Синхронизация потоков

Потоки могут упрощать проектирование и реализацию программ и повышать их производительность, но их использование требует принятия мер по защите разделяемых ресурсов от попыток их изменения одновременно несколькими потоками, а также создания таких условий, при которых потоки выполняются лишь в ответ на запрос или тогда, когда это является необходимым. В настоящей главе представлены способы решения этих задач с помощью объектов синхронизации Windows — критических участков кода, мьютексов, семафоров и событий, а также описаны некоторые из проблем, например, взаимоблокировка потоков и возникновение состязаний между ними, которые могут наблюдаться в результате неправильного использования потоков. Объекты синхронизации могут применяться для синхронизации потоков, принадлежащих как одному и тому же, так и различным процессам.

Примеры иллюстрируют объекты синхронизации, а также создают почву для обсуждения как положительных, так и отрицательных аспектов применения тех или иных методов синхронизации на производительность. В последующих главах демонстрируется использование синхронизации для решения дополнительных задач программирования и повышения производительности программ, а также рассказывается о возможных ловушках и применении более развитых средств.

Синхронизация потоков является одной из важнейших и интереснейших тем и играет существенную роль почти в любом многопоточном приложении.

Необходимость в синхронизации потоков

В главе 7 были продемонстрированы методы создания рабочих потоков и управления ими в условиях, когда каждый рабочий поток обращался к собственным ресурсам. В приведенных в главе 7 примерах каждый поток обрабатывает отдельный файл или отдельную область памяти, но даже и в этом случае возникает необходимость в простейшей синхронизации во время создания и завершения потоков. Так, в программе grepMT все рабочие потоки выполняются независимо друг от друга, но главный поток должен ожидать завершения рабочих потоков, прежде чем вывести сгенерированные ими результаты. Заметьте, что главный поток разделяет общую память с рабочими потоками, но структура программы гарантирует, что главный поток не получит доступа к памяти до тех пор, пока рабочий поток не завершит своего выполнения.

Программа sortMT несколько сложнее, поскольку рабочие потоки должны синхронизировать свое выполнение, ожидая завершения смежных потоков, и не могут быть запущены до тех пор, пока главный поток не создаст все рабочие потоки. Как и в случае программы grepMT, синхронизация достигается за счет ожидания завершения одного или нескольких потоков.

Однако во многих случаях требуется, чтобы выполнение двух и более потоков могло координироваться на протяжении всего времени жизни

каждой из них. Например, несколько потоков могут обращаться к одной и той же переменной или набору переменных, и тогда возникает вопрос о взаимоисключающем доступе. В других случаях поток не может продолжать выполнение до тех пор, пока другой поток не достигнет определенного этапа выполнения. Каким образом программист может получить уверенность в том, что, например, два или более потоков не попытаются одновременно изменить данные, хранящиеся в глобальной памяти, такие, например, как статистические данные о производительности? Как, далее, программист может добиться того, чтобы поток не предпринимал попыток удаления элемента из очереди, если очередь не содержит хотя бы одного элемента?

Несколько примеров иллюстрируют ситуации, которые могут приводить к нарушению условий безопасного выполнения нескольких потоков. (Код считается безопасным в этом смысле, если он может выполняться одновременно несколькими потоками без каких-либо нежелательных последствий.) Условия безопасного выполнения потоков обсуждаются далее в этой и последующих главах.

На рис. 8.1 показано, что может случиться, когда два не синхронизированных потока разделяют общий ресурс, например ячейку памяти. Оба потока увеличивают значение переменной N на единицу, но в силу специфики очередности, в которой могут выполняться потоки, окончательное значение N равно 5, тогда как правильным значением является 6. Заметьте, что представленный здесь частный результат не обладает ни повторяемостью, ни предсказуемостью; другая очередность выполнения потоков могла бы привести к правильному результату. В SMP-системах эта проблема еще более усугубляется.

Критические участки кода

Инкрементирование N при помощи единственного оператора, например, в виде N++, не улучшает ситуацию, поскольку компилятор сгенерирует последовательность из одной или более машинных инструкций, которые вовсе не обязательно должны выполняться *атомарно* (atomically), то есть как одна неделимая единица выполнения.

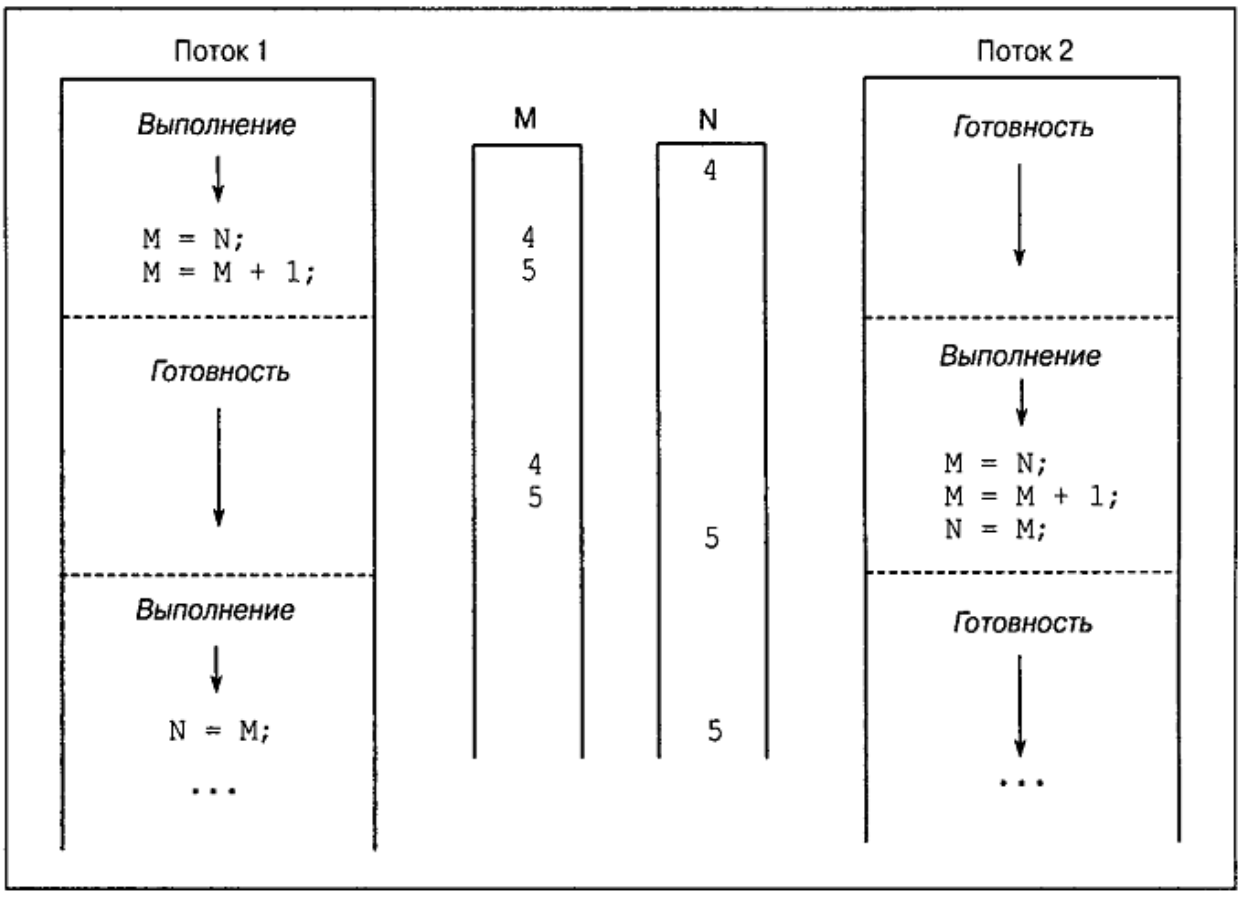

Рис. 8.1. Разделение общей памяти не синхронизированными потоками

Основная проблема состоит в том, что имеется *критический участок кода* ((critical section), (в данном примере — код, который увеличивает N на 1)), характеризующийся тем, что если один из потоков приступил к его выполнению, то никакой другой поток не должен входить в данный код до тех пор, пока его не покинет первый поток. Проблему критических участков кода можно считать разновидностью проблемы состязаний, поскольку первый поток "состязается" со вторым потоком в том, чтобы завершить выполнения критического участка еще до того, как его начнет выполнять любой другой поток. Таким образом, мы должны так синхронизировать выполнение потоков, чтобы можно было гарантировать, что в каждый момент времени код будет выполняться только одним потоком.

Неудачные пути решения проблемы критических участков кода

К аналогичным непредсказуемым результатам будет приводить и код, в котором предпринимается попытка защитить участок инкрементирования переменной путем опроса состояния флага.

```
while (Flag) Sleep (1000);
Flag = TRUE;
N++;
Flag = FALSE;
```

Даже в этом случае поток может быть вытеснен в процессе выполнения программы от момента тестирования значения флага до

момента, когда его значение будет установлено равным TRUE; критический участок кода образуют два оператора, которые не защищены должным образом от параллельного доступа к ним двух и более потоков.

Другая разновидность попытки решения проблемы синхронизации выполнения потоками критического участка кода могла бы состоять в том, чтобы предоставить каждому потоку собственный экземпляр переменной N, например, так, как показано ниже:

```
DWORD WINAPI ThFunc(TH_ARGS pArgs) {
 volatile  DWORD N;
 … N++; …
}
```

Однако такой подход ничем не лучше предыдущего, поскольку каждый поток имеет собственный экземпляр переменной в своем стеке, но может, например, требоваться, чтобы N представляло суммарное число действующих потоков. В то же время, этот тип решения необходим в тех случаях, когда каждый поток должен иметь собственный, независимый от других потоков экземпляр переменной. Эта методика часто встречается в наших примерах.

Заметьте, что проблемы подобного рода не ограничиваются случаем нескольких потоков одного процесса. С этими проблемами приходится сталкиваться также в случаях, когда два процесса разделяют общую память или изменяют один и тот же файл.

Класс памяти volatile

Даже если решить проблему синхронизации, все равно остается еще один скрытый дефект. Оптимизирующие компиляторы могут оставлять значение N в регистре, а не заносить его обратно в ячейку памяти, соответствующую переменной N. Попытка решения этой проблемы путем переустановки переключателей опций компилятора окажет отрицательное воздействие на скорость выполнения остальных участков программы. Правильное решение состоит в том, чтобы использовать определенный в стандарте ANSI C спецификатор памяти volatile, который гарантирует, что после изменения значения переменной оно будет сохраняться в памяти, а при необходимости будет всегда извлекаться из памяти. Ключевое слово volatile сообщает компилятору, что значение переменной может быть в любой момент изменено.

Функции взаимоблокировки

Если все, что требуется — это увеличение, уменьшение или обмен значениями переменных, как в нашем первом простом примере, то *функций взаимоблокировки* (interlocked functions) вам будет вполне достаточно. Функции взаимоблокировки проще в использовании, обеспечивают более высокое быстродействие по сравнению с другими возможными методами и не приводят к блокированию потоков. Двумя членами этого семейства функций, которые представляют для нас интерес, являются функции InterlockedIncrement и InterlockedDecrement. Обе функции применяются по отношению к 32-битовым целым числам со знаком.

Эти функции имеют ограниченную область применимости, но будут использоваться нами при любой удобной возможности.

Задача инкрементирования N, представленная на рис. 8.1, может быть реализована посредством единственной строки кода:

InterlockedIncrement(&N);

N — это целое число типа long со знаком, и функция возвращает его новое значение, несмотря на то что другой поток мог изменить значение N еще до того, как поток, вызвавший функцию InterlockedIncrement, успеет воспользоваться возвращенным значением.

Следует, однако, проявлять осторожность и, например, не вызывать эту функцию два раза подряд, если, значение переменной должно быть увеличено на 2, поскольку поток может быть вытеснен в промежутке между двумя вызовами функции. Вместо этого лучше воспользоваться функцией InterlockedExchangeAdd, описание которой приводится далее в настоящей главе.

Локальная и глобальная память

Суть другого требования, предъявляемого к корректному многопоточному коду, состоит в том, что глобальная память не должна использоваться для локальных целей. Так, применение функции ThFunc, приводившейся ранее в качестве примера, будет необходимым и уместным в тех случаях, когда поток должен располагать собственным экземпляром N. N может быть использовано для хранения временных результатов или размещения аргумента функции. Если же N размещается в глобальной памяти, то все процессы будут разделять единственный экземпляр N, что может стать причиной некорректного поведения программы, как бы тщательно вы ни планировали синхронизацию доступа к этой переменной. Ниже приводится пример подобного некорректного использования N. N должно быть локальной переменной, размещаемой в стеке функции потока.

```
DWORD N;
DWORD WINAPI ThFunc (TH_ARGS pArgs) {
 …
 N = 2 * pArgs->Count; …
}
```

Резюме: безопасный многопоточный код

Прежде чем мы приступим к рассмотрению объектов синхронизации, ознакомьтесь с пятью начальными рекомендациями, соблюдение которых будет гарантировать корректное выполнение программ в многопоточной среде.

1. Переменные, являющиеся локальными по отношению к потоку, не должны быть статическими, и их следует размещать в стеке потока или же в структуре данных или TLS, непосредственный доступ к которым имеет только отдельный поток.

2. В тех случаях, когда функцию могут вызывать несколько потоков, а какой-либо специфический для состояния потока параметр, например счетчик, должен сохранять свое значение в течение промежутков времени, отделяющих один вызов функции от другого, значение параметра состояния должно храниться в TLS или в структуре данных, выделенной специально для этого потока, например, в структуре данных, передаваемой потоку при его создании. Использовать стек для сохранения постоянно хранимых (persistent) значений не следует. Применение необходимой методики при построении безопасных многопоточных DLL иллюстрируют программы 12.4 и 12.5.

3. Старайтесь не создавать предпосылок для формирования условий состязаний наподобие тех, которые возникли бы в программе 7.2 (sortMT), если бы потоки не создавались в приостановленном состоянии. Если предполагается, что в определенной точке программы должно выполняться некоторое условие, используйте ожидание объекта синхронизации для гарантии того, что, например, дескриптор всегда будет ссылаться на существующий поток.

4. Вообще говоря, потоки не должны изменять окружение процесса, поскольку это окажет воздействие на все потоки. Таким образом, поток не должен определять дескрипторы стандартного ввода и вывода или изменять переменные окружения. Это не касается только основного потока, который может вносить такие изменения до создания других потоков.

5. Переменные, разделяемые всеми потоками, должны быть статическими или храниться в глобальной памяти, объявленной с использованием спецификатора volatile, а также должны быть защищены с использованием описанных ниже механизмов синхронизации.

Объекты синхронизации обсуждаются в следующем разделе. Приведенных в нем объяснений вам будет достаточно для того, чтобы разработать простой пример системы "производитель/потребитель" (producer/consumer).

Объекты синхронизации потоков

До сих пор нами были обсуждены только два механизма, обеспечивающие синхронизацию процессов и потоков друг с другом:

1. Поток, выполняющийся в контексте одного процесса, может дожидаться завершения другого процесса с использованием функции ExitProcess путем применения к дескриптору процесса функций ожидания WaitForSingleObject или WaitForMultipleObject. Тем же способом поток может организовать ожидание завершения (с помощью функции ExitThread или выполнения оператора return) другого потока.

2. Блокировки файлов, предназначенные для частного случая синхронизации доступа к файлам.

Windows предоставляет четыре других объекта, предназначенных для синхронизации потоков и процессов. Три из них — мьютексы, семафоры и события — являются объектами ядра, имеющими дескрипторы. События

используются также для других целей, например, для асинхронного ввода/вывода.

Мы начнем обсуждение с четвертого объекта, а именно, объекта критического участка кода CRITICAL_SECTION. В силу своей простоты и предоставляемых ими преимуществ в отношении производительности объекты критических участков кода являются предпочтительным механизмом, если их возможностей достаточно для того, чтобы удовлетворить требования программиста.

В то же время, при этом возникают некоторые проблемы, связанные с производительностью, о чем говорится в главе 9.

Предостережение

Неправильное применение объектов критических участков кода порождает определенные риски. Эти риски, такие, например, как риск блокировки, описываются в этой и последующих главах наряду с изложением методик, предназначенных для разработки надежного кода. Однако, прежде всего мы приведем некоторые примеры синхронизации в реалистических ситуациях.

Рассмотрение двух других объектов синхронизации — таймеров ожидания и портов завершения ввода/вывода — отложено. Эти типы объектов требуют использования методик асинхронного ввода/вывода Windows, которые описываются в указанной главе.

Объекты критических участков кода

Как уже упоминалось ранее, объект критического участка кода — это участок программного кода, который каждый раз должен выполняться только одним потоком; параллельное выполнение этого участка несколькими потоками может приводить к непредсказуемым или неверным результатам.

В качестве простого механизма реализации и применения на практике концепции критических участков кода Windows предоставляет объект CRITICAL_SECTION.

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

Для инициализации и удаления переменной типа CRITICAL_SECTION используются, соответственно, функции InitializeCriticalSection и DeleteCriticalSection:

VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection)

Функция EnterCriticalSection блокирует поток, если на данном критическом участке кода присутствует другой поток. Ожидающий поток разблокируется после того, как другой поток выполнит функцию LeaveCriticalSection. Говорят, что поток *получил права владения* объектом CS, если произошел возврат из функции EnterCriticalSection, тогда как для уступки прав владения используется функция LeaveCriticalSection. *Всегда следите за своевременной переуступкой прав владения объектами CS; несоблюдение этого правила может привести к тому, что другие потоки будут пребывать в состоянии ожидания в течение неопределенного времени даже после завершения выполнения потока-владельца.*

Мы часто будем говорить о *блокировании* и *разблокировании* объектов CS, а вхождение в CS будет означать то же, что и блокирование CS.

VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection)

Поток, владеющий объектом CS, может повторно войти в этот же CS без его блокирования; это означает, что объекты CRITICAL_SECTION являются *рекурсивными* (recursive). Поддерживается счетчик вхождений в объект CS, и поэтому поток должен покинуть данный CS столько раз, сколько было вхождений в него, чтобы разблокировать этот объект для других потоков. Эта возможность может оказаться полезной для реализации рекурсивных функций и обеспечения безопасного многопоточного выполнения функций общих (разделяемых) библиотек.

Выход из объекта CS, которым данный поток не владеет, может привести к непредсказуемым результатам, включая блокирование самого потока.

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

BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)

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

Объекты CRITICAL_SECTION обладают тем преимуществом, что они не являются объектами ядра и поддерживаются в пользовательском пространстве. Обычно, но не всегда, это приводит к дополнительному улучшению показателей производительности. К обсуждению аспектов

производительности мы вернемся после того, как ознакомимся с объектами синхронизации, относящимися к ядру.

Настройка спин-счетчика

Обычно, если в результате выполнения функции EnterCriticalSection поток обнаруживает, что объект CS уже принадлежит другому потоку, он входит в ядро и остается блокированным до тех пор, пока не освободится объект CRITICAL_SECTION, что требует определенного времени. Однако в SMP-системах вы можете потребовать, чтобы поток повторил попытку завладеть объектом CS, прежде чем блокироваться, поскольку существует вероятность того, что поток, владеющий CS, выполняется на другом процессоре и в любой момент может освободить CS. Это может оказаться полезным для повышения производительности, если между потоками наблюдается высокая состязательность за право владения единственным объектом CRITICAL_SECTION. Влияние упомянутых факторов на производительность обсуждается далее в этой и последующих главах.

Для настройки счетчика занятости, или спин-счетчика (spin-count), предназначены две функции, одна из которых, SetCriticalSectionSpinCount, обеспечивает динамическую настройку счетчика, а вторая, InitializeCritical-SectionAndSpinCount, выступает в качестве замены функции Initialize-CriticalSection. Настройка спин-счетчика рассматривается в главе 9.

Использование объектов CRITICAL_SECTION для защиты разделяемых переменных

Использование объектов CRITICAL_SECTION не вызывает сложностей, и одним из наиболее распространенных способов их применения является обеспечение доступа потоков к разделяемым глобальным переменным. Рассмотрим, например, многопоточный сервер (аналогичный представленному на рис. 7.1), в котором необходимо вести учет следующих статистических данных:

- Общее количество полученных запросов.
- Общее количество отправленных ответов.
- Количество запросов, обрабатываемых в настоящее время всеми потоками сервера.

Поскольку переменные счетчиков являются глобальными переменными процесса, нельзя допустить того, чтобы одновременно два потока изменяли их значения. Один из методов обеспечения этого, базирующийся на применении объектов CRITICAL_SECTION, иллюстрирует схема, показанная ниже на рис. 8.2. Использование объектов CRITICAL_SECTION демонстрируется на примере программы 8.1, представляющей намного более простую систему, чем серверная.

Объекты CS могут привлекаться для решения задач, аналогичных той, которую иллюстрирует рис. 8.1, где два потока увеличивают значение одной и той же переменной. Приведенный ниже фрагмент кода обеспечивает нечто большее, нежели простое увеличение переменной, поскольку для этого достаточно было бы воспользоваться функциями взаимоблокировки. Обратите внимание на спецификатор volatile, предотвращающий размещение

текущего значения переменной оптимизирующим компилятором в регистре, а не в ячейке памяти, отведенной для хранения переменной. Кроме того, в этом примере используется промежуточная переменная; этот необязательный элемент снижает эффективность программы, однако позволяет более отчетливо продемонстрировать, каким образом решается задача, иллюстрируемая рис. 8.1.

```
CRITICAL_SECTION cs1;
volatile DWORD N = 0, M;
/* N — глобальная переменная, разделяемая всеми потоками. */
InitializeCriticalSection (&cs1);
…
EnterCriticalSection (&cs1);
if (N < N_MAX) { M = N; M += 1; N = M; }
LeaveCriticalSection (&cs1);
…
DeleteCriticalSection (&cs1);
```

На рис. 8.2 представлена одна из возможных последовательностей выполнения программы для случая, изображенного на рис. 8.1, и продемонстрировано, каким образом объекты CS упрощают решение проблемы синхронизации.

Программа 8.1 демонстрирует, насколько полезными могут быть объекты CS.

Пример: простая система "производитель/потребитель"

Программа 8.1 иллюстрирует, насколько полезными могут быть объекты CS. Кроме того, эта программа демонстрирует, как создаются защищенные структуры данных для хранения состояний объектов, и знакомит с понятием *инварианта* (invariant) — свойства состояния объекта, относительно которого гарантируется (путем соответствующей реализации программы), что оно будет истинным за пределами критического участка кода.

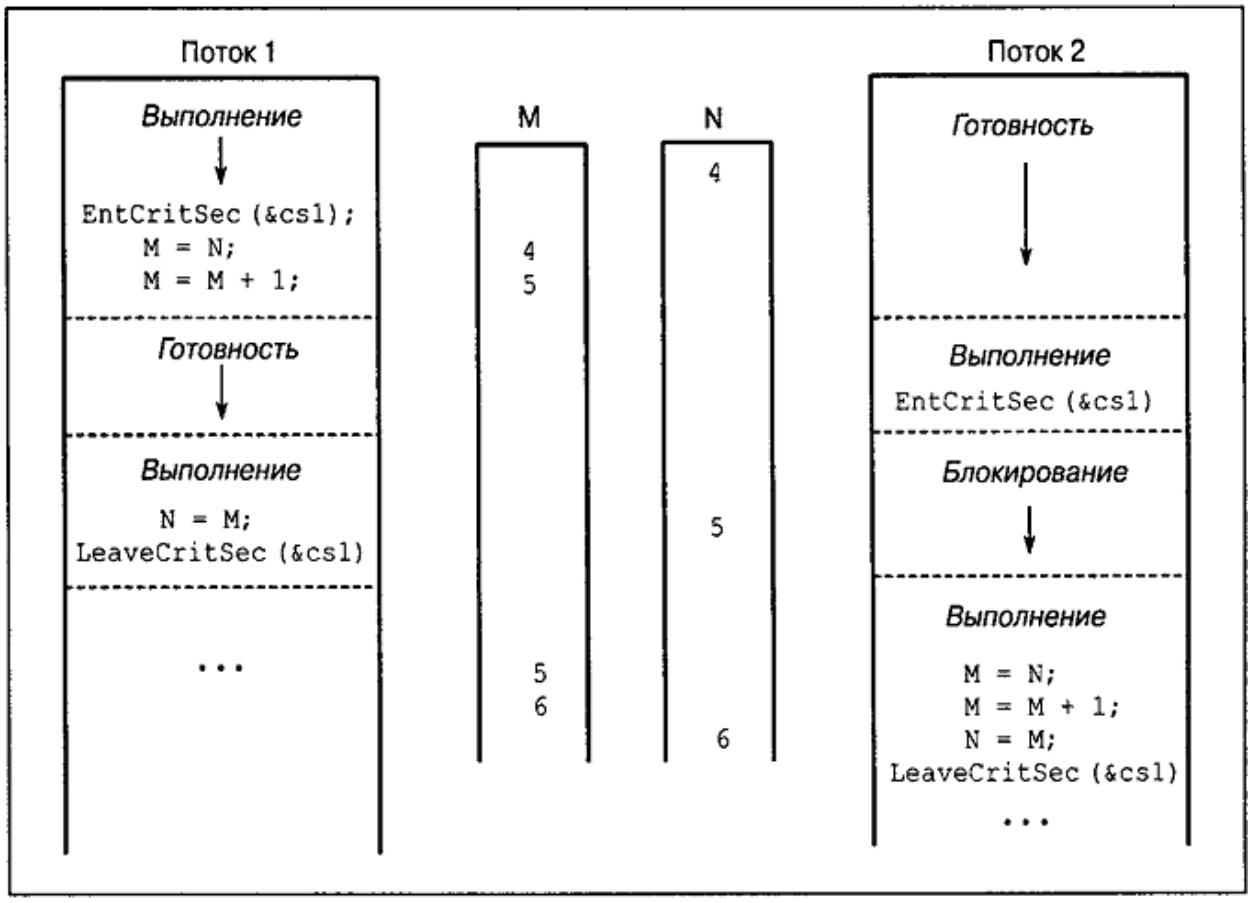

Рис. 8.2. Разделение общей памяти синхронизированными потоками

Описание задачи приводится ниже.

• Имеются два потока, *производитель* (producer) и *потребитель* (consumer), работающие в полностью асинхронном режиме.

• Производитель периодически создает сообщения, содержащие таблицу чисел, например, таблицу биржевых котировок, которая периодически обновляется.

• По требованию пользователя потребитель отображает текущие данные. Требуется, чтобы отображаемые данные представляли собой *самый последний полный набор данных, но никакие данные не должны отображаться дважды.*

• Данные не должны отображаться в те промежутки времени, когда они обновляются производителем; устаревшие данные также не должны отображаться. Обратите внимание на то, что многие сообщения вообще никогда не используются и, таким образом, "теряются". Этот пример является частным случаем конвейерной модели, в которой данные передаются из одного потока в другой.

• В качестве средства контроля целостности данных производитель вычисляет простую контрольную сумму[28] данных таблицы, которая далее сравнивается с аналогичной суммой, вычисленной потребителем, дабы удостовериться в том, что данные не были повреждены при их передаче из одного потока в другой. Данные, полученные при обращении к таблице в моменты ее обновления, будут недействительными; использование объектов

CS гарантирует, что этого никогда не произойдет. Инвариантом блока сообщения (message block invariant) является корректность контрольной суммы для содержимого текущего сообщения.

• Обоими потоками поддерживается статистика суммарного количества отправленных, полученных и утерянных сообщений.

Программа 8.1.simplePC: простая система "производитель/потребитель"

```
/* Глава 8. simplePC.c */
/* Поддерживает два потока — производителя и потребителя. */
/* Производитель периодически создает буферные данные с контрольными */
/* суммами, или "блоки сообщений", отображаемые потребителем по запросу */
/* пользователя. */

#include "EvryThng.h"
#include <time.h>
#define DATA_SIZE 256

typedef struct msg_block_tag { /* Блок сообщения. */
 volatile DWORD f_ready, f_stop; /* Флаги готовности и прекращения сообщений. */
 volatile DWORD sequence; /* Порядковый номер блока сообщения. */
 volatile DWORD nCons, nLost;
 time_t timestamp;
 CRITICAL_SECTION mguard; /* Структура защиты блока сообщения. */
 DWORD checksum; /* Контрольная сумма содержимого сообщения. */
 DWORD data[DATA_SIZE]; /* Содержимое сообщения. */
} MSG_BLOCK;

/* Одиночный блок, подготовленный к заполнению новым сообщением. */
MSG_BLOCK mblock = { 0, 0, 0, 0, 0 };

DWORD WINAPI produce(void*);
DWORD WINAPI consume(void*);
void MessageFill(MSG_BLOCK*);
void MessageDisplay(MSG_BLOCK*);

DWORD _tmain(DWORD argc, LPTSTR argv[]) {
 DWORD Status, ThId;
 HANDLE produce h, consume_h;
```

```
        /* Инициализировать критический участок блока сообщения. */
        InitializeCriticalSection  (&mblock.mguard);
        /* Создать два потока. */
        produce_h = (HANDLE)_beginthreadex(NULL, 0, produce, NULL, 0,
&ThId);
        consume_h = (HANDLE)_beginthreadex (NULL, 0, consume, NULL, 0,
&ThId);
        /* Ожидать завершения потоков производителя и потребителя. */
        WaitForSingleObject(consume_h, INFINITE);
        WaitForSingleObject(produce_h, INFINITE);
        DeleteCriticalSection(&mblock.mguard);
        _tprintf(_T("Потоки производителя и потребителя завершили
выполнение\n"));
        _tprintf(_T("Отправлено: %d, Получено: %d, Известные потери:
%d\n"), mblock.sequence, mblock.nCons, mblock.nLost);
        return 0;
       }

       DWORD WINAPI produce(void *arg)
       /* Поток производителя — создание новых сообщений через
случайные */
       /* интервалы времени. */
       {
        srand((DWORD)time(NULL)); /* Создать начальное число для
генератора случайных чисел. */
        while (!mblock.f_stop) {
         /* Случайная задержка. */
         Sleep(rand() / 100);
         /* Получить и заполнить буфер. */
         EnterCriticalSection(&mblock.mguard);
         __try {
          if (!mblock.f_stop) {
           mblock.f_ready = 0;
           MessageFill(&mblock);
           mblock.f_ready = 1;
           mblock.sequence++;
          }
         } __finally { LeaveCriticalSection (&mblock.mguard); }
        }
        return 0;
       }

       DWORD WINAPI consume (void *arg) {
        DWORD ShutDown = 0;
        CHAR command, extra;
```

```
/* Принять ОЧЕРЕДНОЕ сообщение по запросу пользователя. */
while (!ShutDown) { /* Единственный поток, получающий доступ к стандартным устройствам ввода/вывода. */
 _tprintf(_T("\n**Введите 'c' для приема; 's' для прекращения работы: "));
 _tscanf("%c%c", &command, &extra);
 if (command == 's') {
  EnterCriticalSection(&mblock.mguard);
  ShutDown = mblock.f_stop = 1;
  LeaveCriticalSection(&mblock.mguard);
  } else if (command == 'c') { /* Получить новый буфер для принимаемых сообщений. */
  EnterCriticalSection(&mblock.mguard);
  __try {
   if (mblock.f_ready == 0) _tprintf(_T("Новые сообщения отсутствуют. Повторите попытку.\n"));
   else {
    MessageDisplay(&mblock);
    mblock.nCons++;
    mblock.nLost = mblock.sequence – mblock.nCons;
    mblock.f_ready = 0; /* Новые сообщения отсутствуют. */
    }
  } __finally { LeaveCriticalSection (&mblock.mguard); }
 } else {
  tprintf(_T("Такая команда отсутствует. Повторите попытку.\n"));
 }
}
return 0;
}

void MessageFill(MSG_BLOCK *mblock) {
 /* Заполнить буфер сообщения содержимым, включая контрольную сумму и отметку времени. */
 DWORD i;
 mblock->checksum = 0;
 for (i = 0; i < DATA_SIZE; i++) {
  mblock->data[i] = rand();
  mblock->checksum ^= mblock->data[i];
 }
 mblock->timestamp = time(NULL);
 return;
}

void MessageDisplay(MSG_BLOCK *mblock) {
```

```
/* Отобразить буфер сообщения, отметку времени и контрольную сумму. */
DWORD i, tcheck = 0;
for (i = 0; i < DATA_SIZE; i++) tcheck ^= mblock->data[i];
_tprintf(_T("\nВремя генерации сообщения № %d: %s"), mblock->sequence, _tctime(&(mblock->timestamp)));
_tprintf(_T("Первая и последняя записи: %x %x\n"), mblock->data[0], mblock->data[DATA_SIZE – 1]);
if (tcheck == mblock->checksum) _tprintf(_T("УСПЕШНАЯ ОБРАБОТКА –>Контрольная сумма совпадает.\n"));
else tprintf(_T("СБОЙ –>Несовпадение контрольной суммы. Сообщение запорчено.\n"));
return;
}
```

Комментарии к примеру простой системы "производитель/потребитель"

Этот пример иллюстрирует некоторые моменты и соглашения, касающиеся программирования, которые будут важны для нас на протяжении этой и последующих глав.

• Объект CRITICAL_SECTION является частью объекта (блока сообщения), защиту которого он обеспечивает.

• Каждый доступ к сообщению осуществляется на критическом участке кода.

• Типом переменных, доступ к которым осуществляется разными потоками, является volatile.

• Использование обработчиков завершения гарантирует, что объекты CS будут обязательно освобождены. Хотя в данном случае эта методика и не является для нас существенной, она дополнительно гарантирует, что вызов функции LeaveCriticalSection не будет случайно опущен впоследствии при изменении кода программы. Имейте также в виду, что обработчик завершения ограничен использованием средств C, и его не следует использовать совместно с C++.

• Функции MessageFill и MessageDisplay вызываются лишь на критических участках кода и используют для нужд своих вычислений не глобальную, а локальную память. Кстати, обе они будут применяться и в последующих примерах, но их листинги больше приводиться не будут.

• Не существует удобного способа, при помощи которого поток производителя мог бы известить поток потребителя о наличии нового сообщения, и поэтому поток потребителя должен просто ожидать, пока не будет установлен флаг готовности, который используется для индикации появления нового сообщения. Устранить этот недостаток нам помогут объекты событий ядра.

• Одним из инвариантных свойств, которые гарантируются этой программой, является то, что контрольная сумма блока сообщения будет

всегда корректной *вне* критических участков кода. Другим инвариантным свойством является следующее:

0 <= nLost + nCons <= sequence

Об этом важном свойстве далее еще будет идти речь.

• О необходимости прекращения передачи поток производителя узнает лишь после проверки флага, устанавливаемого в блоке сообщения потока потребителя. Поскольку потоки не могут обмениваться между собой никакими сигналами, а вызов функции TerminateThread чреват нежелательными побочными эффектами, эта методика является простейшим способом остановки другого потока. Разумеется, чтобы эта методика была эффективной, работа потоков должна быть скоординированной. В то же время, подобное решение требует, чтобы поток не блокировался, иначе он не сможет тестировать флаг; способы решения проблемы блокированных потоков обсуждаются в главе 10.

Объекты CRITICAL_SECTION предоставляют в наше распоряжение мощный механизм синхронизации, но, тем не менее, они не в состоянии обеспечить всю полноту необходимых функциональных возможностей. О невозможности отправки сигналов одним потоком другому уже говорилось, кроме того, эти объекты не позволяют воспользоваться конечными интервалами ожидания (time-out). Объекты синхронизации ядра Windows позволяют снизить остроту не только этих, но и других ограничений.