытаться получить доступ к библиотеке и изменить данные, содержащиеся в ее глобальной памяти. Принципы проектирования многопоточных программ будут вновь обсуждаться в главе 8, в которой описывается синхронизация объектов Windows.

Пример функции strtok показывает, почему при написании некоторых функций библиотеки C не учитывалась многопоточная поддержка. Функция strtok, просматривающая строку в поиске очередного вхождения определенной лексемы, поддерживает сохранение состояния (persistent state) между последовательными вызовами функции, и это состояние хранится в области статической памяти, совместный доступ к которой имеют все потоки, вызывающие эту функцию.

Microsoft C решает эту проблему, предлагая реализацию библиотеки C под названием LIBCMT.LIB, которая обеспечивает многопоточную поддержку. Однако, это еще не все. Вы не должны использовать функцию CreateThread; для запуска потока и создания специфической для него области рабочей памяти библиотеки LIBCMT.LIB необходимо пользоваться специальной функцией C, а именно, функцией _beginthreadex. Для завершения потока вместо функции ExitThread применяется функция _endthreadex.

Примечание

В качестве упрощенного варианта функции _beginthreadex предусмотрена функция _beginthread, однако *использовать ее не рекомендуется.* Прежде всего, функция _beginthread не имеет ни атрибутов, ни флагов защиты и не возвращает идентификатор потока. Более того, в действительности она закрывает дескриптор потока, который создает, в результате чего возвращенное значение дескриптора может оказаться недействительным на момент его сохранения родительским потоком. Не следует вызывать и функцию _endthread; она не позволяет пользоваться возвращаемым значением.

Аргументы функции _beginthreadex в точности совпадают с аргументами функций Windows, однако типы данных Windows для этой функции не определены, и поэтому тип возвращаемого значения функции _beginthread необходимо привести к типу HANDLE, что позволит избежать появления предупреждающих сообщений. Убедитесь в том, что определение символической константы _MT предшествует любому из включаемых файлов; в примерах программ это определение содержится в файле Envirmnt.h. Больше от вас ничего не требуется. Резюмируя, перечислим действия, которые вы должны выполнить, если имеете дело со средой разработки Visual C++.

• Подключите библиотеку LIBCMT.LIB и откажитесь от использования библиотеки, заданной по умолчанию.

• Включите директиву #define _MT во все исходные файлы, в которых используется библиотека C.

• Добавьте включаемый файл <process.h>, содержащий определения функций _beginthreadex и _endthreadex.

• Создайте потоки с помощью функции _beginthreadex; не применяйте для этой цели функцию CreateThread.

• Завершите потоки посредством функции _endthreadex или просто воспользуйтесь оператором return в конце функции потока.

В приложении A вы найдете указания относительно того, как создавать многопоточные приложения. В частности, можно, и даже рекомендуется, указывать библиотеку и определять константу _MT непосредственно в среде разработки.

Именно так будут построены все наши примеры, и функция CreateThread никогда не будет непосредственно применяться в программах даже в тех случаях, когда библиотека C в функциях потоков не используется.

Библиотеки с многопоточной поддержкой

При проектировании пользовательских библиотек следует уделять самое пристальное внимание тому, чтобы избежать возникновения проблем, связанных с параллельным выполнением нескольких потоков, особенно в тех случаях, когда речь идет о сохранении информации о состоянии процессов. Одна из возможных стратегий демонстрируется в примере в главе 12 (программа 12.4), где библиотека DLL для сохранения информации о состоянии использует отдельный параметр.

Еще один пример в главе 12 (программа 12.5) иллюстрирует альтернативный подход, в котором применяется функция DllMain и TLS, описанные далее в настоящей главе.

Пример: многопоточный поиск контекста

В программе 6.1 (grepMP) для выполнения одновременного поиска текстового шаблона в нескольких файлах использовались процессы. Программа 7.1 (grepMT), которая включает исходный код функции поиска текстового шаблона grep, обеспечивает выполнение поиска несколькими потоками в рамках одного процесса. Код функции поиска основан на вызовах функций файлового ввода/вывода библиотеки C. Основная программа аналогична той, которая предлагалась в варианте реализации, основанном на использовании процессов.

Этот пример также показывает, что применение потоков позволяет выполнять асинхронные операции ввода/вывода даже без привлечения специально для этого предназначенных методов, описанных в главе 14. В данном примере параллельным вводом/выводом с участием нескольких файлов управляет программа, в то время как основной или любого другого потока предоставляется возможность в ожидании завершения ввода/вывода выполнять дополнительную обработку. По мнению автора, способ реализации асинхронного ввода/вывода, обеспечиваемый потоками, является более простым, а сравнительный анализ эффективности различных методов, представленный в главе 14, поможет вам выработать собственное мнение на этот счет.

Мы увидим, однако, что в сочетании с портами завершения ввода/вывода операции асинхронного ввода/вывода становятся очень полезным, а часто и необходимым средством в тех случаях, когда количество потоков очень велико.

В иллюстративных целях в программу grepMT введено дополнительное отличие по сравнению с программой grepMP. В данном случае функция WaiForMultipleObjects ожидает завершения не всех потоков, а только одного. Соответствующая информация выводится без ожидания завершения других потоков. В большинстве случае порядок завершения потоков будет меняться от одного запуска программы к другому. Программу легко видоизменить таким образом, чтобы результаты отображались в порядке указания аргументов в командной строке; для этого будет достаточно сымитировать программу grepMP.

Наконец, обратите внимание на ограничение в 64 потока, обусловленное значением константы MAXIMUM_WAIT_OBJECTS, которая ограничивает количество дескрипторов при вызове функции WaitForMultipleObjects. Если у вас возникнет необходимость в большем количестве потоков, организуйте для функций WaitForSingleObjects или WaitForMultipleObjects соответствующий цикл.

Предостережение

Программа grepMP осуществляет асинхронный ввод/вывод в том смысле, что отдельные потоки выполняют параллельное синхронное чтение различных файлов, которые блокируются до момента завершения операции чтения. Можно также организовать параллельное чтение одного и того же файла, если у него имеются различные дескрипторы (обычно, по одному дескриптору для каждого потока). Эти дескрипторы должны быть сгенерированы функцией CreateFile, а не функцией DuplicateHandle. В главе 14 описывается асинхронный ввод/вывод, осуществляемый как с использованием, так и без использования пользовательских потоков, а в примере, доступном на Web-сайте (программа atouMT, описанная в главе 14), операции ввода/вывода выполняются с использованием нескольких потоков по отношению к одному и тому же файлу.

Программа 7.1. grepMT: многопоточный поиск текстового шаблона

```
/* Глава 7. grepMT. */
/* Параллельный поиск текстового шаблона — версия, использующая несколько потоков. */
#include "EvryThng.h"

typedef struct { /* Структура данных потока поиска. */
 int argc;
 TCHAR targv[4][MAX_PATH];
} GREP_THREAD_ARG;
typedef GREP_THREAD_ARG *PGR_ARGS;
static DWORD WINAPI ThGrep(PGR_ARGS pArgs);

int _tmain(int  argc, LPTSTR argv[]) {
 GREP_THREAD_ARG * gArg;
 HANDLE  * tHandle;
 DWORD ThdIdxP, ThId, ExitCode;
 TCHAR CmdLine[MAX_COMMAND_LINE];
 int iThrd, ThdCnt;
 STARTUPINFO Startup;
 PROCESS_INFORMATION ProcessInfo;
 GetStartupInfo(&StartUp);
 /* Основной поток: создает отдельные потоки поиска на основе функции "grep" для каждого файла. */
```

```
        tHandle = malloc((argc – 2) * sizeof(HANDLE));
        gArg = malloc((argc – 2) * sizeof(GREP_THREAD_ARG));
        for (iThrd = 0; iThrd < argc – 2; iThrd++) {
         _tcscpy(gArg[iThrd].targv[1], argv[1]); /* Pattern. */
         _tcscpy(gArg[iThrd].targv[2], argv[iThrd + 2]);
         GetTempFileName /* Имя временного файла. */
          (".", "Gre", 0, gArg[iThrd].targv[3]);
         gArg[iThrd].argc = 4;
         /* Создать рабочий поток для выполнения командной строки. */
           tHandle[iThrd] = (HANDLE)_beginthreadex(NULL, 0, ThGrep,
&gArg[iThrd], 0, &ThId);
        }
        /* Перенаправить стандартный вывод для вывода списка файлов. */
        Startup.dwFlags = STARTF_USESTDHANDLES;
        Startup.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
        /* Выполняются все рабочие потоки. Ожидать их завершения. */
        ThdCnt = argc – 2;
        while (ThdCnt > 0) {
           ThdIdxP = WaitForMultipleObjects(ThdCnt, tHandle, FALSE,
INFINITE);
         iThrd = (int)ThdIdxP – (int)WAIT_OBJECT_0;
         GetExitCodeThread(tHandle [iThrd], &ExitCode);
         CloseHandle(tHandle [iThrd]);
         if (ExitCode ==0) { /* Шаблон найден. */
          if (argc > 3) {
           /* Вывести имя файла, если имеется несколько файлов. */
                _tprintf(_T("\n**Результаты поиска – файл: %s\n"),
gArg[iThrd].targv [2]);
           fflush(stdout);
          }
             /* Использовать программу "cat" для перечисления
результирующих файлов. */
          _stprintf(CmdLine, _T("%s%s"), _T("cat "), gArg [iThrd].targv[3]);
           CreateProcess(NULL, CmdLine, NULL, NULL, TRUE, 0, NULL,
NULL, &StartUp, &ProcessInfo);
          WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
          CloseHandle(ProcessInfo.hProcess);
          CloseHandle(ProcessInfo.hThread);
         }
         DeleteFile(gArg[iThrd].targv[3]);
         /* Скорректировать массивы потоков и имен файлов. */
         tHandle[iThrd] = tHandle[ThdCnt – 1];
         _tcscpy(gArg[iThrd].targv[3], gArg[ThdCnt – 1].targv[3]);
         _tcscpy(gArg[iThrd].targv[2], gArg[ThdCnt – 1].targv[2]);
         ThdCnt--;
```

```
 }
}

/* Прототип функции контекстного поиска:
static DWORD WINAPI ThGrep(PGR_ARGS pArgs){ } */
```

Потоки и производительность

Программы grepMP и grepMT по своей структуре и сложности сопоставимы друг с другом, однако, как и следовало ожидать, программа grepMT характеризуется более высокой производительностью, так как переключение между потоками осуществляется ядром намного эффективнее, чем переключение между процессами. В приложении В показано, что эти теоретические ожидания отвечают действительности, и это особенно заметно в тех случаях, когда файлы размещены на различных дисках. Оба варианта реализации способны работать в SMP-системах, существенно улучшая показатели производительности в терминах общего времени выполнения (истекшего времени); потоки, независимо от того, принадлежат ли они одному и тому же или разным процессам, параллельно выполняются на различных процессорах. Измеренное пользовательское время в действительности превышает общее время выполнения, поскольку рассчитывается в виде суммарной величины для всех процессоров.

В то же время, существует весьма распространенное заблуждение, суть которого состоит в том, что отмеченный параллелизм, независимо от того, касается ли он использования нескольких процессов, как в случае grepMP, или же применения нескольких потоков, как в случае grepMT, способен приводить к повышению производительности лишь в случае SMP-систем. Выигрыш в производительности можно получить и при использовании нескольких дисков, а также при любом другом распараллеливании в системе хранения. Во всех подобных случаях операции ввода/вывода с участием нескольких файлов будут осуществляться в параллельном режиме.

Модель "хозяин/рабочий" и другие модели многопоточных приложений

Программа grepMT демонстрирует модель многопоточных приложений, носящую название модели "хозяин/рабочий" ("boss/worker"), а рис. 6.3, после замены в нем термина "процесс" на термин "поток", может служить графической иллюстрацией соответствующих отношений. Главный поток (основной поток в данном случае) поручает выполнение отдельных задач рабочим потокам. Каждый рабочий, поток получает файл, в котором она должна выполнить поиск, а полученные рабочим потоком результаты передаются главному потоку во временном файле.

Существуют многочисленные вариации этой модели, одной из которых является модель рабочей группы (work crew model), в которой рабочие потоки объединяют свои усилия для решения одной задачи, причем каждый отдельный поток выполняет свою небольшую часть работы. Модель рабочей группы используется в нашем следующем примере (рис. 7.2). Рабочие группы даже могут самостоятельно распределять работу между собой без получения каких-либо указаний со стороны главного потока. В многопоточных программах может быть применена практически любая из схем управления, разработанных для коллективов в человеческом обществе.

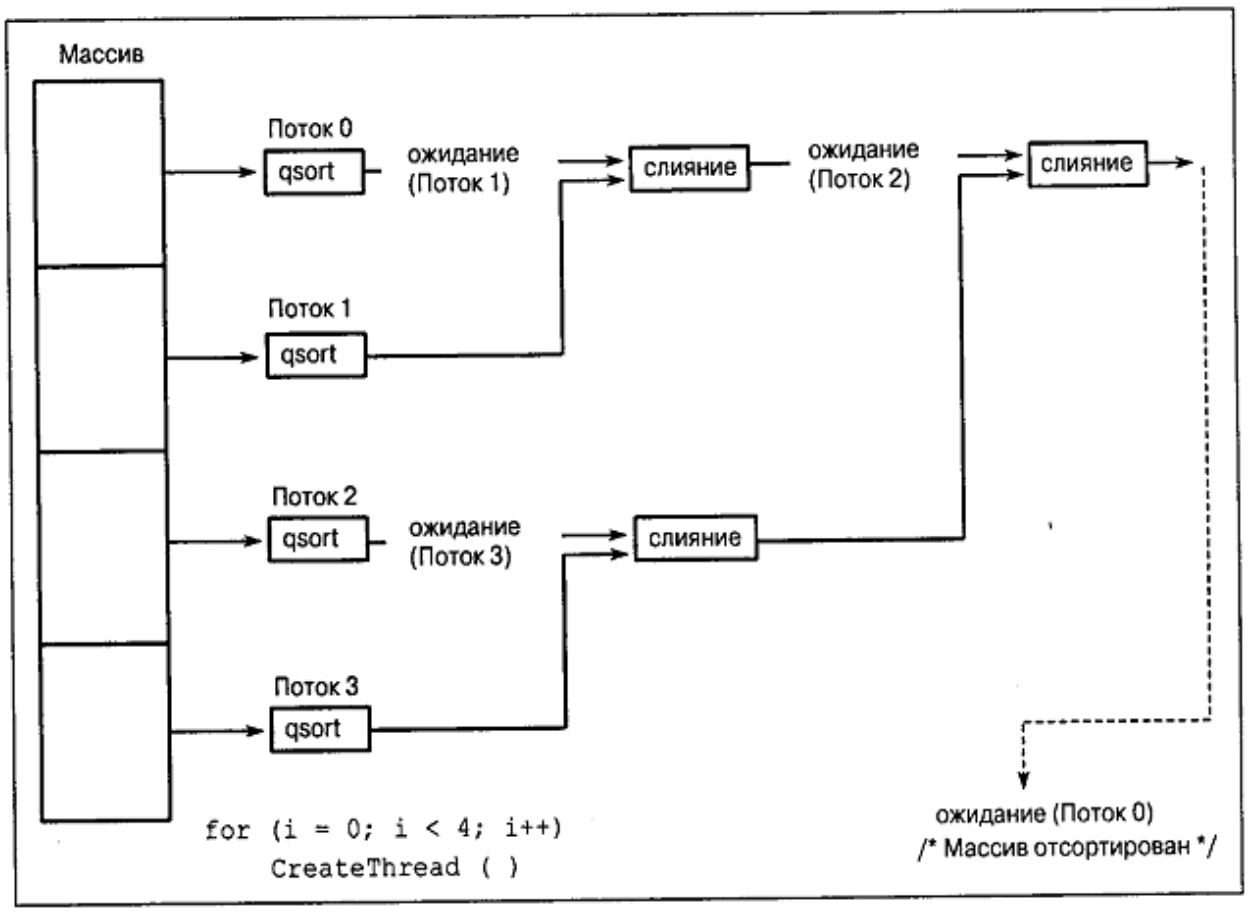

Рис. 7.2. Выполнение сортировки слиянием с использованием нескольких потоков

Двумя другими основными моделями являются модель *"клиент/сервер"* (client/server) (проиллюстрирована на рис. 7.1, а пример ее практической реализации рассматривается в главе 11) и *конвейерная модель* (pipeline model), в которой выполнение задания передается от одного потока к другому (пример многоступенчатого конвейера рассматривается в главе 10 и иллюстрируется на рис. 10.1).

При проектировании многопоточных систем эти модели обладают целым рядом преимуществ, к числу которых можно отнести следующие:

• Большинство проблем многопоточного программирования могут быть разрешены с использованием одной из стандартных моделей, облегчающих проектирование, разработку и отладку программ.

• Применение понятных и проверенных моделей не только позволяет избежать многих ошибок, которые легко допустить при написании многопоточных программ, но и способствует повышению производительности результирующих приложений.

• Эти модели естественным образом соответствуют структуре большинства обычных задач программирования.

• Программистам, сопровождающим программу, будет гораздо легче понять ее устройство, если она будет описана в документации на понятном языке.

• Находить неполадки в незнакомой программе гораздо легче, если ее можно анализировать в терминах моделей. Очень часто главную причину неполадок удается установить на основании видимых нарушений базовых принципов одной из моделей.

• Многие распространенные дефекты программ, например, нарушение условий состязаний задач и их блокирование, также можно описать с использованием простых моделей, к числу которых относятся эффективные методы использования объектов синхронизации, описанные в главах 9 и 10.

Эти классические модели потоков реализованы во многих ОС. В модели компонентных объектов (Component Object Model, COM), широко используемой во многих Windows-системах, применяется другая терминология, и хотя рассмотрение модели COM выходит за рамки данной книги, об этих моделях говорится в конце главы 11, где они сравниваются с другими примерами программ.

Пример: применение принципа "разделяй и властвуй" для решения задачи сортировки слиянием в SMP-системах

Этот пример демонстрирует возможности значительного повышения производительности за счет использования потоков, особенно в случае SMP-систем. Основная идея заключается в разбиении задачи на более мелкие составляющие, распределении выполнения подзадач между отдельными потоками и последующем объединении результатов для получения окончательного решения. Планировщик Windows автоматически назначит потокам отдельные процессоры, в результате чего задачи будут выполняться параллельно, снижая общее время выполнения приложения.

Эта стратегия, которую часто называют стратегией "разделяй и властвуй" (divide and conquer), или *моделью рабочей группы* (work crew model), оказалась весьма полезной и в качестве средства повышения производительности, и в качестве метода проектирования алгоритмов. Одним из примеров ее применения служит программа grepMT (программа 7.1), в которой для каждой файловой операции ввода/вывода и для поиска

шаблона создается отдельный поток. Как показано в приложении B, в случае SMP-систем производительность повышается, поскольку планировщик может распределять выполнение потоков между различными процессорами.

Далее мы рассмотрим другой пример, в котором задача сортировки содержимого файла разбивается на ряд подзадач, выполнение которых делегируется отдельным потокам.

Решение задачи сортировки слиянием (merge-sort), в которой сортируемый массив разбивается на несколько массивов меньшего размера, является классическим примером алгоритма, построенного на принципе "разделяй и властвуй". Каждый из массивов небольшого размера сортируется по отдельности, после чего отсортированные массивы попарно объединяются с образованием отсортированных массивов большего размера. Описанное слияние массивов попарно осуществляется вплоть до завершения всего процесса сортировки. В общем случае, сортировка слиянием начинается с массивов размерности 1, которые сами по себе не нуждаются в сортировке. В данном примере сортировка начинается с массивов большей размерности, чтобы на каждый процессор приходилось по одному массиву. Блок-схема используемого алгоритма показана на рис. 7.2.

Детали реализации представлены в программе 7.2. Число задач задается пользователем в командной строке. Временные показатели сортировки приведены в приложении B. В упражнении 7.9 вам предлагается изменить программу sortMT таким образом, чтобы она сначала определяла количество доступных процессоров, используя для этого функцию GetSystemInfo, а затем создавала по одному потоку для каждого процессора.

Заметьте, что эта программа эффективно выполняется в однопроцессорных системах, в которых имеется достаточно большой запас оперативной памяти, и обеспечивает значительное повышение производительности в SMP-системах. *Предостережение.* Представленный алгоритм будет корректно работать лишь при условии, что число записей в сортируемом файле нацело делится на число потоков, а число потоков выражается степенью 2. В упражнении 7.8 упомянутые ограничения снимаются.

Примечание

Изучая работу этой программы, постарайтесь отделить логику управления потоками от логики определения части массива, которую должна сортировать тот или иной поток. Обратите также внимание на использование функции qsort из библиотеки C, применение которой избавляет нас от необходимости самостоятельно разрабатывать эффективную функцию сортировки.

Программа 7.2. sortMT: сортировка слиянием с использованием нескольких потоков

```
/* Глава 7. SortMT.
```

Сортировка файлов с использованием нескольких потоков (рабочая группа).

sortMT [параметры] число_задач файл */

```
#include "EvryThng.h"
#define DATALEN 56 /* Данные: 56 байт; ключ: 8 байт. */
#define KEYLEN 8

typedef struct _RECORD {
 CHAR Key[KEYLEN];
 TCHAR Data[DATALEN];
} RECORD;

#define RECSIZE sizeof (RECORD)
typedef RECORD * LPRECORD;

typedef struct _THREADARG { /* Аргумент потока */
 DWORD iTh; /* Номер потока: 0, 1, 2, … */
 LPRECORD LowRec; /* Младшая часть указателя записи */
 LPRECORD HighRec; /* Старшая часть указателя записи */
} THREADARG, *PTHREADARG;

static int KeyCompare(LPCTSTR, LPCTSTR);
static DWORD WINAPI ThSort(PTHREADARG pThArg);
static DWORD nRec; /* Общее число записей, подлежащих сортировке. */
static HANDLE* ThreadHandle;

int _tmain(int argc, LPTSTR argv[]) {
 HANDLE hFile;
 LPRECORD pRecords = NULL;
 DWORD FsLow, nRead, LowRecNo, nRecTh, NPr, ThId, iTh;
 BOOL NoPrint;
 int iFF, iNP;
 PTHREADARG ThArg;
 LPTSTR StringEnd;
 iNP = Options(argc, argv, _T("n"), &NoPrint, NULL);
 iFF = iNP + 1;
 NPr = _ttoi(argv[iNP]); /* Количество потоков. */
 hFile = CreateFile(argv[iFF], GENERIC_READ | GENERIC_WRITE, 0,
NULL, OPEN_EXISTING, 0, NULL);
 FsLow = GetFileSize(hFile, NULL);
 nRec = FsLow / RECSIZE; /* Общее число записей. */
 nRecTh = nRec / NPr; /* Количество записей на один поток. */
```

```
/* Распределить память для аргументов потока и массива дескрипторов и выделить в памяти место для файла. Считать весь файл. */
ThArg = malloc(NPr * sizeof(THREADARG));
/* Аргументы потоков. */
ThreadHandle = malloc(NPr * sizeof(HANDLE));
pRecords = malloc(FsLow + sizeof(TCHAR));
ReadFile(hFile, pRecords, FsLow, &nRead, NULL);
CloseHandle(hFile);
LowRecNo = 0; /* Создать потоки, выполняющие сортировку. */
for (iTh = 0; iTh < NPr; iTh++) {
 ThArg[iTh].iTh = iTh;
 ThArg[iTh].LowRec = pRecords + LowRecNo;
 ThArg[iTh].HighRec = pRecords + (LowRecNo + nRecTh);
 LowRecNo += nRecTh;
  ThreadHandle[iTh] = (HANDLE)_beginthreadex (NULL, 0, ThSort, &ThArg[iTh], CREATE_SUSPENDED, &ThId);
}
for (iTh = 0; iTh < NPr; iTh++) /* Запустить все потоки сортировки. */
 ResumeThread(ThreadHandle [iTh]);
WaitForSingleObject(ThreadHandle[0], INFINITE);
for (iTh = 0; iTh < NPr; iTh++) CloseHandle(ThreadHandle [iTh]);
StringEnd = (LPTSTR)pRecords + FsLow;
*StringEnd = '\0';
if (!NoPrint) printf("\n%s", (LPCTSTR)pRecords);
free(pRecords);
free(ThArg);
free(ThreadHandle);
return 0;
} /* Конец tmain. */

static VOID MergeArrays(LPRECORD, LPRECORD);

DWORD WINAPI ThSort(PTHREADARG pThArg) {
 DWORD GrpSize = 2, RecsInGrp, MyNumber, TwoToI = 1;
 LPRECORD First;
 MyNumber = pThArg->iTh;
 First = pThArg->LowRec;
 RecsInGrp = pThArg->HighRec – First;
 qsort(First, RecsInGrp, RECSIZE, KeyCompare);
 while ((MyNumber % GrpSize) == 0 && RecsInGrp < nRec) {
 /* Объединить слиянием отсортированные массивы. */
 WaitForSingleObject(ThreadHandle[MyNumber + TwoToI], INFINITE);
 MergeArrays(First, First + RecsInGrp);
 RecsInGrp *= 2;
 GrpSize *= 2;
```

```
 TwoToI *= 2;
 }
 _endthreadex(0);
 return 0; /* Подавить вывод предупреждающих сообщений. */
}

static VOID MergeArrays(LPRECORD p1, LPRECORD p2) {
 DWORD iRec = 0, nRecs, i1 = 0, i2 = 0;
 LPRECORD pDest, p1Hold, pDestHold;
 nRecs = p2 – p1;
 pDest = pDestHold = malloc(2 * nRecs * RECSIZE);
 p1Hold = p1;
 while (i1 < nRecs && i2 < nRecs) {
  if (KeyCompare((LPCTSTR)p1, (LPCTSTR)p2) <= 0) {
   memcpy(pDest, p1, RECSIZE);
   i1++;
   p1++;
   pDest++;
  } else {
   memcpy(pDest, p2, RECSIZE);
   i2++;
   p2++;
   pDest++;
  }
 }
 if (i1 >= nRecs) memcpy(pDest, p2, RECSIZE * (nRecs – i2));
 else memcpy(pDest, p1, RECSIZE * (nRecs – i1));
 memcpy(p1Hold, pDestHold, 2 * nRecs * RECSIZE);
 free (pDestHold);
 return;
}
```

Производительность

В приложении B представлены результаты сортировки файлов большого размера, содержащих записи длиной 64 байта, для случаев использования одной, двух и четырех потоков. SMP-системы позволяют получать значительно лучшие результаты. Упомянутый принцип "разделяй и властвуй" обеспечивает нечто большее, чем просто стратегию проектирования алгоритмов; он также служит ключом к использованию потоков и SMP. Результаты для однопроцессорных систем могут быть различными в зависимости от остальных характеристик системы. В системах с ограниченным объемом памяти (то есть объема физической памяти не достаточно для того, чтобы наряду с ОС и другими активными процессами в ней уместился весь файл) использование нескольких потоков увеличивает

время сортировки, поскольку потоки состязаются между собой в захвате доступной физической памяти. С другой стороны, если памяти имеется достаточно, то многопоточный вариант может привести к повышению производительности и в случае однопроцессорных систем. Кроме того, как следует из приложения B, получаемые результаты существенно зависят от начального распределения данных.

Локальные области хранения потоков

Потокам могут требоваться собственные, независимо распределяемые и управляемые ими области памяти, защищенные от других потоков того же процесса. Одним из методов создания таких областей является вызов функции CreateThread (или _beginthreadex) с параметром lpvThreadParm, указывающим на структуру данных, уникальную для каждого потока. После этого поток может распределять память для дополнительных структур данных и получать доступ к ним через указатель lpvThreadParm. Эта методика используется в программе 7.1.

Кроме того, Windows предоставляет локальные области хранения потоков (Thread Local Storage, TLS), обеспечивающие каждый из потоков собственным массивом указателей. Организация TLS показана на рис. 7.3.

Индексы (строки) TLS первоначально не распределены, но в любой момент времени можно добавлять новые строки и освобождать существующие, причем минимально возможное число строк для любого процесса определяется значением TLS_MINIMUM_AVAILABLE (равным, по крайней мере, 64). Число столбцов может изменяться по мере создания новых потоков и завершения существующих.

Сначала мы рассмотрим управление индексами TLS. Логическим пространством для этого служит основной поток, но управлять индексами может любой поток.

Функция TlsAlloc возвращает распределенный индекс (> 0) или –1 (0xFFFFFFFF) в случае отсутствия доступных индексов.

DWORD TlsAlloc(VOID)
BOOL TlsFree(DWORD dwIndex)

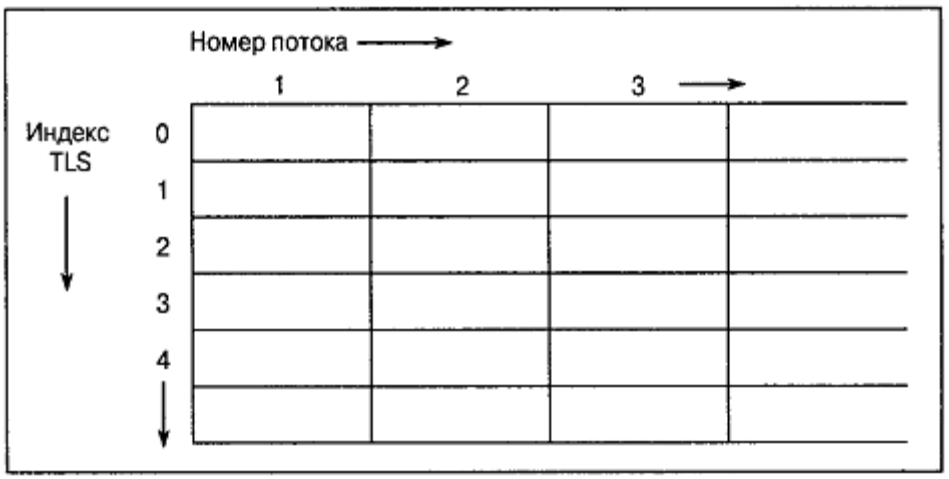

Рис. 7.3. Локальные области хранения потоков в контексте процесса

Каждый отдельный поток может выбирать и устанавливать значения (указатели типа void), связанные с ее областью памяти, используя индексы TLS.

Программист всегда должен убеждаться в том, что параметр индекса TLS является действительным, то есть что он был распределен с помощью функции TlsAlloc, но не был освобожден.

LPVOID TlsGetValue(DWORD dwTlsIndex)

BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue)

TLS предоставляют удобный механизм организации памяти, являющейся глобальной в контексте потока, но недоступной другим потокам. Обычные глобальные хранилища разделяются всеми потоками. Несмотря на то что ни один поток не может получить доступа к TLS другого потока, любой поток может уничтожить индекс TLS другого потока, вызвав функцию TlsFree, так что этой функцией следует пользоваться с осторожностью. TLS часто используются DLL в качестве замены глобальной памяти библиотеки; в результате этого каждый поток получает в свое распоряжение собственную глобальную память. Кроме того, TLS обеспечивают вызывающим программам удобный способ взаимодействия с функциями DLL, и именно этот способ применения TLS является наиболее распространенным. В качестве примера в главе 12 (программа 12.4) TLS используются для создания библиотеки DLL с многопоточной поддержкой; другим важным элементом этого решения являются уведомления DLL о присоединении/отсоединении потоков и процессов путем вызова функции DllMain (глава 5).

Приоритеты процессов и потоков и планирование выполнения

Ядро Windows всегда запускает тот из потоков, готовых к выполнению, который обладает наивысшим приоритетом. Поток не является готовым к выполнению, если он находится в состоянии ожидания, приостановлен или блокирован по той или иной причине.

Потоки получают приоритеты на базе классов приоритета своих процессов. Как обсуждалось в главе б, первоначально функцией CreateProcess устанавливаются четыре класса приоритета, каждый из которых имеет *базовый приоритет* (base priority):

- IDLE_PRIORITY_CLASS, базовый приоритет 4.
- NORMAL_PRIORITY_CLASS, базовый приоритет 9 или 7.
- HIGH_PRIORITY_CLASS, базовый приоритет 13.
- REALTIME_PRIORITY_CLASS, базовый приоритет 24.

Оба предельных класса используются редко, и в большинстве случаев можно обойтись нормальным (normal) классом. Windows NT (все версии) не является ОС реального времени (real-time), чего нельзя сказать, например, о Windows CE, и в случаях, аналогичных последнему, классом

REALTIME_PRIORITY_CLASS следует пользоваться с осторожностью, чтобы не допустить вытеснения других процессов. Нормальный базовый приоритет равен 9, если фокус ввода с клавиатуры находится в окне; в противном случае этот приоритет равен 7.

Один процесс может изменить или установить свой собственный приоритет или приоритет другого процесса, если это разрешено атрибутами защиты.

BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriority)
DWORD GetPriorityClass(HANDLE hProcess)

Приоритеты потоков устанавливаются относительно базового приоритета процесса, и во время создания потока его приоритет устанавливается равным приоритету процесса. Приоритеты потоков могут принимать значения в интервале ±2 относительно базового приоритета процесса. Результирующим пяти значениям приоритета присвоены следующие символические имена:

- THREAD_PRIORITY_LOWEST
- THREAD_PRIORITY_BELOW_NORMAL
- THREAD_PRIORITY_NORMAL
- THREAD_PRIORITY_HIGHEST

Для установки и выборки относительного приоритета потока следует использовать эти значения. Обратите внимание на использование целых чисел со знаком вместо чисел типа DWORD.

BOOL SetThreadPriority(HANDLE hThread, int nPriority)
int GetThreadPriority(HANDLE hThread)

Существуют два дополнительных значения приоритета потоков. Они являются абсолютными, а не относительными, и используются только в специальных случаях.

- THREAD_PRIORITY_IDLE имеет значение 1 (или 16 — для процессов, выполняющихся в режиме реального времени).
- THREAD_PRIORITY_TIME_CRITICAL имеет значение 15 (или 31 — для процессов, выполняющихся в режиме реального времени).

Приоритеты потоков автоматически изменяются при изменении приоритета процесса. Помимо этого, ОС может регулировать приоритеты потоков динамическим путем на основании поведения потоков. Вы можете активизировать или отключить это средство с помощью функции SetThreadPriorityBoost.

Предостережение относительно использования приоритетов потоков и процессов

Высокими приоритетами потоков и высокими классами приоритета процессов необходимо пользоваться с осторожностью. Следует решительно избегать использования приоритетов реального времени для обычных пользовательских процессов; эти приоритеты должны использоваться лишь в тех случаях, когда приложения действительно являются приложениями

реального времени. Нарушение этого правила чревато тем, что пользовательские потоки будут тормозить выполнение потоков операционной системы.

Кроме того, приводимые в последующих главах высказывания относительно корректности многопоточных программ справедливы лишь при условии соблюдения принципа *равноправия* (fairness) потоков. Равноправие потоков означает, что все они, в конечном счете, будут выполняться. Если не соблюдать этот принцип, то потоки с более низким приоритетом смогут удерживать ресурсы, необходимые потокам, имеющим более высокий приоритет. При описании недостатков планирования, осуществляемого с нарушением принципа равноправия, используют термины зависание потоков (thread starvation) и инверсия приоритетов (priority inversion).

Состояния потоков

На рис. 7.4, взятом из [9] (см. также [38], версию, обновленную Соломоном (Solomon) и Руссиновичем (Russinovych)), представлена схема планирования потоков и показаны их возможные состояния. Кроме того, этот рисунок иллюстрирует результаты работы программы. Такие диаграммы состояния являются общими для всех многозадачных ОС и помогают выяснить, каким образом планируется выполнение потоков и как они переходят из одного состояния в другое.

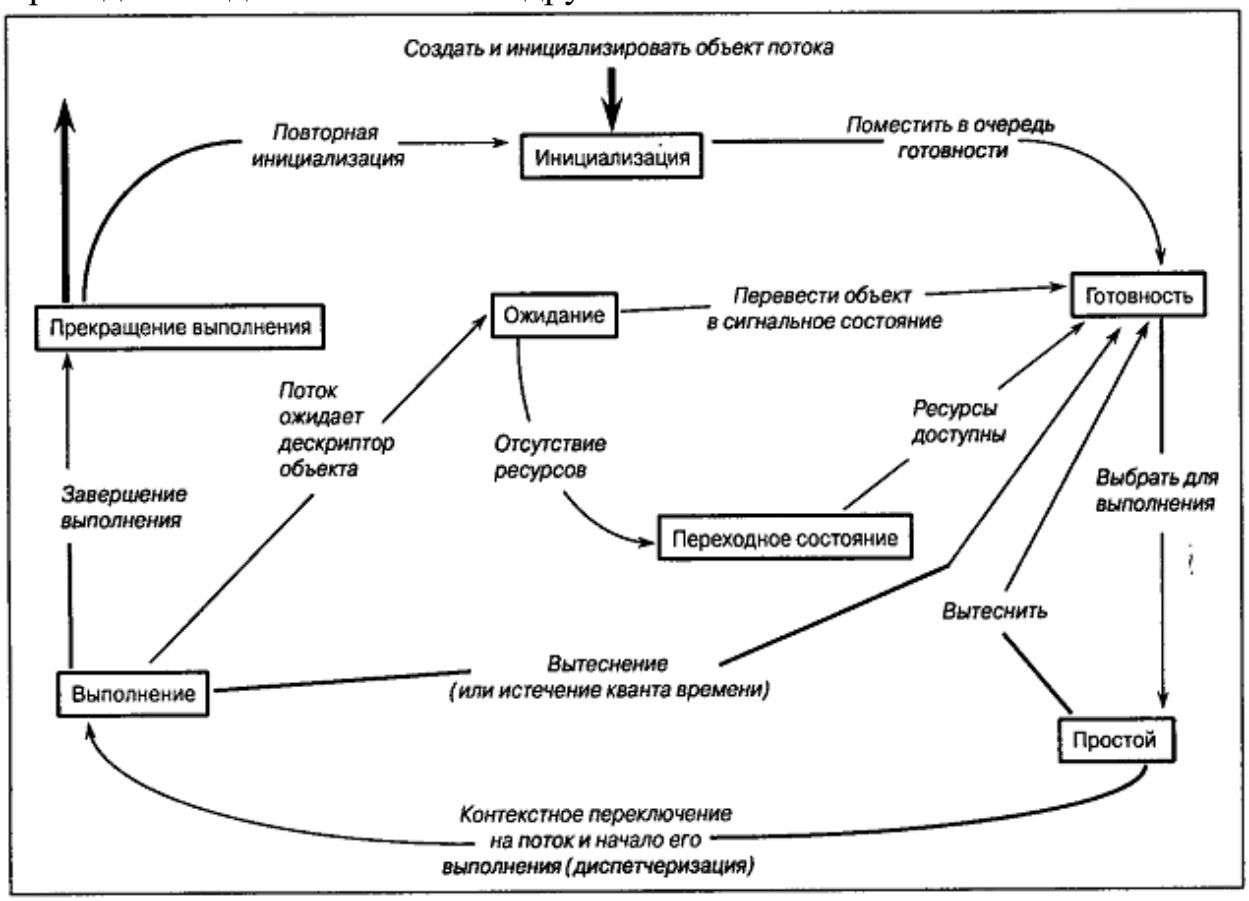

Рис. 7.4. Состояния потоков и переходы между состояниями (Источник: *Inside Windows NT*, Copyright © 1993, by Helen Custer. Copyright

Microsoft Press. Воспроизводится с разрешения Microsoft Press. Все права сохранены.)

Ниже приводится краткая сводка основных положений. Для получения более подробной информации по этому вопросу обратитесь в [38] или к руководству по ОС.

• Поток находится в *состоянии выполнения* (running state), если она фактически выполняется процессором. В SMP-системах в состоянии выполнения могут находиться одновременно несколько потоков.

• Планировщик переводит поток в *состояние ожидания* (wait state), если он выполняет функцию ожидания несигнализирующих объектов, например, потоков или процессов, или перехода в сигнальное состояние объектов синхронизации, о чем говорится в главе 8. Операции ввода/вывода также будут ожидать завершения передачи дисковых или иных данных, но ожидание может быть вызвано и другими многочисленными функциями. О потоках, находящихся в состоянии ожидания, нередко говорят как о *блокированных* (blocked) или *спящих* (sleeping).

• Поток находится в *состоянии готовности* (ready state), если она может выполняться. Планировщик в любой момент может перевести такой поток в состояние выполнения. Когда процессор станет доступным, планировщик запустит тот из потоков, находящихся в состоянии готовности, который обладает наивысшим приоритетом, а при наличии нескольких потоков с одинаковым высшим приоритетом запущен будет та, который пребывал в состоянии готовности дольше всех. При этом поток проходит через *состояние простоя* (standby state), или *резервное состояние.*

• Обычно, в соответствии с приведенным описанием, планировщик помещает поток, находящийся в состоянии готовности, на любой доступный процессор. Программист может указать *маску родства процессоров* (processor affinity mask) для потока (см. главу 9), предоставляя потоку процессоры, на которых он может выполняться. Используя этот способ, программист может распределять процессоры между потоками. Соответствующими функциями являются SetProcessorAffinityMask и GetProcessorAffinityMask. Функция SetThreadIdealProcessor позволяет указать предпочтительный процессор, подлежащий использованию планировщиком при первой же возможности.

• После истечения кванта времени, отведенного выполняющемуся потоку, планировщик без ожидания переводит его в состояние готовности. В результате выполнения функции Sleep(0) поток также будет переведен из состояния выполнения в состояние готовности.

• Планировщик переводит ожидающий поток в состояние готовности сразу же, как только соответствующие дескрипторы оказываются в сигнальном состоянии, хотя при этом поток фактически проходит через промежуточное *переходное состояние* (transition state). В подобных случаях принято говорить о том, что поток *пробуждается* (wakes).

• Не существует способа, позволяющего программе определить состояние другого потока (разумеется, если поток выполняется, то он находится в состоянии выполнения, и поэтому ему нет никакого смысла определять свое состояние). Даже если бы такой способ и существовал, то состояние потока может измениться еще до того, как опрашивающий поток успеет предпринять какие-либо действия в ответ на полученную информацию.

• Поток, независимо от его состояния, может быть приостановлен (suspended), и приостановленный поток не будет запущен, даже если он находится в состоянии готовности. В случае приостановки выполняющегося потока, независимо от того, по собственной ли инициативе или по инициативе потока, выполняющегося на другом процессоре, он переводится в состояние готовности.

• Поток переходит в *состояние завершения* (terminated state) тогда, когда его выполнение завершается, и остается в этом состоянии до тех пор, пока остается открытым хотя бы один из ее дескрипторов. Это позволяет другим потокам запрашивать состояние данного потока и его код завершения.

Возможные ловушки и распространенные ошибки

Существует ряд факторов, о которых следует всегда помнить при разработке многопоточных программ. Пренебрежение некоторыми базовыми принципами может привести к появлению серьезных дефектов в программе, и лучше заранее стремиться к тому, чтобы не допустить ошибок, чем впоследствии затрачивать время на тестирование и отладку программ.

Существенно то, что потоки выполняются в асинхронном режиме. По отношению к ним не действует никакая система упорядочения, если только вы не создали ее явно. Именно асинхронное поведение потоков делает их столь полезными, однако при отсутствии должного внимания можно столкнуться с серьезными трудностями.

Часть соответствующих рекомендаций представлена ниже, а остальные будут даваться по мере изложения материала в последующих главах.

• Не делайте никаких предположений относительно очередности выполнения родительских и дочерних потоков. Вполне возможно, что дочерний поток будет выполняться вплоть до своего завершения, прежде чем родительский поток вернется из функции CreateThread, или наоборот, дочерний поток может вообще не выполняться в течение длительного периода времени. В случае же SMP-систем возможно параллельное выполнение родительского и одного или нескольких дочерних потоков.

• Убедитесь в том, что до вызова функции CreateThread были завершены все действия по инициализации данных, необходимые для правильной работы дочернего потока, либо приостановите поток или же воспользуйтесь любой другой подходящей методикой. Несвоевременная

инициализация данных, требуемых дочерним потоком, может создать "условия состязаний" ("race conditions"), суть которых заключается в том, что родительский поток "состязается" с дочерним, чтобы инициализировать данные до того, как они начнут использоваться дочерним потоком.

• Проследите за тем, чтобы каждый отдельный поток имел собственную структуру данных, переданную ему через параметр функции потока. Не делайте никаких предположений относительно очередности завершения дочерних потоков (иначе можете столкнуться с другой разновидностью "проблемы состязаний").

• Выполнение любого потока может быть прервано в любой момент, и точно так же выполнение любого потока в любой момент может быть возобновлено.

• Не пользуйтесь приоритетами потоков в качестве замены явной синхронизации.

• Никогда не прибегайте к аргументации наподобие "вряд ли это может произойти" при анализе корректности программы. Если что-то *может* произойти, оно *обязательно* произойдет, причем тогда, когда вы меньше всего этого ожидаете.

• В еще большей степени, чем в случае однопоточных программ, справедливо утверждение о том, что, хотя тестирование и необходимо, но его одного еще не достаточно для проверки корректности многопоточной программы. Довольно часто программы способны успешно пройти через самые различные тесты, несмотря на наличие в них дефектов. Ничто не может заменить тщательно выполненных проектирования, реализации и анализа кода.

• Поведение многопоточных программ может заметно меняться в зависимости от быстродействия и количества процессоров, версии ОС и множества других факторов. Тестирование программы в самых различных системах помогает выявлению всевозможных дефектов, но предыдущее предостережение остается в силе.

• Убедитесь в том, что предусмотрели для потоков достаточно большой объем стека, хотя заданного по умолчанию стека размером 1 Мбайт в большинстве случаев вам должно хватить.

• Потоки следует использовать только тогда, когда это действительно необходимо. Таким образом, если по самой своей природе некоторые операции допускают параллелизм, то каждое такое действие может быть представлено потоком. В то же время, если операциям присуща очередность, то потоки лишь усложнят программу, а связанный с ними расход системных ресурсов может привести к снижению производительности.

• К счастью, чаще всего корректно работают самые простые программы и те, отличительной чертой которых является элегантность. При малейшей возможности избегайте усложнения программ.

Ожидание в течение конечного интервала времени

Наконец, рассмотрим функцию Sleep, позволяющую потоку отказаться от процессора и перейти из состояния выполнения в состояние ожидания, которое будет длиться в течение заданного промежутка времени. Например, выполнение задачи потоком может продолжаться в течение некоторого периода времени, после чего поток приостанавливается. По истечении периода ожидания планировщик вновь переводит поток в состояние готовности. Именно эта техника применена в одной из программ в главе 11 (программа 11.4).

VOID Sleep(DWORD dwMilliseconds)

Длительность интервала ожидания указывается в миллисекундах, и одним из ее возможных значений является INFINITE, что соответствует бесконечному периоду ожидания, при котором выполнение приостанавливается на неопределенное время. Значению 0 соответствует отказ потока от оставшейся части отведенного ей временного промежутка; в этом случае ядро переводит поток из состояния выполнения в состояние готовности, как показано на рис. 7.4.

Функция **SwitchToThread** предоставляет потоку еще один способ уступить процессор другому потоку из числа тех, которые находятся в состоянии готовности, если таковые имеются.

UNIX-функция sleep аналогична функции Sleep, но длительность периода ожидания измеряется в секундах. Чтобы получить миллисекундное разрешение, используйте функции select или poll без дескрипторов файлов.

Облегченные потоки

Примечание

Облегченные потоки относятся к специальной тематике. Ознакомьтесь с комментарием, включенным в конце первого абзаца приведенного ниже списка, и решите для себя, стоит ли вам читать данный раздел.

Облегченные потоки (fibers), как говорит само их название, являются элементами потока. Точнее, облегченный поток — это единица выполнения в контексте потока, планируемая приложением, а не ядром. В потоке могут быть запланированы несколько облегченных потоков, и облегченные потоки сами определяют, какой из них должен выполняться следующим. Облегченные потоки имеют независимые стеки, но во всем остальном выполняются исключительно в контексте потока, имея, например, доступ к TLS потока и любому мьютексу[27], владельцем которого является данный поток. Более того, вся работа с облегченными потоками осуществляется вне ядра исключительно в пользовательском пространстве. Между потоками и облегченными потоками существуют многочисленные отличия.

Облегченные потоки могут использоваться в нескольких целях.

• Следует отметить тот немаловажный факт, что во многих приложениях, особенно в приложениях UNIX, использующих патентованные реализации потоков, которые в настоящее время можно, как правило, считать устаревшими, предусмотрено планирование собственных потоков. Использование облегченных потоков упрощает перенос таких приложений в среду Windows. *Поскольку для большинства читателей этот вопрос не является актуальным, они, вероятно, предпочтут пропустить данный раздел.*

• Потоки вовсе не обязательно должны блокироваться в ожидании блокировки файла, мьютекса, именованного входного канала или иных ресурсов. Вместо этого один облегченный поток может опрашивать ресурсы и, если эти ресурсы остаются недоступными, передавать управление другому указанному облегченному потоку.

• Облегченные потоки действуют в контексте потока и имеют доступ к ресурсам потока и процесса. В отличии от потоков вытесняющее планирование к облегченным потокам не применяется. В действительности планировщику Windows об облегченных потоках ничего не известно; управление такими потоками осуществляется из DLL облегченных потоков исключительно в пользовательском пространстве.

• Облегченные потоки позволяют реализовать механизм *сопрограмм* (co-routines), посредством которого приложение может переключаться между несколькими взаимосвязанными задачами. Добиться этого с помощью потоков невозможно, поскольку в распоряжении программиста нет средств, обеспечивающих непосредственное управление очередностью выполнения потоков.

• Основные разработчики программного обеспечения, использующие облегченные потоки, представляют этот элемент как фактор повышения производительности. Так, в приложении Oracle Database 10g предусмотрена возможность переключения в "режим облегченных потоков" ("fiber mode") (см. http://download.oracle.com/owsf_2003/40171_colello.ppt; там же находится описание многопоточной модели).

API облегченных потоков представлен шестью функциями. Порядок их использования описывается ниже и иллюстрируется рис. 7.5.

1. Прежде всего, поток должен сделать возможным выполнение облегченного потока, вызвав функцию ConvertThreadToFiber. В результате этого поток становится облегченным потоком, который может рассматриваться в качестве *основного* (primary). Вызов упомянутой функции обеспечивает получение указателя на данные облегченного потока, который может быть использован во многом так же, как аргумент потока использовался для создания специфических для этого потока данных.

2. Основной или другие облегченные потоки создают дополнительные облегченные потоки с помощью функции CreateFiber. Каждый облегченный поток характеризуется начальным адресом, размером стека и параметром. Каждый новый облегченный поток идентифицируется с помощью адреса, а не дескриптора.

3. Отдельный облегченный поток может получить свои данные, назначенные ему функцией CreateFiber, обратившись к функции GetFiberData.

4. Аналогично, облегченный поток может идентифицировать себя при помощи функции GetCurrentFiber.

5. Выполняющийся облегченный поток может уступить управление другому облегченному потоку, указав его адрес в вызове функции SwitchToFiber. Облегченные потоки должны явно указывать очередной облегченный поток, который должен выполняться в контексте данного потока.

6. Функция DeleteFiber уничтожает существующий облегченный поток и все относящиеся к нему данные.

7. В Windows XP (NT 5.1) наряду с локальными областями хранения облегченных потоков введены новые функции, такие, например, как ConvertFiberToThread (которая освобождает ресурсы, созданные функцией ConvertThreadToFiber).

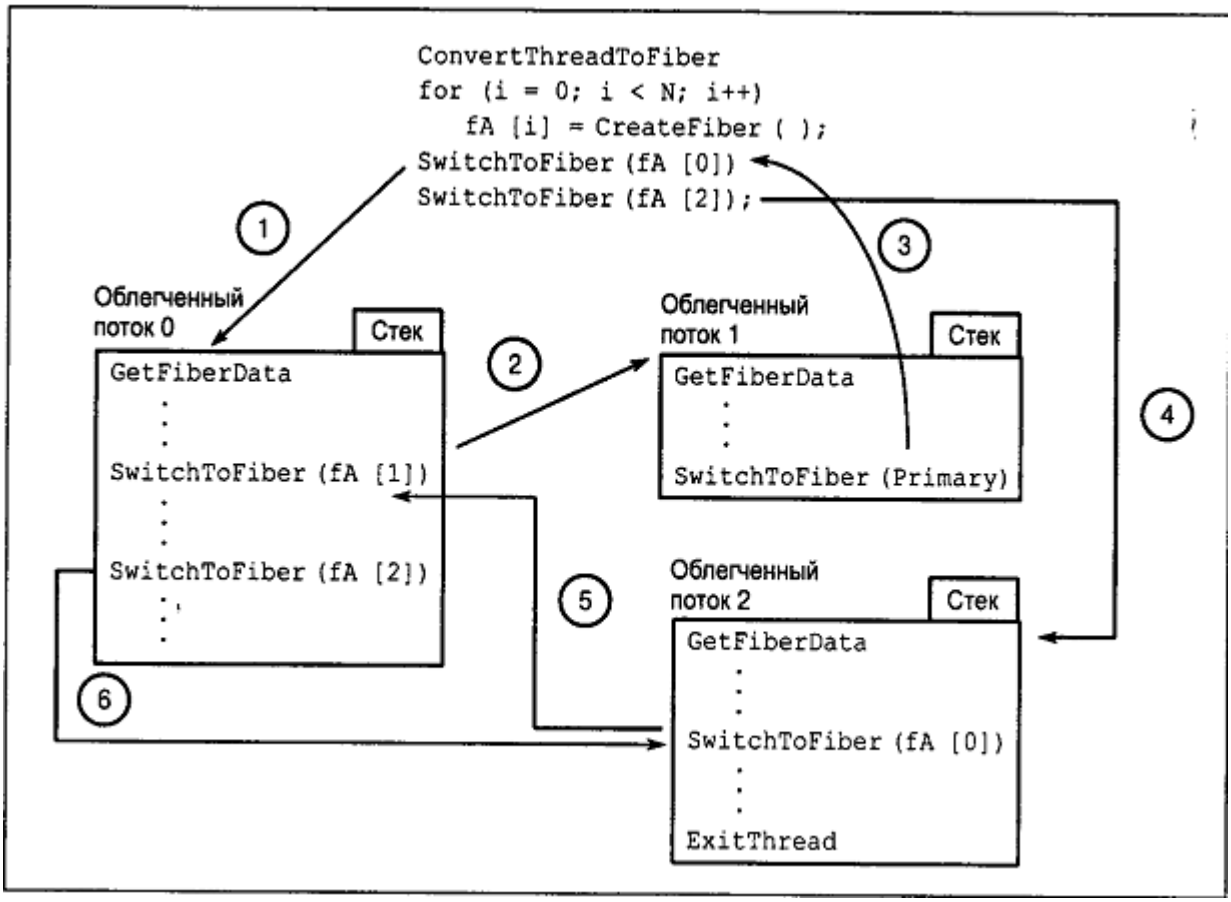

Рис. 7.5. Передача управления между облегченными потоками внутри потока

Схема взаимодействия между облегченными потоками в контексте потока представлена на рис. 7.5. Этот пример иллюстрирует два способа вытеснения одного потока другим.

• **Подчиненное планирование (master-slave scheduling).** Только один, главный (master) облегченный поток, в данном случае — основной, принимает решения относительно того, какой облегченный поток должен выполняться, и этот облегченный поток всегда уступает управление главному облегченный потоке. На рис. 7.5 главным является облегченный поток 1.

• **Равноправное планирование (peer-to-peer scheduling).** Облегченный поток сам определяет, какой из других облегченных потоков должен выполняться следующим. Определение очередного облегченного потока может базироваться на таких стратегиях, как круговое планирование (round-robin scheduling), приоритетное планирование на основании схемы приоритетов и тому подобное. По принципу равноправного планирования реализуются сопрограммы. На рис. 7.5 такого типа передача управления осуществляется между облегченными потоками 0 и 2.

Резюме

Windows поддерживает потоки, которые планируются независимо друг от друга, но разделяют адресное пространство и ресурсы одного и того же процесса. Потоки дают программисту возможность упростить программу и использовать параллелизм выполнения задач для повышения производительности приложения. Потоки могут обеспечивать выигрыш в производительности даже в однопроцессорных системах.

В следующих главах

Рассмотрение темы синхронизации, которое начинается в главе 8 с описания и сравнительного анализа объектов синхронизации Windows, продолжается в главах 9 и 10 обсуждением более сложных вопросов синхронизации с привлечением многочисленных примеров. В главе 11 реализуется сервер с многопоточной поддержкой; он показан на рис. 7.1.

Дополнительная литература

Windows

Книга [1] полностью посвящена потокам Win32. Также заслуживают внимания книги [26] и [7]. В то же время во многих из этих и других книг не учтены новшества, появившиеся в Windows 2000, XP и Server 2003.

UNIX и Pthreads

В книге [40] применение потоков в UNIX не рассматривается, однако для изучения этой темы можно порекомендовать книгу [6]. В этой книге даются многочисленные рекомендации, касающиеся проектирования и реализации многопоточных программ. Приведенная в ней информация

применима в равной степени как к потокам Pthreads, так и к потокам Windows, и многие примеры без труда переносятся в Windows. В ней также хорошо изложены модели "хозяин/рабочий", "клиент/сервер" и конвейерная модель, и представление Бутенхофа (Butenhof) было положено в основу описания указанных моделей в данной главе.

Упражнения

7.1. Реализуйте набор функций, позволяющий приостанавливать и возобновлять выполнение потоков, и, кроме того, получать значение счетчика приостановок потоков.

7.2. Сравните производительность программ параллельного поиска, одна из которых использует потоки (программа 7.1, GrepMT), а другая — процессы (программа 6.1, GrepMP). Сравните полученные результаты с теми, которые приведены в приложении В.

7.3. Проведите дополнительное исследование производительности программы GrepMT для случаев, когда файлы находятся на различных дисках или являются сетевыми файлами. Определите также выигрыш в производительности, если таковой будет наблюдаться, в случае SMP-систем.

7.4. Измените программу 7.1, GrepMT, таким образом, чтобы она выводила результаты в той очередности, в какой файлы указаны в командной строке. Сказываются ли эти изменения каким-либо образом на показателях производительности?

7.5. Дополнительно усовершенствуйте программу 7.1, GrepMT, таким образом, чтобы она выводила время, потребовавшееся для выполнения каждого из потоков. Для этого вам понадобится функция GetThreadTimes, аналогичная функции GetProcessTimes, которая была использована в главе 6. Указанное усовершенствование сможет работать лишь в Windows NT4 и более поздних версиях.

7.6. На Web-сайте книги находится многопоточная программа подсчета слов, wcMT.c, структура которой аналогична структуре программы grepMT.c. Там же находится версия этой программы, wcMTx, в которую были намеренно введены некоторые дефекты. Попытайтесь найти и устранить указанные дефекты, в том числе и синтаксические ошибки, не сверяясь с корректным решением. Кроме того, создайте тестовые примеры, иллюстрирующие эти дефекты, и выполните эксперименты по определению производительно сти, аналогичные тем, которые предлагались для программы grepMT. На Web-сайте находится также однопоточная версия упомянутой программы, wcST.c, которую можно использовать для того, чтобы определить, обеспечивают ли потоки выигрыш в производительности по сравнению с последовательной обработкой.

7.7. На Web-сайте находится программа grepMTx.c, в которой имеются дефекты, связанные с нарушением базовых правил, соблюдение которых необходимо для безопасного выполнения нескольких потоков. Опишите, к чему это приводит, а также найдите и устраните ошибки.

7.8. Для правильного выполнения программы sortMT требуется, чтобы количество записей в массиве нацело делилось на количество потоков, а количество потоков равнялось степени 2. Устраните эти ограничения.

7.9. Усовершенствуйте программу sortMT таким образом, чтобы в случаях, когда количество потоков, заданное в командной строке, равно 0, программа определяла количество процессоров, установленных в локальной системе, с помощью функции GetSystemInfo. Задавая количество потоков равным различным кратным количества процессоров (используя коэффициенты кратности 1, 2, 4 и так далее), определите, изменяется ли при этом производительность.

7.10. Видоизмените программу sortMT таким образом, чтобы рабочие потоки не приостанавливались при их создании. Сказываются ли, и если да, то каким именно образом, возникающие при этом нежелательные условия состязаний на работе программы?

7.11. В программе sortMT, еще до того, как создаются потоки, выполняющие сортировку, осуществляется считывание всего файла основным потоком. Видоизмените программу таким образом, чтобы каждый поток самостоятельно считывал необходимую часть файла.

7.12. Видоизмените одну из приведенных в данной главе программ (grepMT или sortMT) таким образом, чтобы специфическая для потоков информация частично или полностью передавалась через TLS, а не через структуры данных.

7.13. Наблюдается ли выигрыш в производительности в случае предоставления некоторым потокам в программе sortMT более высокого приоритета по сравнению со всеми остальными? Например, может оказаться выгодным предоставить таким потокам, как поток 3 на рис. 7.2, которая занята лишь сортировкой без слияния, более высокий приоритет, чем всем остальным. Объясните результаты.

7.14. В программе sortMT все потоки создаются в приостановленном состоянии с той целью, чтобы избежать создания условий состязаний. Видоизмените эту программу таким образом, чтобы потоки создавались в обратном порядке и в состоянии выполнения. Продолжают ли после этого оставаться какие-либо предпосылки для существования условий состязаний? Сравните показатели производительности измененной и исходной версий программы.

7.15. Алгоритм быстрой сортировки (Quicksort), обычно используемый функцией qsort библиотеки C, как правило, характеризуется высоким быстродействием, но в некоторых случаях может замедляться. В большинстве учебников по алгоритмам рассматривается его версия, которая работает быстрее всего в тех случаях, когда массив отсортирован в обратном порядке, и медленнее всего, когда массив является уже отсортированным. Однако реализация этого алгоритма в библиотеке Microsoft C ведет себя иначе. Определите из кода библиотечной программы, какого типа последовательности элементов массива будут приводить к наилучшим и наихудшим результатам, и исследуйте показатели производительности

программы sortMT для этих двух крайних случаев. Как влияет на результаты увеличение или уменьшение количества потоков? *Примечание.* Исходный код библиотеки C мог быть установлен в подкаталоге CRT основного каталога Visual Studio на вашей машине. Ищите функцию qsort.c. Кроме того, вы можете попытаться отыскать эту функцию на установочном компакт-диске.

7.16. На Web-сайте содержится программа sortMTx, в которую были намеренно внесены некоторые дефекты. Продемонстрируйте наличие дефектов на тестовых примерах, а затем объясните и устраните указанные дефекты, не сверяясь с корректными решениями. *Предостережение.* Версии программ, в которых присутствуют дефекты, могут содержать как синтаксические ошибки, так и ошибки в логике организации выполнения потоков.

7.17. Прочитайте статью Джейсона Кларка (Jason Clark) "Waiting for More than 64 Objects" ("Ожидание более чем 64 объектов"), опубликованную в октябрьском номере журнала *Windows Developer's Journal* за 1997 год. Примените описанное в ней решение к программе grepMT. Найти старые выпуски журналов иногда бывает очень трудно, поэтому воспользуйтесь каким-нибудь поисковым механизмом для поиска сразу по нескольким ключевым словам. Мною для этого поиска была использована фраза "wait for multiple objects more than 64", хотя другие варианты могут оказаться более эффективными.