**Лабораторная № 10**

**Тема: Многопоточность систем. Синхронизация потоков. Мьютексы. Семафоры.**

Контрольные вопросы:

1. Что такое"Ппроцесс"? Приведите пример процесса.
2. Что выполняют процессы?
3. Что такое "Поток"? Приведите пример потока.
4. Для чего используются потоки?
5. Что такое "Вытесняющая многозадачность системы"?
6. Приведите диаграмму, которая отображает порядок событий во времени и состояние потока выполняющегося приложения между этими событиями.
7. Что значит VCL-поток?
8. Опишите понятие "Синхронизация потоков".
9. Что значит "Атомарность действия"?
10. Опишите конкуренцию потоков.
11. Опишите завершение потока (несколько методов).
12. Опишите критические секции.
13. Дайте определение понятию "Мьютекс".
14. Дайте определение понятию "Семафоры".
15. Опишите роль семафоров при управлении потоками данных.
16. Опишите дополнительные механизмы синхронизации потоков.

Содержание

Глава 1. Что такое потоки. Для чего их использовать?   
Глава 2. Создание потока в Дельфи.   
Глава 3. Основы синхронизации.   
Глава 4. Простое разрушение потока.   
Глава 5. Снова о разрушении потока. Тупик или зацикливание (Deadlock).   
Глава 6. Снова о синхронизации: Критические секции и мьютексы.   
Глава 7. Программирование с использованием мьютексов. Управление конкуренцией.   
Глава 8. Потокобезопасные классы в Дельфи и приоритеты.   
Глава 9. Семафоры. Управление потоками данных. Взаимосвязь источник-приемник.   
Глава 10. Ввод/вывод и потоки данных: от блокировки к асинхронности и обратно.   
Глава 11. Синхронизаторы и события (Events).

Глава 1. Что такое потоки (threads)? Для чего их использовать?  
  
  
Содержание:

* Из истории.
* Определения.
* Пример.
* Разделение времени.
* Для чего используют потоки?

Из истории.  
  
На заре компьютерной эры все программирование было в основном однопоточным. Вы создавали программу, пробивая дырочки в перфокартах или ленте, относили стопку карт в местный вычислительный центр, и через несколько дней получали другую стопку, в удачных случаях содержащую требуемые результаты. Вся обработка велась по принципу - что раньше поступило, раньше выполняется, и при запуске вашей программы она одна использовала компьютерное время.   
Но все меняется. Концепция многопоточного исполнения впервые появилась на системах с разделением времени (time sharing), где несколько человек могли одновременно работать на центральном компьютере. Важно отметить, что процессорное время просто делилось между пользователями, а уже сама операционная система использовала концепции "процессов" и "потоков". Настольные компьютеры прошли тот же путь. Раньше DOS и Windows были однозадачными. На компьютере могла исполняться лишь единственная программа. С усложнением приложений и ростом требований к персональному компьютеру, особенно в отношении высокой производительности графики и сетевых возможностей, потребовались многопроцессные и многопоточные операционные системы.   
  
Определения.  
  
Сначала определим, что называется **процессом**. Большинство пользователей Windows 95, 98 и NT имеют об этом хорошее интуитивное представление. Они рассматривают процесс как программу, которая выполняется на машине, сосуществуя и разделяя ресурсы процессора, диска и памяти с другими программами. Программисты знают, что процесс - это вызов исполняемого кода, причем этот код уникален и его инструкции выполняются в определенном порядке. В общем, процессы изолированы. Используемые ими ресурсы (память, диск, ввод-вывод,процессорное время) виртуальны, так что каждый процесс имеет свой набор ресурсов, не разделяя их с другими, что обеспечивается операционной системой. Процессы выполняют **модули кода**. Они могут быть раздельными, например, модули кода Windows Explorer и Microsoft Word - различны. Однако они могут быть и общими, как в случае библиотек DLL. Код DLL обычно исполняется во многих процессах, и часто одновременно. Выполнение инструкций в целом для разных процессов не упорядочено: Microsoft Word не останавливает открытие документа во время посылки данных из очереди принтера! Конечно, когда разные процессы взаимодействуют, програмист должен следить за порядком, о чем будет рассказано ниже.   
Следующее определение - **поток** (нить, thread). Концепция потоков появилась, когда стало ясно, что желательно иметь приложения, осуществляющие набор действий с наименьшими затратами на переключение, по возможности одновременно. В ситуациях, когда некие действия могут вызвать существенную задержку в одном из потоков (например, ожидание ввода от пользователя), часто желательно, чтобы программа имела возможность осуществлять другие действия независимо (например, фоновую проверку орфографии или обработку входящих сетевых сообщений). Однако создание нового процесса для каждого действия (с соответственно возникающей задачей обеспечения их взаимодействия) часто является неоправданным.   
  
Пример.  
  
Если нужно показать хороший пример многопоточности, то на эту роль прекрасно подходит Проводник (т.е. Windows Shell). Сделайте двойной щелчок на "My Computer", и откройте несколько папок, создавая новое окно для каждой. Теперь запустите длительное копирование в одном из окон. Появляется индикатор копирования, и это окно не отвечает на действия пользователя. Однако все другие окна можно использовать. Очевидно, что несколько действий может исполняться одновременно при единственной запущенной копии explorer.exe. В этом и состоит сущность многопоточности.   
  
Разделение времени.  
  
В большинстве систем, поддерживающих многопоточность, может быть много пользователей, делающих одновременные запросы к вычислительной системе. Обычно число процессоров в системе меньше, чем число потоков, которые могут исполняться параллельно. Большинство систем поддерживает **разделение времени** (time slicing), известное также как **вытесняющая многозадачность** (pre-emptive multitasking) для решения этой проблемы. В системе с разделением времени потоки запускаются на короткое время, а затем вытесняются; т.е. таймер периодически заставляет ОС заново решать, какие потоки должны исполняться, потенциально останавливая уже выполняющиеся потоки, и запуская другие, которые были приостановлены. Это позволяет даже единственному процессору выполнять много потоков. На PC эти промежутки времени составляют порядка 55 миллисекунд.   
  
Для чего используют потоки?  
  
Потоки не должны изменять семантику программы. Они просто изменяют время выполнения операций. В результате они почти всегда используются для изящного решения проблем, связанных с производительностью. Вот несколько примеров ситуаций, в которых можно использовать потоки:

* Выполнение длительных действий: когда приложение ведет расчеты, оно не отвечает на сообщения, в результате не обновляется экран.
* Выполнение фоновых действий: некоторые задачи не критичны ко времени, но должны выполняться постоянно.
* Выполнение действий по вводу-выводу (I/O): работа с диском или сетью может привести к неопределенным задержкам. Потоки дадут возможность в таких случаях не останавливать исполнение других частей программы.

Все эти примеры имеют общее: некоторые действия в программе могут вызвать потенциально большую задержку в работе CPU, недопустимую для других операций, которые нужно провести *именно сейчас* . Конечно, есть и другие преимущества, например:

* При использовании многопроцессорных систем: приложение с единственным потоком не будет использовать два или более процессоров! В Главе 3 это объяснено более детально.
* Эффективное разделение времени: используя приоритеты процессов и потоков, вы обеспечите наиболее правильное использование времени CPU.

Мудрое использование потоков делает медленные, плохо взаимодействующие с пользователем программы быстрыми и удобными, эффективными, и может значительно улучшить как производительность, так и удобство использования.

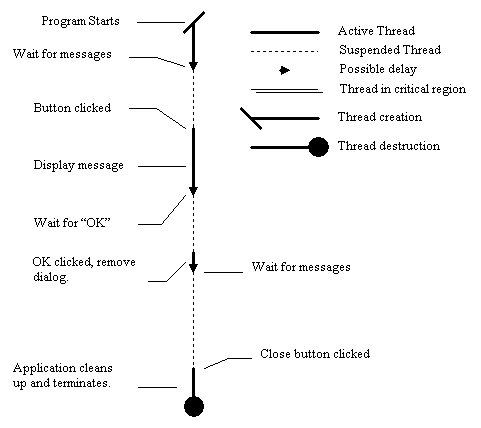
Глава 2. Создание потока в Delphi.  
  
Содержание:

* Предисловие с диаграммой.
* Наш первый не-VCL поток .
* Что именно делает эта программа?
* Проблемы и сюрпризы.
* Проблемы запуска.
* Проблемы взаимодействия.
* Проблемы завершения.

Предисловие с диаграммой.  
  
До расмотрения деталей создания потока и выполнения его кода независимо от основного потока приложения необходимо разобраться в диаграмме, иллюстрирующей динамику выполнения потока. Это значительно нам поможет, когда мы начнем разрабатывать многопоточные программы. Рассмотрим пример 1

|  |
| --- |
| **Код** |
| unit test;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls;  type    TForm1 = class(TForm)     Button1: TButton;     procedure Button1Click(Sender: TObject);   private     { Private declarations }   public     { Public declarations }   end;  var   Form1: TForm1;  implementation  {$R \*.DFM}  procedure TForm1.Button1Click(Sender: TObject); begin   ShowMessage('Hello World!'); end;  end. |

.   
Приложение имеет один исполняемый поток: основной поток VCL. Его работу можно проиллюстрировать диаграммой, показывающей состояние потока в течение времени выполнения. Ось времени направлена вниз. Описание этой диаграммы будет относиться и ко всем последующим диаграммам выполнения потоков.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig1.gif)

Заметьте, что эта диаграмма не показывает деталей выполнения алгоритмов. Вместо этого она отображает порядок событий во времени и состояние потока приложения между этими событиями. Имеет значение не фактическое расстояние между разными точками на диаграмме, а их вертикальное упорядочение. Некоторые части этой диаграммы следует рассмотреть особенно подробно.   
Поток приложения не выполняется непрерывно Могут быть длинные периоды времени, когда он не получает никаких внешних стимулов, и совсем не выполняет вычислений или действий. Память и ресурсы приложением заняты, и окно находится на экране, но CPU не исполняет кода.   
Приложение запущено, и выполняется основной поток. Как только создано главное окно, работы больше нет, и поток попадает в часть кода VCL, которая называется цикл обработки сообщений, опрашивающий операционную систему о наличии сообщений. Если нет сообщений, требующих обработки, операционная система **приостанавливает** (suspend) поток.

* В некоторый следующий момент, пользователь нажимает на кнопку, чтобы отобразить текстовое сообщение. Операционная система **возобновляет** (resume) основной поток и передает ему сообщение, указывающее, что кнопка нажата. Основной поток теперь снова **активен**.
* Этот процесс resume - suspend происходит несколько раз. Для иллюстрации я ввел ожидание подтверждения пользователем закрытия окна сообщения, и ожидание нажатия кнопки закрытия. На практике могут быть получены и многие другие сообщения.

Наш первый не-VCL поток.  
  
Хотя Win32 API обеспечивает исчерпывающую поддержку многопоточности, для создания и уничтожения потоков, в VCL имеется полезный класс, TThread, который предоставляет более высокоуровневый подход, значительно упрощает работу и помогает программисту избегать некоторых неприятных ловушек, в которые можно попасть при недостатке опыта. Я рекомендую использовать именно его. Система помощи Дельфи дает неплохое введение в создание класса потока, так что я не буду подробно рассказывать о последовательности действий для создания потока, за исключением того, что предложу выбрать пункт меню *File| New...* и затем *Thread Object .*  
Этот пример содержит программу, которая вычисляет, является ли данное число простым. Она состоит из двух модулей, один с обычной формой, и один с объектом потока. Она более или менее работоспособна, но обладает несколькими неприятными особенностями, которые иллюстрируют основные проблемы, с которыми встречаются программисты, разрабатываюшие многопоточные приложения. Мы обсудим пути их преодоления позже. Модуль формы

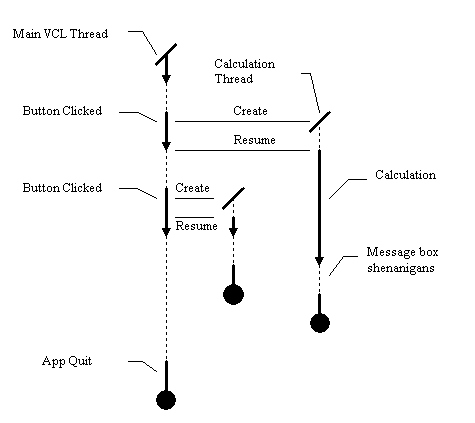
|  |
| --- |
| **Код** |
| unit PrimeForm;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls;  type   TPrimeFrm = class(TForm)     NumEdit: TEdit;     SpawnButton: TButton;     procedure SpawnButtonClick(Sender: TObject);   private       { Private declarations }   public       { Public declarations }   end;  var   PrimeFrm: TPrimeFrm;  implementation  uses PrimeThread;  {$R \*.DFM}  procedure TPrimeFrm.SpawnButtonClick(Sender: TObject);  var   NewThread: TPrimeThrd;  begin   NewThread := TPrimeThrd.Create(True);   NewThread.FreeOnTerminate := True;   try     NewThread.TestNumber := StrToInt(NumEdit.Text);     NewThread.Resume;   except on EConvertError do     begin       NewThread.Free;       ShowMessage('That is not a valid number!');     end;   end; end;  end. |

и модуль объекта потока.

|  |
| --- |
| **Код** |
| unit PrimeThread;  interface  uses   Classes;  type   TPrimeThrd = class(TThread)   private     FTestNumber: integer;   protected     function IsPrime: boolean;     procedure Execute; override;   public     property TestNumber: integer write FTestNumber;   end;  implementation  uses SysUtils, Dialogs;  function TPrimeThrd.IsPrime: boolean;  var   iter: integer;  begin   result := true;   if FTestNumber < 0 then   begin     result := false;     exit;   end;   if FTestNumber <= 2 then     exit;   for iter := 2 to FTestNumber - 1 do   begin     if (FTestNumber mod iter) = 0 then     begin       result := false;       {exit;}     end;   end; end;  procedure TPrimeThrd.Execute; begin   if IsPrime then     ShowMessage(IntToStr(FTestNumber) + 'is prime.')   else     ShowMessage(IntToStr(FTestNumber) + 'is not prime.'); end;  end. |

Что именно делает эта программа?  
  
Всякий раз, когда нажата кнопка "Spawn", программа создает новый объект потока, инициализирует несколько его полей, затем запускает выполнение потока. В зависимости от величины входного числа, поток работает, вычисляя, простое ли это число, и как только вычисления завершаются, поток отображает сообщение, показывая, простое оно или нет. Эти потоки конкурируют между собой, и, независимо от того, одно- или многопроцессорная у вас машина, с точки зрения пользователя они выполняются одновременно. Кроме того, эта программа не ограничивает числа созданных потоков. В результате вы можете продемонстрировать истинный параллелизм так:

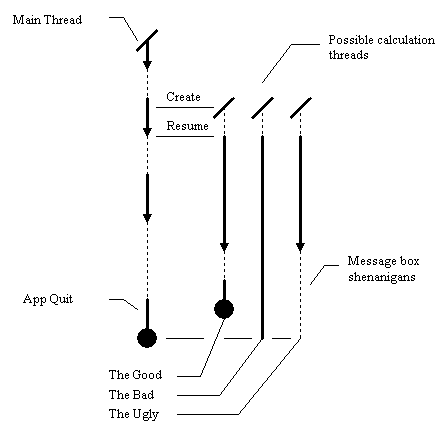
* Поскольку я закомментировал оператор выхода в подпрограмме определения простого числа, время потраченное потоком, приблизительно пропорционально величине входного числа. У меня для аргумента порядка 2^24 поток до завершения работает около 10-20 секунд. Подберите число для обеспечения такой задержки на вашей машине.
* Запустите программу, введите большое число, нажмите кнопку.
* Сразу же введите небольшое число (например, 42) и нажмите кнопку снова. Вы увидите, что результат для маленького числа появится раньше результата для большого, несмотря на то, что сначала мы запустили поток для большого числа. Диаграмма иллюстрирует эту ситуацию.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig2.gif)

Проблемы и сюрпризы.  
  
На этом этапе появляется проблема синхронизации. Когда основной поток возобновляет выполнение (вызывает resume) "рабочего" потока, основной поток программы не может ничего знать о состоянии рабочего потока и наоборот. Вполне возможно, что рабочий поток может завершить свое выполнение прежде, чем в основном потоке VCL выполнится хоть один оператор. Фактически для маленьких чисел, расчет для которых займет менее чем 1/20 секунды, это весьма вероятно. Аналогично, рабочий поток не может ничего предполагать о состоянии основного потока. Остается лишь полагаться на планировщик Win32. Рассмотрим три основные проблемы: **запуск, взаимодействие и завершение**.   
  
Проблемы запуска.  
  
Delphi облегчает запуск потока. Перед началом исполнения порожденного потока часто нужно установить некоторое его начальное состояние. Создавая поток приостановленным (параметр конструктора потока), можно быть уверенным, что код потока не будет выполняться пока его не активируют. Это означает, что основной поток VCL может безопасно прочитать и модифицировать данные объекта TThread, гарантируя, что они будут правильными, когда порожденный поток начнет выполняться.  
В данной программе свойства потока "FreeOnTerminate" и "TestNumber" установлены до начала выполнения. Если бы это не было сделано, то поведение потока должно было быть неопределенным. Если вы не хотите создавать поток приостановленным, то просто отодвигаете проблемы запуска до следующего этапа: проблемы взаимодействия.   
  
Проблемы взаимодействия.  
  
Эти проблемы появляются, если у вас есть два работающих потока, и вам нужно каким-то способом связываться между ними. Эта программа не затрагивает взаимодействие потоков. На данный момент достаточно отметить, что если вы не защищаете все операции над разделяемыми данными, ваша программа, вероятно, будет вести себя непредсказуемо. Если вы не обеспечиваете требуемую синхронизацию и управление параллельным доступом, недопустимо следующее:

* Доступ к любой форме или общим ресурсам из двух потоков.
* Доступ к потоко-незащищенным частям VCL из не-VCL потока.
* Попытка делать графические операции из отдельного потока.

Даже такая простая операция, как доступ к общей целой переменной из двух потоков, может закончиться полным беспорядком, а несинхронизированный доступ к общим ресурсам или вызовы VCL приведут ко многим часам непростой отладки, значительной неразберихи, и возможно к обращению в ближайшую психиатрическую лечебницу. Пока вы не изучили подходящие методы в следующих главах, не делайте этого.  
Есть ли хорошие новости? Вы можете делать все три вышеуказанные действия, если используете правильные механизмы для управления параллельным выполнением, и это не так уж и трудно! Мы рассмотрим простой путь разрешения вопросов взаимодействия через VCL в следующей главе, а более изящные (но и более сложные) методы позже.   
  
Проблемы завершения.  
  
Поток, подобно любому другому объекту Delphi, использует распределение памяти и других ресурсов, так что не должен вызывать удивления факт, что важно обращаться с завершением потока очень аккуратно, а наша программа этого не делает. Есть два возможных подхода к проблеме освобождения ресурсов.  
Первый - позволить потоку решить все самому. Это главным образом используется для потоков, которые:   
а) Передают результаты выполнения потока в основной поток VCL перед остановкой.  
б) Не содержат ко времени завершения никакой информации, необходимой другому потоку.  
В этих случаях программист может установить флаг "FreeOnTerminate" для объекта потока, и он корректно освободит ресурсы при своем завершении.  
Второй подход - основной поток VCL должен прочитать данные из объекта рабочего потока после его завершения, а затем уничтожить его. Это описано в Главе 4.   
Я не затрагивал проблем передачи результатов в основной поток, поскольку рабочий поток сам сообщает пользователю ответ путем вызова ShowMessage. При этом не используется связь с основным потоком VCL, и вызов ShowMessage можно рассматривать как потокобезопасный, так что работа VCL не нарушается. В результате я могу использовать первый метод, для разрушения потока и разрешить потоку самоуничтожиться. Несмотря на это, программа иллюстрирует одну неприятную особенность, проявляющуюся при саморазрушении потока:

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig3.gif)

Как можно заметить, могут произойти две вещи. Во-первых, мы можем попытаться выйти из программы, когда поток еще активен и ведет вычисления. Во-вторых - мы можем попытаться выйти из программы, когда поток приостановлен. Первый случай довольно благоприятный: приложение закрывается, не считаясь с потоком. Код завершения Delphi и Windows cделает все, как нужно. Второй вариант несколько хуже, поскольку поток приостанавливается где-то в недрах подсистемы обмена сообщениями Win32. При этом Delphi производит работу по очистке в обоих случаях. Тем не менее, принудительный выход из потока без учета того, в каком он состоянии - плохой стиль программирования. Например, рабочий поток может в это время вести запись в файл. Если пользователь выходит из программы до завершения процесса записи, то файл может быть поврежден. Вот почему правильнее, когда порожденные потоки завершают работу согласованно с основным потоком VCL, даже если не требуется передача данных: при этом возможно чистое завершение процесса и потока. В Главе 4 обсуждаются решения этой проблемы.

Глава 3. Основы синхронизации.  
  
Содержание:

* Как разделять данные между потоками?
* Атомарность при доступе к общим данным.
* Дополнительные проблемы с VCL.
* Многопроцессорные машины.
* Решение для Delphi: TThread.Synchronize.
* Как это работает? Что делает Synchronize?
* Синхронизация для не-VCL потоков.

Как разделять данные между потоками?  
  
Прежде всего, необходимо знать, что для каждого процесса и потока сохраняется его состояние. У всякого потока имеется собственный программный счетчик и состояние процессора. Это означает, что код каждого потока исполняется независимо. Каждый поток обладает также своим стеком, так что локальные переменные в сущности видны лишь внутри каждого отдельного потока, и для этих переменных не существует вопросов синхронизации . Глобальные же данные программы могут быть общими для нескольких потоков, и для них, таким образом, может появиться проблема синхронизации. Конечно, это не страшно, если переменная глобальна, но используется только одним потоком. Такая же ситуация и для для памяти, распределенной в куче (обычно для объектов): в принципе, любой поток может иметь доступ к конкретному объекту, но если программа написана так, чтобы только у одного потока был указатель на конкретный объект, то только он и может обращаться к этому объекту, и проблемы синхронизации не возникает.   
В Delphi есть зарезервированное слово **threadvar**, что позволяет объявлять "глобальные" переменные, копия которых создается для каждого потока. Эта возможность используется нечасто, поскольку обычно удобнее размещать такие переменные в классе TThread, создавая, таким образом, один экземпляр переменной для каждого созданного потомка TThread.   
  
Атомарность при доступе к общим данным.  
  
Для того, чтобы понять, как заставить потоки работать вместе, необходимо понимать концепцию **атомарности**. Действие или последовательность действий называются **атомарными,** если они неделимы. Когда поток выполняет атомарное действие, то все другие потоки видят это действие или как еще не начатое, или как уже завершенное. Невозможно одному потоку застать другой "в действии". Если потоки несинхронизированы, то все действия неатомарные. Давайте рассмотрим простой пример:

|  |
| --- |
| **Код** |
| var   a: integer; {a is global}  begin   a := a + 1; end; |

Что может быть проще? К несчастью, даже этот простой код может вызвать проблемы, если два потока используют его для увеличения общей переменной A. Этот единственный оператор Паскаля транслируется в три действия на ассемблерном уровне.

* Чтение A из ячейки памяти в регистр процессора.
* Увеличение регистра процессора на 1.
* Запись содержимого регистра процессора в ячейку памяти A.

Даже на однопроцессорной машине выполнение этого кода несколькими потоками может вызвать проблемы. Причина этого - в **scheduling** (планировщике действий). Когда есть только один процессор, в любой момент действительно выполняется только один поток, но планировщик Win32 переключается между потоками приблизительно 18 раз в секунду. Планировщик может остановить выполнение одного потока и запустить другой в любое время: вытесняющая многозадачность. Операционная система не ждет разрешения перед остановкой одного потока и запуском другого: переключение может случиться в любой момент. Поскольку переключение может произойти между любыми двумя инструкциями процессора, оно может случиться и произойти в некой точке в середине функции, и даже на полпути выполнения одного конкретного оператора программы. Давайте представим себе, что два потока (X и Y) выполняют код примера на однопроцессорной машине. В удачном случае программа может работать, и действия планировщика могут и не захватить эту критическую точку, что даст ожидаемые результаты: А увеличится на два.

|  |
| --- |
| **Код** |
| Инструкции, выполняемые                   Инструкции, выполняемые          Значение       потоком X                                  потоком Y               переменной A     <другие операции>                   поток приостановлен                    1 Чтение переменной A в регистр.         поток приостановлен                    1 Увеличение регистра на 1.              поток приостановлен                    1 Запись из регистра в переменную A.     поток приостановлен                    2    <другие операции>                   поток приостановлен                    2 Переключение потоков                   Переключение потоков                   2 поток приостановлен                      <другие операции>                    2 поток приостановлен                    Чтение переменной A в регистр.         2 поток приостановлен                    Увеличение регистра на 1.              2 поток приостановлен                    Запись из регистра в переменную A.     3 поток приостановлен                      <другие операции>                    3 |

Однако нет никакой гарантии, что все именно так и произойдет. Закон Мерфи гласит, что может случиться следующее:

|  |
| --- |
| **Код** |
| Инструкции, выполняемые                   Инструкции, выполняемые          Значение       потоком X                                  потоком Y               переменной A     <другие операции>                   поток приостановлен                    1 Чтение переменной A в регистр.         поток приостановлен                    1 Увеличение регистра на 1.              поток приостановлен                    1 Переключение потоков                   Переключение потоков                   1 поток приостановлен                      <другие операции>                    1 поток приостановлен                    Чтение переменной A в регистр.         1 поток приостановлен                    Увеличение регистра на 1.              1 поток приостановлен                    Запись из регистра в переменную A.     2 Переключение потоков                   Переключение потоков                   2 Запись из регистра в переменную A.     поток приостановлен                    2    <другие операции>                   поток приостановлен                    2 |

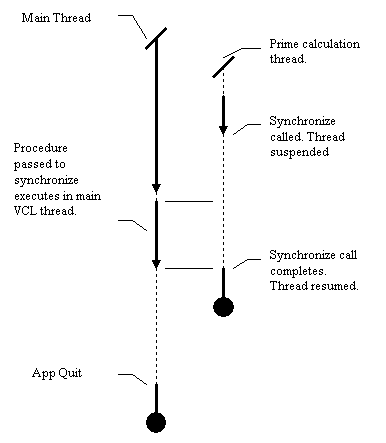
В этом случае А увеличивается не на два, а только на единицу. Конечно, если А является положением индикатора, то это, скорее всего, не проблема, но если А - что-нибудь более важное, подобно счетчику количества элементов в списке, тогда жди беды. Если общая переменная является указателем, то можно наткнуться на самые разные неприятные результаты. Это иногда называют **race condition** (конфликт, конкуренция потоков)**.**   
  
Дополнительные проблемы с VCL.  
  
VCL не содержит никакой защиты от этих конфликтов. Это означает, что переключение потоков может произойти, когда один или более потоков выполняют код VCL. Большая часть VCL организована в этом отношении достаточно хорошо . К несчастью, компоненты, и, в частности, потомки TControl, содержат различные механизмы, которые не согласуются с переключением потоков. Переключение в неподходящий момент может привести к полному хаосу, искажению счетчиков общих дескрипторов, уничтожению не только данных, но и связей между компонентами.   
Даже когда поток не выполняет код VCL, отсутствие синхронизации все равно может вызвать проблемы: недостаточно убедиться, что основной поток VCL остановлен прежде, чем другие потоки что-то модифицируют. Часть кода VCL может все-таки выполняться (например, появление диалогового окна или запись на диск), приостанавливая основной поток. Если другой поток модифицирует разделяемые данные, это может отразиться на основном потоке, так что глобальные данные волшебным образом изменятся в результате вызова диалога или записи в файл. Очевидно, это неприемлемо, и означает, что либо только один поток может выполнять код VCL, либо должен быть найден механизм, который гарантирует, что отдельные потоки не влияют друг на друга.   
  
Многопроцессорные машины.  
  
К счастью для программиста, эта проблема не становится сложнее для машин с более чем одним процессором. Методы синхронизации, обеспечиваемые Delphi и Windows, работают одинаково хорошо независимо от количества процессоров. Разработчикам операционной системы Windows пришлось писать дополнительный код, чтобы справиться со многопроцессорной обработкой: Windows NT 4 сообщает пользователю при загрузке, используется ли одно- или многопроцессорное ядро. Тем не менее, для программиста все это невидимо. Вам не нужно заботиться о том, сколько процессоров есть на машине, так же как и о типе чипсета, использованного на материнской плате.  
  
Решение для Delphi: TThread.Synchronize.  
  
Delphi обеспечивает решение, идеальное для начинающих работать с потоками. Оно простое и решает все вышеуказанные проблемы. У класса TThread есть метод **Synchronize**. Этот метод принимает как параметр другой метод без параметров, который вы хотите выполнить. Таким обрахом гарантируется, что код метода без параметров будет выполнен в результате синхронизированного вызова, и не будет конфликтов с потоком VCL.   
Звучит интригующе? Вполне возможно. Я проиллюстрирую это на примере. Мы изменим нашу программу для простых чисел так, что вместо того, чтобы показывать окно сообщения, она добавит текст, говорящий о том, простое число или нет, в Memo на главной форме. Во-первых, добавим Memo (ResultsMemo) к главной форме:

|  |
| --- |
| **Код** |
| type   TPrimeFrm = class(TForm)     NumEdit: TEdit;     SpawnButton: TButton;     ResultsMemo: TMemo;     procedure SpawnButtonClick(Sender: TObject);   private       { Private declarations }   public       { Public declarations }   end; |

. Теперь добавим в наш поток новый метод (UpdateResults) , который покажет результаты в Мemo, а вместо вызова ShowMessage вызовем Synchronize, передавая этот метод как параметр. Объявление класса потока и измененные части теперь выглядят так:

|  |
| --- |
| **Код** |
| unit PrimeThread;  interface  uses   Classes;  type   TPrimeThrd = class(TThread)   private     FTestNumber: integer;     FResultString: string;   protected     function IsPrime: boolean;     procedure UpdateResults;     procedure Execute; override;   public     property TestNumber: integer write FTestNumber;   end;  implementation  uses SysUtils, Dialogs, PrimeForm;  procedure TPrimeThrd.UpdateResults; begin   PrimeFrm.ResultsMemo.Lines.Add(FResultString); end;  function TPrimeThrd.IsPrime: boolean; {omitted for brevity}    procedure TPrimeThrd.Execute;   begin     if IsPrime then       FResultString := IntToStr(FTestNumber) + ' is prime.'     else       FResultString := IntToStr(FTestNumber) + ' is not prime.';     Synchronize(UpdateResults);   end;  end. |

. Заметьте, что UpdateResults имеет доступ и к главной форме, и к строке результата. С точки зрения главного потока VCL, кажется, что главная форма изменяется в ответ на событие. С точки зрения рабочего потока, к строке результата осуществляется доступ во время вызова Synchronize.   
  
Как это работает? Что делает Synchronize?  
  
В коде, который исполняется при вызове Synchronize, можно делать все то же самое, что и в основном потоке VCL. Кроме того, можно также модифицировать данные, связанные со своим собственным объектом потока, причем безопасно, зная, что выполнение своего потока находится в конкретной точке (точке вызова Synchronize). То, что происходит на самом деле - довольно любопытно, и наилучшим образом иллюстрируется другой диаграммой.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig4.gif)

Когда вызывается Synchronize, рабочий поток приостанавливается. На этой стадии основной поток VCL может быть приостановлен в состоянии ожидания (idle), может быть временно приостановлен для операций ввода-вывода, а может и выполняться. Рабочий поток ждет, пока главный не перейдет в состояние ожидания (цикл обработки сообщений). Как только основной поток приостановится, *метод без параметров, переданный в Synchronize, выполняется в контексте основного потока VCL* . В нашем случае метод без параметров называется UpdateResults и работает он c Memo. Это гарантирует, что никаких конфликтов с основным потоком VCL не произойдет, и в сущности, выполнение этого кода очень похоже на выполнение любого кода Delphi, который срабатывает в ответ на сообщение, посланное приложению. Никаких конфликтов с потоком, вызвавшим Synchronize, не происходит, поскольку он приостановлен в известной безопасной точке (в коде TThread.Synchronize).   
Когда это "выполнение кода по доверенности" завершается, основной поток VCL снова свободно может исполнять свои прямые обязанности, а поток, вызвавший Synchronize, продолжает свою работу после возврата из вызова. Таким образом, вызов Synchronize в основном потоке VCL выглядит подобно обработке сообщения, а в счетном потоке - как вызов функции. Код потоков находится в известных точках, и конкуренции нет. Конфликты исключены. Проблема решена.   
  
Синхронизация для не-VCL потоков.  
  
Мой предыдущий пример показывает, как можно создать дополнительный поток, взаимодействующий с основным потоком VCL. Для этого он заимствует время основного потока VCL. Но такой подход не сработает при взаимодействии нескольких дополнительных потоков между собой. Если у вас есть два не-VCL потока, X и Y, то вы не можете вызвать Synchronize в одном лишь потоке X, и при этом модифицировать данные, хранимые в Y. Необходимо вызывать Synchronize из *обои* х потоков при чтении или записи разделяемых данных. На деле это означает, что данные модифицируются основным потоком VCL, а все другие потоки синхронизируются с основным каждый раз, когда им нужен доступ к этим данным. Это выполнимо, но неэффективно, особенно если основной поток занят: каждый раз, когда двум потокам нужно связаться, они должны ждать, пока третий не перейдет в режим ожидания. Позже мы увидим, как следует управлять параллельным выполнением потоков и их прямым взаимодействием.

Глава 4. Простое разрушение потока.  
  
Содержание:

* Проблемы завершения, остановки и разрушения потоков.
* Досрочная остановка потока.
* Событие OnTerminate.
* Контролируемая остановка потока - Подход 1.

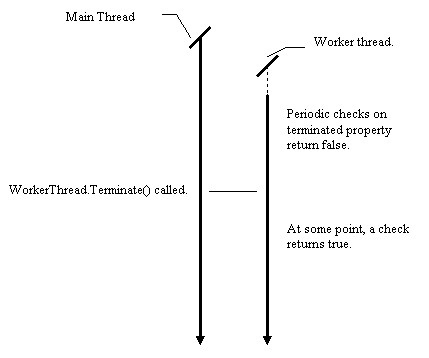
Проблемы завершения, остановки и разрушения потоков.  
  
В Главе 2 было упомянуто несколько проблем, возникающих при завершении потока, главные из них:

* Выход из потока, освобождение занятых ресурсов.
* Получение результатов из потока по его завершении.

Эти темы тесно взаимосвязаны. Если потоку не нужно передавать никакую информацию в основной поток VCL, когда он завершен, или если используются методы, описанные в предыдущей части (передача результатов прямо перед остановкой потока), тогда и нет необходимости основному потоку VCL заниматься очисткой порожденного потока. В этих случаях можно установить для потока FreeOnTerminate в True и позволить потоку выполнить освобождение собственной памяти самому. Помните только, что в этом случае пользователь может выйти из программы в любой момент, в результате чего все потоки завершатся, и вероятны непредвиденные последствия. Если поток только записывает что-то в память или связывается с другими частями приложения, то это не проблема. Если же он пишет данные в файл или работает с разделяемыми системными ресурсами, то такой метод неприемлем.   
Если поток должен обмениваться информацией с VCL после завершения, то следует обеспечить механизм синхронизации главного потока VCL с рабочим потоком, и основному потоку VCL придется выполнить очистку (вы должны написать код освобождения потока). Два механизма для этого будут описаны ниже.   
Следует помнить еще об одной вещи:

* Остановка потока до его естественного завершения.

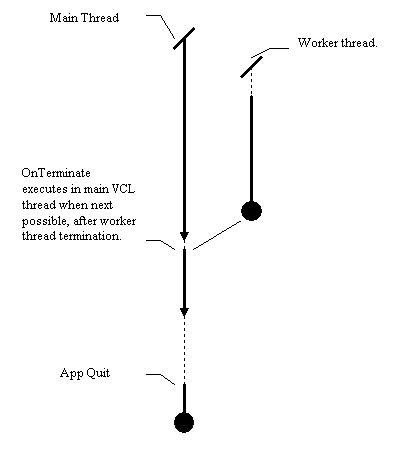
Это происходит довольно часто. Некоторые потоки, особенно те, что занимаются вводом-выводом, содержат в себе выполнение бесконечного цикла: программа всегда может получить новые данные, а поток должен быть готов к их приему и обработке до тех пор, пока программа работает.   
Итак, рассмотрим эти вопросы в обратном порядке...   
  
Досрочная остановка потока.  
  
В некоторых ситуациях одному потоку может потребоваться уведомить другой поток о своем завершении. Это обычно происходит, когда поток выполняет длительную операцию, и пользователь решает выйти из приложения, или операция должна быть прервана. TThread обеспечивает простой механизм для поддержки таких действий, а именно, метод **Terminate** и свойство **Terminated**. Когда поток создается, свойство Terminated установлено в False, а всякий раз, когда вызывается метод Terminate, свойство Terminated для этого потока устанавливается в True. Таким образом, на всех потоках лежит ответственность за периодическую проверку, не были ли они остановлены, и если это случается, за корректное завершение своей работы. Заметьте, что никакой крупномасштабной синхронизации при этом не происходит: когда один поток устанавливает свойство Terminated другого, нельзя предполагать, что другой поток тут же прочитает значение своего свойства Terminated и начнет процесс завершения. Свойство Terminated является просто флагом, говорящим "пожалуйста, завершайся как можно скорее". Диаграмма иллюстрирует эту ситуацию.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig5.gif)

При проектировании объектов потока стоит уделить внимание проверке свойства Terminated, если это может потребоваться. Если же ваш поток блокирован в результате действия любого из механизмов синхронизации, обсуждаемых ниже, вы можете перекрыть метод Terminate для его разблокировки. Не забудьте, что нужно вызывать унаследованный метод Terminate перед разблокированием вашего потока, если хотите, чтобы следующая проверка Terminated возвратила True. Вот пример - небольшая модификация

|  |
| --- |
| **Код** |
| function TPrimeThrd.IsPrime: boolean;  var   iter: integer;  begin   result := true;   if FTestNumber < 0 then   begin     result := false;     exit;   end;   if FTestNumber <= 2 then     exit;   iter := 2;   while (iter < FTestNumber) and (not terminated) do   begin     if (FTestNumber mod iter) = 0 then     begin       result := false;       {exit;}     end;     Inc(iter);   end; end; |

потока расчета простых чисел из предыдущей части с добавлением проверки свойства Terminated. Я полагаю, что допустимо, если поток вернет неверный результат при установленном свойстве Terminated.   
  
Событие OnTerminate.  
  
Событие OnTerminate происходит, когда поток в самом деле завершается. Оно *не случается*, когда вызывается метод потока Terminate. Это событие может быть весьма полезным, поскольку оно выполняется в контексте основного потока VCL, подобно методам, вызываемым с помощью Synchronize. Таким образом, если есть желание выполнять какие-то действия VCL с потоком, который автоматически освобождается по окончании, то обработчик этого события - прекрасное место для таких действий. Для большинства программистов, начинающих работать с потоками, это наиболее удобный путь получения данных из не-VCL потока без особых усилий, не требующий явных вызовов синхронизации.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig6.gif)

Как можно видеть на диаграмме, OnTerminate работает в основном так же, как и Synchronize, и семантически это почти идентично вызову Synchronize в конце потока. Основная польза от такой ситуации заключается в том, что используя флаг, например, "AppCanQuit" или счетчик работающих потоков в главном потоке VCL, можно обеспечить простые механизмы проверки того, что основной поток VCL завершается только тогда, когда все другие потоки остановлены. Существуют некоторые тонкости синхронизации, особенно если программист должен помещать вызов Application.Terminate в событие OnTerminate потока, но эти проблемы будут рассмотрены позже.   
  
Контролируемая остановка потока - Подход 1.  
  
В данном примере мы используем код для простых чисел из Главы 3 и модифицируем его так, чтобы пользователь не мог неумышленно закрыть приложение, когда выполняется рабочий поток. Это довольно просто. Фактически нам и не нужно модифицировать код потока. Мы просто добавим поле счетчика ссылок к основной форме, увеличивая его при создании потоков, создадим обработчик события OnTerminate, который будет уменьшать счетчик ссылок, и когда пользователь попытается закрыть программу, мы покажем, если потребуется, диалоговое окно предупреждения.   
Пример

|  |
| --- |
| **Код** |
| unit PrimeForm;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls;  type   TPrimeFrm = class(TForm)     NumEdit: TEdit;     SpawnButton: TButton;     ResultsMemo: TMemo;     procedure SpawnButtonClick(Sender: TObject);     procedure FormCreate(Sender: TObject);     procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);   private     { Private declarations }     FThreadRefCount: integer;     procedure HandleTerminate(Sender: TObject);   public     { Public declarations }   end;  var   PrimeFrm: TPrimeFrm;  implementation  uses PrimeThread;  {$R \*.DFM}  procedure TPrimeFrm.SpawnButtonClick(Sender: TObject);  var   NewThread: TPrimeThrd;  begin   NewThread := TPrimeThrd.Create(True);   NewThread.FreeOnTerminate := True;   try     with NewThread do     begin       TestNumber := StrToInt(NumEdit.Text);       Inc(FThreadRefCount);       OnTerminate := HandleTerminate;       Resume;     end;   except on EConvertError do     begin       NewThread.Free;       ShowMessage('That is not a valid number!');     end;   end; end;  procedure TPrimeFrm.FormCreate(Sender: TObject); begin   FThreadRefCount := 0; end;  procedure TPrimeFrm.FormCloseQuery(Sender: TObject; var CanClose: Boolean); begin   CanClose := true;   if FThreadRefCount > 0 then   begin     if MessageDlg('Threads active. Do you still want to quit?',       mtWarning, [mbYes, mbNo], 0) = mrNo then       CanClose := false;   end; end;  procedure TPrimeFrm.HandleTerminate(Sender: TObject); begin   Dec(FThreadRefCount); end;  end. |

показывает как несложно этого достичь: весь код, относящийся к отслеживанию числа работающих потоков, исполняется в основном потоке VCL, и этот код управляется событиями, как обычно и делается в Delphi-приложениях. В следующей части мы рассмотрим более сложный подход, который имеет некоторые преимущества при использовании продвинутых механизмов синхронизации.

Глава 5. Снова о разрушении потока. Тупик или зацикливание (Deadlock).   
  
Содержание:

* Метод WaitFor.
* Контролируемое завершение потока - Подход 2.
* Введение в обработку сообщений и отложенное уведомление.
* WaitFor может вызвать долгую задержку.
* Вы заметили ошибку?
* Как избежать такого тупика.

Метод WaitFor.  
  
OnTerminate, как обсуждалось в предыдущей части, полезно, если вы используете поток в режиме "выполнить и забыть", с автоматическим разрушением. Но что, если в некий момент выполнения главного потока VCL вы должны быть уверены, что все остальные потоки завершены? Решение состоит в использовании метода **WaitFor**, который пригодится в следующих случаях:

* Главному потоку VCL нужен доступ к объекту рабочего потока после его остановки для чтения или записи содержащихся в нем данных .
* Принудительное завершение потоков при закрытии программы неприемлемо.

Попросту говоря, когда поток А вызывает метод WaitFor потока B, сам он приостановливается, пока поток B не завершится. И когда поток А продолжит свое выполнение, можно быть уверенным, что результаты из потока B можно прочесть, и что объект потока B можно уничтожать. Обычно при завершении программы основной поток VCL вызывает Terminate всех вторичных потоков, и затем ожидает их завершения (WaitFor), после чего осуществляется выход из программы.   
  
Контролируемое завершение потока - Подход 2.  
  
В этом примере мы модифицируем код программы для простых чисел так, чтобы в каждый момент выполнялся только один поток, и программа перед выходом будет ждать завершения потока. Хотя в этой программе и не обязательно основному потоку ждать завершения других, но упражнение будет полезным и продемонстрирует несколько свойств WaitFor, которые не всегда желательны, а также проиллюстрирует пару довольно тонких моментов, которые могут быть упущены новичками в программировании потоков. Сначала код главной формы

|  |
| --- |
| **Код** |
| unit PrimeForm;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls, PrimeThread;  const   WM\_THREAD\_COMPLETE = WM\_APP + 5437; { Just a magic number }  type   TPrimeFrm = class(TForm)     NumEdit: TEdit;     SpawnButton: TButton;     ResultsMemo: TMemo;     procedure SpawnButtonClick(Sender: TObject);     procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);   private     { Private declarations }     FThread: TPrimeThrd;     procedure HandleThreadCompletion(var Message: TMessage); message WM\_THREAD\_COMPLETE;   public     { Public declarations }   end;  var   PrimeFrm: TPrimeFrm;  implementation  {$R \*.DFM}  procedure TPrimeFrm.HandleThreadCompletion(var Message: TMessage); begin   if Assigned(FThread) then   begin     FThread.WaitFor;     FThread.Free;     FThread := nil;   end; end;  procedure TPrimeFrm.SpawnButtonClick(Sender: TObject);  begin   if not Assigned(FThread) then   begin     FThread := TPrimeThrd.Create(True);     FThread.FreeOnTerminate := false;     try       with FThread do       begin         TestNumber := StrToInt(NumEdit.Text);         Resume;       end;     except on EConvertError do       begin         FThread.Free;         FThread := nil;         ShowMessage('That is not a valid number!');       end;     end;   end; end;  procedure TPrimeFrm.FormCloseQuery(Sender: TObject; var CanClose: Boolean); begin   CanClose := true;   if Assigned(FThread) then   begin     if MessageDlg('Threads active. Do you still want to quit?',       mtWarning, [mbYes, mbNo], 0) = mrNo then       CanClose := false;   end;   {Sleep(50000);}{Line C}   if CanClose then   begin     if Assigned(FThread) then     begin       FThread.Terminate;       FThread.WaitFor;       FThread.Free;       FThread := nil;     end;   end; end;  end. |

Можно увидеть несколько отличий от предыдущего примера:

* В начале модуля объявлено "магическое число" . Это относительный номер сообщения, а значение его не важно, главное, чтобы оно было уникальным.
* Вместо счетчика потоков мы следим только за одним потоком, которому отвечает переменная FThread главной формы.
* Мы хотим, чтобы в каждый момент исполнялся только один поток, поскольку имеется единственная переменная, указывающая на рабочий поток. Поэтому код перед созданием потока проверяет, нет ли уже запущенных потоков.
* Код создания потока не устанавливает свойство FreeOnTerminate в True, вместо этого основной поток VCL будет освобождать рабочий поток позже.
* У главной формы есть обработчик сообщения, который ждет завершения рабочего потока, и затем разрушает его.
* Соответственно, код, исполняемый, когда пользователь хочет закрыть программу, ожидает завершения рабочего потока и освобождает его.

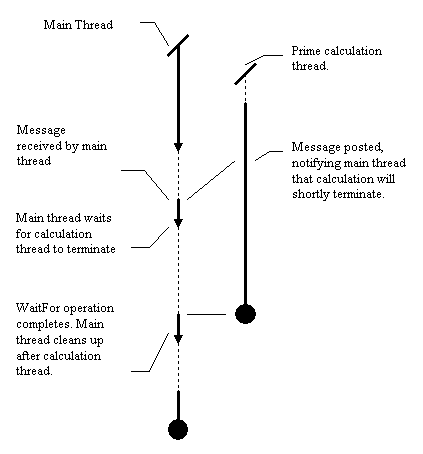
Учтите эти пункты, а здесь код рабочего потока

|  |
| --- |
| **Код** |
| unit PrimeThread;  interface  uses   Classes;  type   TPrimeThrd = class(TThread)   private     FTestNumber: integer;     FResultString: string;   protected     function IsPrime: boolean;     procedure UpdateResults;     procedure Execute; override;   public     property TestNumber: integer write FTestNumber;   end;  implementation  uses SysUtils, Dialogs, PrimeForm, Windows;  procedure TPrimeThrd.UpdateResults; begin   PrimeFrm.ResultsMemo.Lines.Add(FResultString); end;  function TPrimeThrd.IsPrime: boolean;  var   iter: integer;  begin   result := true;   if FTestNumber < 0 then   begin     result := false;     exit;   end;   if FTestNumber <= 2 then     exit;   iter := 2;   while (iter < FTestNumber) and (not terminated) do {Line A}   begin     if (FTestNumber mod iter) = 0 then     begin       result := false;       {exit;}     end;     Inc(iter);   end; end;  procedure TPrimeThrd.Execute; begin   if IsPrime then     FResultString := IntToStr(FTestNumber) + ' is prime.'   else     FResultString := IntToStr(FTestNumber) + ' is not prime.';   if not Terminated then {Line B}   begin     Synchronize(UpdateResults);     PostMessage(PrimeFrm.Handle, WM\_THREAD\_COMPLETE, 0, 0);   end; end;  end. |

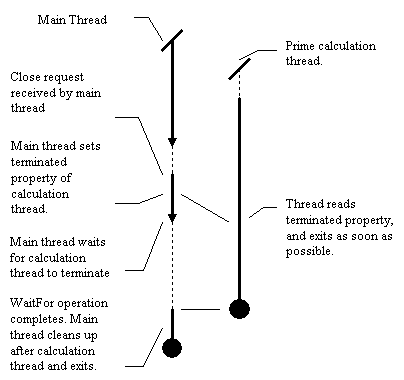
В нем тоже есть небольшие отличия от того, что было в Главе 3:

* Функция IsPrime теперь проверяет запросы на завершение потока, обеспечивая быстрый выход, если установлено свойство Terminated.
* Процедура Execute делает проверку на нештатное завершение.
* При нормальном завершении используется Synchronize для показа результатов и главной форме посылается сообщение - запрос на освобождение потока.

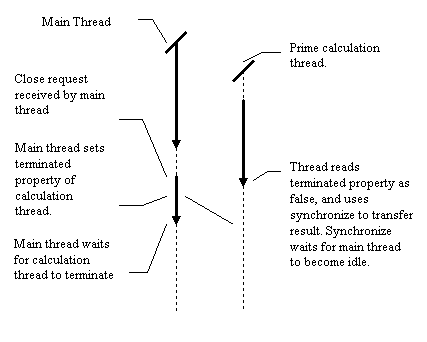
Введение в обработку сообщений и отложенное уведомление  
  
При нормальном ходе дел поток выполняется, использует Synchronize для показа результатов, а затем посылает сообщение главной форме. Это сообщение асинхронно: главная форма получит его в некоторый момент чуть позже. **PostMessage** не приостанавливает рабочий поток, он продолжает свою работу до завершения. Это очень полезное свойство: мы не можем использовать Synchronize для того, чтобы сообщить главной форме, что пора освободить поток, потому что тогда мы бы "вернулись" из вызова Synchronize в несуществующий уже поток. Вместо этого, происходит лишь уведомление (notification), вежливое напоминание главной форме о том, что следует освободить поток при первой возможности.   
В некоторый момент главный программный поток получает сообщение и выполняется его обработчик. Этот обработчик проверяет, существует ли поток, если существует, ждет, пока он закончит выполняться. Этот шаг необходим, поскольку, хотя рабочий поток и близок к завершению (после PostMessage операторов мало), но гарантии нет. Когда ожидание закончено, главный поток очищает рабочий поток.   
Диаграмма иллюстрирует первый случай. Ради упрощения детали Synchronize на диаграмме не приводятся. Кроме того, вызов PostMessage показан как происходящий несколько раньше окончания кода рабочего потока, чтобы продемонстрировать функционирование WaitFor.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig7.gif)

Позднее мы покажем преимущества посылки сообщений более детально. Сейчас достаточно сказать, что эта техника полезна при взаимодействии с главным потоком VCL.   
В случае принудительного завершения пользователь пытается выйти из программы и подтверждает, чтo хочет это сделать немедленно. Главный поток устанавливает свойство Terminated рабочего потока, что приводит к довольно быстрой его остановке, и затем ждет его завершения. После завершения, как и в предыдущем случае, производится очистка. Диаграмма иллюстрирует второй случай.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig8.gif)

Многие читатели в настоящий момент могут полностью удовлетвориться такой ситуацией. Однако подводные камни остаются, и, как часто бывает при рассмотрении многопоточной синхронизации, они проявляются в деталях.   
  
WaitFor может вызвать долгую задержку.  
  
Преимущество метода WaitFor является также и его крупнейшим недостатком: он переводит главный поток в состояние, в котором тот не может принимать сообщения. Это означает, что программа не может предпринять никаких операций, связанных обычно с обработкой сообщений: в состоянии такого ожидания приложение не будет перерисовываться, изменять размер формы или отвечать на внешние воздействия. Когда пользователь это заметит, он решит, что программ зависла. Это не беда в случае нормального завершения потока; вызывая PostMessage самым последним оператором рабочего потока, мы гарантируем, что главному потоку не придется долго ждать. В случае же нештатного завершения потока время, потраченное главным потоком на ожидание, зависит в основном от того, как часто рабочий поток проверяет свое свойство Terminate. Код PrimeThread содержит строку, помеченную "Line A". Если удалить "and not terminated", то вы можете поэкспериментировать с выходом из приложения во время исполнения длительного потока.   
В целом же проще писать потоки, которые регулярно проверяют свойство Terminated. Если это невозможно, тогда лучше выдавать пользователю предупреждение о потенциальной невосприимчивости в течение некоторого времени (подобно Microsoft Exchange.)   
  
Вы заметили ошибку? WaitFor и Synchronize: зацикливание.  
  
Задержка, вызываемая WaitFor - незначительная проблема по сравнению с другой. В приложениях которые используют как Synchronize, так и WaitFor, вполне возможно вызвать зависание, зацикливание приложения (deadlock, тупик). Тупиком можно считать случай, когда в приложении нет алгоритмических ошибок, но оно заторможено, не отзывается на действия пользователя. Обычно оно происходит, если потоки циклически ожидают друг друга. Поток А может ждать завершения потоком B некоторой операции, в то время как поток C ждет поток D, и т.д.. А вот поток D может ждать завершения некоторых действий потоком А. К сожалению поток А не может завершить операцию, поскольку он приостановлен. Это программный эквивалент проблемы "A: Сначала проезжайте Вы... B: Нет, Вы... A: Нет, я настаиваю!", которая приводит к автомобильным пробкам, когда примущественное право проезда не очевидно. Это поведение документировано и в файлах помощи VCL.  
В этом конкретном случае зацикливание может произойти для двух потоков, если вычислительный поток вызывает Synchronize прямо перед тем,как основной поток вызывает WaitFor. Тогда вычислительный поток будет ждать, пока основной поток не вернется в цикл обработки сообщений, а основной будет ждать завершения вычислительного. Произойдет зацикливание. Возможно также, что основной поток VCL вызовет WaitFor незадолго до вызова Synchronize рабочим потоком. Это тоже может привести к зацикливанию. К счастью, разработчики VCL предусмотрели перехват такой ошибки: в рабочем потоке возбуждается исключение, таким образом цикл прерывается, и поток завершается.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig9.gif)

Реализация примера делает это маловероятным. Рабочий поток вызывает Synchronize только при чтении свойства Terminated, если оно установлено в False, незадолго до окончания выполнения. Основной поток приложения устанавливает свойство Terminated прямо перед вызовом WaitFor. Таким образом, для того, чтобы произошло зацикливание, рабочий поток должен был бы определить, что свойство Terminated=False, выполнить Synchronize, и затем управление должно быть передано в основной поток точно в тот момент, когда пользователь подтвердил принудительный выход из программы.   
Несмотря на то, что в этом случае зацикливание маловероятно, события подобного типа явно могут привести к конфликтам. Все зависит от точных временных интервалов между событиями, которые могут меняться от запуска к запуску и от и от машины к машине. В 99.9% случаев принудительное закрытие сработает, но один раз из тысячи все может заблокироваться: этой проблемы следует избегать во что бы то ни стало. Читатель может вспомнить, что я прежде упоминал, что никакой серьезной синхронизации не происходит при чтении или записи свойства Terminated. Это означает, что невозможно использовать свойство Terminated для полного исключения указанной проблемы, как и доказывает предыдущая диаграмма.   
Любознательный читатель может захотеть воспроизвести проблему зацикливания. Это нетрудно сделать, осуществив следующие изменения в коде:

* Удалить "and not terminated" в строке Line A
* Заменить "not terminated" в строке Line B на "true".
* Удалить комментарий для строки Line C.

Теперь можно спровоцировать зацикливание, запустив поток, чье выполнение займет порядка 20 секунд, и попытавшись выйти из приложения сразу после создания потока. Вы можете также захотеть подобрать временной интервал, в течение которого главный поток приложения "засыпает", чтобы добиться "правильного" порядка событий:

* Пользователь запускает вычислительный поток.
* Пользователь пытается выйти и говорит "Да, я хочу выйти, несмотря на тот факт, что потоки еще работают".
* Главный поток приложения засыпает (строка C)
* Вычислительный поток попадает в конец исполняемого кода и вызывает Synchronize. (Достигается модификацией строк A и B).
* Главный поток приложения просыпается и вызывает WaitFor.

Как избежать такого тупика.  
  
Наилучший метод не допускать этой формы зацикливания - не использовать WaitFor и Synchronize в одном приложении. От WaitFor можно избавиться, применяя событие OnTerminate, как обсуждалось выше. Данный пример оказался довольно удачен в этом отношении, поскольку возвращаемые потоком результаты очень просты, так что мы можем избежать использования Synchronize. Используя WaitFor, основной поток может легально иметь доступ к свойствам рабочего потока после его завершения, и все, что нам нужно - переменная "result" для хранения текстовой строки, полученной в рабочем потоке. Необходимые модификации:

* Удаление метода потока "DisplayResults".
* Добавление подходящего свойства рабочего потока.
* Изменение обработчика в главной форме.

Изменения

|  |
| --- |
| **Код** |
| { Unit PrimeThread }  type   TPrimeThrd = class(TThread)   private     FTestNumber: integer;     FResultString: string;   protected     function IsPrime: boolean;     procedure Execute; override;   public     property TestNumber: integer write FTestNumber;     property ResultString: string read FResultString;   end;  procedure TPrimeThrd.Execute; begin   if IsPrime then     FResultString := IntToStr(FTestNumber) + ' is prime.'   else     FResultString := IntToStr(FTestNumber) + ' is not prime.';   if not Terminated then {Line B}     PostMessage(PrimeFrm.Handle, WM\_THREAD\_COMPLETE, 0, 0); end;  { Unit PrimeForm }  procedure TPrimeFrm.HandleThreadCompletion(var Message: TMessage); begin   if Assigned(FThread) then   begin     FThread.WaitFor;     ResultsMemo.Lines.Add(FThread.ResultString);     FThread.Free;     FThread := nil;   end; end; |

Обсуждение механизмов синхронизации, общих для всех 32-битовых версий Delphi, почти закончено. Я еще не рассмотрел методы TThread.Suspend и TThread.Resume. Они будут обсуждаться в Главе 10. В дальнейших частях исследуются средства, предоставляемые Win32 API и последними версиями Delphi. Я хотел бы предложить, чтобы, как только читатель освоится с основами работы с потоками в Delphi, он нашел время для изучения этих более продвинутых методов, поскольку они намного более гибкие, чем встроенные в Delphi, и позволяют программисту согласовывать работу потоков изящнее и эффективнее, а также уменьшают возможность написания кода, ведущего к зацикливанию.

Часть 6. Снова о синхронизации: Критические секции и мьютексы.  
  
Содержание:

* Ограничения Synchronize.
* Критические секции.
* Что это все значит для программиста на Delphi?
* На заметку.
* Могут ли данные пропасть или остаться недоступными в буфере?
* Как насчет сообщений out of date?
* Проблемы Flow Control и неэффективность списка.
* Мьютексы.

Ограничения Synchronize.  
  
У метода Synchronize есть несколько недостатков, благодаря которым он подходит лишь для простых многопоточных приложений.

* Synchronize полезен лишь при взаимодействии между рабочим потоком и основным потоком VCL.
* Использование Synchronize подразумевает,что рабочий поток ждет, пока основной поток VCL будет в состоянии ожидания, даже когда это не так уж и необходимо.
* Если приложение часто использует Synchronize, главный поток VCL становится "узким местом", и возникают проблемы с производительностью.
* Если Synchronize используется для непосредственного взаимодействия двух рабочих потоков, оба они могут быть приостановлены, ожидая главный поток.
* Synchronize может вызвать зацикливание, если главный поток VCL ожидает другие потоки.

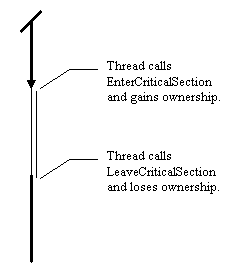
Правда, у Synchronize есть и одно преимущество над другими механизмами синхронизации:

* В методе, вызываемом с помощью Synchronize, может быть любой код, в том числе и потоко-небезопасный код VCL.

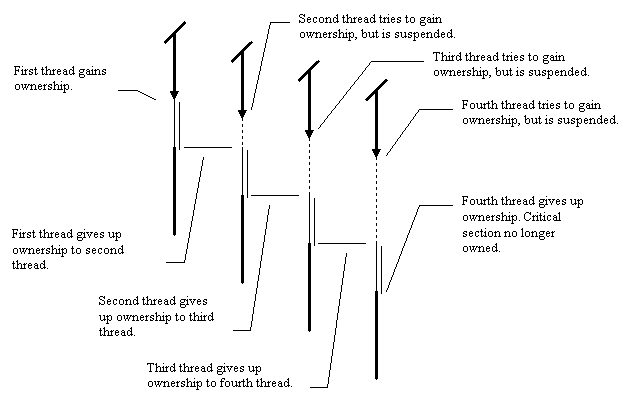
Важно помнить, для чего в приложении используются потоки. Основная причина для большинства Delphi-программистов в том, что они хотят, чтобы их приложение оставалось восприимчивым к действиям пользователя, пока выполняются длительные операции или используется передача данных с блокировкой или ввод-вывод. Это часто означает, что основной поток приложения должен выполнять краткие, основанные на событиях подпрограммы,а также отвечать за пользовательский интерфейс, т.е. обеспечивать прием ввода и отображать результаты. Другие потоки приложения будут выполнять "черную работу". Основываясь на этой философии, часто приходится делать так, что большая часть кода, выполняющегося в рабочих потоках, не использует код VCL, который не является потокобезопасным. Рабочие потоки могут выполнять операции с файлами или базами данных, но они редко используют потомков TControl. В этом свете Synchronize может привести к проблемам с производительностью.   
Многим потокам нужно связываться с VCL только в простых случаях, как например, передача потока (stream) данных, или выполнение запроса к базе данных и возвращение структуры данных как результат этого запроса. Как отмечено в Главе 3, при модификации общих данных нам нужно только поддерживать атомарность. В качестве простого примера можно рассмотреть поток данных (stream), который записывается рабочим потоком, и периодически читается основным потоком VCL. Нужно ли нам гарантировать, что поток VCL никогда не выполняется одновременно с рабочим? Конечно, нет! Все, что нужно обеспечить - так это то, что только один поток модифицирует этот разделяемый ресурс в каждый момент, таким образом устраняя условия для конфликтов, и делая операции с коллективным ресурсом атомарными. Такой режим называется взаимное исключение (**mutual exclusion**). Есть много примитивов синхронизации, которые могут быть использованы для осуществления такого режима. Простейшие из них - мьютекс (**Mutex**), встроенный в Win32, и близкие к мьютексам критические секции (**Critical Section**). Последние версии Delphi содержат класс, который инкапсулирует вызовы критических секций Win32. Этот класс здесь не обсуждается, поскольку он имеется не во всех 32-битовых версиях Delphi. У программистов, использущих этот класс, не должно быть больших трудностей в использовании соответствующих его методов для достижения таких же эффектов, как и обсуждаемые здесь.   
  
Критические секции.  
  
Критическая секция (Critical Section) позволяет добиться взаимного исключения. Win32 API поддерживает несколько операций с ними:

* InitializeCriticalSection.
* DeleteCriticalSection.
* EnterCriticalSection.
* LeaveCriticalSection.
* TryEnterCriticalSection (только Windows NT).

Операции InitializeCriticalSection и DeleteCriticalSection можно рассматривать подобно созданию и освобождению объектов в куче. Обычно имеет смысл проводить действия по созданию и разрушению критических секций в одном потоке, причем в наиболее долгоживущем. Очевидно, что все потоки, которые хотят синхронизовать доступ, используя критическую секцию, должны иметь дескриптор или указатель на нее. Это может быть достигнуто прямым путем через общую переменную или независимо, что возможно, поскольку критическая секция встроена в потокобезопасный класс, к которому имеют доступ оба потока.  
Когда объект критической секции создан, его можно использовать для контроля за общими ресурсами. Две главных операции - **EnterCriticalSection** и **LeaveCriticalSection**. В большей части литературы, касающейся темы синхронизации, эти операции называют **Wait** и **Signal**, или **Lock** и **Unlock** соответственно. Эти альтернативные термины используются также и для других примитивов синхронизации, где они имеют приблизительно эквивалентные значения. По умолчанию при создании критической секции ни один из потоков приложения не владеет ей (**ownership**). Чтобы управлять критической секцией, поток вызывает EnterCriticalSection, и если критическая секция еще не имеет владельца, то поток становится им. Затем поток обычно совершает действия с общими ресурсами (критическая часть кода, показана двойной линией), а когда заканчивает эти действия, то отказывается от владения критической секцией вызовом LeaveCriticalSection.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig10.gif)

Важно, что у критической секции в каждый момент времени может быть только один поток-владелец. Если поток пытается войти в критическую секцию, когда другой поток уже находится внутри нее, то он будет приостановлен, и возобновит свою работу, только когда другой поток выйдет из критической секции. Это обеспечивает нам требуемое взаимоисключение при работе с общим ресурсом. Может быть приостановлено в ожидании освобождения критической секции и несколько потоков, так что критические секции можно использовать для синхронизации более чем двух потоков. Вот что происходит в примере, если четыре потока пытаются получит доступ к одной критической секции примерно в одно время.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig11.gif)

Как видно из диаграммы, в каждый момент только один поток исполняет критический код, так что нет ни условий для конфликтов, ни проблем атомарности.   
  
Что это все значит для Delphi-программиста?  
  
Это означает, что если не нужно проводить действия с VCL, а только обеспечить доступ к данным и их изменение, при написании программ с использование потоков на Delphi программист избавлен от бремени TThread.Synchronize.

* Потоку VCL не нужно находиться в состоянии ожидания, чтобы рабочий поток мог модифицировать общие ресурсы, он только не должен быть внутри критической секции.
* Критические секции не знают и не заботятся о том, является ли их владелец основным потоком VCL или экземпляром объекта TThread, так как только один из потоков может использовать критическую секцию.
* Программист, разрабатывающий многопоточное приложение, может теперь безопасно (почти) использовать WaitFor, избегая проблемы зацикливания.

В последнем пункте указано "почти", поскольку все-таки возможно вызвать зацикливание точно таким же способом, как и прежде. Все, что нужно для этого сделать - вызвать WaitFor в основном потоке, когда он находится в критической секции. Как мы увидим позже, остановка потока на большой промежуток времени, когда он внутри критической секций - плохая идея. Теперь, когда я более-менее объяснил теорию, представлю еще один пример. Это чуть более изящная и интересная программа нахождения простых чисел. Вначале она пытается найти простые числа, стартуя с числа 2, и продвигаясь по числовому ряду вверх. Каждый раз, определив, что число простое, она обновляет общую структуру данных (список строк) и сообщает основному потоку, что к списку добавлены новые данные. Вот код главной формы

|  |
| --- |
| **Код** |
| unit PrimeForm2;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls, PrimeThread;  const   WM\_DATA\_IN\_BUF = WM\_APP + 1000;   MaxMemoLines = 20;  type   TPrimeFrm = class(TForm)     ResultMemo: TMemo;     StartBtn: TButton;     StartNumEdit: TEdit;     StopBtn: TButton;     procedure StartBtnClick(Sender: TObject);     procedure StopBtnClick(Sender: TObject);     procedure FormClose(Sender: TObject; var Action: TCloseAction);   private     { Private declarations }     FStringSectInit: boolean;     FPrimeThread: TPrimeThrd2;     FStringBuf: TStringList;     procedure UpdateButtons;     procedure HandleNewData(var Message: TMessage); message WM\_DATA\_IN\_BUF;   public     { Public declarations }     StringSection: TRTLCriticalSection;     property StringBuf: TStringList read FStringBuf write FStringBuf;   end;  var   PrimeFrm: TPrimeFrm;  implementation  {$R \*.DFM}  procedure TPrimeFrm.UpdateButtons; begin   StopBtn.Enabled := FStringSectInit;   StartBtn.Enabled := not FStringSectInit; end;  procedure TPrimeFrm.StartBtnClick(Sender: TObject); begin   if not FStringSectInit then   begin     InitializeCriticalSection(StringSection);     FStringBuf := TStringList.Create;     FStringSectInit := true;     FPrimeThread := TPrimeThrd2.Create(true);     SetThreadPriority(FPrimeThread.Handle, THREAD\_PRIORITY\_BELOW\_NORMAL);     try       FPrimeThread.StartNum := StrToInt(StartNumEdit.Text);     except       on EConvertError do FPrimeThread.StartNum := 2;     end;     FPrimeThread.Resume;   end;   UpdateButtons; end;  procedure TPrimeFrm.StopBtnClick(Sender: TObject); begin   if FStringSectInit then   begin     with FPrimeThread do     begin       Terminate;       WaitFor;       Free;     end;     FPrimeThread := nil;     FStringBuf.Free;     FStringBuf := nil;     DeleteCriticalSection(StringSection);     FStringSectInit := false;   end;   UpdateButtons; end;  procedure TPrimeFrm.HandleNewData(var Message: TMessage); begin   if FStringSectInit then {Not necessarily the case!}   begin     EnterCriticalSection(StringSection);     ResultMemo.Lines.Add(FStringBuf.Strings[0]);     FStringBuf.Delete(0);     LeaveCriticalSection(StringSection);     {Now trim the Result Memo.}     if ResultMemo.Lines.Count > MaxMemoLines then       ResultMemo.Lines.Delete(0);   end; end;  procedure TPrimeFrm.FormClose(Sender: TObject; var Action: TCloseAction); begin   StopBtnClick(Self); end;  end. |

Это довольно похоже на предыдущие примеры в том, что касается создания потока, но есть и несколько дополнительных полей основной формы, которые следует создать. StringSection - это критическая секция, которая контролирует доступ к ресурсам, разделяемым между потоками. FStringBuf - список строк, который выступает буфером между основной формой и рабочим потоком. Рабочий поток посылает результаты в основную форму, добавляя их к этому списку, который является единственным общим ресурсом в этой программе. И наконец, имеется логическая переменная, FStringSectInit. Эта переменная служит для проверки того, что необходимые объекты синхронизации реально созданы прежде, чем их начали использовать. Общие ресурсы создаются, когда мы запускаем рабочий поток, и уничтожаются сразу после того, как мы убедимся, что рабочий поток завершился. Заметьте, что поскольку список строк, выступающий в роли буфера, распределен динамически , мы *должны* использовать WaitFor при уничтожения потока, чтобы убедиться, что рабочий поток перестал использовать буфер до его освобождения.   
Мы можем использовать WaitFor в этой программе, не беспокоясь о зацикливании, поскольку можно доказать, что никогда не бывает ситуации, в которой оба потока ждут друг друга. Доказать это просто:

1 Рабочий поток ждет только при попытке получить доступ к критической секции.   
2 Основной поток программы ждет только при ожидании завершения рабочего потока.   
3 Основной поток программы не ожидает, когда он завладеет критической секцией.   
4 Если рабочий поток ожидает критической секции, главная программа освободит критическую секцию до того, как она будет ждать рабочий поток.

Вот код рабочего потока

|  |
| --- |
| **Код** |
| unit PrimeThread;  interface  uses   Classes, Windows;  type   TPrimeThrd2 = class(TThread)   private     { Private declarations }     FStartNum: integer;     function IsPrime(TestNo: integer): boolean;   protected     procedure Execute; override;   public     property StartNum: integer read FStartNum write FStartNum;   end;  implementation  uses PrimeForm2, SysUtils;  function TPrimeThrd2.IsPrime(TestNo: integer): boolean;  var   iter: integer;  begin   result := true;   if TestNo < 0 then     result := false;   if TestNo <= 2 then     exit;   iter := 2;   while (iter < TestNo) and (not terminated) do   begin     if (TestNo mod iter) = 0 then     begin       result := false;       exit;     end;     Inc(iter);   end; end;  procedure TPrimeThrd2.Execute;  var   CurrentNum: integer;  begin   CurrentNum := FStartNum;   while not Terminated do   begin     if IsPrime(CurrentNum) then     begin       EnterCriticalSection(PrimeFrm.StringSection);       PrimeFrm.StringBuf.Add(IntToStr(CurrentNum) + ' is prime.');       LeaveCriticalSection(PrimeFrm.StringSection);       PostMessage(PrimeFrm.Handle, WM\_DATA\_IN\_BUF, 0, 0);     end;     Inc(CurrentNum);   end; end;  end. |

Рабочий поток последовательно пробегает положительные целые числа, пытаясь найти простые. В случае успеха он получает доступ к критической секции, модифицирует буфер, выходит из критической секции, затем посылает сообщение в основную форму, указывая, что в буфере есть данные.   
  
На заметку.  
  
Этот пример более сложен, чем предыдущие, поскольку у нас есть довольно большой буфер между двумя потоками, и в результате появляются различные проблемы, которые нужно учесть и избавиться от них, а также некоторые особенности кода, имеющие дело с необычными ситуациями. Сведем эти пункты вместе:

* Могут ли данные пропасть или остаться недоступными в буфере?
* Как насчет запоздавших (out of date) сообщений ?
* Проблемы Flow Control.
* Неэффективность списка строк, статическое и динамическое задание размера.

Могут ли данные пропасть или остаться недоступными в буфере?  
  
Рабочий поток указывает основному программному потоку, что в буфере есть данные, которые нужно обрабатывать, посылая ему сообщение. Важно отметить, что при использовании таким образом системы обмена сообщениями Windows синхронизация потоков никак не связывает конкретное сообщение с конкретной модификацией буфера . К счастью, в этом случае причинно-следственная связь работает в нашу пользу: когда буфер обновляется, сообщение посылается *после* модификации. Это означает, что основной программный поток всегда получает сообщение о модификации буфера после этой модификации. Следовательно, не может случиться так, что данные останутся в буфере на неопределенное время. Если данные сейчас в буфере, то рабочий и основной потоки находятся в процессе посылки или получения сообщения о модификации буфера. Заметьте, что если бы рабочий поток посылал сообщение прямо перед обновлением буфера, могло бы случиться, что основной поток обработал это сообщение, прочитав буфер, прежде, чем рабочий добавил в буфер самые последние результаты, а это означало бы, что последний результат мог бы остаться в буфере в течение некоторого времени.   
  
Как насчет запоздавших сообщений (out of date)?   
  
Законы причины и следствия в предыдущем случае работали хорошо, но, к несчастью, существует также и обратная проблема . Если основная поток занят обновлением долго, возможно, что сообщения выстроятся в очередь, так что мы получим обновления намного позже того, как рабочий поток пошлет сообщения. В большинстве ситуаций это не составит проблем. Тем не менее, надо рассмотреть один частный случай, когда пользователь останавливает рабочий поток или непосредственно, нажав кнопку "Stop", или косвенно, закрывая программу. В этом случае вполне возможно, что основной поток VCL завершает рабочий поток, удаляет все объекты синхронизации и буфера, а затем последовательно получает сообщения, которые находились в очереди в течение некоторого времени. В данном примере я проверял наличие этой проблемы, убеждаясь, что перед обработкой сообщения критическая секция и буферные объекты все еще существуют (строка кода с комментарием *Not necessarily the case!* ). Этого метода обычно достаточно для большинства приложений.   
  
Проблемы Flow Control и неэффективность списка.  
  
В Главе 2 я утверждал, что при создании потока никакой неявной синхронизации не существует. Это было очевидно уже в первых примерах и продемонстрировано проблемами переключения потоков (проблема синхронизации состояния потоков) . Такая же проблема существует и для *синхронизации скорости*. Ничто в последнем примере не гарантирует, что рабочий поток будет выдавать результаты достаточно медленно для того, чтобы основной поток VCL был способен успеть их отобразить. Фактически, если программа выполняется так, что рабочий поток начинает поиск небольших простых чисел, вполне вероятно, что при равных долях времени процессора рабочий поток намного опередит поток VCL. Эта проблема решается посредством **flow control** (управления потоками).   
Flow control называют метод, при котором скорость выполнения нескольких потоков сбалансирована так, что скорость ввода в буфер и скорость чтения из него примерно равны. Последний пример очень прост, но подобное происходит и в многих других случаях. Почти каждый механизм ввода-вывода или передачи данных между потоками или процессами использует какой-либо способ управления потоками. В простых случаях можно разрешать только одному куску данных находиться в процессе передачи, задерживая или **источник** (поток, который выводит данные в буфер) или **приемник** (поток, который их читает). В более сложных случаях потоки могут выполняться на разных машинах, и "буфер" может состоять из внутренних буферов в каждой машине и буферных возможностей сети, их соединяющей. Большая часть протокола TCP занимается как раз таким управлением потоками. Каждый раз, когда вы загружаете web-страницу, протокол TCP согласует передачу между двумя компьютерами, гарантируя, что независимо от относительной скорости процессора и дисков, вся передача данных происходит со скоростью, с которой обе машины могут справиться[1]. В последнем примере сделана довольно грубая попытка управления потоками. Приоритет рабочего потока установлен так, чтобы планировщик отдавал основному потоку VCL предпочтение перед рабочим всякий раз, когда оба не простаивают. Под управлением Win32 проблема снимается, но абсолютной гарантии нет.   
Еще один аспект управления потоками - неограниченный размер буфера в последнем примере. Во-первых, это создает проблему неэффективности - основной поток должен выполнить много перемещений памяти при удалении первого элемента большого списка строк, а во-вторых, это означает, что с использованием описанного метода управления потоками буфер может расти без предела. Попробуйте удалить строку, которая устанавливает приоритет потока. Вы увидите, что рабочий поток выдает результаты быстрее, чем поток VCL может их обработать, что приводит к увеличению списка. Это замедляет поток VCL еще сильнее (так как действия по удалению строк происходят дольше для большого списка), и проблема усугубляется. В конечном вы увидите, что список становится достаточно большим, заполняет всю память, машина начинает сбоить, и все останавливается. Фактически при испытании примера я не мог заставить Delphi реагировать на запрос выхода из программы и пришлось прибегать к использованию менеджера задач Windows NT, чтобы закрыть процесс!   
Хотя эта программа на первый взгляд и проста, но она иллюстрирует большое количество потенциальных нежелательных эффектов. Более устойчивые решения этих проблем обсуждаются во последующих главах данного руководства.   
  
Мьютексы.  
  
Читатель может подумать, что я потратил столько времени на объяснение критических секций, и совсем забыл о мьютексах. Но это не так - просто мьютексы не представляют собой никаких новые концепций. Мьютекс работает точно так же, как и критическая секция. Единственное различие в реализациях Win32 - в том, что критическая секция ограничена использованием в пределах только одного процесса. Если у вас есть единая программа, которая использует несколько потоков, то критическая секция - легкий и удобный способ обеспечения ваших потребностей. Тем не менее, при написании DLL часто возможно использование DLL несколькими разными процессами одновременно. В этом случае вы должны использовать вместо критических секций мьютексы. Хотя Win32 API обеспечивает более обширный диапазон функций для работы с мьютексами и другими объектами синхронизации, чем будет рассмотрено нами, следующие функции аналогичны функциям, приведенным выше для критических секций:

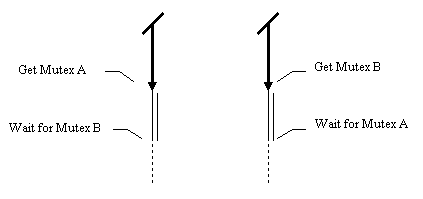
* CreateMutex / OpenMutex
* CloseHandle
* WaitForSingleObject(Ex)
* ReleaseMutex

Эти функции хорошо документированы в справке Win32 API, и будут более детально обсуждаться позже.   
  
[1] Протокол TCP выполняет также много других странных и удивительных функций, среди которых, например, копирование с потерей информации или оптимизация размеров окна так, чтобы поток данных соответствовал возможностям не только обеих машин, но также и связывающей их сети, минимизируя задержки и увеличивая пропускную способность. Он также содержит back-off алгоритмы для гарантии того, что несколько TCP-соединений могут разделять одно физическое соединение без монополизации физического ресурса одним из них.

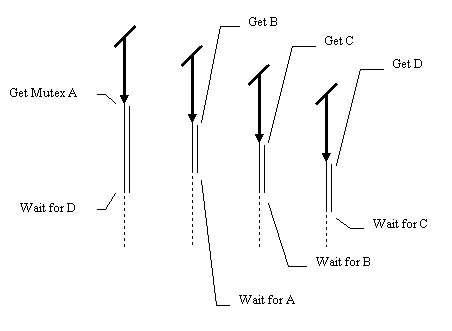
Глава 7. Программирование с использованием мьютексов. Управление конкуренцией.  
  
Содержание:

* Пора позаботиться о стиле.
* Тупик из-за упорядочения мьютексов.
* Избавляемся от зацикливания потоков путем ожидания.
* Избавляемся от зацикливания, устанавливая упорядочение захвата мьютексов.
* Из огня да в полымя!
* Избавляемся от зацикливания "ленивым способом", давая Win32 сделать это за нас.
* Атомарность составных операций - управление конкуренцией оптимистическим и пессимистическим образом.
* Оптимистическое управление.
* Пессимистическое управление.
* Избавляемся от недостатков в схеме блокировки.
* Еще не все ясно? Можно и попроще!

Пора позаботиться о стиле?  
  
Большинство представленных до сих пор в этом руководстве примеров было довольно грубыми. При проектировании многократно используемых компонентов или структуры большого многопоточного приложения, метод "полета вслепую" не подходит. Разработчик приложений или компонентов должен создавать классы со встроенными средствами обеспечения потокобезопасности, то есть классы, к которым возможен доступ из других потоков, содержащие подходящие внутренние механизмы, гарантирующие сохранность данных. Для этого разработчик компонентов должен позаботиться о решении некоторых проблем, которые возникают при использовании мьютексов в очень сложных приложениях. Если вы впервые пытаетесь написать потокобезопасный класс, не откладывайте из-за кажущейся сложности изучение вопросов, рассматриваемых в этой главе. Довольно часто может быть принято упрощенное решение, которое ценой некоторой эффективности позволяет избежать многих упомянутых здесь проблем. Заметьте, что решения, в которых упоминаются мьютексы, ообычно так же хорошо применимы и к критическим секциям. Я для краткости не буду каждый раз это отмечать.   
  
Тупик из-за упорядочения мьютексов.  
  
Если в программе имеется несколько мьютексов, то бывает нетрудно ввести ее в тупик неправильным кодом синхронизации. Наиболее часто это происходит, если существует циклическая зависимость порядка, в котором захватываются мьютексы. В академической литературе это часто называют проблемой **трапезы философов**. Как мы видели ранее, критерий возникновения зацикливания состоит в том, что потоки ждут, пока другой поток освободит объект синхронизации. Простейший пример - для двух потоков, один из которых захватывает мьютекс A до захвата мьютекса B, а другой захватывает мьютекс B до захвата мьютекса A.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig12.gif)

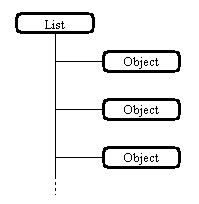
Конечно, вполне возможно получить зацикливание программы и более сложным образом, с циклической цепочкой зависимостей, как показано ниже для четырех потоков и четырех мьютексов от A до D.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig13.gif)

Очевидно, что подобные ситуации недопустимы в большинстве приложений. Существует несколько способов решения этой задачи и методы снятия проблем с такими зависимостями, что и позволяет избавиться от зацикливания.   
  
Избавляемся от зацикливания потоков путем ожидания.  
  
Функции Win32, работающие с мьютексами, *не требуют*, чтобы поток вечно ждал воэможности захвата объекта мьютекса. Функция WaitForSingleObject позволяет определить время, в течение которого поток будет ждать. По его истечении поток будет разблокирован, и функция вернет код ошибки, показывающий, что время ожидания вышло. При использовании мьютексов для обеспечения доступа к критическому участку кода обычно не предполагается, что потоку придется ждать очень долго, так что установка периода ожидания (time-out) в пределах нескольких секунд вполне разумна. Если ваш поток использует этот метод, то он должен, конечно, правильно обрабатывать ошибки, например, повтором попытки или отказом от действия. При использовании критической секции такой возможности нет, так как функции ожидания критической секций ждут бесконечно.   
  
Избавляемся от зацикливания, устанавливая упорядочение захвата мьютексов.  
  
Хотя возможность справиться с проблемами приобретения мьютекса и есть, лучше все-таки заранее гарантировать, чтобы тупиковые ситуации вообще не возникли. Поскольку такое зацикливание вызвано циклическими зависимостями, оно может быть устранено насильным упорядоченим приобретения мьютексов. Это упорядочение очень просто. Пусть у нас есть программа с мьютексами *M1*, *M2, M3*, ... *Mn*, где одним или несколькими мьютексами могут владеть потоки программы.

* Зацикливание не произойдет, если потоки, которые пытаются захватить некоторый мьютекс *Mx*, в этот момент не владеют никакими мьютексами "более высокого приоритета", т.е., *M(x+1)* ... *Mn*.

Звучит несколько абстрактно? Рассмотрим простой конкретный пример. В этой части данной главы я упоминаю "блокировку" и "разблокирование" объектов. Эта терминология вполне уместна, когда мьютекс связан с куском данных, и требуется атомарный доступ к этим данным. Следует отметить, что это также означает, что каждый поток, получает доступ (захватывает) мьютекс до доступа к объекту, и освобождает мьютекс после операций с ним: такие действия уже обсуждались ранее, отличие только в терминах, которые более подходят в случае объектно-ориентированной (OO) модели. В этом смысле **Object.Lock** можно рассматривать как эквивалент **EnterCriticalSection(Object.CriticalSection)** или, возможно, **WaitForSingleObject(Object.Mutex,INFINITE)**.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig14.gif)

У нас есть структура данных - список, к которому имеют доступ несколько потоков. В списке несколько объектов, у каждого из которых есть собственный мьютекс. На данный момент мы считаем, что структура списка статическая, не изменяется, и, таким образом, потоки могут проводить чтение без какой-либо блокировки. Потоки, работающие с этой структурой данных, могут совершить одно из следующих действий:

* Чтение элемента с блокировкой его, чтением данных, затем разблокировка.
* Запись в элемент с блокировкой его, записью данных, затем разблокировка.
* Сравнение двух элементов, с поиском в списке, блокировкой обоих элементов, затем осуществляется сравнение.

Простой псевдокод для этих функций, без учета приведения типов, обработки исключений и других вторичных проблем, может выглядеть примерно так

|  |
| --- |
| **Код** |
| function Read(L: TList; Index: integer): integer; begin   if (Index > 0) and (L.Count > Index) then   begin     with L.Items[Index] do     begin       Lock;       Result := Value;       Unlock;     end;   end   else     raise ENotFound; end;  procedure Write(L: TList; Index: integer; NewVal: integer); begin   if (Index > 0) and (L.Count > Index) then   begin     with L.Items[Index] do     begin       Lock;       Value := NewVal;       Unlock;     end;   end   else     raise ENotFound; end;  function Compare(L: TList; Ind1, Ind2: integer): integer; begin   if (Ind1 > 0) and (Ind2 > 0) and (L.Count > Ind1) and (L.Count > Ind2) then   begin     L.Items[Ind1].Lock;     L.Items[Ind2}.Lock;     Result := L.Items[Ind2].Value - L.Items[Ind1].Value;     L.Items[Ind2].Unlock;     L.Items[Ind1].Unlock;   end   else     raise ENotFound; end; |

Представим момент, в который потоку нужно сравнить элементы списка X и Y. Если поток всегда блокирует X, а потом Y, то может возникнуть зацикливание, если одному потоку нужно сравнивать элементы 1 и 2, а другому потоку - сравнивать 2 и 1. Одно из простых решений состоит в том, чтобы всегда блокировать сначала элемент с меньшим номером, или сортировать входные индексы, осуществлять блокировку, и правильно получать результаты сравнения. Однако более интересная ситуация создается, если объект содержит информацию о другом объекте, с которым требуется сравнение. В этом случае поток может блокировать первый объект, получить индекс второго объекта в списке, найти, что он располагается в списке ниже, блокировать его, и затем произвести сравнение. Все очень легко. Проблема возникает, когда второй объект находится в списке выше первого. Мы не можем блокировать его немедленно, так как это приведет к тупику. Теперь нам придется разблокировать первый объект, блокировать второй, и затем снова блокировать первый объект. Вот так удастся избежать тупика. Вот пример процедуры косвенного сравнения, представляющей данный подход

|  |
| --- |
| **Код** |
| function CompareIndirect(L: TList; Ind1: integer): integer; var   Ind2: integer;  begin   if (Ind1 > 0) and (L.Count > Ind1) then   begin     L.Items[Ind1].Lock;     Ind2 := L.Items[Ind1];     Assert(Ind2 <> Ind1); {I'm not even going to consider this nasty case in any more detail!}     if Ind2 > Ind1 then       L.Items[Ind2].Lock     else     begin       L.Items[Ind1].Unlock;       L.Items[Ind2].Lock;       L.Items[Ind1].Lock;     end;     Result := L.Items[Ind2].Value - L.Items[Ind1].Value;     L.Items[Ind1].Unlock;     L.Items[Ind2].Unlock;   end   else     raise ENotFound; end; |

Из огня да в полымя!  
  
Хотя так удается избежать тупика, появляются новые проблемы. При задержке между разблокировкой и вторичной блокировкой первого объекта мы не можем быть уверены, что другой поток за нашей спиной не модифицирует первый объект. Все это потому, что мы совершали составную операцию: действие в целом теперь уже не атомарно. Решения этой проблемы обсуждаются ниже.   
  
Избавляемся от зацикливания "ленивым способом", давая Win32 сделать это за нас.  
  
Зная, что могут появиться такие проблемы, разработчики операционных систем Microsoft предусмотрели еще один путь их решения через другую функцию синхронизации Win32: WaitForMultipleObjects(Ex). Эта функция позволяет программисту ожидать и захватывать многочисленные объекты синхронизации (включая мьютексы) одновременно. В частности, она позволяет потоку ожидать, пока один или все из набора объектов не будут свободны (signalled) (в случае мьютексов - у них не будет владельцев), и тогда захватить объекты. Большое преимущество этого способа состоит в том, что если два потока ожидают мьютексы A и B, то не имеет значения порядок, в котором они идут в наборе объектов, подлежащих ожиданию, и либо ни один из объектов не будут захвачен, либо все они будут захвачены атомарно, так что зацикливание становится невозможным.   
У этого метод также имеется несколько недостатков. Первый недостаток в том, что поскольку *все* объекты синхронизации должны быть свободны прежде, чем *любой* из них будет захвачен, то возможно, что поток, ждущий много объектов, долгое время не сможет захватить их, если другие потоки владеют какими-нибудь из этих же объектов синхронизации поодиночке. Например, самый левый поток на диаграмме мог бы ожидать мьютексы A, B и C, в то время как другие три потока захватывали каждый мьютекс отдельно. В самом неблагоприятном случае поток, ожидающий освобождения многочисленных объектов, вообще никогда не сможет их захватить.   
Второй недостаток в том, что *все-таки* возможно попасть в тупик, но на этот раз не с отдельными мьютексами, а с набором из нескольких мьютексов сразу! Следующая диаграмма (прим.переводчика: диаграмма в оригинале отстутствует) иллюстрирует ситуацию, которая, несомненно, приведет к тупику, как и в примере, представленном в начале этой главы.   
Третий недостаток этого метода, как и метода исключения тупика путем "ожидания" - невозможно использовать эти функции, если вы применяете критические секции; функция EnterCriticalSection не позволяет задать время ожидания, и не возвращает кода ошибки.   
  
Атомарность составных операций - управление конкуренцией оптимистическим и пессимистическим образом.  
  
Рассматривая выше упорядочение мьютексов, мы встречались с ситуацией, когда было нужно разблокировать и потом заново блокировать объект для правильного упорядочения мьютексов. Это означает, что над объектом совершалось несколько действий, и блокировка его снималась в несколько стадий.   
  
Оптимистическое управление.  
  
Один из путей работы с этой проблемой - предположить, что конфликт потоков очень маловероятен, и просто проверять, не случился ли он, и возвращать ошибку если это произошло. Часто это вполне работоспособный метод решения проблемы в сложных ситуациях, если "загрузка" структуры данных различными потоками не слишком высока. В случае, представленном ранее, мы можем тривиально узнать о наличии этого конфликта, храня локальную копию данных, и проверяя, что данные верны после разблокировки обоих объектов в требуемом порядке. Вот измененная процедура

|  |
| --- |
| **Код** |
| function CompareIndirect(L: TList; Ind1: integer): integer; var   Ind2: integer;   TempValue: integer;  begin   if (Ind1 > 0) and (L.Count > Ind1) then   begin     L.Items[Ind1].Lock;     Ind2 := L.Items[Ind1];     Assert(Ind2 <> Ind1); {I'm not even going to consider this nasty case in any more detail!}     if Ind2 > Ind1 then       L.Items[Ind2].Lock     else     begin       TempValue := L.Items[Ind1].Value;       L.Items[Ind1].Unlock;       L.Items[Ind2].Lock;       L.Items[Ind1].Lock;     end;     if TempValue := L.Items[Ind1].Value then       Result := L.Items[Ind2].Value - L.Items[Ind1].Value     else       {Perhaps some retry mechanism?};     L.Items[Ind1].Unlock;     L.Items[Ind2].Unlock;   end   else     raise ENotFound; end; |

С более сложными структурами данных можно иногда прибегать к использованию глобально уникальных идентификаторов или меток версии элементов данных. Личное примечание: я работал вместе с группой других студентов над университетским проектом, и этот метод зарекомендовал себя очень хорошо: метка-число последовательно увеличивалась всякий раз, когда часть данных была изменена (в этом случае данные состояли из записей в многопользовательском дневнике). Данные во время чтения блокировались и после этого показывались пользователю, и если пользователь редактировал данные, то число сравнивалось с тем, что пользователь получал при последнем чтении, и коррекция не принималась, если числа не совпадали.   
  
Пессимистическое управление.  
  
Мы можем использовать и несколько другой подход к этой проблеме, считая, что список, вероятно, будет модифицирован, и, таким образом, требуется его блокировка. Все действия, которые читают или пишут в список, включая поиск, должны его сначала блокировать. Это обеспечивает альтернативное решение проблемы корректной блокировки нескольких объектов списка. Давайте снова рассмотрим действия, которые мы хотим выполнять, обратив внимание на несколько модифицированную схему блокировки.   
Поток может читать и модифицировать содержимое объекта, имеющегося в списке, но не изменять размещение его в списке и не удалять объект. Эта операция может занять много времени, и мы не хотим воспрепятствовать работе других потоков, которые могут захотеть обратиться к другим объектам, так что поток, модифицирующий объект, должен выполнить следующие действия:

* Блокировать список.
* Найти объект в списке.
* Блокировать объект.
* Разблокировать список.
* Провести действия над объектом.
* Разблокировать объект.

Это прекрасно, потому что даже если операция чтения или записи будет длительной, весь список не блокируется надолго, и другие потоки без задержки могут изменять другие объекты списка.   
Поток может уничтожить объект, реализуя такой алгоритм:

* Блокировать список.
* Блокировать объект.
* Удалить объект из списка.
* Разблокировать список.
* Удалить объект (учитывая возможные ограничения при уничтожении блокированного мьютекса).

Заметьте, что возможно разблокировать список до окончательного удаления объекта, поскольку мы убрали объект из списка, и знаем, что никакие другие действия над объектом или списком не осуществляются (и тот, и другой блокированы).   
Мы подошли к интересной части. Поток может сравнить два объекта, выполняя более простой алгоритм, чем приведенные выше:

* Блокировать список.
* Найти в списке первый объект.
* Блокировать первый объект.
* Найти в списке второй объект.
* Блокировать второй объект.
* Разблокировать список.
* Провести сравнение.
* Разблокировать оба объекта (в любом порядке).

Заметьте, что в операции сравнения я не устанавливал никаких ограничений на порядок, в котором осуществляется блокировка объектов. Не приведет ли это к тупику? Приведенным алгоритмам не нужно следовать никаким критериям, представленным в начале этой главы, чтобы избежать тупика, но все-таки он не случится. Этого не происходит, поскольку когда поток блокирует мьютекс объекта, он уже владеет мьютексом списка, и, таким образом, блокирует несколько объектов, не освобождая мьютексы списка, так что составная блокировка нескольких объектов становится атомарной. В результате мы можем пересмотреть вышеуказаные критерии:

* Зацикливание не произойдет, если потоки, которые пытаются захватить некоторый мьютекс *Mx*, в этот момент не владеют никакими мьютексами "более высокого приоритета", т.е., *M(x+1)* ... *Mn*.
* Кроме того, оно не случится, и если мьютексы захватываются не в указанном порядке (когда нарушен первый критерий), и для любой группы мьютексов, участвующей в неупорядоченной блокировке, все наборы блокировочных операций над этими мьютексами атомарны, причем действия по блокировке следует заключать в критическую секцию (полученную блокировкой другого мьютекса).

Избавляемся от недостатков в схеме блокировки.  
  
Сейчас следует отметить, что вышеприведенный пример характерен для кода блокировки, который очень чувствителен к упорядочению. Прежде всего это должно показать, что при разработке нетривиальных схем блокировки надо обращать особое внимание на порядок событий.   
Если вы можете быть уверены, что ваша программа будет работать только под Windows NT (или 2K), то Windows API на самом деле предоставляет еще одно решение проблемы составных действий при разблокировке и новой блокировке объектов. Функция API **SignalObjectAndWait** позволяет вам атомарно освобождать (signal) один объект синхронизации и ждать другого. Сохраняя эти два действия атомарными, вы можете передать состояние блокировки одного объекта на другой, в то же время гарантируя, что никакие другие потоки не изменят состояние объектов во время передачи. Это означает, что оптимистическое управление параллелизмом в таких ситуациях не требуется.   
  
Еще не все ясно? Можно и попроще!  
  
Если вам удалось продолжить чтение до этого момента, то я вас поздравляю - вы достигли базового понимания проблем, которые дают авторам многопоточных приложений существенную пищу для размышлений. Полезно подчеркнуть, что усложненные схемы блокировки внутренних структур данных обычно необходимы только для высопроизводительных систем. Для небольших программ часто можно обойтись менее сложными методами. Вот пара методов "победы малой кровью".

* Не заботьтесь об эффективности, блокируйте все,что можно.
* При работе с данными положитесь на BDE.

Блокировка всех разделяемых данных часто неплохо работает, если вы можете несколько пожертвовать эффективностью. Большинство пользователей предпочитают программы, которые работают чуть медленнее, чем те, которые непредсказуемо зависают из-за ошибок в схеме блокировки. Если имеется большой объем данных, сохранность которых очень важна, то подойдет работа с данными через BDE. Все системы управления базами данных непременно являются потокобезопасными, то есть вы можете без проблем получать доступ к вашим данным из отдельных потоков. Если вы будете использовать базы данных, то вам придется кое-что узнать об управлении транзакциями, т.е. о **reservation** и использовании семантики **prepare, commit** и **rollback**, но пока примите на веру, что подход, основанный на транзакциях, решает проблемы конфликтов потоков; большая часть сложной работы по кодированию для вас уже сделана. Использование BDE c потоками будет описано позже.

Глава 8. Потокобезопасные классы в Дельфи и приоритеты.  
  
Содержание:

* Для чего писать потокобезопасные классы?
* Типы потокобезопасных классов.
* Потокобезопасная инкапсуляция или наследники существующих классов.
* Классы управления потоками данных.
* Мониторы.
* Классы Interlock (взаимоблокировки).
* Поддержка потоков в VCL.
* TThreadList
* TSychroObject
* TCriticalSection
* TEvent и TSimpleEvent
* TMultiReadExclusiveWriteSynchroniser.
* Руководство разработчика потокобезопасных классов.
* Управление приоритетами.
* Что такое приоритет? Как это делается в Win32.
* Какой приоритет дать моему потоку?

Для чего писать потокобезопасные классы?  
  
В простых программах на Delphi, написанных начинающими работать с потоками, синхронизация часто является частью логики приложения. Как продемонстрировано в предыдущей главе, очень легко допустить неуловимые ошибки в логике синхронизации, а разработка отдельной схемы синхронизации для каждого приложения требует большого труда. Лишь немногие механизмы синхронизации используется неоднократно: почти весь потоки, связанные с вводом-выводом, обмениваются данными через общие буферы, и часто используются списки и очереди без встроенной синхронизации. Эти факторы указывают, что следует уделить внимание построению библиотеки потокобезопасных объектов и структур данных: проблемы, возникающие в межпотоковом обмене- непростые, но несколько общих решений подойдут почти во всех случаях.   
Иногда необходимо написать потокобезопасный класс, поскольку никакой другой метод неопустим. Код в DLL, который имеет доступов к уникальным системным данным, должен содержать синхронизацию потоков, даже если DLL и не содержит никаких объектов потоков. Так как большинство программистов на Delphi использует средства языка (классы) для обеспечения возможности модульной разработки и повторного использования кода, эти DLL будут содержать классы, и эти классы должны быть потокобезопасными. Некоторые могут быть довольно простыми, как, например,вышеописанные общие буферные классы. Тем не менее, вполне вероятно что некоторые из этих классов могли осуществлять блокировку ресурсов или другие механизмы синхронизации специфическими средствами ради решения конкретной задачи.   
  
Типы потокобезопасных классов.  
  
Классы могут быть самыми разными, программисты с определенным опытом в Delphi знают, что концепция класса используется многими способами. Некоторые классы используются в основном как структуры данных, другие - как обертки для упрощения сложной внутренней структуры. Иногда семейства совместно работающих классов использутся, чтобы обеспечивать гибкость в достиженеии общей цели, как хорошо демонстрирует механизм потков данных (streams) в Delphi. Аналогичное разнообразие существует и среди потокобезопасных классов. В некоторых случаях классификация может получиться немного расплывчатой, но тем не менее, можно выделить четыре различных типа потокобезопасных классов.   
  
Потокобезопасная инкапсуляция или наследники существующих классов.  
  
Это самый простой тип многопоточного класса. Обычно расширяемый класс имеет довольно ограниченную функциональность и самодостаточен. В простейшем случае создание поттокобезопасного класса может состоять просто из добавления мьютекса и двух дополнительных функций - методов класса, Lock (блокировка) и UnLock. Кроме того, функции, манипулирующие данными класса, могут выполнять блокировку и операции разблокировки автоматически. Какой метод использовать, зависит в основном от количества возможных операций с объектом, и желания программиста самостоятельно создать функции блокировки для обеспечения атомарности составных действий.   
  
Классы управления потоками данных.  
  
Это небольшое расширение вышеуказанного типа, обычно они состоят из буферных классов: списков, стеков и очередей. Дополнительно к поддержке атомарности, эти классы могут выполнять автоматическое управление потоками данных в работающем с буфером потоке. Это часто состоит в задержке потоков, пытающихся читать из пустого буфера или писать в заполненный. Разработка таких классов более полно обсуждается в Главе 10. Множество оперыций может поддерживаться одним классом: с одной стороны, будут обеспечиваться полностью неблокировки действия, с другой стороны, все действия могут блокироваться если они не могут успешно завершить. Компромисс часто достигается, когда действия асинхронны, но обеспечивают обратную связь или обмен сообщениями, когда прежде неудачное действие, вероятно, достигнет цели. Сокеты Win32 API является хорошим примером интерфейса потока данных, которые осуществляют все вышеуказанные возможности в том,что касается управление потоками.   
  
Мониторы.  
  
Мониторы являются логическим шагом вперед от классов управления потоками данных. Они обычно допускают параллельный доступ к данным, которые требуют более сложной синхронизации и блокировки, чем обеспечивает простая потокобезопасная инкапсуляция существующего класса Delphi. Системы управления базами данных относятся к наиболее высокой категории мониторов: обычно в них предусмотрена сложная блокировка и схема управления транзакциями для обеспечения максимальной степени параллелизма при доступе к общим данным с минимальным ущербом производительности из-за конфликтов потоков. СУБД работают довольно специфическим образом, используя управление транзакциями для тщательного контроля над составными операциями, и при этом они также обеспечивают гарантии непрерывности выполняемых операций до самого их завершения. Другим хорошим примером монитора является файловая система. Файловые системы Win32 позволяют нескольким потокам иметь доступ ко многочисленным файлам, которые могут быть открыты несколькими разными процессами одновременно в различных режимах. Большая часть хорошей файловой системы состоит из управления дескрипторами и блокировочных схем, которые обеспечивают оптимальную производительность, гарантируя при этом сохранение атомарности и непрерывности операций. Все потоки могут обращаться к файловой системе, и она гарантирует, что никакие операции не будут вступать в конфликт, и как только операция завершится, ее результат непременно будет записан на диск. В частности файловая система NTFS базируется на журнале событий ("log based"), и обеспечивает сохранность данных даже в случае отказа питания или зависания операционной системы.   
  
Классы Interlock (взаимоблокировки).  
  
Классы взаимоблокировки в этой классификации стоят особняком, посколькуони не содержат никаких данных. Некоторые механизмы блокировки полезны тогда, когда код блокировки легко отделить от кода манипуляции с общими данными. Лучший пример этого в Delphi - класс "Multiple reader single writer interlock", который разрешает разделяемое чтение и атомарную запись некоего ресурса. Его работа будет рассмотрена ниже, в внутренняя реализация класса будет изучена в следующей главе.   
  
Поддержка потоков в VCL.  
  
В Delphi 2 не было классов для поддержки многопоточного программирования, всю синхронизацию приходилось делать только полагаясь на собственные силы. С тех пор библиотека VCL в этом отношении была значительно улучшена. Я опишу классы, имеющиеся в Delphi 4, так как у меня именно эта версия. Пользователи Delphi 3 и 2 увидят, что некоторые классы отсутствуют в этих версиях, а пользователи Delphi 5 и более новых версий найдут многочисленные расширения этих классов. В этой главе я представлю краткое описание этих классов и их использования. Замечу, что в общем-то многие из встроенных в Delphi классов не слишком полезны: они предоставляют слишком мало преимуществ по сравнению с механизмами, обеспечиваемыми Win32 API.   
  
TThreadList  
  
Как уже упоминалось ранее, списки, стеки и очереди часто применяются при реализации задачи взаимодействия потоков. Класс TThreadList осуществляет базовый вид синхронизации, требующийся для потоков. Вдобавок ко всем методам, имеющимся в TList, предусмотрены два дополнительных: Lock и UnLock. Использование их должно быть довольно очевидно для читателей, которые проработали предыдущие главы: список блокируется перед манипуляциями с данными и разблокируется после них. Если поток выполняет над списком многочисленные действия , который должны быть атомарны, то список должен оставаться блокированным. Список не осуществляет никакой неявной синхронизации для принадлежащих ему объектов. Программист может при желании разработать механизмы дополнительной блокировки для обеспечения таких возможностей, а кроме того, использовать блокировку списка для контроля всех операций со структурами данных, принадлежащими ему.   
  
TSychroObject  
  
Этот класс предоставляет виртуальные методы Acquire и Release, которые используются во всех главных классах синхронизации Delphi, предоставляя основу для реализации концепции владения, блокировки или захвата простых объектов синхронизации, подобно тому, как уже обсуждалось ранее. Классы критической секции и классы событий (event) являются его наследниками.   
**TCriticalSection**Этот класс не нуждается в подробном описании. Я подозреваю, что его включение в Delphi - просто дань тем программистам, кто испытывает антипатию к Win32 API. Следует только отметить, что он предоставляет четыре метода: Acquire, Release, Enter и Leave. Два последних только вызывают два первых, для удобства программистов, предпочитающих один из наборов терминов.   
  
TEvent и TSimpleEvent  
  
События (Event) - несколько другой способ обеспечения синхронизации. Вместо осуществления взаимного исключения, они применяются, чтобы заставить какое-то количество потоков ожидать, пока что-то не произойдет, а при возникновении этого события освобождать один или все эти потоки. TSimpleEvent - класс события, который определяет различные параметры по умолчанию, наиболее вероятно использующиеся в приложениях Delphi. События тесно связаны с семафорами и обсуждаются в последующей главе.   
  
TMultiReadExclusiveWriteSynchroniser.  
  
Это объект синхронизации полезен в ситуациях, когда многим потокам нужно читать из коллективного ресурса, но запись в него ведется сравнительно редко. В таком случае часто нет необходимости полностью блоктровать ресурс. В первых главах я утверждал, что любое несинхронизированное использование общих ресурсов, вероятно, приведет к конфликтам потоков. Хотя это и верно, не всегда обязательно использовать полное взаимное исключение. Полное взаимное исключение подразумевает, что в каждый момент только один поток осуществляет какую-то операцию. Мы можем ослабить это требование, если понимаем, что есть два основных типа конфликтов потоков:

* запись после чтения
* запись после записи

Конфликты первого рода происходят, если один поток пишется в раздел ресурса после того, как другой поток прочитал это значение, и считает его верным. Этот тип конфликтов проиллюстрирован в Главе 3. Конфликт второго рода бывает, когда два потока пишут в общий ресурс, один после другого, причем читающий поток не знает о более ранней записи. Это приводит к уничтожению первой записи. Конечно, некоторые действия вполне допустимы: чтение после чтения, чтение после записи. Эти две операции постоянно выполняются в однопоточных программах! Это, очевидно, указывает, что мы можем немного ослабить критерии согласованности работы с данными. Минимальные условия:

* Несколько потоков могут читать одновременно.
* Только один поток может писать в каждый момент.
* Если поток пишет, ни один из потоков не может читать.

Синхронизатор MultiReadExclusiveWriteSynchroniser обеспечивает эти условия с помощью четырех функций: BeginRead, BeginWrite, EndRead и EndWrite. При вызове их до и после записи достигается необходимая синхронизация. Что же касается программиста, то он может рассматривать этот синхронизатор как очень похожий на критическую секцию, с тем исключением, что поток использует его или для чтения или для записи.   
  
Руководство разработчика потокобезопасных классов.  
  
Хотя в последующих главах и рассматриваюся детали создания потокобезопасных классов и различные преимущества и ловушки, в которые можно попасть при их проектировании, видимо, стоит отметить несколько простых пунктов, которые нужно будет всегда учитывать.

* Кто блокирует?
* Экономия блокировочных ресурсов.
* Обработка ошибок.

Ответственность за блокировку в потокобезопасном классе можно отдать на откуп или разработчику класса, или программисту - пользователю класса. Если класс обеспечивает только простую функциональность, обычно лучше, чтобы за блокировку отвечали пользователи класса. Они, вероятно, будут используют несколько экземпляров данного класса, и дав им ответственность за блокировку, мы гарантируем, что не будет неожиданных зацикливаний, а также даем им выбор уровня блокировки, чтобы добиться либо простоты, либо эффективности. Для более сложных классов, например, мониторов, ответственность за блокировку обычно перекладывают на сам класс (или набор классов), скрывая таким образом сложность блокировки от конечного пользователся класса  
В целом, ресурсы должны блокироваться по возможности минимально, и их блокировка должна тщательно настраиваться. Хотя упрощенные схемы блокировки схем и уменьшают шансы внесения в код трудноуловимых ошибок, они могут значительно снизить преимущества в производительности при использовании потоков. Конечно, нет ничего плохого в том,чтобы начинать с простого, но при возникновении проблем с производительностью схему блокировки придется изучатьи проверять более тщательно.   
Ничто не работает всегда без ошибок. При использовании вызовов Win32 API, учтите возможность неудачи операции. Если вы относитесь к тем программистам, кто не против проверять тысячи кодов ошибок, то это выполнимо. Иначе вы можете захотеть написать класс-обертку, который инкапсулирует возможности объектов синхронизации Win32, вызывая исключения, когда происходят ошибки. В любом случае хорошо подумайте об использовании блоков **[b]try**... **finally**[/b], чтобы в случае неудачи гарантировать, что объекты синхронизации останутся в предсказуемом состоянии.   
  
Управление приоритетами.  
  
Вся потоки созданы равными, но некоторые более равны, чем другие (Orwell, Animal Farm). Планировщик должен поделить время CPU между всеми потоками, в любой момент работающими на машине. Для этого ему нужно иметь некоторое представление о том, насколько каждый поток будет использовать CPU, и как важно исполнять конкретный поток, когда он готов работать. Большинство потоков ведут себя одним из двух способов: во время выполнения они задействуют в основном или CPU или ввод/вывод.   
Интенсивно использующие CPU потоки обычно выполняют долгие численные расчеты в фоновом режиме. Они будут занимать все отведенные им ресурсы процессора, но редко будут приостанавливаться для ожидания ввода-вывода или взаимодействия с другими потоками. Довольно часто время их выполнения не особенно критично. Например, поток в программе компьютерной графики может осуществлять долгую операцию по обработке изображения (фильтрация или вращение картинки), и она может занять несколько секунд или даже минут. С точки зрения планировщика, выделяющего кванты процессорного времени, этому потоку никогда не нужно запускаться безотлагательно, так как пользователя не волнует, двенадцать или тринадцать секунд займет выполнение этой операции, и никакой другой поток в системе не ждет результатов этого действия как можно скорее.   
По другому обстоит дело с выделением времени потокам, связанным с вводом-выводом. Они обычно не занимают много процессорного времени, и могут состоять из сравнительно небольших кусков кода обработки. Они очень часто приостанавливаются (блокируются) устройствами ввода-вывода, а когда они получают информацию, то обычно запускаются на короткое время, обрабатывают ввод, и почти немедленно приостанавливаются снова, если больше нет доступной для обработки информации. Примером может служить поток, обрабатывающий действия по перемещению мыши и коррекции положения курсора. Каждый раз при передвижении мыши поток запускается на очень малую долю секунды, обновляя курсор, и затем приостановливается. Потоки подобного типа обычно гораздо более критичны ко времени: они не запускаются надолго, но их запуск должен происходить немедленно. В большинстве GUI систем неприемлемо, чтобы курсор мыши не откликался на ввод даже в течение короткого периода времени, и, следовательно, поток, отвечающий за работу с мышью, является довольно критичным ко времени. Пользователи WinNT могут обратить внимание, что даже когда компьютер занимается интенсивными вычислениями, курсор мыши реагирует немедленно. Весь операционные системы с вытесняющей могозадачностью, включая Win32, обеспечивают поддержку этих концепций, разрешая программисту назначать "приоритеты" потоков. Обычно потоки с более высокими приоритетами связаны со вводом-выводом, а потоки с более низкими приоритетами связаны с вычислительными задачами процессора. Реализация приоритетов потоков в Win32 слегка отличается от реализации, например, в UNIX, так что обсуждаемые здесь детали относятся только к Win32.   
  
Что такое приоритет? Как это делается в Win32.  
  
Большинство операционных системы назначают потокам приоритет для определения, сколько времени CPU должен получать каждый поток. В Win32 фактический приоритет каждого потока вычисляется динамически исходя из множества факторов, некоторые из которых могут непосредственно быть установлены программистом, а некоторые - нет. К этим факторам относятся класс приоритета (**Priority Class**) процесса, уровень приоритета (**Priority Level**)потока, используемые вместе для определения базового приоритета (**Base Priority**) и уровня повышения приоритета (**Priority Boost**), действующих для этого потока. Класс приоритета устанавливается для каждого запущенного процесса. Почти для всех приложений Дельфи он будет Normal, за исключением скринсейверов, которым можно установить класс приоритета Idle. Обычно программисту в Дельфи не нужно изменять класс приоритета запущенного процесса. Уровень приоритета каждого потока можно затем установить в рамках класса , назначенного для процесса, что более полезно, и программист может использовать вызов API **SetThreadPriority** для изменения уровня приоритета потока. Допустимые значения параметра: **THREAD\_PRIORITY\_HIGHEST, THREAD\_PRIORITY\_ABOVE\_NORMAL, THREAD\_PRIORITY\_NORMAL, THREAD\_PRIORITY\_BELOW\_NORMAL, THREAD\_PRIORITY\_LOWEST** и **THREAD\_PRIORITY\_IDLE**. Поскольку реальный базовый приоритет потока вычисляется на основе *как* уровня приоритета, *так и* класса приоритета процесса, потоки с уровнем приоритета Above Normal в процессе с классом приоритета Normal будут обладать большим базовый приоритетом, чем потоки с уровнем приоритета Above Normal в процессе с классом приоритета Below Normal. Как только базовый приоритет потока вычислен, он остается фиксированным на все время жизни потока или пока уровень приоритета (или класс процесса) не изменится. Тем не менее, фактический приоритет, используемый в каждый момент, планировщик задач слегка изменяет в результате повышения приоритета.  
Повышение приоритета является механизмом, используемым планировщиком, чтобы постараться учесть поведение потоков во время выполнения. Некоторые потоки будут полностью занимать CPU или ввод/вывод во время своего исполнения, и планировщик может повысить приоритет этих потоков, но не выделит им полный квант времени. Кроме того, потокам, которые владеют дескрипторами окон, находящихся в данный момент на переднем плане (foreground) также дается небольшое повышение приоритета для улучшения реакции на действия пользователя  
  
Какой приоритет дать моему потоку?  
  
Поняв основы обращения с приоритетам, мы можем теперь попытаться назначить нужные уровни приоритетов уровни потокам нашего приложения. Учтите, что по умолчанию поток VCL выполняется с уровнем приоритета normal. В обшем большинство Дельфи-приложений пишутся так, чтобы по возможности обеспечивать пользователю наиболее быструю ответную реакцию, так что редко нужно увеличивать приоритет потока выше normal - при этом всякий раз при выполнении потока будет происходить задержка многих действий, например, перерисовки окна. Большинство потоков, имеющих дело со вводом/выводом или передачей данных в приложениях Дельфи можно оставить приоритетом normal, так как, если нужно, планировщик задач даст потоку больше времени, а если поток занимает слишком много процессорного времени, то увеличения доли времени не произойдет, что приводит к разумной скорости операций в основном потоке VCL. И наоборот, понижение приоритетов может быть очень полезно. Если вы уменьшите приоритет потока, выполняющего фоновую интенсивную обработку, требующую вычислительных ресурсов, машина покажется пользователю лучше откликающейся на его действия, чем если бы у этого потока оставить нормальный приоритет. Обычно пользователь значительно терпимее относится к небольшим задержкам в выполнении низкоприоритетных потоков: он может переключиться на другие задачи, и при этом компьютер и приложение не теряют восприимчивости к вводу.

Глава 9. Семафоры. Управление потоками данных. Взаимосвязь источник-приемник.  
  
Содержание:

* Семафоры.
* Счетчик больше единицы? "Не вполне критические" секции.
* Новое применение семафоров: управление потоками данных.
* Ограниченный буфер.
* Реализация ограниченного буфера в Delphi.
* Создание: Корректная инициализация счетчиков семафоров.
* Работа: правильные времена ожидания.
* Разрушение: Очистка.
* Разрушение: тонкости остаются.
* Доступ к дескрипторам синхронизации должен быть синхронизован!
* Управление дескрипторами в Win32.
* Решение.
* Использование ограниченного буфера: пример.
* В завершение...

Семафоры.  
  
Семафор представляет собой другой тип примитива синхронизации, с несколько более широкими возможностями по сравнению с мьютексом. В наиболее простых случаях его можно использовать точно так же, как и мьютекс. В общем же семафоры позволяют реализовать в программе более продвинутые механизмы синхронизации.   
Сначала давайте вспомним, как работают мьютексы. Мьютекс может быть или занят или свободен (signalled). Если он свободен, то действие ожидания мьютекса не блокируется. Если он занят, операция ожидания этого мьютекса блокирована. Если мьютекс занят, то он принадлежит конкретному потоку, и, следовательно, только один поток может обладать мьютексом в каждый момент времени.   
Семафоры можно заставить действовать точно так же. Вместо понятия владения, захвата мьютекса, у семафора имеется счетчик. Когда этот счетчик больше нуля, семафор *свободен*, и операции ожидания для него *не блокируются*. Когда счетчик равен 0, то семафор *занят*, и операции ожидания *заблокированы*. Мьютекс по существу является разновидностью семафора, счетчик которого может быть только 0 или 1. Аналогично, семафоры можно рассматривать как воображаемые мьютексы, которые могут одновременно иметь более одного владельца. Функции Win32 API, работающие с семафорами, очень похожи на функции для работы с мьютексами.

* CreateSemaphore. Эта функция подобна CreateMutex. Вместо флага, указывающего, что поток, создающий мьютекс, сразу им будет владеть, эта функция принимает аргумент, задающий начальный счетчик. Создание занятого мьютекса подобно созданию семафора со счетчиком 0: в обоих случаях любой другой поток, ожидающий освобождения объекта, будет заблокирован. Аналогично, создание свободного мьютекса подобно созданию семафора со счетчиком 1: в обоих случаях единственный поток не будет заблокирован в ожидании объекта синхронизации.
* Функции ожидания (Wait). Они для обоих случаев идентичны. Для мьютексов успешный результат ожидания приводит к тому, что поток захватывает мьютекс, а для семафоров*успешный результат ожидания уменьшает счетчик семафора, а если счеичик обнуляется, то вызывающий поток блокируется*.
* ReleaseSemaphore. Это подобно ReleaseMutex, но вместо освобождения мьютекса потоком, ReleaseSemaphore принимает дополнительный целочисленный аргумент, определяющий, на какую величину увеличится счетчик. *ReleaseSemaphore либо увеличивает счетчик семафора, либо активирует соответствующее число потоков, которые были блокированы эти семафором.*

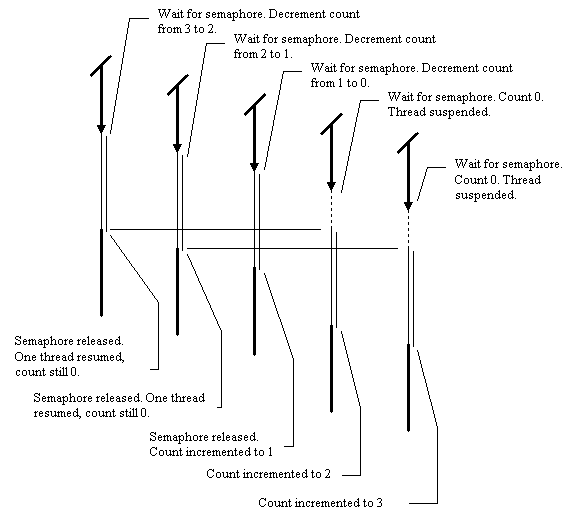
Следующая таблица показывает, как превратить код, использующий мьютексы, в код с применением семафоров, и эквивалентные операции.

|  |
| --- |
| **Код** |
| Мьютексы.                                  Семафоры.  MyMutex := CreateMutex(nil,FALSE,<name>);  MySemaphore := CreateSemaphore(nil,1,1,<name>);  MyMutex := CreateMutex(nil,TRUE,<name>);   MySemaphore := CreateSemaphore(nil,0,1,<name>);  WaitForSingleObject(MyMutex,INFINITE);     WaitForSingleObject(MySemaphore,INFINITE);  ReleaseMutex(MyMutex);                     ReleaseSemaphore(MySemaphore,1);  CloseHandle(MyMutex);                      CloseHandle(MySemaphore); |

Вот простой пример: изменения, требующиеся для кода, представленного в 6 Главе, чтобы программа использовала семафоры вместо критических секций.

|  |
| --- |
| **Код** |
| type   TPrimeFrm = class(TForm)     { No change here until public declarations }   public     { Public declarations }     StringSemaphore: THandle; { Now a semaphore instead of a critical section }     property StringBuf: TStringList read FStringBuf write FStringBuf;   end;  procedure TPrimeFrm.StartBtnClick(Sender: TObject); begin   if not FStringSectInit then   begin     StringSemaphore := CreateSemaphore(nil, 1, 1, SemName); { Now creating a semaphore instead of a critical section }     FStringBuf := TStringList.Create;     FStringSectInit := true;     FPrimeThread := TPrimeThrd2.Create(true);     SetThreadPriority(FPrimeThread.Handle, THREAD\_PRIORITY\_BELOW\_NORMAL);     try       FPrimeThread.StartNum := StrToInt(StartNumEdit.Text);     except       on EConvertError do FPrimeThread.StartNum := 2;     end;     FPrimeThread.Resume;   end;   UpdateButtons; end;  procedure TPrimeFrm.StopBtnClick(Sender: TObject); begin   if FStringSectInit then   begin     with FPrimeThread do     begin       Terminate;       WaitFor;       Free;     end;     FPrimeThread := nil;     FStringBuf.Free;     FStringBuf := nil;     CloseHandle(StringSemaphore); { Deleting semaphore }     FStringSectInit := false;   end;   UpdateButtons; end;  procedure TPrimeFrm.HandleNewData(var Message: TMessage); begin   if FStringSectInit then {Not necessarily the case!}   begin     WaitForSingleObject(StringSemaphore, INFINITE); { New wait call }     ResultMemo.Lines.Add(FStringBuf.Strings[0]);     FStringBuf.Delete(0);     ReleaseSemaphore(StringSemaphore, 1, nil); { New release call }     {Now trim the Result Memo.}     if ResultMemo.Lines.Count > MaxMemoLines then       ResultMemo.Lines.Delete(0);   end; end;  procedure TPrimeThrd2.Execute;  var   CurrentNum: integer;  begin   CurrentNum := FStartNum;   while not Terminated do   begin     if IsPrime(CurrentNum) then     begin       WaitForSingleObject(PrimeFrm.StringSemaphore, INFINITE); { New wait call }       PrimeFrm.StringBuf.Add(IntToStr(CurrentNum) + ' is prime.');       ReleaseSemaphore(PrimeFrm.StringSemaphore, 1, nil); { New release call }       PostMessage(PrimeFrm.Handle, WM\_DATA\_IN\_BUF, 0, 0);     end;     Inc(CurrentNum);   end; end; |

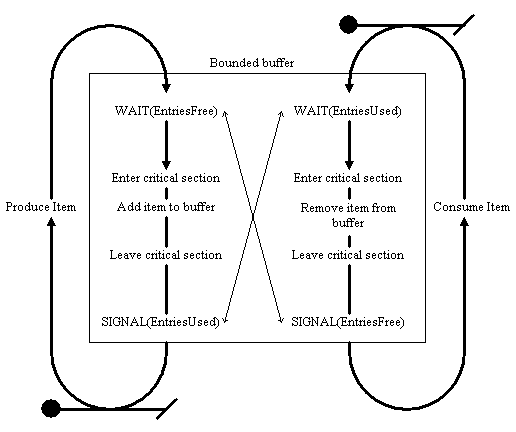
Счетчик больше единицы? "Не вполне критические" секции.  
  
Возможность семафоров иметь счетчик более единицы отчасти аналогична разрешению мьютексу иметь более одного владельца. Таким образом семафоры позволяют создавать критические секции, который разрешают определенному количеству потоков иметь доступ к одному конкретному участку кода или к конкретному объекту. Это по большей части полезно в ситуациях, когда коллективный ресурс состоит из множества буферов или множества потоков, которые можно использовать и другими потоками системы. Давайте рассмотрим конкретный пример и предположим, что до трех потоков могут работать с определенным участком кода. Семафор создается с начальным и максимальным счетчиком 3, и предположим, что ни один поток не работает с критическим участком. Выполнение пяти потоков, пытающихся получить доступ к коллективному ресурсу, может выглядеть приблизительно так:

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig15.gif)

Это конкретное применение семафоров, вероятно, не особенно полезно для программистов на Delphi, главным образом потому, что есть несколько подобных статических структур для уровня приложения. Тем не менее, оно оказывается значительно более важным для ОС, где дескрипторы или ресурсы, такие как системные буферы, вероятно будут статически распределены во время загрузки.   
  
Новое применение семафоров: управление потоками данных.  
  
В Главе 6 было указано на потребность в управлении потоками данных при их прохождении между программными потоками . Кроме того, в Главе 8 эта тема была затронута при обсуждении мониторов. В данной главе рассматривается ситуация для примера, где часто требуется управление потоками данных: ограниченный буфер с единственным потоком-поставщиком данных, выводящий некоторые элементы в буфер, и единственный поток-потребитель, забирающий их оттуда.   
  
Ограниченный буфер.  
  
Ограниченный буфер представляет собой простую разделяемую структуру данных, которая обеспечивает и управление потоками данных, и общий доступ к данным. Буфер, рассмотренный здесь, будет простой очередью: первым вошел - первым вышел (FIFO). . Это будет реализовано в виде циклического буфера, то есть содержать фиксированное количество элементов и иметь два указателя "get" и "put", показывающие, в каком именно месте буфера данные будут вставлены и удалены. Обычно разрешается четыре операции с буфером:

* Create Buffer. Создаются и инициализируются буфер и связанные с ним механизмы синхронизации.
* Put Item. Попытка вставить элемент в буфер с учетом потокобезопасности. Если это невозможно из-за заполнения буфера, то поток, пытающийся вставить элемент, блокируется (приостанавливается), пока буфер не перейдет в состояние, в котором в него разрешено добавлять данные.
* Get Item. Попытка забрать элемент из буфера с учетом потокобезопасности. Если это невозможно из-за того, что буфер пуст, то поток, пытающийся получить элемент, блокируется, пока буфер не перейдет в состояние, в котором из него разрешено удалять данные..
* Destroy Buffer. Разблокирует все потоки, ожидающие буфера, разрушает буфер.

Очевидно, для обращения с коллективными данными потребуются мьютексы. Тем не менее, мы можем использовать семафоры для выполнения необходимых операции блокировки, когда буфер полон или пуст, устраняя потребность в контроле выхода за границы, и даже подсчете того, сколько элементов находится в буфере. Для того, чтобы это сделать, потребуется некоторое изменение концепций. Вместо ожидания семафора и затем освобождения его при выполнении операций, имеющих отношение к буферу, мы используем счетчик двух семафоров, чтобы следить за тем, сколько входов в буфере пусты или заполнены. Давайте назовем эти семафоры "EntriesFree" и "EntriesUsed".   
Обычно с буфером взаимодействуют два потока. Поток-производитель (или писатель) пытается вставить данные в буфер, а поток-потребитель (читатель) пытается их извлечь, как показано на следующей диаграмме. Третий поток (возможно, поток VCL), может вмешиваться для того, чтобы создавать и уничтожать буфер.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig16.gif)

Как можно видеть, потоки - читатель и писатель - выполняются в цикле. Поток-писатель создает элемент и пытается поместить его в буфер. Сначала поток ожидает семафора EntriesFree. Если счетчик EntriesFree нулевой, то поток будет заблокирован, так как буфер полный, и больше данных добавить нельзя. Как только это возможное ожидание закончится, поток добавляет элемент в буфер, а затем увеличивает счетчик EntriesUsed, и, если необходимо, активирует поток-потребитель. Соответственно поток-потребитель заблокируется, если счетчик EntriesUsed нулевой, а когда он удаляет элемент, то увеличивает счетчик EntriesFree, разрешая потоку-производителю добавлять новый элемент.   
Блокировка нужного потока всякий раз, когда буфер становится полным или пустым, оставляет один или другой поток "вне игры". При данном размере буфера N, поток-производитель может только быть на N элементов впереди потока-потребителя до своей остановки, и аналогично, поток-потребитель не может отставать более, чем на N элементов. Это дает несколько преимуществ:

* Один поток не может выдать лишние данные, таким образом мы избавляемся от проблем, указанных в Главе 6, где у нас один поток создавал очередь все увеличивающегося размера.
* Буфер имеет конечный размер, в противоположность списку, рассмотренному ранее, так что мы можем предусмотреть наихудший вариант использования памяти.
* Здесь не будет "ожидания при занятости". Когда потоку нечего делать, он приостанавливается. При этом не возникает ситуации, когда программист пишет маленькие циклы, которые ничего не делают, кроме ожидания новых данных при снятии блокировки. Этого следует избегать, так как впустую тратится процессорное время.

Для полной ясности я дам пример последовательности событий. У нас есть буфер, который имеет 4 возможных входа, и инициализируется он так, что все элементы свободны. Возможно много путей выполнения в зависимости от работы планировщика, но я проиллюстрирую тот, где каждый поток выполняется столько, сколько возможно, прежде чем он будет приостановлен.

|  |
| --- |
| **Код** |
| Действия потока-читателя              Действия потока-писателя            Число     Число                                                                         свободных  занятых                                                                         элементов  элементов Thread starts                         Thread inactive (not scheduled)       4       0 Wait(EntriesUsed) blocks. Suspended.                                        4       0                                       Wait(EntriesFree) flows through       3       0                                       Item Added. Signal(EntriesUsed)       3       1                                       Wait(EntriesFree) flows through       2       1                                       Item Added. Signal(EntriesUsed)       2       2                                       Wait(EntriesFree) flows through       1       2                                       Item Added. Signal(EntriesUsed)       1       3                                       Wait(EntriesFree) flows through       0       3                                       Item Added. Signal(EntriesUsed)       0       4                                       Wait(EntriesFree) blocks. Suspended   0       4 Wait(EntriesUsed) completes                                                 0       3 Item Removed. Signal(EntriesFree)                                           1       3 Wait(EntriesUsed) flows through                                             1       2 Item Removed. Signal(EntriesFree)                                           2       2 Wait(EntriesUsed) flows through                                             2       1 Item Removed. Signal(EntriesFree)                                           3       1 Wait(EntriesUsed) flows through                                             3       0 Item Removed. Signal(EntriesFree)                                           4       0 Wait(EntriesUsed) blocks. Suspended                                         4       0 |

Реализация ограниченного буфера в Delphi.  
  
Вот первая реализация ограниченного буфера на Delphi.

|  |
| --- |
| **Код** |
| unit BoundedBuf;  {Martin Harvey 24/4/2000}  interface  uses Windows, SysUtils;  const   DefaultWaitTime = 5000; { Five second wait on mutexes }  type   { I don't particularly like dynamic arrays, so I'm going to do things     the "C" way here, explicitly allocating memory     Think of TBufferEntries as ^(array of pointer) }    TBufferEntries = ^Pointer;    TBoundedBuffer = class   private     FBufInit: boolean;     FBufSize: integer;     FBuf: TBufferEntries;     FReadPtr, { ReadPtr points to next used entry in buffer}     FWritePtr: integer; { WritePtr points to next free entry in buffer}     FEntriesFree, FEntriesUsed: THandle; { Flow control semaphores }     FCriticalMutex: THandle; { Critical section mutex }   protected     procedure SetSize(NewSize: integer);   public     procedure ResetState;     destructor Destroy; override;     function PutItem(NewItem: Pointer): boolean;     function GetItem: Pointer;   published     property Size: integer read FBufSize write SetSize;   end;    { No constructor required because default values of 0, false etc acceptable }  implementation  const   FailMsg1 = 'Flow control failed, or buffer not initialised';   FailMsg2 = 'Critical section failed, or buffer not initialised';  procedure TBoundedBuffer.SetSize(NewSize: integer);  { Initialises handles and allocates memory.   If the buffer size has previously been set, then this may invoke a buffer   reset }  begin   if FBufInit then ResetState;   if NewSize < 2 then NewSize := 2;   FBufSize := NewSize;   GetMem(FBuf, Sizeof(Pointer) \* FBufSize);   FillMemory(FBuf, Sizeof(Pointer) \* FBufSize, 0);   FBufInit := true;   FCriticalMutex := CreateMutex(nil, false, nil); { note lack of name }   { The initial count on the semaphores requires some thought,     The maximum count requires more thought.     Again, all synchronisation objects are anonymous }   FEntriesFree := CreateSemaphore(nil, FBufSize - 1, FBufSize, nil);   FEntriesUsed := CreateSemaphore(nil, 0, FBufSize, nil);   if (FCriticalMutex = 0)     or (FEntriesFree = 0)     or (FEntriesUsed = 0) then ResetState end;  procedure TBoundedBuffer.ResetState;  { Closes handles and deallocates memory.   Note that this must unblock threads in such a manner that they quit cleanly }  begin   if FBufInit then   begin     WaitForSingleObject(FCriticalMutex, DefaultWaitTime);     FBufInit := false;     FBufSize := 0;     FreeMem(FBuf);     { Now wake up all threads currently waiting.       Currently assumes only 1 producer and 1 consumer.       Plenty of ordering subtleties and pitfalls to be discussed here }     ReleaseSemaphore(FEntriesFree, 1, nil);     ReleaseSemaphore(FEntriesUsed, 1, nil);     CloseHandle(FEntriesFree);     CloseHandle(FEntriesUsed);     { If reader or writer threads are waiting,       then they will be waiting on the mutex.       We will close the handle and let them time out }     CloseHandle(FCriticalMutex);   end; end;  function TBoundedBuffer.PutItem(NewItem: Pointer): boolean;  { Called by producer thread } var   NthItem: TBufferEntries;  begin   result := false;   { WAIT(EntriesFree) }   if WaitForSingleObject(FEntriesFree, INFINITE) <> WAIT\_OBJECT\_0 then     exit;   if (WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0)     or not FBufInit then     exit;   NthItem := FBuf;   Inc(NthItem, FWritePtr);   NthItem^ := NewItem;   FWritePtr := (FWritePtr + 1) mod FBufSize;   ReleaseMutex(FCriticalMutex);   { SIGNAL(EntriesUsed) }   ReleaseSemaphore(FEntriesUsed, 1, nil);   result := true; end;  function TBoundedBuffer.GetItem: Pointer;  { Called by consumer thread } var   NthItem: TBufferEntries;  begin   result := nil;   { WAIT(EntriesUsed) }   if WaitForSingleObject(FEntriesUsed, INFINITE) <> WAIT\_OBJECT\_0 then     exit;   if (WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0)     or not FBufInit then     exit;   NthItem := FBuf;   Inc(NthItem, FReadPtr);   Result := NthItem^;   FReadPtr := (FReadPtr + 1) mod FBufSize;   ReleaseMutex(FCriticalMutex);   { SIGNAL(EntriesFree) }   ReleaseSemaphore(FEntriesFree, 1, nil); end;  destructor TBoundedBuffer.Destroy; begin   ResetState;   inherited Destroy; end;  end. |

Как обычно, появляется несколько вопросов, на которые следует обратить внимание, и несколько проблем, которые будут решены позже.

* Какие значения следует передавать при вызовах создания семафоров?
* Сколько нужно ожидать освобождения мьютексов и критических секций?
* Сколько следует ожидать освобождения семафоров?
* Какой наилучший метод разрушения буфера?

Создание: Корректная инициализация счетчиков семафоров.  
  
При такой реализации ограниченного буфера данные хранятся как массив указателей с индексами чтения и записи в этот массив. В целях отладки я сделал так, что если буфер содержит *N* элементов, то он будет объявлен полным, когда заполнено *N-1*элементов. Такая задача чаще всего решается с помощью циклического буфера, где индексы чтения и записи сравнивают для определения, полон буфер или нет. Если буфер пуст, индексы чтения и записи одинаковы. К несчастью, то же самое будет и для случая, если буфер совершенно заполнен, так что часто в коде циклического буфера делают один всегда пустой вход, что позволяет различить эти два условия. В нашем случае, поскольку мы используем семафоры, это не обязательно. Тем не менее, я решил соблюсти это соглашение для облегчения отладки.   
Учитывая это, мы можем инициализировать семафор EntriesUsed нулем. Поскольку заполненных элементов нет, мы хотим, чтобы потоки-читатели сразу же были блокированы. По условию, мы хотим, чтобы потоки-писатели добавили в буфер*N-1* элементов, и поэтому инициализируем EntriesFree значением *N-1*.   
Нам также нужно учитывать максимальный счетчик, разрешенный для семафоров. Процедура, которая уничтожает буфер, всегда выполняет действие SIGNAL (свободны) для обоих семафоров. Поэтому, когда буфер разрушается, в нем может находиться любое количество элементов, т.е. он может быть и полностью заполнен и совершенно пуст, и мы установили максимальный счетчик в N, допуская таким образом одно действие освобождения семафоров для всех возможных состояний буфера.   
  
Работа: правильные времена ожидания.  
  
Я использовал мьютексы вместо критических секций в этой части программы, поскольку они позволяют разработчику более точно контролировать ошибочные ситуации. Кроме того, они также допускают выход по таймауту. Время ожидания для wait-функций семафоров в самом деле должно быть бесконечным; возможно, что буфер остается полным или пустым в течение длинных периодов времени, и нам нужно заблокировать поток на столько, сколько буфер будет пуст. Параноидально настроенные или пишущие небезопасный код программисты могут использовать задержки на нескольких секунд для этих примитивов, чтобы учесть непредвиденные ошибочные ситуации, когда поток блокируется навсегда. Я достаточно уверен в своем коде, чтобы считать это необязательным, по крайней мере в данном случае...   
Задержка для мьютекса - совсем другое дело. Операции в критической секции быстрые; до N записей в память, и, если обеспечить сравнительное небольшое N (то есть меньше миллиона), то эти действия не должны занять более 5 секунд. В качестве бесплатного приложения часть кода очистки захватывает этот мьютекс, а вместо освобождения его - закрывает дескриптор. При установке таймаута гарантируется, что потоки, ожидающие мьютекс, будут разблокированы, и вернут код ошибки.   
  
Разрушение: Очистка.  
  
К настоящему моменту большинство читателей уже понимают, что операции очистки часто являются наиболее трудными в многопоточном программировании. Ограниченный буфер не является исключением. Процедура ResetState выполняет очистку. Первое, что она делает - проверяет FBufInit. Я предположил, что при этом не требуется синхронизировать доступ, поскольку поток, который создает буфер, должен также и уничтожить его, а раз все делается одним потоком, и все операции записи происходят в критической секции (по крайней мере после создания), то никаких конфликтов не произойдет. Процедуре очистки теперь нужно проверить, что все правильно уничтожено, и что все потоки, находящиеся в ожидании, в процессе чтения или записи, завершаются корректно, сообщая в противном случае о неудаче.   
Процедура очистки сначала захватывает мьютекс для разделяемых данных в буфере, затем разблокирует потоки чтения и записи, освобождая оба семафора. Операции производятся в этом порядке, поскольку, когда оба семафора свободны, состояние буфера больше не отражает истинного положения дел: счетчики семафоров не согласованы с содержимым буфера. Захватывая сначала мьютекс, мы можем уничтожить буфер раньше того, как разблокированные потоки приступят к его чтению. Уничтожая буфер, и устанавливая FBufInit в False, мы гарантируем, что разблокированные потоки вернут код ошибки, а не будут обращаться к неправильным данным.   
Затем мы разблокируем оба потока, освободив оба семафора, а потом закрываем все дескрипторы синхронизации. После этого уничтожаем мьютекс, не освобождая его. Это не страшно, поскольку все действия ожидания мьютекса закончились, то мы можем быть уверены, что как поток чтения, так и поток записи в конечном счете разблокируются. Кроме того, так как имеется только по одному потоку - читателю и писателю, мы можем гарантировать, что никакие другие потоки в течение этого процесса не станут ожидать освобождения семафоров. Это означает, что одного действия по освобождению обоих семафоров было достаточно, чтобы возобновить все потоки, а поскольку мы уничтожаем дескрипторы семафоров, пока удерживаем мьютекс, то дальнейшие действия по записи или чтению обречены на неудачу, если при них будет попытка обращения к одному из семафоров.   
  
Разрушение: тонкости остаются.  
  
***Этот код гарантированно работает только с одним потоком чтения, с одним потоком записи и одним управляющим. Почему?***   
Если существует более одного потока чтения или записи, то более, чем один поток может ожидать один из семафоров в любом момент времени. Следовательно, мы не могли бы активировать все ожидающие потоки-читатели или писатели, когда состояние буфера сброшено. Первой реакцией программиста на это может быть модификация подпрограммы очистки, чтобы продолжалось освобождение одного или другого семафора, пока все потоки не будут разблокированы, что можно сделать приблизительно так

|  |
| --- |
| **Код** |
| procedure TBoundedBuffer.ResetState;  { Closes handles and deallocates memory.   Note that this must unblock threads in such a manner that they quit cleanly }  var   SemCount: integer;  begin   if FBufInit then   begin     WaitForSingleObject(FCriticalMutex, DefaultWaitTime);     FBufInit := false;     FBufSize := 0;     FreeMem(FBuf);     repeat       ReleaseSemaphore(FEntriesFree, 1, @SemCount);     until SemCount = 0;     repeat       ReleaseSemaphore(FEntriesUsed, 1, @SemCount);     until SemCount = 0;     CloseHandle(FEntriesFree);     CloseHandle(FEntriesUsed);     CloseHandle(FCriticalMutex);   end; end; |

К несчастью, этого *все еще* недостаточно, поскольку один из повторяющихся циклов в процедуре очистки может завершиться *прямо перед тем*, как другой поток приступит к действиям чтения или записи и станет ожидать семафор. Очевидно, нам нужно нечто атомарное, но мы не можем ввести действия с семафором в критическую секцию, поскольку потоки, блокирующие семафор, останутся в критической секции, и вся потоки придут к тупику.   
  
Доступ к дескрипторам синхронизации должен быть синхронизован!   
  
Следующая возможность - обнулить дескриптор семафора прямо перед "отключением" его, сделав нечто подобное

|  |
| --- |
| **Код** |
| procedure TBoundedBuffer.ResetState;  { Closes handles and deallocates memory.   Note that this must unblock threads in such a manner that they quit cleanly }  var   SemCount: integer;   LocalHandle: THandle;  begin   if FBufInit then   begin     WaitForSingleObject(FCriticalMutex, DefaultWaitTime);     FBufInit := false;     FBufSize := 0;     FreeMem(FBuf);     LocalHandle := FEntriesFree;     FEntriesFree := 0;     repeat       ReleaseSemaphore(LocalHandle, 1, @SemCount);     until SemCount = 0;     CloseHandle(LocalHandle);     LocalHandle := FEntriesUsed;     FEntriesUsed := 0;     repeat       ReleaseSemaphore(LocalHandle, 1, @SemCount);     until SemCount = 0;     CloseHandle(LocalHandle);     CloseHandle(FCriticalMutex);   end; end; |

(*Автор будет честен и признает, что это жалкое неполноценное решение пришло ему в голову*). Однако это ничем не лучше. Вместо проблемы тупика мы получим конфликт потоков непростого типа. Этот конфликт представляет собой запись после чтения для самого дескриптора семафора! Да... Вы должны синхронизировать даже ваши объекты синхронизации! Вот что может случиться: рабочий поток читает значение дескриптора мьютекса из буферного объекта и приостанавливается, ожидая; в этот момент поток очистки, уничтожающий буфер, освобождает мьютекс необходимое число раз, и именно в этот момент рабочий поток возобновляется и обращается к мьютексу, который, как мы считаем, только что был уничтожен! Интервал, в котором это может случиться, очень небольшой, но тем не менее, это решение неприемлемо.   
  
Управление дескрипторами в Win32.  
  
Эта проблема достаточно сложна, так что имеет смысл рассмотреть что именно происходит, когда мы запускаем функцию Win32 для закрытия мьютекса или семафора. В частности, полезное знать вот что:

* Разблокирует ли закрытие дескриптора потоки, ожидающие данный мьютекс или семафор?
* В случае мьютексов, есть ли разница, кто владеет дескриптором при освобождении мьютекса?

Чтобы узнать это, мы можем использовать два текстовых приложения, для мьютекса

|  |
| --- |
| **Код** |
| unit HandleForm;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls;  type   THandleFrm = class(TForm)     CreateBtn: TButton;     CloseOwnerBtn: TButton;     CloseNonOwnerBtn: TButton;     procedure CreateBtnClick(Sender: TObject);     procedure CloseOwnerBtnClick(Sender: TObject);     procedure CloseNonOwnerBtnClick(Sender: TObject);   private     { Private declarations }   public     { Public declarations }     Mutex: THandle;   end;  var   HandleFrm: THandleFrm;  implementation  uses HandleThreads;  {$R \*.DFM}  procedure THandleFrm.CreateBtnClick(Sender: TObject);  var   NewThread: THandleThread;  begin   Mutex := CreateMutex(nil, false, nil);   WaitForSingleObject(Mutex, INFINITE);   NewThread := THandleThread.Create(false);   NewThread := THandleThread.Create(false);   ShowMessage('Threads Created.'); end;  procedure THandleFrm.CloseOwnerBtnClick(Sender: TObject); begin   CloseHandle(Mutex); end;  procedure THandleFrm.CloseNonOwnerBtnClick(Sender: TObject); begin   ReleaseMutex(Mutex);   CloseHandle(Mutex); end;  end.  unit HandleThreads;  interface  uses   Classes, Windows, SysUtils, Dialogs;  type   THandleThread = class(TThread)   private     { Private declarations }   protected     procedure Execute; override;   end;  implementation  uses HandleForm;  procedure THandleThread.Execute;  var   RetVal: integer;  begin   RetVal := WaitForSingleObject(HandleFrm.Mutex, INFINITE);   case RetVal of     WAIT\_OBJECT\_0: ShowMessage('Unblocked: WAIT\_OBJECT\_0');     WAIT\_ABANDONED: ShowMessage('Unblocked: WAIT\_ABANDONED');     WAIT\_TIMEOUT: ShowMessage('Unblocked: WAIT\_TIMEOUT');   else     ShowMessage('Unblocked. Unknown return code.');   end; end;  end. |

и для семафора

|  |
| --- |
| **Код** |
| unit HandleForm;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls;  type   THandleFrm = class(TForm)     CreateBtn: TButton;     CloseOwnerBtn: TButton;     CloseNonOwnerBtn: TButton;     RelBtn: TButton;     procedure CreateBtnClick(Sender: TObject);     procedure CloseOwnerBtnClick(Sender: TObject);     procedure CloseNonOwnerBtnClick(Sender: TObject);     procedure RelBtnClick(Sender: TObject);   private     { Private declarations }   public     { Public declarations }     Semaphore: THandle;   end;  var   HandleFrm: THandleFrm;  implementation  uses HandleThreads;  {$R \*.DFM}  procedure THandleFrm.CreateBtnClick(Sender: TObject);  begin   Semaphore := CreateSemaphore(nil, 1, 1, nil);   WaitForSingleObject(Semaphore, INFINITE);   THandleThread.Create(false);   THandleThread.Create(false);   ShowMessage('Threads Created.'); end;  procedure THandleFrm.CloseOwnerBtnClick(Sender: TObject); begin   CloseHandle(Semaphore); end;  procedure THandleFrm.CloseNonOwnerBtnClick(Sender: TObject); begin   ReleaseSemaphore(Semaphore, 1, nil);   CloseHandle(Semaphore); end;  procedure THandleFrm.RelBtnClick(Sender: TObject); begin   ReleaseSemaphore(Semaphore, 1, nil); end;  end.  unit HandleThreads;  interface  uses   Classes, Windows, SysUtils, Dialogs;  type   THandleThread = class(TThread)   private     { Private declarations }   protected     procedure Execute; override;   end;  implementation  uses HandleForm;  procedure THandleThread.Execute;  var   RetVal: integer;  begin   RetVal := WaitForSingleObject(HandleFrm.Semaphore, 10000);   case RetVal of     WAIT\_OBJECT\_0: ShowMessage('Unblocked: WAIT\_OBJECT\_0');     WAIT\_ABANDONED: ShowMessage('Unblocked: WAIT\_ABANDONED');     WAIT\_TIMEOUT: ShowMessage('Unblocked: WAIT\_TIMEOUT');   else     ShowMessage('Unblocked. Unknown return code.');   end; end;  end. |

С помощью этих программ можно определить, что при закрытии дескриптора объекта синхронизации Win32 не разблокирует потоки, ожидающие этот объект. Это, наиболее вероятно, происходит благодаря механизму подсчета ссылок, который Win32 использует, чтобы следить за дескрипторами: потоки, ожидающие объект синхронизации, могут поддерживать внутренний счетчик ссылок так, чтобы он не обнулялся, и закрывая дескриптор объекта для приложения, мы только лишаемся всякого управления этим объектом синхронизации. В нашей ситуации это серьезная проблема. В идеале при очистке хотелось бы надеяться, что попытка ожидания для закрытого дескриптора должна разблокировать потоки, ждущие освобождения данного объекта синхронизации через этот конкретный дескриптор. Это бы позволило программисту войти в критическую секцию, очистить данные в этой критической секции, затем закрыть дескриптор, таким образом разблокировав потоки, ожидающие данный объект с неким значением ошибки (возможно, **WAIT\_ABANDONED**? (ждать, пока не исчезнет)).

Решение.   
  
В результате всего этого мы определили, что закрытие дескрипторов необходимо, оно приводит к тому,что поток не будет выполнять ожидание дескриптора неопределенное время. Применяя это к ограниченному буферу при очистке, мы можем гарантированно разблокировать все потоки, ожидающие семафора, только если мы знаем, сколько потоков ожидают освобождения мьютексов. В общем случае нам нужно проверять, что потоки не выполняет бесконечного ожидания мьютексов. Вот измененный буфер, который работает с любым количеством потоков:

|  |
| --- |
| **Код** |
| unit BoundedBuf;  {Martin Harvey 24/4/2000}  interface  uses Windows, SysUtils;  const   DefaultWaitTime = 1000; { One second wait on all synchronisation primitives }  type   { I don't particularly like dynamic arrays, so I'm going to do things     the "C" way here, explicitly allocating memory     Think of TBufferEntries as ^(array of pointer) }    TBufferEntries = ^Pointer;    TBoundedBuffer = class   private     FBufInit: boolean;     FBufSize: integer;     FBuf: TBufferEntries;     FReadPtr, { ReadPtr points to next used entry in buffer}     FWritePtr: integer; { WritePtr points to next free entry in buffer}     FEntriesFree, FEntriesUsed: THandle; { Flow control semaphores }     FCriticalMutex: THandle; { Critical section mutex }   protected     procedure SetSize(NewSize: integer);     function ControlledWait(Semaphore: THandle): boolean;     { Returns whether wait returned OK, or an error occurred }   public     procedure ResetState;     destructor Destroy; override;     function PutItem(NewItem: Pointer): boolean;     function GetItem: Pointer;   published     property Size: integer read FBufSize write SetSize;   end;    { No constructor required because default values of 0, false etc acceptable }  implementation  procedure TBoundedBuffer.SetSize(NewSize: integer);  { Initialises handles and allocates memory.   If the buffer size has previously been set, then this may invoke a buffer   reset }  begin   if FBufInit then ResetState;   if NewSize < 2 then NewSize := 2;   FBufSize := NewSize;   GetMem(FBuf, Sizeof(Pointer) \* FBufSize);   FillMemory(FBuf, Sizeof(Pointer) \* FBufSize, 0);   FBufInit := true;   FCriticalMutex := CreateMutex(nil, false, nil); { note lack of name }   { The initial count on the semaphores requires some thought,     The maximum count requires more thought.     Again, all synchronisation objects are anonymous }   FEntriesFree := CreateSemaphore(nil, FBufSize - 1, FBufSize, nil);   FEntriesUsed := CreateSemaphore(nil, 0, FBufSize, nil);   if (FCriticalMutex = 0)     or (FEntriesFree = 0)     or (FEntriesUsed = 0) then ResetState end;  procedure TBoundedBuffer.ResetState;  { Closes handles and deallocates memory.   Note that this must unblock threads in such a manner that they quit cleanly }  begin   if FBufInit then   begin     WaitForSingleObject(FCriticalMutex, DefaultWaitTime);     FBufInit := false;     FBufSize := 0;     FreeMem(FBuf);     ReleaseSemaphore(FEntriesUsed, 1, nil);     ReleaseSemaphore(FEntriesFree, 1, nil);     CloseHandle(FEntriesFree);     CloseHandle(FEntriesUsed);     ReleaseMutex(FCriticalMutex);     CloseHandle(FCriticalMutex);   end; end;  function TBoundedBuffer.ControlledWait(Semaphore: THandle): boolean;  var   ErrCode: integer;  begin   repeat     ErrCode := WaitForSingleObject(Semaphore, DefaultWaitTime);     if (ErrCode = WAIT\_OBJECT\_0) or (ErrCode = WAIT\_ABANDONED) then     begin       { If wait abandoned, return failure. Buffer not properly cleaned up }       result := ErrCode = WAIT\_OBJECT\_0;       exit;     end;     { Wait timed out. Check whether buffer state initialised }     if WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0 then     begin       result := false;       exit;     end     else     begin       result := FBufInit;       ReleaseMutex(FCriticalMutex);     end;   until not Result; end;  function TBoundedBuffer.PutItem(NewItem: Pointer): boolean;  { Called by producer thread } var   NthItem: TBufferEntries;  begin   result := false;   { WAIT(EntriesFree) }   if not ControlledWait(FEntriesFree) then     exit;   if (WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0)     or not FBufInit then { NB.This condition depends on L -> R lazy evaluation }     exit;   NthItem := FBuf;   Inc(NthItem, FWritePtr);   NthItem^ := NewItem;   FWritePtr := (FWritePtr + 1) mod FBufSize;   ReleaseMutex(FCriticalMutex);   { SIGNAL(EntriesUsed) }   ReleaseSemaphore(FEntriesUsed, 1, nil);   result := true; end;  function TBoundedBuffer.GetItem: Pointer;  { Called by consumer thread } var   NthItem: TBufferEntries;  begin   result := nil;   { WAIT(EntriesUsed) }   if not ControlledWait(FEntriesUsed) then     exit;   if (WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0)     or not FBufInit then { NB.This condition depends on L -> R lazy evaluation }     exit;   NthItem := FBuf;   Inc(NthItem, FReadPtr);   Result := NthItem^;   FReadPtr := (FReadPtr + 1) mod FBufSize;   ReleaseMutex(FCriticalMutex);   { SIGNAL(EntriesFree) }   ReleaseSemaphore(FEntriesFree, 1, nil); end;  destructor TBoundedBuffer.Destroy; begin   ResetState;   inherited Destroy; end;  end. |

Функции ожидания семафоров в нем были модифицированы, процедура очистки также претерпела некоторые изменения.   
Вместо выполнения бесконечного ожидания соответствующего мьютекса, потоки чтения и записи теперь вызывают функцию "Управляемого Ожидания". В этой функции каждый поток ждет семафор только определенное временя. Такое ожидание семафора может возвращать одно из трех значений, в соответствии с описанием в справке Win32.

* **WAIT\_OBJECT\_0** (Успех).
* **WAIT\_ABANDONED**
* **WAIT\_TIMEOUT**

Во-первых, если семафор освобожден, функция возвращает "успех", и никаких дальнейших действий не требуется... Во-вторых, в случае, когда функция Win32 WaitFor дает **WAIT\_ABANDONED**, функция возвращает ошибку; это значение ошибки указывает, что поток завершается без корректного освобождения объекта синхронизации. Случай, который нам наиболее интересен - время ожидания истекло **(WAIT\_TIMEOUT)**. Это может быть по одной из двух возможных причин:

* Поток мог быть долгое время блокирован.
* Содержимое буфера было разрушено без уведомления соответствующего потока.

Для того, чтобы это проверить, мы пытаемся войти в критическую секцию и проверить, что переменная "буфер инициализирован" все еще True. Если любое из этих действий терпит неудачу, то мы знаем, что внутреннее содержимое буфера было разрушено, и функция завершается, возвратив ошибку. Если оба этих действия успешны, то мы возвращаемся в цикл, снова ожидая мьютекс. Эта функция гарантирует, что когда буфер разрушен, заблокированный поток в конечном счете выйдет по таймауту, и возвратит ошибку в вызвавший поток.   
Подпрограмма очистки также немного модифицирована. Она теперь и освобождает семафоры, и мьютекс критической секции. Такой подход гарантирует, что первый поток-читатель и писатель разблокируются немедленно, как только состояние буфера будет сброшено. Конечно, дополнительным потокам, возможно, придется ждать завершения вплоть до истечения задержки, определенной по умолчанию.   
  
Использование ограниченного буфера: пример.   
  
Чтобы создать основу этого примера, было разработано простое приложение с использованием двух потоков. Эта программа ищет простые числа-палиндромы (перевертыши). Пара палиндромов существует, когда два числа, *X* и *Y* оба простые, и *Y* является палиндромом *X*. Ни *X* , ни *Y* не должныбыть палиндромами сами по себе, хотя это предсталяет собой особый случай X = Y. Примеры простых чисел - палиндромов включают: (101, 101), (131, 131) - это особый случай, а (16127, 72161) , (15737, 73751) и (15683, 38651) - не особый.   
В сущности, два потока осуществляют слегка разные задачи:

|  |
| --- |
| **Код** |
| unit PrimeThreads;  interface  uses   Windows, Classes, SysUtils, BoundedBuf, Forms;  type   TIntRec = record     Num: integer;   end;   PIntRec = ^TIntRec;    TPrimeThread = class(TThread)   private     FBuffer: TBoundedBuffer;   protected     function IsPrime(TestNum: integer): boolean;   public     property Buffer: TBoundedBuffer read FBuffer write FBuffer;   end;    TForwardPrimeThread = class(TPrimeThread)   private   protected     procedure SendToBackThread(TestNum: integer);     procedure Execute; override;   end;    TBackwardPrimeThread = class(TPrimeThread)   private     FDestSection: PRTLCriticalSection;     FDestMsgNum: integer;     FDestForm: TForm;     FDestList: TStrings;   protected     function ReverseNumber(Input: integer): integer;     function RecieveFromForwardThread(var TestNum: integer): boolean;     procedure SendToVCLThread(CurrentNumber, ReversedNumber: integer);     procedure Execute; override;   public     property DestSection: PRTLCriticalSection read FDestSection write FDestSection;     property DestMsgNum: integer read FDestMsgNum write FDestMsgNum;     property DestForm: TForm read FDestForm write FDestForm;     property DestList: TStrings read FDestList write FDestList;   end;  var   ForwardThread: TForwardPrimeThread;   BackwardThread: TBackwardPrimeThread;   Buffer: TBoundedBuffer;  procedure StartThreads(Form: TForm;   Section: PRTLCriticalSection;   MsgNum: integer;   List: TStrings); procedure StopThreads;  implementation  const   DefBufSize = 16;  { Ancillary procedures }  procedure StartThreads(Form: TForm;   Section: PRTLCriticalSection;   MsgNum: integer;   List: TStrings); begin   ForwardThread := TForwardPrimeThread.Create(true);   BackwardThread := TBackwardPrimeThread.Create(true);   SetThreadPriority(ForwardThread.Handle, THREAD\_PRIORITY\_BELOW\_NORMAL);   SetThreadPriority(BackwardThread.Handle, THREAD\_PRIORITY\_BELOW\_NORMAL);   Buffer := TBoundedBuffer.Create;   Buffer.Size := DefBufSize;   ForwardThread.Buffer := Buffer;   BackwardThread.Buffer := Buffer;   with BackwardThread do   begin     DestForm := Form;     DestSection := Section;     DestMsgNum := MsgNum;     DestList := List;   end;   ForwardThread.Resume;   BackwardThread.Resume; end;  procedure StopThreads; begin   ForwardThread.Terminate;   BackwardThread.Terminate;   Buffer.ResetState;   ForwardThread.WaitFor;   BackwardThread.WaitFor;   Buffer.Free;   ForwardThread.Free;   BackwardThread.Free; end;  { TPrimeThread }  function TPrimeThread.IsPrime(TestNum: integer): boolean;  var   iter: integer;  begin   result := true;   if TestNum < 0 then     result := false;   if TestNum <= 2 then     exit;   iter := 2;   while (iter < TestNum) and (not terminated) do {Line A}   begin     if (TestNum mod iter) = 0 then     begin       result := false;       exit;     end;     Inc(iter);   end; end;  { TForwardPrimeThread }  procedure TForwardPrimeThread.SendToBackThread(TestNum: integer);  var   NewRec: PIntRec;  begin   New(NewRec);   NewRec.Num := TestNum;   if not Buffer.PutItem(NewRec) then Dispose(NewRec); end;  procedure TForwardPrimeThread.Execute;  var   CurrentNumber: integer;  begin   CurrentNumber := 2;   while not Terminated do   begin     if IsPrime(CurrentNumber) then       SendToBackThread(CurrentNumber);     Inc(CurrentNumber);   end; end;  { TBackwardPrimeThread }  function TBackwardPrimeThread.RecieveFromForwardThread(var TestNum: integer): boolean;  var   NewRec: PIntRec;  begin   NewRec := Buffer.GetItem;   Result := Assigned(NewRec);   if Result then TestNum := NewRec^.Num; end;  procedure TBackwardPrimeThread.SendToVCLThread(CurrentNumber, ReversedNumber: integer);  var   Msg: string;  begin   Msg := 'Palindromic primes: ' + IntToStr(CurrentNumber) + ' and '     + IntToStr(ReversedNumber);   EnterCriticalSection(FDestSection^);   DestList.Add(Msg);   LeaveCriticalSection(FDestSection^);   PostMessage(DestForm.Handle, DestMsgNum, 0, 0); end;  function TBackwardPrimeThread.ReverseNumber(Input: integer): integer;  var   InStr, OutStr: string;   Len, Iter: integer;  begin   Input := Abs(Input);   InStr := IntToStr(Input);   OutStr := '';   Len := Length(InStr);   for Iter := Len downto 1 do     OutStr := OutStr + InStr[Iter];   try     Result := StrToInt(OutStr);   except     on EConvertError do Result := Input;   end; end;  procedure TBackwardPrimeThread.Execute;  var   CurrentNumber,     ReversedNumber: integer;  begin   while not Terminated do   begin     if RecieveFromForwardThread(CurrentNumber) then     begin       ReversedNumber := ReverseNumber(CurrentNumber);       if IsPrime(ReversedNumber) then         SendToVCLThread(CurrentNumber, ReversedNumber);     end;   end; end;  end. |

Первый поток ("прямой") ищет простые числа. При нахождении он помещает это число в ограниченный буфер. Второй поток ждет, пока в буфере будут элементы. Когда они появляются, он удаляет элемент, переворачивает цифры, проверяет, является ли обращенное число простым, и если это так, то посылает текстовую строку, содержащую два числа, главной форме:

|  |
| --- |
| **Код** |
| unit PalPrimeForm;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls;  const   WM\_DATA\_IN\_BUF = WM\_APP + 1000;   MaxMemoLines = 20;  type   TPalFrm = class(TForm)     ResultsMemo: TMemo;     StartButton: TButton;     StopButton: TButton;     procedure StartButtonClick(Sender: TObject);     procedure StopButtonClick(Sender: TObject);     procedure FormClose(Sender: TObject; var Action: TCloseAction);   private     { Private declarations }     FStringSectInit: boolean;     FStringBuf: TStringList;     StringSection: TRTLCriticalSection;     procedure UpdateButtons;     procedure HandleNewData(var Message: TMessage); message WM\_DATA\_IN\_BUF;   public     { Public declarations }   end;  var   PalFrm: TPalFrm;  implementation  uses PrimeThreads;  {$R \*.DFM}  procedure TPalFrm.UpdateButtons; begin   StopButton.Enabled := FStringSectInit;   StartButton.Enabled := not FStringSectInit; end;  procedure TPalFrm.StartButtonClick(Sender: TObject); begin   if not FStringSectInit then   begin     InitializeCriticalSection(StringSection);     FStringBuf := TStringList.Create;     FStringSectInit := true;     StartThreads(Self, @StringSection, WM\_DATA\_IN\_BUF, FStringBuf);   end;   UpdateButtons; end;  procedure TPalFrm.StopButtonClick(Sender: TObject); begin   if FStringSectInit then   begin     ResultsMemo.Lines.Add('Please wait...');     StopThreads;     ResultsMemo.Lines.Add('Done!');     FStringBuf.Free;     FStringBuf := nil;     DeleteCriticalSection(StringSection);     FStringSectInit := false;   end;   UpdateButtons; end;  procedure TPalFrm.HandleNewData(var Message: TMessage); begin   if FStringSectInit then   begin     EnterCriticalSection(StringSection);     ResultsMemo.Lines.Add(FStringBuf.Strings[0]);     FStringBuf.Delete(0);     LeaveCriticalSection(StringSection);     if ResultsMemo.Lines.Count > MaxMemoLines then       ResultsMemo.Lines.Delete(0);   end; end;  procedure TPalFrm.FormClose(Sender: TObject; var Action: TCloseAction); begin   StopButtonClick(Self); end;  end. |

Хотя кода довольно много, нового для обсуждения почти нет. Я советую читателю рассмотреть методы Execute каждого потока, так как они дают ясное представление о том, что происходит. Перенос данных из второго потока в поток VCL и соответствующие методы главной формы обсуждались в предыдущих главах. Остался только один момент, который стоит затронуть... вы, наверно, догадались! Освобождение ресурсов и очистка.   
  
В завершение...  
  
Вы наверно, подумали, не могло ли что-нибудь еще остаться недосказанным про разрушение? Надо упомянуть еще один тонкий момент. Код ограниченного буфера допускает, что потоки могут попытаться получить доступ к полям объекта после сброса буфера. Это хорошо, но это означает, что, уничтожая два потока и буфер между ними, мы должны сбросить состояние буфера, *затем подождать завершения всех потоков*, и только потом действительно разрушать буфер, освобождая таким образом память, содержащую сам объект. Если так не сделать, может произойти нарушение доступа (access violation). Функция StopThreads правильно это выполняет, гарантируя корректный выход.   
В данный момент стоит упомянуть, что дополнительные проблемы с синхронизацией имеются для процедуры SetSize. В примере я считал, что размер буфера установлен единожды и навсегда, причем раньше, чем какой-либо поток станет использовать буфер. Возможно установить размер буфера и тогда, когда он используется. В общем, это плохая идея, поскольку означает, что если два потока используют буфер; один читатель и один писатель, то они не могут правильно обнаружить разрушение буфера. Если у буфера должен быть изменен размер, то все потоки, использующие буфер, должны быть остановлены или приостановлены в известной безопасной точке. Затем у буфера можно поменять размер, и перезапустить потоки записи и чтения. Некоторые программисты могут захотеть написать расширенную версию буфера, который корректно обрабатывает операции изменения размера.

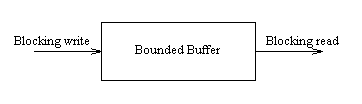
Глава 10. Ввод/вывод и потоки данных: от блокировки к асинхронности и обратно.  
  
Содержание.

* Отличия от потока VCL и разработка I/O интерфейса.
* Обзор.
* Реализация преобразования блокировка-асинхронность
* Добавление к ограниченному буферу операций просмотра.
* Создание двунаправленного ограниченного буфера.
* Детальное рассмотрение Блокировочно-Асинхронного Буфера (БАБ).
* Создание БАБ.
* Разрушение БАБ.
* Пример программы с использованием БАБ.
* Мы достигли нашей цели!
* Заметили утечку памяти?
* Избавляемся от утечки.
* Проблемы просмотра.
* Промежуточный буфер.
* Различные ограничения.
* Обратная сторона монеты: Потоковые буферы.

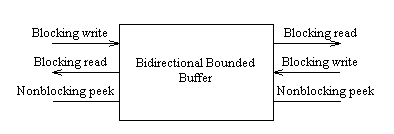
Отличия от потока VCL и разработка интерфейса I/O (ввода/вывода).  
  
Для рабочих потоков имеет смысл делать I/O блокирующим, так как обычно это самый простой путь. С точки зрения потока, использующего I/O ресурс c блокировкой, успех или неудача такого вызова проявляются немедленно, и при разработке логики программы не нужно заботиться о периоде времени между началом операции с I/O и ее завершением.   
Выполнение главного потока VCL обычно нельзя блокировать на длинные периоды времени: поток должен всегда быть способен обработать новые сообщения с минимальной задержкой. Обычно дисковые I/O операции блокируют поток, поскольку задержки для конечного пользователя невелики, но другие I/O операции делают асинхронными, особенно действия по связи между потоками, процессами, или машинами, так как величину этих задержек нельзя предсказать заранее. Преимущество асинхронных операций, как уже говорилось прежде, состоит в том, что код, исполняемый потоком VCL, всегда способен воспринимать новые сообщения. Основной же недостаток - в том, что код, выполняющийся в потоке VCL, должен следить за статусом завершения всех незаконченных I/O операций. Это может быть довольно трудно, учитывая расходы на хранение возможно многочисленных описаний состояния. Иногда приходится создавать машину состояний, особенно при выполнении предопределенных протоколов, например, HTTP, FTP или NNTP. Чаще, конечно, проблема не настолько сложна, и должно быть решена разово. В таких случаях достаточно разработать конкретное решение.   
При разработке набора функций передачи данных следует учитывать это различие . Если, например, рассмотреть взаимодействие, то самый общий набор операций для коммуникационного канала включает: **Open**, **Close**, **Read** и **Write**. Блокирующие I/O интерфейсы предоставляют простые функции для реализации этих действий. Асинхронные интерфейсы обеспечивают четыре основные функции, и вдобавок до четырех уведомлений (**notifications)**, инициируемых либо **call back** (обратным вызовом), либо по событию (**event)**. Эти уведомления или показывают, что завершилась начатая операция, или что возможно ее повторить, или и то и другое. Пример интерфейса может быть таким:

* Функция **Open**, и связанное с ней событие **OnOpen**, которое показывает, то действие открытия завершено, сообщая об успехе или ошибке.
* Функция **Read**, и событие **CanRead** (или **OnRead**). Событие обычно показывает, что при вызове Read будут прочитаны новые данные, и/или что со времени последнего чтения поступили новые данные.
* Функция **Write**, и событие **CanWrite** (или **OnWrite**). Событие обычно показывает, что вызов Write запишет данные, и/или что со времени последней записи посланы новые данные, в результате чего в буфере есть место для новых операций Write. В зависимости от семантики это событие может включаться или нет после успешного вызова Open.
* Функция **Close**, и событие **OnClose**. Событие обычно показывает, что коммуникационный канал был закрыт, и что больше нельзя посылать или принимать данные. Это событие обычно нужно в ситуациях, когда еще возможно читать данные с удаленного конца коммуникационного канала после вызова Close, и успешно работает вместе с механизмами настройки и разрыва коммуникации, которые используют соответствующие протоколы (например, TCP).

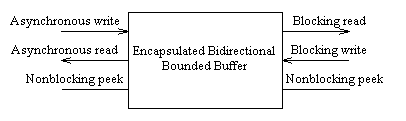
Обзор.  
  
Для лучшего понимания этой главы будет полезно рассмотреть существующий механизм передачи данных между потоками, отметив методы, с помощью которых он будет расширен. Я хотел бы настоятельно рекомендовать читателям проработать эту главу, несмотря на то, что в ней много кода, который следует изучить. Наиболее важный момент, на который нужно обратить внимание - многие детали реализации, хотя и полезны для тех, кто хочет писать функциональные программы, включающие эти методы, но не имеют первостепенной важности для желающих приобрести фундаментальное понимание описанных вопросов. До сих пор единственный механизм передачи данных, который мы рассматривали - ограниченный буфер, который можно представить таким образом:

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig17.gif)

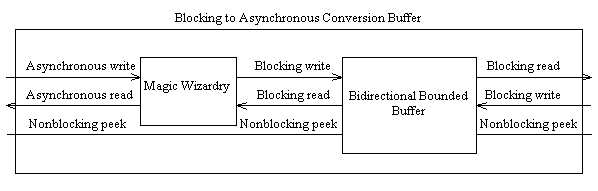
В этой главе будут продемонстрированы различные расширения этого буфера. Первая модификация будет довольно проста: установить два буфера комплементарно и добавить неблокирующую операцию извлечения с обоих сторон получившегося двунаправленного буфера.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig18.gif)

Хорошо. Это не должно вызвать удивления у читателей, и тем, кто дошел до этого момента, прорабатывая весь материал, нетрудно будет реализовать такую конструкцию. Следующая модификация более существенная: вместо проведения операций чтения и записи с блокированием буфера, сделаем один набор действий асинхронным:

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig19.gif)

Точнее, мы создадим компонент, который преобразует блокирующие действия в асинхронные и наоборот. В данном случае он будет просто инкапсулировать операции чтения и записи для двунаправленного буфера, но при желании можно перекрыть эту функциональность, чтобы преобразовывать различные действия ввода/вывода между блокировочным и асинхронным режимами.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig20.gif)

Появляется вопрос: Почему? Ответ должен быть очевиден: Если мы делаем буфер, обеспечивающий двунаправленное сообщение между двумя потоками, где один поток использует блокировку, а другой поток- асинхронные операции, то:

* Мы можем использовать его для связи между потоком VCL и рабочими потоками нашего приложения без блокировки потока VCL.
* Все сложности будут спрятаны внутри: никаких сообщений с магическими числами, никакого использования Synchronize, никаких явно видимых критических секций.
* Он будет осуществлять контроль потоков данных между потоком VCL и рабочими потоками; пока нерешенная задача.
* Его может использовать как готовое решение проблем взаимодействия между потоком VCL и другими потоками даже человек, не имеющий представления о проблемах синхронизации.

Реализация компонента преобразования блокировка-асинхронность.  
  
Компонент, который мы создадим, предполагает, что выполняется только один поток VCL, так что асинхронный интерфейс будет предусмотрен только для одного потока. Операции блокировки, обеспечиваемые этим буфером, будут работать с точно теми же ограничениями, как и в примере ограниченного буфера в предыдущей главе, и следовательно, любое число блокирующих потоков будет способно параллельно иметь доступ к блокировочному интерфейсу. Подобно тому, как ограниченный буфер допускал простые операции Get и Put, вовлекающие только один элемент, для блокировочно-асинхронного буфера (в дальнейшем называемого БАБ), также допустимы простые действия, включающие только один элемент. Семантика интерфейса будет такой:

* **Creation:** При создании компонент БАБ создаст требуемые внутренние буферные структуры данных и потоки и будет генерировать событие OnWrite, сигнализируя, то данные могут быть записаны в буфер потоком VCL.
* **Reading:** Компонент предоставит две функции чтения; BlockingRead и AsyncRead. BlockingRead будет использоваться рабочими потоками, а AsyncRead - потоком VCL.
* **Read Notifications:** Компонент вызовет в главном потоке VCL событие OnRead, когда возможна асинхронная операция чтения, т.е., данные ждут чтения потоком VCL. Так как эта реализация компонента имеет дело только с чтением и записью указателей, можно считать, что только один элемент может быть прочитан при каждом уведомлении, и поток VCL должен ожидать следующего уведомления до попытки нового чтения.
* **Writing:** Компонент БАБ предоставит две функции; BlockingWrite и AsyncWrite. BlockingWrite будет использоваться рабочими потоками, а AsyncWrite - потоком VCL.
* **Write Notifications:** Компонент вызовет в главном потоке VCL событие OnWrite, когда может пройти асинхронная операция записи, т.е., в буфере имеется свободное место, куда может быть записан элемент. И опять же поддерживается взаимосвязь один-к-одному между уведомлениями и успешными записями, и поток VCL должен выполнять только одну попытку записи и ожидать следующего уведомления.
* **Peek operations:** Любой поток будет способен узнать, сколько входов буфера свободны или заняты в конкретном направлении. Эта операция может быть полезна для рабочих потоков, которые таким образом узнают, будут ли BlockingRead или BlockingWrite в действительности вызывать блокировку. Поток VCL не должен использовать эти функции для определения, можно ли читать или писать, и полагаться на уведомления.

Добавление к ограниченному буферу операций просмотра.  
  
Вот модификация ограниченного буфера, реализующая операции просмотра.

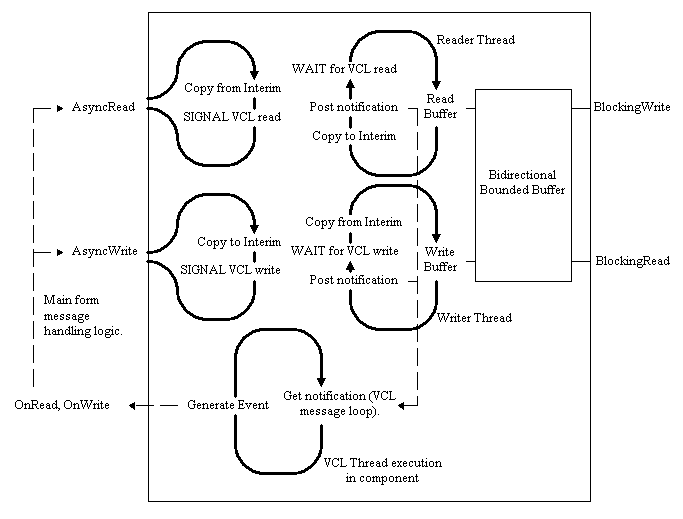
|  |
| --- |
| **Код** |
| unit BoundedBuf;  {Martin Harvey 24/4/2000}  interface  uses Windows, SysUtils;  const   DefaultWaitTime = 1000; { One second wait on all synchronisation primitives }  type   { I don't particularly like dynamic arrays, so I'm going to do things     the "C" way here, explicitly allocating memory     Think of TBufferEntries as ^(array of pointer) }    TBufferEntries = ^Pointer;    TBoundedBuffer = class   private     FBufInit: boolean;     FBufSize: integer;     FBuf: TBufferEntries;     FReadPtr, { ReadPtr points to next used entry in buffer}     FWritePtr: integer; { WritePtr points to next free entry in buffer}     FEntriesFree, FEntriesUsed: THandle; { Flow control semaphores }     FCriticalMutex: THandle; { Critical section mutex }     FEntryCountFree, FEntryCountUsed: integer; { Used for peeking operations }   protected     procedure SetSize(NewSize: integer);     function ControlledWait(Semaphore: THandle): boolean;     { Returns whether wait returned OK, or an error occurred }   public     procedure ResetState;     destructor Destroy; override;     function PutItem(NewItem: Pointer): boolean;     function GetItem: Pointer;     { New peeking operations. Note that we can't use simple properties, since       we have to communicate success or failure of the operation, in addition       to providing a result }     function GetEntriesFree(var Free: integer): boolean;     function GetEntriesUsed(var Used: integer): boolean;   published     property Size: integer read FBufSize write SetSize;   end;    { No constructor required because default values of 0, false etc acceptable }  implementation  procedure TBoundedBuffer.SetSize(NewSize: integer);  { Initialises handles and allocates memory.   If the buffer size has previously been set, then this may invoke a buffer   reset }  begin   if FBufInit then ResetState;   if NewSize < 2 then NewSize := 2;   FBufSize := NewSize;   GetMem(FBuf, Sizeof(Pointer) \* FBufSize);   FillMemory(FBuf, Sizeof(Pointer) \* FBufSize, 0);   FCriticalMutex := CreateMutex(nil, false, nil); { note lack of name }   WaitForSingleObject(FCriticalMutex, INFINITE);   FBufInit := true;   { The initial count on the semaphores requires some thought,     The maximum count requires more thought.     Again, all synchronisation objects are anonymous }   FEntriesFree := CreateSemaphore(nil, FBufSize - 1, FBufSize, nil);   FEntriesUsed := CreateSemaphore(nil, 0, FBufSize, nil);   FEntryCountFree := FBufSize - 1;   FEntryCountUsed := 0;   ReleaseMutex(FCriticalMutex);   if (FCriticalMutex = 0)     or (FEntriesFree = 0)     or (FEntriesUsed = 0) then ResetState end;  procedure TBoundedBuffer.ResetState;  { Closes handles and deallocates memory.   Note that this must unblock threads in such a manner that they quit cleanly }  begin   if FBufInit then   begin     WaitForSingleObject(FCriticalMutex, DefaultWaitTime);     FBufInit := false;     FBufSize := 0;     FreeMem(FBuf);     ReleaseSemaphore(FEntriesUsed, 1, nil);     ReleaseSemaphore(FEntriesFree, 1, nil);     CloseHandle(FEntriesFree);     CloseHandle(FEntriesUsed);     ReleaseMutex(FCriticalMutex);     CloseHandle(FCriticalMutex);   end; end;  function TBoundedBuffer.ControlledWait(Semaphore: THandle): boolean;  var   ErrCode: integer;  begin   repeat     ErrCode := WaitForSingleObject(Semaphore, DefaultWaitTime);     if (ErrCode = WAIT\_OBJECT\_0) or (ErrCode = WAIT\_ABANDONED) then     begin       { If wait abandoned, return failure. Buffer not properly cleaned up }       result := ErrCode = WAIT\_OBJECT\_0;       exit;     end;     { Wait timed out. Check whether buffer state initialised }     if WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0 then     begin       result := false;       exit;     end     else     begin       result := FBufInit;       ReleaseMutex(FCriticalMutex);     end;   until not Result; end;  function TBoundedBuffer.PutItem(NewItem: Pointer): boolean;  { Called by producer thread } var   NthItem: TBufferEntries;  begin   result := false;   { WAIT(EntriesFree) }   if not ControlledWait(FEntriesFree) then     exit;   if (WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0)     or not FBufInit then { NB.This condition depends on L -> R lazy evaluation }     exit;   NthItem := FBuf;   Inc(NthItem, FWritePtr);   NthItem^ := NewItem;   FWritePtr := (FWritePtr + 1) mod FBufSize;   Inc(FEntryCountUsed);   Dec(FEntryCountFree);   ReleaseMutex(FCriticalMutex);   { SIGNAL(EntriesUsed) }   ReleaseSemaphore(FEntriesUsed, 1, nil);   result := true; end;  function TBoundedBuffer.GetItem: Pointer;  { Called by consumer thread } var   NthItem: TBufferEntries;  begin   result := nil;   { WAIT(EntriesUsed) }   if not ControlledWait(FEntriesUsed) then     exit;   if (WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0)     or not FBufInit then { NB.This condition depends on L -> R lazy evaluation }     exit;   NthItem := FBuf;   Inc(NthItem, FReadPtr);   Result := NthItem^;   FReadPtr := (FReadPtr + 1) mod FBufSize;   Inc(FEntryCountFree);   Dec(FEntryCountUsed);   ReleaseMutex(FCriticalMutex);   { SIGNAL(EntriesFree) }   ReleaseSemaphore(FEntriesFree, 1, nil); end;  destructor TBoundedBuffer.Destroy; begin   ResetState;   inherited Destroy; end;  function TBoundedBuffer.GetEntriesFree(var Free: integer): boolean; begin   result := false;   if (WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0)     or not FBufInit then     exit;   Free := FEntryCountFree;   result := true;   ReleaseMutex(FCriticalMutex); end;  function TBoundedBuffer.GetEntriesUsed(var Used: integer): boolean; begin   result := false;   if (WaitForSingleObject(FCriticalMutex, DefaultWaitTime) <> WAIT\_OBJECT\_0)     or not FBufInit then     exit;   Used := FEntryCountUsed;   result := true;   ReleaseMutex(FCriticalMutex); end;  end. |

Обратите внимание, что хотя и возможно считывать счетчики семафоров во время определенных действий, я решил сохранять счетчики вручную, используя дополнительные переменные FEntryCountFree и FEntryCountUsed. Для чтения этих переменных предусмотрены новые методы. Многие программисты на Delphi могут сразу подумать об объявлении этих атрибутов буфера как свойств. К несчастью, мы должны учесть, что операции синхронизации, требующиеся для доступа к этим переменным могут потерпеть неудачу. Кажется более подходящим оставить операции просмотра в виде функций, таким образом напоминая программисту, что требуется некоторая работа по доступу к необходимым данным, и что функция может вернуть ошибку. Некоторые могут посчитать, что при таком обосновании будет иметь смысл также оформить атрибут буфера Size как явную функцию чтения. Это скорее вопрос стиля, так как размер буфера можно читать непосредственно без потребности в какой-либо синхронизации.   
  
Создание двунаправленного ограниченного буфера.  
  
Эта операция почти тривиальна и не требует сложного объяснения. Я осуществил ее как простую инкапсуляцию двух ограниченных буферных объектов. Все действия, поддерживаемые ограниченным буфером, также поддерживаются и двунаправленным буфером, с той небольшой модификацией, что поток, использующий этот объект, должен определить, с какой частью буфера он хочет иметь дело. Обычно один поток будет работать с частью A, а другой - с частью B.

|  |
| --- |
| **Код** |
| unit BiDirBuf;  {Martin Harvey 7/5/2000}  interface  uses BoundedBuf;  type   TBufferSide = (bsSideA, bsSideB);   TBufferOp = (boWriting, boReading);    TBiDirBuf = class   private     FAtoBBuf, FBtoABuf: TBoundedBuffer;   protected     function GetBuf(Side: TBufferSide; Op: TBufferOp): TBoundedBuffer;     function GetSize: integer;     procedure SetSize(NewSize: integer);   public     constructor Create;     destructor Destroy; override;     procedure ResetState;     function PutItem(Side: TBufferSide; Item: pointer): boolean;     function GetItem(Side: TBufferSide): pointer;     { Entries used function peeks buffer one is reading from, and       Entried free function peeks buffer one is writing to. It seems       a bit useless to allow the other two operations: why worry about       your neighbour when you have plenty else to worry about? }     function GetEntriesUsed(Side: TBufferSide; var Used: integer): boolean;     function GetEntriesFree(Side: TBufferSide; var Free: integer): boolean;   published     property Size: integer read GetSize write SetSize;   end;  implementation  { TBiDirBuf }  constructor TBiDirBuf.Create; begin   inherited Create;   FAToBBuf := TBoundedBuffer.Create;   FBToABuf := TBoundedBuffer.Create; end;  destructor TBiDirBuf.Destroy; begin   FAToBBuf.Free;   FBToABuf.Free;   inherited Destroy; end;  procedure TBiDirBuf.ResetState; begin   FAToBBuf.ResetState;   FBToABuf.ResetState; end;  function TBiDirBuf.GetBuf(Side: TBufferSide; Op: TBufferOp): TBoundedBuffer; begin   if ((Side = bsSideA) and (Op = boWriting))     or ((Side = bsSideB) and (Op = boReading)) then     result := FAToBBuf   else if ((Side = bsSideA) and (Op = boReading))     or ((Side = bsSideB) and (Op = boWriting)) then     result := FBToABuf   else   begin     result := FAToBBuf;     Assert(false);   end; end;  function TBidirBuf.GetSize: integer; begin   Assert(FAToBBuf.Size = FBToABuf.Size);   result := FAToBBuf.Size; end;  procedure TBiDirBuf.SetSize(NewSize: integer); begin   FAToBBuf.Size := NewSize;   FBToABuf.Size := NewSize;   Assert(FAToBBuf.Size = FBToABuf.Size); end;  function TBiDirBuf.PutItem(Side: TBufferSide; Item: Pointer): boolean; begin   result := GetBuf(Side, boWriting).PutItem(Item); end;  function TBiDirBuf.GetItem(Side: TBufferSide): Pointer; begin   result := GetBuf(Side, boReading).GetItem; end;  function TBiDirBuf.GetEntriesUsed(Side: TBufferSide; var Used: integer): boolean; begin   result := GetBuf(Side, boReading).GetEntriesUsed(Used); end;  function TBiDirBuf.GetEntriesFree(Side: TBufferSide; var Free: integer): boolean; begin   result := GetBuf(Side, boWriting).GetEntriesFree(Free); end;  end. |

Этот класс реализует функциональность, описанную графически на вышеприведенной диаграмме, представляющей БАБ.

Детальное рассмотрение Блокировочно-Асинхронного Буфера (БАБ).  
  
Сделав всю подготовительную работу, теперь можно рассказать о БАБ более подробно. Он содержит двунаправленный буфер и два потока, один чтения и один записи. Эти потоки выполняют действия записи и чтения с ограниченным буфером от имени потока VCL. Выполнение всех потоков можно представить графически с некоторыми отклонениями от существующих соглашений:

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig21.gif)

Эта диаграмма выглядит довольно устрашающе, так что, видимо, легче рассмотреть рабочий пример. Давайте возьмем тот случай, когда рабочий поток выполняет блокирующую запись в БАБ.

* Рабочий поток выполняет блокирующую запись.
* Поток чтения БАБ в данный момент блокирован при попытке прочесть из двунаправленного буфера. В результате записи он разблокируется и успешно проводит чтение.
* Он копирует прочитанные данные во вспомогательный буфер, локальный для класса потока, и генерирует событие прохождения данных, обрабатываемое БАБ.
* Код обработки события прохождения данных БАБ, исполняемый в контексте потока чтения, посылает сообщение дескриптору своего окна, свидетельствуя, что данные были успешно прочитаны потоком чтения.
* Поток чтения теперь ждет семафора, который сигнализирует о том, что данные прочитаны главным потоком VCL.
* Несколько позже главный поток VCL обрабатывает сообщения из очереди, предназначенные компоненту, точно таким же образом, как и для всех компонентов, имеющих дескриптор окна.
* Среди сообщений для компонента есть сообщение уведомления, посланное потоком чтения. Оно обрабатывается и генерирует событие OnRead для компонента.
* Событие OnRead обрабатывается логикой приложения (вероятно, главной формой), и в результате, очевидно, поток VCL пытается прочитать данные.
* Поток VCL вызывает метод БАБ AsyncRead.
* AsyncRead копирует данные из вспомогательного буфера в главный поток VCL. Затем он освобождает семафор, которым блокирован поток чтения, разрешая ему попытаться осуществить новую операцию чтения из двунаправленного буфера.

Точно так же БАБ работает и при записи. Она осуществляется асинхронно потоком VCL, внутреннй поток записи БАБ пробуждается и проводит блокирующую запись в двунаправленный буфер, и по окончании записи поток VCL уведомляется через событие, что можно пытаться делать новые операции записи.   
В сущности интерфейс между блокировкой и асинхронной операцией через отправления сообщения идентичен тому, который был неформально введен в более ранних примерах. Отличие для этого компонента в том, что детали спрятаны от конечного пользователя, и проблема решена более формальным, хорошо определенным способом.   
Вот код этого компонента

|  |
| --- |
| **Код** |
| unit BlockToAsyncBuf;  { Martin Harvey 10/5/2000 }  interface  uses Classes, Forms, Messages, Windows, BiDirBuf;  const   InternalBufferSize = 4;   WM\_BLOCK\_ASYNC = WM\_USER + 2876;  type    { With this component, as with previous buffering schemes, one cannot read     or write nil pointers. }    TThreadNotify = (tnReaderDataFlow, tnWriterDataFlow);    TBlockAsyncThread = class(TThread)   private     FDataSection: TRTLCriticalSection;     FIdleSemaphore: THandle;     FInterimBuf: Pointer;     FOnDataFlow: TNotifyEvent;     FBuffer: TBiDirBuf;   protected     procedure DataFlow; virtual;     function GetItemsInTransit: integer;   public     constructor Create(CreateSuspended: boolean);     destructor Destroy; override;   published     property OnDataFlow: TNotifyEvent read FOnDataFlow write FOnDataFlow;     property Buffer: TBiDirBuf write FBuffer;     property ItemsInTransit: integer read GetItemsInTransit;   end;    TBAWriterThread = class(TBlockAsyncThread)   private   protected     procedure Execute; override;   public     function WriteItem(Item: Pointer): boolean;   published   end;    TBAReaderThread = class(TBlockAsyncThread)   private   protected     procedure Execute; override;   public     function ReadItem: pointer;   published   end;    TBlockToAsyncBuf = class(TComponent)   private     FHWND: THandle;     FBuffer: TBiDirBuf;     FReaderThread: TBAReaderThread;     FWriterThread: TBAWriterThread;     FOnRead, FOnWrite: TNotifyEvent;   protected     procedure MessageHandler(var Msg: TMessage);     procedure ReaderDataFlow(Sender: TObject);     procedure WriterDataFlow(Sender: TObject);     procedure Read; virtual;     procedure Write; virtual;     function GetItemsInTransit: integer;   public     constructor Create(AOwner: TComponent); override;     destructor Destroy; override;     function BlockingRead: pointer;     function BlockingWrite(Item: pointer): boolean;     function AsyncRead: pointer;     function AsyncWrite(Item: pointer): boolean;     procedure ResetState;   published     property OnRead: TNotifyEvent read FOnRead write FOnRead;     property OnWrite: TNotifyEvent read FOnWrite write FOnWrite;     property ItemsInTransit: integer read GetItemsInTransit;   end;  implementation  procedure TBlockAsyncThread.DataFlow; begin   if Assigned(FOnDataFlow) then FOnDataFlow(Self); end;  constructor TBlockAsyncThread.Create(CreateSuspended: boolean); begin   inherited Create(CreateSuspended);   InitializeCriticalSection(FDataSection);   FIdleSemaphore := CreateSemaphore(nil, 0, High(Integer), nil); end;  destructor TBlockAsyncThread.Destroy; begin   ReleaseSemaphore(FIdleSemaphore, 1, nil);   WaitFor;   DeleteCriticalSection(FDataSection);   CloseHandle(FIdleSemaphore); end;  function TBlockAsyncThread.GetItemsInTransit: integer; begin   EnterCriticalSection(FDataSection);   if Assigned(FInterimBuf) then     result := 1   else     result := 0;   LeaveCriticalSection(FDataSection); end;  { Buffer error handling needs to be discussed }  procedure TBAWriterThread.Execute;  var   Temp: Pointer;  begin   while not Terminated do   begin     DataFlow;     WaitForSingleObject(FIdleSemaphore, INFINITE);     EnterCriticalSection(FDataSection);     Temp := FInterimBuf;     FInterimBuf := nil;     LeaveCriticalSection(FDataSection);     if not FBuffer.PutItem(bsSideA, Temp) then Terminate;   end; end;  function TBAWriterThread.WriteItem(Item: Pointer): boolean; begin   result := false;   EnterCriticalSection(FDataSection);   if not Assigned(FInterimBuf) then   begin     FInterimBuf := Item;     result := true;   end;   LeaveCriticalSection(FDataSection);   if Result then ReleaseSemaphore(FIdleSemaphore, 1, nil); end;  procedure TBAReaderThread.Execute;  var   Temp: Pointer;  begin   while not Terminated do   begin     Temp := FBuffer.GetItem(bsSideA);     if Assigned(Temp) then     begin       EnterCriticalSection(FDataSection);       FInterimBuf := Temp;       LeaveCriticalSection(FDataSection);       DataFlow;       WaitForSingleObject(FIdleSemaphore, INFINITE);     end     else Terminate;   end; end;  function TBAReaderThread.ReadItem: pointer; begin   EnterCriticalSection(FDataSection);   result := FInterimBuf;   LeaveCriticalSection(FDataSection);   if Assigned(Result) then ReleaseSemaphore(FIdleSemaphore, 1, nil); end;  procedure TBlockToAsyncBuf.MessageHandler(var Msg: TMessage); begin   if (Msg.Msg = WM\_BLOCK\_ASYNC) then   begin     case TThreadNotify(Msg.LParam) of       tnReaderDataflow: Read;       tnWriterDataflow: Write;     else       Assert(false);     end;   end; end;  procedure TBlockToAsyncBuf.ReaderDataFlow(Sender: TObject); begin   PostMessage(FHWND, WM\_BLOCK\_ASYNC, 0, Integer(tnReaderDataflow)); end;  procedure TBlockToAsyncBuf.WriterDataFlow(Sender: TObject); begin   PostMessage(FHWND, WM\_BLOCK\_ASYNC, 0, Integer(tnWriterDataflow)); end;  procedure TBlockToAsyncBuf.Read; begin   if Assigned(FOnRead) then FOnRead(Self); end;  procedure TBlockToAsyncBuf.Write; begin   if Assigned(FOnWrite) then FOnWrite(Self); end;  constructor TBlockToAsyncBuf.Create(AOwner: TComponent); begin   inherited Create(AOwner);   FHWND := AllocateHWnd(MessageHandler);   FBuffer := TBiDirBuf.Create;   FBuffer.Size := InternalBufferSize;   FReaderThread := TBAReaderThread.Create(true);   FReaderThread.Buffer := Self.FBuffer;   FReaderThread.OnDataFlow := ReaderDataFlow;   FWriterThread := TBAWriterThread.Create(true);   FWriterThread.Buffer := Self.FBuffer;   FWriterThread.OnDataFlow := WriterDataFlow;   FReaderThread.Resume;   FWriterThread.Resume; end;  procedure TBlockToAsyncBuf.ResetState; begin   if Assigned(FReaderThread) then FReaderThread.Terminate;   if Assigned(FWriterThread) then FWriterThread.Terminate;   FBuffer.ResetState;   FReaderThread.Free;   FWriterThread.Free;   FReaderThread := nil;   FWriterThread := nil; end;  destructor TBlockToAsyncBuf.Destroy; begin   { A few destruction subtleties here }   ResetState;   FBuffer.Free;   DeallocateHWnd(FHWND);   inherited Destroy; end;  function TBlockToAsyncBuf.BlockingRead: pointer; begin   result := FBuffer.GetItem(bsSideB); end;  function TBlockToAsyncBuf.BlockingWrite(Item: pointer): boolean; begin   result := FBuffer.PutItem(bsSideB, Item); end;  function TBlockToAsyncBuf.AsyncRead: pointer; begin   result := FReaderThread.ReadItem; end;  function TBlockToAsyncBuf.AsyncWrite(Item: pointer): boolean; begin   result := FWriterThread.WriteItem(Item); end;  function TBlockToAsyncBuf.GetItemsInTransit: integer;  var   Entries: integer;  begin   result := FReaderThread.ItemsInTransit + FWriterThread.ItemsInTransit;   if FBuffer.GetEntriesUsed(bsSideA, Entries) then     Inc(result, Entries);   if FBuffer.GetEntriesUsed(bsSideB, Entries) then     Inc(result, Entries); end;  end. |

Некоторые моменты стоит отметить особо. Обычно потомки TThread нечасто используют наследование, однако в данном случае потоки чтения и записи имеют много общего в своей функциональности, которая и реализована в базовом классе TBlockAsyncThread. Он содержит:

* Вспомогательный буфер, содержащий единственный указатель.
* Критическую секцию для обеспечения атомарного доступа ко вспомогательному буферу.
* Указатель на двунаправленный буфер для использования операций блокировки. Он устанавливается в компоненте, указывая на двунаправленный буфер, используемый внутри БАБ.
* Событие "OnDataFlow", которое обрабатывается компонентом БАБ.
* Семафор ожидания. Он используется для реализации операций "Ждать записи VCL" и "Ждать чтения VCL" общепринятым способом.

Базовый класс потока также обеспечивает необходимый минимум общей функциональности: создание потока,его уничтожение, и переключатель события для OnDataFlow. У базового класса есть два потомка: TBAWriterThread и TBAReaderThread. Они реализуют конкретные методы выполнения потока, а также предоставляют методы чтения и записи, которые косвенно будут исполняться потоком VCL. Сам компонент БАБ хранит двунаправленный буфер и два потока. Кроме того, в нем также хранится дескриптор окна FHWND, который используется для специализированной обработки сообщений.   
  
Создание БАБ.  
  
Давайте теперь кратко познакомимся с реализацией. При создании компонент БАБ получает дескриптор окна, используя AllocateHWnd. Эта полезная функция описана в книге Danny Thorpe "Создание компонентов Delphi". Компонент БАБ довольно необычен, потому что ему требуется дескриптор окна для выполнения обработки сообщения, но в действительности он - не визуальный компонент. Возможно дать компоненту БАБ дескриптор окна, сделав его потомком TWinControl. Тем не менее это неподходящий родитель для такого компонента, поскольку БАБ не является оконным элементом. Используя AllocateHWnd, компонент может осуществлять собственную обработку сообщений, не приобретая большого количества бесполезных для него методов. Обеспечивается также некоторое улучшение эффективности благодаря тому, что процедура обработки сообщений компонента выполняет только минимальную работу, имея дело лишь с одним сообщением, и игнорируя все другие.   
При создании компонент БАБ также устанавливает обработчики событий от потоков себе же. Эти обработчики выполняются в контексте потоков чтения и записи и осуществляют отправление уведомления, которое связывает потоки чтения, записи, и основной поток VCL.   
В результате создания компонента инициализируются потоки. Все действия, требуемые для этого - общие для потоков чтения и записи, так что они находятся в конструкторе TBlockAsyncThread. При этом просто устанавливается критическая секция, требующаяся для обеспечения атомарного доступа к промежуточному буферу в каждом потоке, а также создается семафор ожидания для каждого потока, который гарантирует, что рабочий поток будет ожидать поток VCL перед проведением операций чтения или записи.   
  
Разрушение БАБ.  
  
Уничтожение компонента несколько сложнее, но использует принципы, обсужденные в предыдущих главах. Двунаправленный буфер, содержащийся в БАБ, подобен ограниченному буферу, рассмотренному в последней главе, и его разрушение проходит в три стадии. Первый этап - разблокировка всех потоков, выполняющих действия по вводу/выводу в буфер путем вызова ResetState. Второй этап - ожидание, пока все потоки не остановятся, или по крайней мере, будут в том состоянии, в котором они не выполняют больше действий с буфером. Как только это условие выполнено, может начаться третий этап, на котором уничтожаются физические структуры данных.   
Разрушение БАБ, таким образом, работает так:

* Состояние БАБ сбрасывается. При этом завершаются оба внутренних потока, затем сбрасывается состояние двунаправленного буфера, разблокируя, таким образом, все выполняющиеся операции с буфером.
* Вызывается деструктор каждого потока. В нем освобождается семафор ожидания потока, затем поисходит ожидание завершения потока до разрушения критической секции и семафора. Некоторых читателей может удивить, что деструктор потока может вызвать WaitFor. Здесь все правильно, так как мы можем быть уверены, что поток никогда не вызовет свой собственный деструктор. В данном случае деструкторы потоков чтения и записи будут вызываться главным потоком VCL, так что зацикливания не будет.
* Потоки чтения и записи устанавливаются в Nil, что разрешает проводить неоднократные вызовы ResetState.
* Двунаправленный буфер разрушается, дескриптор окна освобождается.

Так как потоки внутренние для БАБ, у этих процедур очистки имеется тот положительный эффект, что деструктор БАБ может разблокировать и очистить все потоки и объекты синхронизации внутри компонента, а пользователю компонента не надо заботиться о потенциальных проблемах, присущих операциям очистки. Достаточно просто вызвать Free для БАБ. Это, очевидно, весьма желательно.   
Несмотря не это, компонент все-таки дает доступ к методу ResetState. Причина в том, что компонент не контролирует другие рабочие потоки, которые могут проводить блокирующие операции с буфером. В подобных ситуациях главное приложение обязано само завершить рабочие потоки, сбросить состояние БАБ, и дождаться завершения рабочих потоков до физического разрушения БАБ.   
  
Пример программы с использованием БАБ.  
  
Вот еще один вариант программы для простых чисел. Главная форма

|  |
| --- |
| **Код** |
| unit BlockAsyncForm;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   StdCtrls, BlockToAsyncBuf, PrimeRangeThread;  const   MaxCount = 20;  type   TBlockAsyncFrm = class(TForm)     Label1: TLabel;     StartRangeEdit: TEdit;     Label2: TLabel;     EndRangeEdit: TEdit;     SubmitBtn: TButton;     ResultsMemo: TMemo;     procedure FormCreate(Sender: TObject);     procedure SubmitBtnClick(Sender: TObject);     procedure FormCloseQuery(Sender: TObject; var CanClose: Boolean);     procedure FormDestroy(Sender: TObject);   private     { Private declarations }     FItemsInTransit: integer;     FBuf: TBlockToAsyncBuf;     FWorkerThread: TPrimeRangeThread;     procedure BufRead(Sender: TObject);     procedure BufWrite(Sender: TObject);   public     { Public declarations }   end;  var   BlockAsyncFrm: TBlockAsyncFrm;  implementation  {$R \*.DFM}  procedure TBlockAsyncFrm.FormCreate(Sender: TObject); begin   FWorkerThread := TPrimeRangeThread.Create(true);   FBuf := TBlockToAsyncBuf.Create(Self);   with FBuf do   begin     { Note that these changes will take effect before       events from this component could possibly occur }     OnRead := BufRead;     OnWrite := BufWrite;   end;   SetThreadPriority(FWorkerThread.Handle, THREAD\_PRIORITY\_BELOW\_NORMAL);   with FWorkerThread do   begin     Buf := FBuf;     Resume;   end; end;  procedure TBlockAsyncFrm.SubmitBtnClick(Sender: TObject);  var   Request: PRangeRequestType;   Temp: integer;  begin   New(Request);   try     Request.Low := StrToInt(StartRangeEdit.Text);     Request.High := StrToInt(EndRangeEdit.Text);     if Request.Low > Request.High then     begin       Temp := Request.Low;       Request.Low := Request.High;       Request.High := Temp;     end;     if FBuf.AsyncWrite(Request) then     begin       Request := nil;       SubmitBtn.Enabled := false;       Inc(FItemsInTransit);     end;   finally     if Assigned(Request) then Dispose(Request);   end; end;  procedure TBlockAsyncFrm.BufWrite(Sender: TObject); begin   { Buffer has indicated that there is space for us to write }   SubmitBtn.Enabled := true; end;  procedure TBlockAsyncFrm.BufRead(Sender: TObject);  var   Reply: TStringList;  begin   { We have received a notification that we may read. }   Reply := TStringList(FBuf.AsyncRead);   if Assigned(Reply) then   begin     Dec(FItemsInTransit);     ResultsMemo.Lines.BeginUpdate;     ResultsMemo.Lines.AddStrings(Reply);     while ResultsMemo.Lines.Count > MaxCount do       ResultsMemo.Lines.Delete(0);     ResultsMemo.Lines.EndUpdate;   end; end;  procedure TBlockAsyncFrm.FormCloseQuery(Sender: TObject;   var CanClose: Boolean); begin   CanClose := true;   if FItemsInTransit > 0 then     if MessageDlg('Some requests in transit, close anyway?', mtWarning,       mbOKCancel, 0) <> mrOK then       CanClose := false; end;  procedure TBlockAsyncFrm.FormDestroy(Sender: TObject); begin   FWorkerThread.Terminate;   FBuf.ResetState;   FWorkerThread.WaitFor;   FBuf.Free;   FWorkerThread.Free; end;  end. |

запрашивает у пользователя два числа - начало и конец диапазона. Эти числа поступают в структуру запроса, а указатель на нее асинхронно записывается в БАБ. Несколько позже рабочий поток

|  |
| --- |
| **Код** |
| unit PrimeRangeThread;  interface  uses   Classes, BlockToAsyncBuf;  type   TPrimeRangeThread = class(TThread)   private     { Private declarations }     FBuf: TBlockToAsyncBuf;   protected     function IsPrime(TestNum: integer): boolean;     procedure Execute; override;   public   published     property Buf: TBlockToAsyncBuf read FBuf write FBuf;   end;    TRangeRequestType = record     Low, High: integer;   end;    PRangeRequestType = ^TRangeRequestType;    { Results returned in a string list }  implementation  uses SysUtils;  { TPrimeRangeThread }  function TPrimeRangeThread.IsPrime(TestNum: integer): boolean;  var   iter: integer;  begin   result := true;   if TestNum < 0 then     result := false;   if TestNum <= 2 then     exit;   iter := 2;   while (iter < TestNum) and (not terminated) do {Line A}   begin     if (TestNum mod iter) = 0 then     begin       result := false;       exit;     end;     Inc(iter);   end; end;  procedure TPrimeRangeThread.Execute;  var   PRange: PRangeRequestType;   TestNum: integer;   Results: TStringList;  begin   while not Terminated do   begin     PRange := PRangeRequestType(FBuf.BlockingRead);     if Assigned(PRange) then     begin       Assert(PRange.Low <= PRange.High);       Results := TStringList.Create;       Results.Add('Primes from: ' + IntToStr(PRange.Low) +         ' to: ' + IntToStr(PRange.High));       for TestNum := PRange.Low to PRange.High do       begin         if IsPrime(TestNum) then           Results.Add(IntToStr(TestNum) + ' is prime.');       end;       if not FBuf.BlockingWrite(Results) then       begin         Results.Free;         Terminate;       end;     end     else Terminate;   end; end;  end. |

проводит блокирующее чтение и извлекает запрос. Затем он тратит какое-то (различное) время на обработку запроса, определяя, какие числа из этого диапазоне простые. По окончании обработки он выполняет блокирующую запись, передавая указатель на список строк, заполненный результатами. Основная форма оповещается, что в компоненте есть данные, для чтения, и тогда она читает список строк из БАБ и копирует результаты в Memo.   
Стоит отметить два основных момента кода основной формы. Первый состоит в том, что интерфейс пользователя корректно обновляется в соответствии с управлением потоками данных для буфера. После того, как запрос будет подан, кнопка запроса будет запрещена. Она восстанавливается, только когда форма получает от БАБ событие OnWrite, указывающее, что можно безопасно записывать новые данные. Данная реализация устанавливает размер двунаправленного буфера 4. Это совсем немного, так что пользователь может убедиться, что после посылки четырех запросов, которые долго обрабатываются, кнопка остается запрещенной, пока один из запросов не обработан. Аналогично, если основная форма не может обработать уведомления о чтении из BAB достаточно быстро, рабочий поток будет заблокирован.   
Второй момент - когда основная форма уничтожена, деструктор использует метод БАБ ResetState, как описано ранее, чтобы гарантировать корректную очистку потоков и освобождение буфера. Невозможность осуществления этого может привести к нарушению доступа (access violation).   
Код рабочего потока довольно прост. Имеет смысл отметить, что поскольку он использует блокировку при чтении и записи, то использует процессор только при активной обработке запроса: если он не может получить запрос или послать ответ из-за перегрузки буфера, то он заблокирован.   
  
Мы достигли нашей цели!   
  
Небольшое резюме о том, чего мы добились с помощью этого компонента:

* Невидимая передача данных между потоком VCL и рабочими потоками.
* Все подробности синхронизации спрятаны внутри БАБ (за исключением ResetState).
* Управление потоками данных между VCL и рабочими потоками.
* Никаких циклов опроса или занятости: процессор используется эффективно.
* Не используется Synchronize. Потоки без необходимости не блокируются.

Читателя можно простить, если он думает, что все его проблемы позади...   
  
Заметили утечку памяти?  
  
Как в предыдущей, так и в этой главе мы уклонились от решения основной проблемы: элементы в созданных нами буферах не уничтожаются корректно при разрушении буфера. При первоначальном проектировании этих буферных структур, был принят подход, подобный используемому в TList: список или буфер только обеспечивает хранение и синхронизацию. Правильное распределение памяти для объектов и их освобождение - задача потока, использующего буфер.   
Этот упрощенный метод создает существенные трудности. В общем случае чрезвычайно трудно проверить, что буфер обоих направлениях пустой перед его уничтожением. В вышеописанном примере, наиболее просто использующем буфер, есть четыре потока, четыре мьютекса или критические секции, и шесть семафоров для всей системы. Определение состояния всех потоков и обеспечение совершенно корректного выхода в такой ситуации, очевидно, невозможно.   
В данном примере это было решено хранением счетчиков того, сколько запросов необслужены в каждый момент времени. Если мы получили столько же ответов, сколько было запросов, то мы можем быть уверены, что все буферы пусты.   
  
Избавляемся от утечки.  
  
Один подход - позволить разным буферам осуществлять callback-вызовы, которые во время очистки уничтожат различные объекты, содержащиеся в этих буферах. В общем случае это будет работать, но есть вероятность злоупотреблений, и реализация такой схемы, скорее всего, на практике получится неаккуратной.   
Другая возможность - создать общую буферную схему управления, которая отслеживает конкретные типы объектов, храня информацию о том, когда они входят и выходят из разных буферов приложения. И опять же, реализация, вероятно, получится довольно сомнительной, и потребует довольно сложного механизм отслеживания ссылок, выполняя работу, которая должна быть в действительности проста.   
Наилучшее решение - сделать буферные структуры аналогичными TObjectList, т.е. все элементы, помещаемые в буферы - *экземпляры классов*. Это позволит потоку, выполняющему операцию очистки, вызывать подходящий деструктор для каждого элемента в буфере. Даже более того - используя типы ссылок на класс, мы можем автоматически выполнять по время выполнения проверку типов объектов, проходящих через буфер, и создать безопасный по отношению к типу набор буферов.   
Реализация подобной схемы оставим как упражнение читателю. Не требуется никаких изменений в основных механизмах синхронизации, но потребуют модификации процедуры для чтения и записи и реализация деструкторов для ограниченного буфера и классов потоков.   
  
Проблемы просмотра.  
  
При реализации двунаправленного буфера возможно обеспечить достаточно разумный механизм для просмотра буферов, чтобы увидеть, сколько в них элементов. Возможно, что при просмотре двунаправленного буфера счетчики свободных и использованных элементов не всегда будут точны, поскольку оба действия не могут выполняться атомарно. Тем не менее, гарантировано, что при только одном потоке чтения и записи в каждом направлении операции просмотра могут быть использованы как верный признак того, что действие будет успешным, без блокировки.   
С асинхронным буфером проблема сложнее, потому что при данной реализации невозможно обеспечить гарантированно правильный контроль за состоянием буфера. Дело в том, что есть по существу два буфера в каждом направлении, ограниченный буфер и вспомогательный, для единственного элемента. Не предусмотрено никакого механизма для глобальной блокировки обоих буферов, чтобы можно было атомарно определять их статус.   
В компоненте сделана попытка предоставить некоторые возможности просмотра, обеспечивая грубый подсчет элементов, проходящих через буферы, причем сделанно это преднамеренно, чтобы программист не заблуждался, думая, что результаты могут быть точны! Возможно ли сделать лучше?   
  
Промежуточный буфер.   
  
Наилучший путь улучшить ситуацию - полностью удалить промежуточный буфер. Это вполне возможно, если немного подумать, но требует переписывания всего кода буферизации. Мы должны сделать новый ограниченный буфер с несколько другой семантикой. Этот новый ограниченный буфер должен:

* Реализовать блокирующие чтение и запись на одном конце, как и раньше.
* На другом конце реализовать асинхронные запись и чтение ( просто успех/ошибка без блокировки), и вдобавок реализовать методы "Block Until" ("блокировать, пока ..."). Эти методы будут блокировать поток, пока предстоящей операции чтения или записи не будет гарантирован успех.

При таком способе можно использовать потоки чтения и записи, чтобы посылать уведомления, блокируя их, пока действие не будет возможным, а поток VCL мог бы выполнять фактические операции чтения и записывать в ограниченный буфер без блокировки.   
С подобной семантикой унас будет только один набор буферов, которыми нужно управлять, и сравнительно легко обеспечить атомарное действие просмотра, которое даст точные результаты. Оставим реализацию читателю как упражнение...   
  
Различные ограничения.  
  
Для всех буферных структур, описанных в последних главах, считается, что программист посылает указатели на правильную память, а не NIL. Некоторые читатели могут обратить внимание, что часть кода потоков чтения и записи неявно подразумевает, что NIL не будет послан через буфер. Это несложно исправить с помощью введения флагов достоверности буфера , но за счет некоторого ухудшения кода.   
Другое (скорее теоретическое) ограничение состоит в том, что конечный пользователь этого компонента может, в принципе, создать очень большое число буферов. Осеовные принципы программирования потоков для Win32 говорят, что обычно стоит ограничить количество потоков примерно до шестнадцати на приложение, что позволяет использовать восемь компонентов БАБ. Поскольку нет ограничений на число рабочих потоков, выполняющих операции блокировки для БАБ, кажется разумным иметь только один БАБ для приложения и использовать для связи между одним потоком VCL и всеми рабочими потоками. При этом, конечно, подразумевается, что все рабочие потоки выполняет одну и ту же работу. Чаще всего такой вариант вполне приемлем, так как большинство приложений Delphi должны порождать много потоков лишь при выполнении фоновых операций, требующих существенных затрат времени.   
  
Обратная сторона монеты: Потоковые буферы.   
  
Итак, все обсуждаемые буферные структуры использовали буферы указателей для передачи данных. Хотя это и полезно для дискретных операций, большинство I/O операций используют потоки данных. Все буферные структуры имеют приблизительный эквивалент, а именно, потоки данных, которые, в общем и целом, могут обрабатываться аналогичными способами. Есть несколько различий, которые имеет смысл отметить:

* При буферизации потоков данных невозможно использовать семафоры, чтобы следить за конкретным количеством элементов в буфере. Вместо этого семафоры используются в **двоичном** режиме, имея счетчик только 1 или 0. При чтении или записи с потоковыми буферами, приходится вычислять, будет ли буфер заполнен или опустошен каждым действием. Если одно из этих событий произойдет, то именно столько байтов по возможности пересылается, и затем поток, если нужно, блокируется.
* Так как состояние блокировки потоков чтения и записи вычисляется "на лету", его надо сохранять, записывая, блокирован ли каждый поток или работает. Это состояние затем используется при последующих операциях чтении или записи для расчета, должны ли разблокироваться "конкурирующие" потоки для конкретного действия чтения или записи. Это отчасти усложняет расчеты блокировки и разблокирования, но общий принцип остается тем же.
* Схемы уведомления для потоковых буферов модифицируются аналогично. Нынешняя схема уведомлений посылает одно уведомление для каждого действия чтения или записи. Компоненты БАБ, работающие с потоками данных, посылают уведомления, основываясь на том, полон или пуст промежуточный буфер (или его эквивалент). Поскольку уведомления можно считать асинхронным аналогом операций Signal или ReleaseSemaphore, эта модификация делается так же, как описано выше.

Есть еще много интересного, что можно было бы обсуждать по этой теме. Если читатель хочет увидеть рабочий пример потоковой буферизации, он может обратиться к коду в последней главе.

Глава 11. Синхронизаторы и события (Events).   
  
Содержание:

* Дополнительные механизмы синхронизации.
* Как добиться оптимальной эффективности.
* Простой MREWS.
* Важные моменты реализации.
* Пример использования простого MREWS.
* Введение в события (Events).
* Моделирование событий с помощью семафоров.
* Простой MREWS с использованием событий.
* MREWS в Delphi.

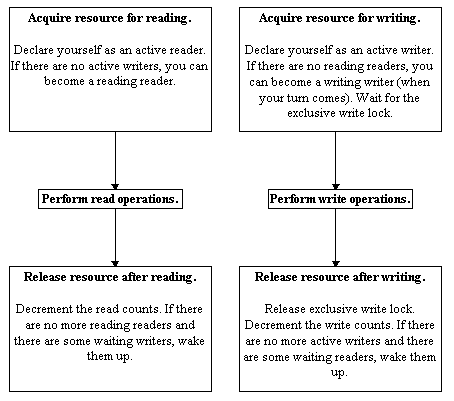
Дополнительные механизмы синхронизации.  
  
В материалах, рассмотренных в предыдущих главах, описаны все основные механизмы синхронизации. В общем, семафоры и мьютексы позволяют программисту, хотя и с некоторыми усилиями, создать все другие механизмы синхронизации. Несмотря на это, есть некоторые ситуации, которые очень часто встречаются в многопоточном программировании, но с использованием описанных механизмов обеспечить синхронихацию в них нелегко. Для решения этих проблем будут введены два новых примитива синхронизации: The Multi Read Exclusive Write Synchronizer (один пишет, многие читают) и Event (событие). Первый теперь является частью библиотеки VCL Delphi, а второй имеется в Win32 API.   
  
Как добиться оптимальной эффективности.  
  
До сих пор все действия с коллективными ресурсами были взаимоисключающими. Все операции чтения и записи были защищенными в том смысле, что в любой момент могло происходить только одно действие чтения или одно - записи. Тем не менее, во многих практически важных ситуациях, когда критический ресурс должен быть доступен большому количеству потоков, такой метод может оказаться неэффективным. Исключающая блокировка фактически скорее является перестраховкой. Напомню, что в Главе 6 отмечалось, что включает необходимая минимальная синхронизация:

* Операции чтения могут выполняться параллельно.
* Операции записи не могут выполняться одновременно со чтением.
* Операции записи не могут выполняться параллельно с другими операциями записи.

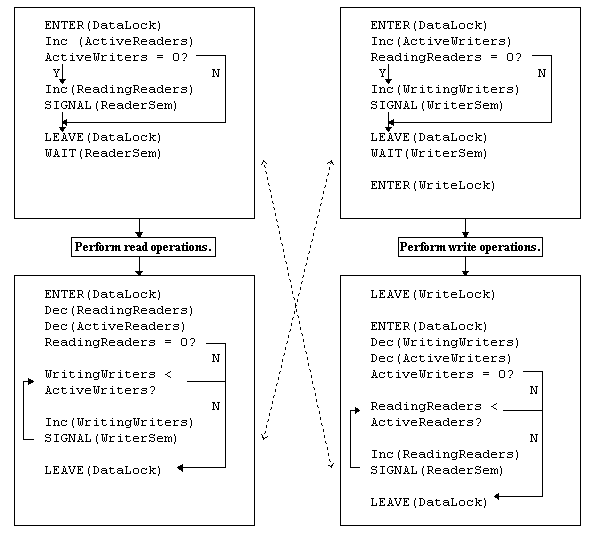
Учитывая абсолютный минимум контроля за параллельным исполнением, возможно добиться значительного увеличения производительности. Наибольшее повышение производительности реализуется, когда из большого числа потоков производится сразу много действий чтения, а операции записи сравнительно редки, и осуществляются лишь небольшим числом потоков.   
Эти условия удовлетворяются на практике довольно часто. Например, база данных склада компании может содержать много элементов, и могут происходить многочисленные действия чтения для определения доступности неких товаров. С другой стороны, база данных корректируется, только когда товары действительно заказываются или отгружаются. Аналогично списки членов общества могут часто проверяться для нахождения адресов, посылки почтовых отправлений или проверки подписки, но введение новых участников, удаление их, или изменение адресов будет происходить довольно редко. Та же самая ситуация и в работе компьютера: основные списки глобальных ресурсов в программе могут часто читаться, но редко изменяться. Требуемый уровень управления параллельным исполнением обеспечивается примитивом, который называется **Multiple Read Exclisive Write Synchronizer**, в дальнейшем **MREWS**.   
Большинство синхронизаторов поддерживают четыре основных действия: **StartRead**, **StartWrite**, **EndRead** and **EndWrite**. Поток вызывает StartRead для конкретного синхронизатора, когда он хочет читать из коллективного ресурса. Затем он выполняет одно или более действий чтения, которые гарантированно будут атомарными и последовательными. Как только поток завершает чтение, то вызывает EndRead. Если две операции чтения выполняются внутри данной пары вызовов StartRead и EndRead, то полученные при этом данные всегда правильные: никаких операций записи между вызовами StartRead и EndRead уже не произойдет.   
Аналогично при выполнении серии операций записи поток вызывает StartWrite. Затем он может выполнять одно или несколько действий записи, и можно быть уверенным, что все операции записи атомарны. По окончании записи поток вызывает EndWrite. Операции записи не будут перекрываться другими операциями записи, а потоки чтения не смогут получить некорректных результатов из-за чтения в процессе записи.   
  
Простой MREWS.  
  
Есть несколько путей осуществления примитива MREWS. В VCL содержится довольно сложная реализация. Для того, чтобы познакомиться с основными принципами, здесь приведена более простая, но несколько менее функциональная реализация с использованием семафоров. Простой MREWS содержит следующие элементы:

* Критическая секция для ограничения доступа к разделяемым данным (DataLock).
* Целый счетчик числа активных потоков чтения (ActRead).
* Целый счетчик числа читающих потоков (ReadRead).
* Целый счетчик числа активных потоков записи (ActWrite).
* Целый счетчик числа записывающих потоков (WriteWrite).
* Два семафора, называемые Reader (читатель) и Writer (писатель) (ReaderSem и WriterSem).
* Критическая секция для осуществления полного исключения записи (WriteLock).

Чтение и запись можно описать так:

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig22.gif)

Чтение или запись проводится в два этапа. Сначала идет активная стадия, когда поток указывает, что он намерен читать или писать. Когда это происходит, поток может быть заблокирован, в зависимости от того, не идет ли уже процесс другого чтения или записи. Когда он разблокируется, то переходит ко второй стадии, выполняет операции чтения или записи, затем освобождает ресурс, устанавливая в соответствующие значения счетчики активных, читающих или записывающих потоков. Если этот поток - последний активный поток чтения или записи, он разблокирует все потоки, которые прежде были заблокированы в результате тех действий, которые он выполнял (чтение или запись). Следующая диаграмма иллюстрирует это более подробно.

[](http://files.vingrad.ru/petrovich/delphi_winapi_potoki/fig23.gif)

К этому моменту реализация такого вида синхронизатора должна быть уже очевидна. Вот она

|  |
| --- |
| **Код** |
| unit SimpleSync;  { Martin Harvey 27/5/2000 }  interface  uses Windows;  type   TSimpleSynchronizer = class(TObject)   private     FDataLock, FWriteLock: TRTLCriticalSection;     FActRead, FReadRead, FActWrite, FWriteWrite: integer;     FReaderSem, FWriterSem: THandle;   protected   public     constructor Create;     destructor Destroy; override;     procedure StartRead;     procedure StartWrite;     procedure EndRead;     procedure EndWrite;   published   end;  implementation  constructor TSimpleSynchronizer.Create; begin   inherited Create;   InitializeCriticalSection(FDataLock);   InitializeCriticalSection(FWriteLock);   FReaderSem := CreateSemaphore(nil, 0, High(Integer), nil);   FWriterSem := CreateSemaphore(nil, 0, High(Integer), nil);   { Initial values of 0 OK for all counts } end;  destructor TSimpleSynchronizer.Destroy; begin   DeleteCriticalSection(FDataLock);   DeleteCriticalSection(FWriteLock);   CloseHandle(FReaderSem);   CloseHandle(FWriterSem);   inherited Destroy; end;  procedure TSimpleSynchronizer.StartRead; begin   EnterCriticalSection(FDataLock);   Inc(FActRead);   if FActWrite = 0 then   begin     Inc(FReadRead);     ReleaseSemaphore(FReaderSem, 1, nil);   end;   LeaveCriticalSection(FDataLock);   WaitForSingleObject(FReaderSem, INFINITE); end;  procedure TSimpleSynchronizer.StartWrite; begin   EnterCriticalSection(FDataLock);   Inc(FActWrite);   if FReadRead = 0 then   begin     Inc(FWriteWrite);     ReleaseSemaphore(FWriterSem, 1, nil);   end;   LeaveCriticalSection(FDataLock);   WaitForSingleObject(FWriterSem, INFINITE);   EnterCriticalSection(FWriteLock); end;  procedure TSimpleSynchronizer.EndRead; begin   EnterCriticalSection(FDataLock);   Dec(FReadRead);   Dec(FActRead);   if FReadRead = 0 then   begin     while FWriteWrite < FActWrite do     begin       Inc(FWriteWrite);       ReleaseSemaphore(FWriterSem, 1, nil);     end;   end;   LeaveCriticalSection(FDataLock); end;  procedure TSimpleSynchronizer.EndWrite; begin   LeaveCriticalSection(FWriteLock);   EnterCriticalSection(FDataLock);   Dec(FWriteWrite);   Dec(FActWrite);   if FActWrite = 0 then   begin     while FReadRead < FActRead do     begin       Inc(FReadRead);       ReleaseSemaphore(FReaderSem, 1, nil);     end;   end;   LeaveCriticalSection(FDataLock); end;  end. |

Если читателю еще не все понятно, не паникуйте! Этот объект синхронизации с первого взгляда понять нелегко! Изучайте его внимательно в течение нескольких минут, и если у вас начнет двоиться в глазах прежде, чем вы разберетесь, то не беспокойтесь об этом и двигайтесь дальше!   
  
Важные моменты реализации.  
  
В схеме синхронизации есть асимметрия: потоки, потенциально желающие читать, будут заблокированы перед чтением, если есть *активные* потоки записи, в то время как потоки, желающие писать, перед записью блокируются, если есть *читающие* потоки. Это дает приоритет потокам записи; это очень существенно, если запись происходит реже, чем чтение. Этот подход не является единственно допустимым, а так как все расчеты - должен ли поток быть заблокирован, или нет - происходят в критической секции, то вполне допустимо сделать синхронизатор симметричным. Недостаток этого метода состоит в том, что если происходит много параллельных операций чтения, то они могут полностью заблокировать запись. Конечно, возможна и противоположная ситуация, когда непрерывная запись останавливает операции чтения.   
Стоит также обратить внимание на использование семафоров для захвата ресурса для чтения или записи: операции ожидания семафоров всегда должны выполняться за пределами критической секции, которая защищает коллективные данные. Таким образом, условная сигнализация семафора внутри критической секции нужна только для проверки, что операция ожидания не блокирующая.   
  
Пример использования простого MREWS.  
  
Чтобы продемонстрировать, что делает MREWS, необходимо немного отойти от рассмотренных до сих пор примеров. Представьте себе, что большому количеству потоков необходимо отслеживать статус множества файлов в определенной директории. Эти потоки хотят знать, изменился ли файл с тех пор, как поток в последний раз проверял его статус. К сожалению, файлы в системе могут быть изменены множеством других программ, так что невозможно одной программе отследить всевозможные файловые операции, выполняемые со всеми файлами.   
В примере имеется рабочий поток, которая пробегает по всем файлам директории, рассчитывая простую контрольную сумму для каждого файла. Он делает это снова и снова, работая бесконечно. Данные хранятся в списке, который содержит синхронизатор MREW, что позволяет, таким образом, многим потокам читать контрольные суммы одного или нескольких файлов.   
Сначала давайте рассмотрим код списка контрольных сумм. Вот он.

|  |
| --- |
| **Код** |
| unit ChecksumList;  { Martin Harvey 29/5/2000 }  interface  uses SimpleSync, Classes, SysUtils;  type   TChecksumList = class   private     FCheckList: TList;     FSync: TSimpleSynchronizer;   protected     { Find function returns -1 if not found }     function FindFileIndex(FileName: string): integer;     function NoLockGetFileList: TStringList;     function NoLockGetChecksum(FileName: string): integer;   public     constructor Create;     destructor Destroy; override;     procedure SetChecksum(FileName: string; Checksum: integer);     procedure RemoveChecksum(FileName: string);     function GetChecksum(FileName: string): integer;     function GetFileList: TStringList;     function GetChecksumList: TStringList;   end;  implementation  type   TCheckSum = record     FileName: string;     Checksum: integer;   end;    PCheckSum = ^TCheckSum;  constructor TChecksumList.Create; begin   inherited Create;   FCheckList := TList.Create;   FSync := TSimpleSynchronizer.Create; end;  destructor TCheckSumList.Destroy;  var   iter: integer;   CurSum: PCheckSum;  begin   if FCheckList.Count > 0 then   begin     for iter := 0 to FCheckList.Count - 1 do     begin       CurSum := PCheckSum(FCheckList.Items[iter]);       if Assigned(CurSum) then Dispose(CurSum);     end;   end;   FCheckList.Free;   FSync.Free;   inherited Destroy; end;  function TCheckSumList.FindFileIndex(FileName: string): integer;  var   iter: integer;   CurSum: PCheckSum;  begin   result := -1;   if FCheckList.Count > 0 then   begin     for iter := 0 to FCheckList.Count - 1 do     begin       CurSum := PCheckSum(FCheckList.Items[iter]);       Assert(Assigned(CurSum));       if AnsiCompareText(FileName, CurSum.FileName) = 0 then       begin         result := iter;         exit;       end;     end;   end; end;  procedure TCheckSumList.SetChecksum(FileName: string; Checksum: integer);  var   CurSum: PCheckSum;   CurIndex: integer;  begin   FSync.StartWrite;   CurIndex := FindFileIndex(FileName);   if CurIndex >= 0 then     CurSum := PCheckSum(FCheckList.Items[CurIndex])   else   begin     New(CurSum);     FCheckList.Add(CurSum);   end;   CurSum.FileName := FileName;   CurSum.Checksum := Checksum;   FSync.EndWrite; end;  procedure TCheckSumList.RemoveChecksum(FileName: string);  var   CurIndex: integer;  begin   FSync.StartWrite;   CurIndex := FindFileIndex(FileName);   if CurIndex >= 0 then   begin     FCheckList.Delete(CurIndex);     FCheckList.Pack;   end;   FSync.EndWrite; end;  function TCheckSumList.NoLockGetChecksum(FileName: string): integer;  var   CurIndex: integer;   CurSum: PCheckSum;  begin   result := 0;   CurIndex := FindFileIndex(FileName);   if CurIndex >= 0 then   begin     CurSum := PCheckSum(FCheckList.Items[CurIndex]);     Assert(Assigned(CurSum));     result := CurSum.Checksum;   end; end;  function TCheckSumList.GetChecksum(FileName: string): integer; begin   FSync.StartRead;   result := NoLockGetChecksum(FileName);   FSync.EndRead; end;  function TCheckSumList.NoLockGetFileList: TStringList;  var   iter: integer;   CurSum: PCheckSum;  begin   result := TStringList.Create;   if FCheckList.Count > 0 then   begin     for iter := 0 to FCheckList.Count - 1 do     begin       CurSum := PCheckSum(FCheckList.Items[iter]);       Assert(Assigned(CurSum));       result.Add(CurSum.FileName);     end;   end;   result.Sort; end;  function TCheckSumList.GetFileList: TStringList; begin   FSync.StartRead;   result := NoLockGetFileList;   FSync.EndRead; end;  function TCheckSumList.GetChecksumList: TStringList;  var   iter: integer;  begin   FSync.StartRead;   result := NoLockGetFileList;   if result.Count > 0 then   begin     for iter := 0 to result.Count - 1 do     begin       result.strings[iter] := result.strings[iter]         + ' ' + IntToStr(NoLockGetChecksum(result.strings[iter]));     end;   end;   FSync.EndRead; end;  end. |

Основные операции:

* Установить контрольную сумму определенного файла. При этом в список добавляется элемент, соответствующий файлу, если его еще не было.
* Получить контрольную сумму определенного файла. Результат будет 0, если файл не найден.
* Удалить файл из списка.
* Получить список строк со всеми именами файлов.
* Получить список строк со всеми именами файлов и соответствующими контрольными суммами.

Для всех этих публично доступных операций в начале и конце их производятся соответствующие вызовы синхронизации.  
Заметьте, что есть методы, название которых начинается с "NoLock". Это методы, которые нужно вызывать из нескольких опубликованных методов. Класс написан таким образом из-за ограничений нашего синхронизатора: *вложенные вызовы начала чтения или записи недопустимы*. Все действия, которые используют простой синхронизатор, должны вызывать только StartRead или StartWrite, если они уже закончили все предыдущие операции чтения или записи. Более подробно это будет обсуждаться позже. За этим исключением, большая часть кода списка контрольных сумм довольно стандартна, представляет собой в основном обычную обработку списка, и не должна составить никаких трудностей для большинства Delphi-программистов.  
Теперь рассмотрим код рабочего потока

|  |
| --- |
| **Код** |
| unit CheckThread;  { Martin Harvey 30/5/2000 }  interface  uses   Classes, Windows, ChecksumList, SysUtils;  type    TState = (sGetCurrentCRCs,     sBuildFileList,     sRemoveCRCs,     sCheckFile,     sDone);    TStateReturn = (rvOK, rvFail1, rvFail2);    TActionFunc = function: TStateReturn of object;    TStateActions = array[TState] of TActionFunc;    TNextStates = array[TState, TStateReturn] of TState;    TCheckThread = class(TThread)   private     FStartDir: string;     FCurrentState: TState;     FActionFuncs: TStateActions;     FNextStates: TNextStates;     FInternalFileList: TStringList;     FExternalFileList: TStringList;     FExternalCRCList: TStringList;     FCheckList: TChecksumList;     FFileToProcess: integer;   protected     procedure InitActionFuncs;     procedure InitNextStates;     function GetCurrentCRCs: TStateReturn;     function BuildFileList: TStateReturn;     function RemoveCRCs: TStateReturn;     function CheckFile: TStateReturn;     procedure Execute; override;   public     constructor Create(CreateSuspended: boolean);     destructor Destroy; override;     property StartDir: string read FStartDir write FStartDir;     property CheckList: TChecksumList read FCheckList write FCheckList;   end;  implementation  { TCheckThread }  {(\*} {Prettyprinter auto-formatting off}  const   BaseStateTransitions:TNextStates = (                       {rvOK}             {rvFail1}        {rvFail2} {sGetCurrentCRCs } ( sBuildFileList,      sDone,           sDone ), {sBuildFileList  } ( sRemoveCRCs,         sDone,           sDone ), {sRemoveCRCs     } ( sCheckFile,          sDone,           sDone ), {sCheckFile      } ( sCheckFile,          sGetCurrentCRCs, sDone ), {sDone           } ( sDone,               sDone,           sDone ));   {\*)}{Prettyprinter auto-formatting on}  procedure TCheckThread.InitActionFuncs; begin   FActionFuncs[sGetCurrentCRCs] := GetCurrentCRCs;   FActionFuncs[sBuildFileList] := BuildFileList;   FActionFuncs[sRemoveCRCs] := RemoveCRCs;   FActionFuncs[sCheckFile] := CheckFile; end;  procedure TCheckThread.InitNextStates; begin   FNextStates := BaseStateTransitions; end;  function TCheckThread.GetCurrentCRCs: TStateReturn; begin   FExternalFileList.Free;   FExternalFileList := nil;   FExternalCRCList.Free;   FExternalCRCList := nil;    FExternalFileList := FCheckList.GetFileList;   FExternalCRCList := FCheckList.GetChecksumList;   result := rvOK; end;  function TCheckThread.BuildFileList: TStateReturn;  var   FindRet: integer;   SearchRec: TSearchRec;  begin   FInternalFileList.Clear;   FindRet := FindFirst(StartDir + '\*.\*', faAnyFile and not faDirectory, SearchRec);   if FindRet <> 0 then     result := rvFail1   else   begin     while FindRet = 0 do     begin       { Found a file.}       FInternalFileList.Add(SearchRec.Name);       FindRet := FindNext(SearchRec);     end;     result := rvOK;   end;   FindClose(SearchRec);   FFileToProcess := 0; end;  function TCheckThread.RemoveCRCs: TStateReturn;  var   iter: integer;   dummy: integer;  begin   FInternalFileList.Sort;   FExternalFileList.Sort;   if FExternalFileList.Count > 0 then   begin     for iter := 0 to FExternalFileList.Count - 1 do     begin       if not FInternalFileList.Find(FExternalFileList[iter], dummy) then         FCheckList.RemoveChecksum(FExternalFileList[iter]);     end;   end;   result := rvOK;  end;  function TCheckThread.CheckFile: TStateReturn;  var   FileData: TFileStream;   MemImage: TMemoryStream;   Data: byte;   Sum: integer;   iter: integer;  begin   if FFileToProcess >= FInternalFileList.Count then   begin     result := rvFail1;     exit;   end;   Sum := 0;   FileData := nil;   MemImage := nil;   try     FileData := TFileStream.Create(StartDir + FInternalFileList[FFileToProcess],       fmOpenRead or fmShareDenyWrite);     FileData.Seek(0, soFromBeginning);     MemImage := TMemoryStream.Create;     MemImage.CopyFrom(FileData, FileData.Size);     MemImage.Seek(0, soFromBeginning);     for iter := 1 to FileData.Size do     begin       MemImage.ReadBuffer(Data, sizeof(Data));       Inc(Sum, Data);     end;     FileData.Free;     MemImage.Free;     if (FCheckList.GetChecksum(FInternalFileList[FFileToProcess]) <> Sum) then       FCheckList.SetChecksum(FInternalFileList[FFileTOProcess], Sum);   except     on EStreamError do     begin       FileData.Free;       MemImage.Free;     end;   end;   Inc(FFileToProcess);   result := rvOK; end;  procedure TCheckThread.Execute; begin   SetThreadPriority(Handle, THREAD\_PRIORITY\_IDLE);   while not (Terminated or (FCurrentState = sDone)) do     FCurrentState := FNextStates[FCurrentState, FActionFuncs[FCurrentState]]; end;  constructor TCheckThread.Create(CreateSuspended: boolean); begin   inherited Create(CreateSuspended);   InitActionFuncs;   InitNextStates;   FInternalFileList := TStringList.Create; end;  destructor TCheckThread.Destroy; begin   FInternalFileList.Free;   FExternalFileList.Free;   FExternalCRCList.Free;   inherited Destroy; end;  end. |

Эта поток несколько отличается от большинства примеров потоков, которые я описывал до сих пор, поскольку он реализован как машина состояний. Метод Execute просто выполняет функцию действия для каждого состояния, и, в зависимости от результата функции, ищет следующее состояние, требуемое согласно таблице переходов. Одна функция действия считывает список файлов из объекта списка с контрольными суммами, вторая удаляет ненужные контрольные суммы из списка, а третья вычисляет контрольную сумму для конкретного файла и при необходимости корректирует ее. Вся прелесть использования машины состояний в том, что она делает завершение потока намного более четким. Метод Execute вызывает функции действия, ищет следующее состояние и проверяет поток на завершение в цикле while. Так как каждой функции действия обычно требуется несколько секунд для выполнения, завершение потока происходит очень быстро. Кроме того, в коде необходима только одна проверка на завершение, что делает код весьма прозрачным. Мне также нравится, что вся логика машины состояний осуществляется одной строкой кода. Во всем этом есть определенная аккуратность.   
И наконец, рассмотрим код главной формы

|  |
| --- |
| **Код** |
| unit SyncForm;  interface  uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,   ExtCtrls, StdCtrls, ChecksumList, CheckThread;  type   TForm1 = class(TForm)     FileMemo: TMemo;     Timer1: TTimer;     procedure FormCreate(Sender: TObject);     procedure FormDestroy(Sender: TObject);     procedure Timer1Timer(Sender: TObject);   private     { Private declarations }     FChecksumList: TChecksumList;     FCheckThread: TCheckThread;   public     { Public declarations }   end;  var   Form1: TForm1;  implementation  {$R \*.DFM}  procedure TForm1.FormCreate(Sender: TObject); begin   FChecksumList := TChecksumList.Create;   FCheckThread := TCheckThread.Create(true);   with FCheckThread do   begin     StartDir := 'D:\Netscape Profiles\Martin\News\host-newsgroups.borland.com\';     CheckList := FChecksumList;     Resume;   end; end;  procedure TForm1.FormDestroy(Sender: TObject); begin   with FCheckThread do   begin     Terminate;     WaitFor;     Free;   end;   FChecksumList.Free; end;  procedure TForm1.Timer1Timer(Sender: TObject);  var   TempList: TStringList;  begin   TempList := FChecksumList.GetChecksumList;   with FileMemo do   begin     with Lines do     begin       BeginUpdate;       Assign(TempList);       EndUpdate;     end;     selstart := gettextlen;     perform(em\_scrollcaret, 0, 0);   end;   TempList.Free; end;  end. |

Он относительно прост: поток и список контрольных сумм создаются при запуске программы и уничтожаются при ее закрытии. Список файлов и их контрольных сумм регулярно отображается по таймеру. Имя директории, за которой будет вестись наблюдение, записано в этом файле; читатели, желающие запустить программу, могут захотеть изменить директорию или, возможно, так модифицировать программу, чтобы задавать название директории при запуске программы.   
Эта программа не выполняет операций над разделяемыми данных строго атомарным способом. Есть несколько мест в потоке обновления, где неявно подразумевается, что локальные данные корректны, в то время как соответствующий файл, возможно, был изменен. Хороший пример этого - функция потока "check file". Когда контрольная сумма файла вычислена, поток читает загруженную контрольную сумму для этого файла и корректирует ее при несовпадении с только что вычисленной. Эти две операции не атомарны, поскольку не атомарны многократные вызовы объекта списка контрольных сумм. Это проистекает главным образом из того, что с нашим простым синхронизатором не работают вложенные вызовы синхронизации. Одно из возможных решение - дать объекту списка контрольных сумм два новых метода: "Блокировка для чтения" и "Блокировка для записи". Блокировку можно было бы применить для монопольного захвата общих данных, либо для чтения, либо для записи, и тогда проводить многократные операции чтения или записи. Тем не менее, это еще не решает всех возможных проблем синхронизации. Более передовые решения будут обсуждаться в этой главе позже.   
Так как внутренняя работа синхронизатора происходит на уровне Delphi, можно получить оценку того, как часто в действительности происходят конфликты потоков. Если установить точку останова в циклы while процедур EndRead и EndWrite, то программа будет приостановлена, если поток чтения или записи были блокированы, пытаясь получить доступ к ресурсу. Реально программа попадает в точку останова, когда ожидающий поток разблокируется, но все равно можно подсчитать конфликты. В данном примере эти конфликты случаются совсем редко, особенно при низкой загрузке, но если количество файлов и контрольных сумм становится большим, конфликты происходят все чаще, поскольку больше времени тратится на получение и копирование разделяемых данных.   
  
Введение в события (Events).  
  
События, возможно, одни из самых простых для понимания примитивов синхронизации, но я предпочел оставить рассказ о них до этого момента, поскольку их лучше всего использовать совместно с другими методами синхронизации. Есть два типа событий: события ручного сброса **manual reset** и автоматического **auto reset**. Сейчас мы рассмотрим события ручного сброса. Событие работает подобно светофору (или стоп-сигналу для читателей из США). У него есть два возможных состояния: сигнальное (аналогично зеленому светофору) и несигнализированное (аналогично красному светофору). Когда событие в сигнальном состоянии, потоки, ожидающие события, не заблокированы и продолжают выполнение. Когда состояние несигнализированное, потоки, ожидающие события, заблокированы, пока оно не перейдет в сигнальное состояние. Win32 API предоставляет набор функций для работы с событиями.

* CreateEvent / OpenEvent: Эти функции подобны другим функциям Win32 для создания или открытия объектов синхронизации. Кроме того, что событие может создаваться или в сигнальном, или в несигнальном состоянии, имеется логический флаг, указывающий, будет ли это событие ручного или автоматического сброса.
* SetEvent: Устанавливает состояние события в сигнальное, запуская таким образом все потоки, ожидающие события, и позволяет новым потокам проходить без блокировки.
* ResetEvent: Устанавливает состояние события в несигнальное, блокируя таким образом все потоки, ожидающие события.
* PulseEvent: Осуществляет установку-сброс события. Следовательно, все потоки, ожидающие события, запускаются, но новые потоки, появившиеся позже, будут блокированы.

Ссобытия с автосбросом являются особым случаем событий с ручным сбросом. Для них сигнальное состояние переходит в несигнализированное, как только один поток пройдет через событие без блокировки, или один поток освобождается. В этом смысле, они работают почти идентично семафорам, и если программист использует события с автосбросом, он должны подумать об использовании вместо них семафоров, чтобы сделать поведение механизма синхронизации более очевидным.   
  
Моделирование событий с помощью семафоров.  
  
Примитив события можно в реальности создать, используя семафоры: возможно использовать семафор для условной блокировки всех потоков, ожидающих примитив события и разблокировать их, когда примитив в сигнальном состоянии. Для того, чтобы так сделать, используется метод, аналогичный алгоритму синхронизатора . У события есть две части состояния: логическая, указывающая, в сигнальном ли состоянии событие, и счетчик числа потоков, блокированных в данный момент семафором события. Вот как реализованы действия:

* CreateEvent: Создается объект события, счетчик блокированных потоков устанавливается нулевым, и сигнальное состояние устанавливается согласно параметру конструктора.
* SetEvent: Устанавливается сигнальное состояние, не блокирующее входящие потоки. Кроме того, проверяется счетчик потоков, блокированных семафором, и если он ненулевой, то семафор повторно сигнализирует, пока все потоки не разблокируются.
* ResetEvent: Сигнальное состояние устанавливается для блокировки входящих потоков.
* PulseEvent: Все потоки, блокированные в данный момент семафором, разблокируются, но изменений в сигнальном состоянии не происходит.
* WaitForEvent: Проверяется сигнальное состояние события. Если событие сигнализировано, то сигнализируется внутренний семафор, и счетчик потоков, блокированных семаформ, уменьшается. Затем счетчик увеличивается и осуществляется ожидание внутреннего семафора.

Вот код моделирования события с использованием семафоров.

|  |
| --- |
| **Код** |
| unit SimulatedEvent;  { Martin Harvey 4/6/2000 }  interface  uses Windows;  type   TSimulatedEvent = class   private     FBlockCount: integer;     FSignalled: boolean;     FDataSection: TRTLCriticalSection;     FBlockSem: THandle;   protected   public     constructor Create(CreateSignalled: boolean);     destructor Destroy; override;     procedure SetEvent;     procedure ResetEvent;     procedure PulseEvent;     procedure WaitFor;   published   end;  implementation  constructor TSimulatedEvent.Create(CreateSignalled: boolean); begin   inherited Create;   FSignalled := CreateSignalled;   InitializeCriticalSection(FDataSection);   FBlockSem := CreateSemaphore(nil, 0, High(Integer), nil); end;  destructor TSimulatedEvent.Destroy; begin   DeleteCriticalSection(FDataSection);   CloseHandle(FBlockSem);   inherited Destroy; end;  procedure TSimulatedEvent.SetEvent; begin   EnterCriticalSection(FDataSection);   FSignalled := true;   while FBlockCount > 0 do   begin     ReleaseSemaphore(FBlockSem, 1, nil);     Dec(FBlockCount);   end;   LeaveCriticalSection(FDataSection); end;  procedure TSimulatedEvent.ResetEvent; begin   EnterCriticalSection(FDataSection);   FSignalled := false;   LeaveCriticalSection(FDataSection); end;  procedure TSimulatedEvent.PulseEvent; begin   EnterCriticalSection(FDataSection);   while FBlockCount > 0 do   begin     ReleaseSemaphore(FBlockSem, 1, nil);     Dec(FBlockCount);   end;   LeaveCriticalSection(FDataSection); end;  procedure TSimulatedEvent.WaitFor; begin   EnterCriticalSection(FDataSection);   if FSignalled then   begin     Dec(FBlockCOunt);     ReleaseSemaphore(FBlockSem, 1, nil);   end;   Inc(FBlockCount);   LeaveCriticalSection(FDataSection);   WaitForSingleObject(FBlockSem, INFINITE); end;  end. |

Если читатель разобрался в работе простого синхронизатора, то этот код должен быть довольно легок для понимания. Реализация могла бы быть немного упрощена заменой циклов while, которые разблокируют потоки, единственным оператором, который увеличивает счетчик семафора на необходимую величину, тем не менее осуществленный здесь метод более соответствует реализации вышеописанного синхронизатора.   
  
Простой MREWS с использованием событий.  
  
Управляющие структуры, требуемые для имитации события с использованием семафоров, подобны структурам, использованным в простом синхронизаторе. Таким образом, имеет смысл попытаться создать синхронизатор, используя события вместо семафоров. Это не слишком трудно:

|  |
| --- |
| **Код** |
| unit EventSync;  { Martin Harvey 5/6/2000 }  interface  uses Windows;  type   TEventSynchronizer = class(TObject)   private     FDataLock, FWriteLock: TRTLCriticalSection;     FReaders, FWriters: integer;     FNoReaders, FNoWriters: THandle;   protected   public     constructor Create;     destructor Destroy; override;     procedure StartRead;     procedure StartWrite;     procedure EndRead;     procedure EndWrite;   published   end;  implementation  constructor TEventSynchronizer.Create; begin   inherited Create;   InitializeCriticalSection(FDataLock);   InitializeCriticalSection(FWriteLock);   FNoReaders := CreateEvent(nil, true, true, nil);   FNoWriters := CreateEvent(nil, true, true, nil); end;  destructor TEventSynchronizer.Destroy; begin   DeleteCriticalSection(FDataLock);   DeleteCriticalSection(FWriteLock);   CloseHandle(FNoReaders);   CloseHandle(FNoWriters);   inherited Destroy; end;  procedure TEventSynchronizer.StartRead;  var   Block: boolean;  begin   EnterCriticalSection(FDatalock);   if FReaders = 0 then     ResetEvent(FNoReaders);   Inc(FReaders);   Block := FWriters > 0;   LeaveCriticalSection(FDataLock);   if Block then     WaitFor |