17 PROGRAMACIÓN CONCURRENTE Y PARALELA

The bearing of a child takes nine months, no matter how many women are assigned

Fred Brooks, The Mythical Man-Month, 1975

Leonardo Paneque y Miguel Katrib

Capítulo del libro Empiece a Programar, empiece con objetos, empiece con C#

Borrador para uso de estudiantes de MATCOM-UH

17.1 INTRODUCCIÓN

Es habitual que se defina a un programa de computadora como una *secuencia de instrucciones a ser ejecutados por un procesador con el propósito de resolver un problema*. Pero lo cierto es desde hace tiempo, las computadoras se diseñan para ejecutar más de una instrucción a la vez por lo que decir secuencia de instrucciones no es lo más exacto.

En un inicio la mayoría de los procesadores (CPU) solo ejecutaban una instrucción por ciclo de reloj, luego se aumentó el escalado[1], añadiéndoles la capacidad de ejecutar más de una instrucción por ciclo. Más tarde se colocaron varios núcleos en un mismo encapsulado físico con lo que se dio lugar al surgimiento de procesadores de múltiples núcleos. Otras tecnologías como el Hyper-Threading de Intel crean dos procesadores lógicos de un único núcleo.

Ya desde los 60s la computadora PDP-8, fabricada por DEC, ofrecía un tipo primitivo de multitarea (*preemtive multitasking*) y tenía un procesador de 12-bits

Estos avances en el hardware han hecho posible que hoy día el paralelismo esté al alcance de la mayoría de los desarrolladores y de los usuarios finales. Ya desde 1998 casi todos los procesadores eran súper escalares, y desde inicio de los 2000 los procesadores con múltiples núcleos pasaron a ser la norma.

Para mejor uniformidad en lo adelante se usará el término **procesador** para referirnos a la unidad de cómputo conformada por un único procesador lógico, es decir la unidad más pequeña capaz de ejecutar una secuencia de instrucciones.

[1] El Pentium de Intel fue el primer súper escalar de la familia x86

La ejecución simultánea de más de un programa en un mismo procesador, es lo que se denomina **multitarea**. Los usuarios de Windows 3.11 ya pudieron empezar a disfrutar de trabajar en un ambiente multitarea cuando podían tener varias aplicaciones ejecutando a la vez (editando un texto y haciendo transparencias). Por supuesto, que ejecutando con un solo procesador este "paralelismo" no era físicamente real, pero en muchos casos daba el plante. El efecto del paralelismo se lograba porque, aunque cada programa se ejecutaba de forma secuencial el sistema operativo era quien administraba y distribuía el tiempo de procesador que le iba dando a cada programa. Este proceso de administración podía llegar a ser bastante complejo ya que requería cargar y descargar en memoria los datos del programa y su estado de ejecución (lo que se denomina cambio de contexto). Esto es lo que daba la sensación, sobre todo cuando se realiza con un procesador suficientemente rápido, de **paralelismo**.

En las arquitecturas de computadoras actuales aún se aplica esta concepción, pero ya reforzada y optimizada para aprovechar la existencia de procesadores más avanzados. Los sistemas operativos modernos se diseñan y desarrollan fuertemente acoplados al hardware para poder aprovechar toda la capacidad de cómputo de estas nuevas arquitecturas.

Pero aun así, no es exacto decir que esta multitarea o concurrencia de varias aplicaciones sea sinónimo de paralelismo...

A una ejecución secuencial de instrucciones se le denomina **hebra** (*thread*). Los términos **concurrente**, **asíncrono**, **multitarea**, **multihebra**, se refieren entonces a la ejecución "simultánea" de varias de estas hebras, pero como ya se ha dicho esta "simultaneidad" puede ser solo en apariencia y no significar necesariamente paralelismo. Tal vez sea más correcto decir también "concurrente" sin que esto implique paralelismo. Mientras que por supuesto decir "en paralelo" sí implica concurrencia.

Para entender mejor la diferencia, note que si se ejecutan los programas A y B (Figura 17.1), de forma concurrente pero sin paralelismo real, el tiempo total de ejecución de sería T=T(A)+T(B); sin embargo si los programas se ejecutaran verdaderamente en paralelo, el tiempo total de ejecución de ambos, sería el del que más demore de los dos, T=Max(T(A), T(B)).

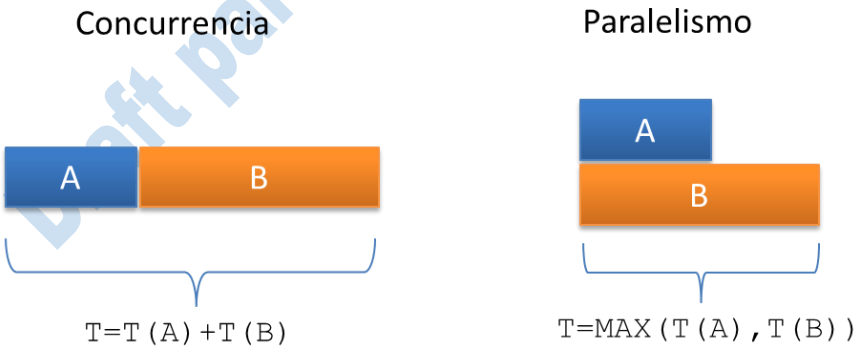


Figura 17.1 Concurrencia y Paralelismo

Puede decirse entonces que **Programación Paralela** es la forma de computación en la que varios cálculos se realizan de forma realmente simultánea (*context switch*). Esta forma

de computación ya data de varias décadas, aunque solo al alcance de programadores en centros de investigación y desarrollo de élite con necesidades más complejas que las del usuario común.

Si el **tiempo de ejecución de un programa** es el total de instrucciones ejecutadas multiplicado por el tiempo promedio de ejecución de cada una; y si cuáles son las instrucciones a ejecutar no varía, entonces la forma de disminuir el tiempo total de ejecución es aumentar la frecuencia del procesador. Esto es lo que se conoce como **escalado por frecuencia** (*frequency scaling*) y fue el factor dominante en la arquitectura de procesadores hasta la irrupción de los múltiples núcleos en el 2004.

La Ley de Moore, una observación empírica publicada en un artículo[2] en 1965, planteaba que la densidad de transistores en un procesador se duplicaría en ciclos de 18-24 meses.

Pero la energía que consume un procesador viene dada por la fórmula $P = C \times V^2 \times F$, donde C es la capacitancia, proporcional a la cantidad de transistores, V es el voltaje y F la frecuencia del procesador. Por lo que a más frecuencia, más energía y más temperatura.

Así que los requerimientos de energía y la dificultad para crear semiconductores adecuados han desacelerado el ritmo del escalado por frecuencia. De modo que la Ley de Moore sigue vigente, aunque a un ritmo menor y el incremento de transistores en los chips va destinado a crear más núcleos en un mismo encapsulado en lugar de un solo núcleo con más velocidad. Lo cual brinda también un aumento de rendimiento comparable o mejor.

Escribir programas concurrentes o paralelos es una tarea complicada ya que los algoritmos tradicionales han sido concebidos pensando en una ejecución secuencial de las instrucciones. Como algunos ingenuamente pudieran pensar, aprovechar ahora la disponibilidad de múltiples núcleos no se va a lograr de forma mágica dividiendo la secuencia de instrucciones entre la cantidad de núcleos. Para resolver un problema con programación paralela hay que adaptar los algoritmos existentes (lo que no siempre es sencillo o siquiera posible). Por ejemplo, el método de Ordenación por Mezcla que se estudió en el Capítulo 8 se puede adaptar para ejecutar en paralelo si cada una de las mitades de un array se ordena en un procesador independiente. Sin embargo, el cálculo de una serie como la de Fibonacci, es imposible de paralelizar porque el cálculo de un valor requiere disponer de los dos valores previos. Para entender el motivo detrás de esta distinción, invitamos al lector a profundizar en el tema estudiando las Condiciones de Bernstein[3].

¿Cuánto podemos mejorar con paralelismo? Para conocerlo recomendamos al lector profundizar con el estudio de la Ley de Amdahl, que plantea la siguiente fórmula:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

S(N) es la **aceleración** (*speedup*) de un programa, dada la cantidad de procesadores

[2] Moore, Gordon E. (1965). "Cramming more components onto integrated circuits". Electronics Magazine. p. 4
[3] Bernstein, A. J. (1 October 1966). "Analysis of Programs for Parallel Processing". IEEE Transactions on Electronic Computers. EC-15 (5): 757–763. doi:10.1109/PGEC.1966.264565.

N y P es la porción del programa que se puede paralelizar.

En este capítulo se tratan los temas relativos a las hebras, así como mecanismos de sincronización entre hebras lo mismo si ejecutan concurrentemente o en paralelo. Se presentan también los conceptos de paralelismo de datos y paralelismo de tareas, sus ventajas y peculiaridades. Se introducen también nuevos patrones de programación concurrente y se exponen los recursos de .NET y del lenguaje C# para esta metodología de programación.

17.2 SINCRONIZACIÓN ENTRE HEBRAS

Una hebra de ejecución es una secuencia de instrucciones que se ejecuta de manera secuencial; ésta es la forma más común de programar. Algoritmos que creamos con métodos de ordenación, búsqueda, ciclos y condicionales, se ejecutan típicamente de forma secuencial por un procesador. Si trabajamos con más de una hebra, es necesario tener una forma de manipular y consultar el contexto de ejecución de éstas. La clase `System.Threading.Thread` es la que en .NET representa una hebra de ejecución y nos facilitará realizar operaciones sobre ellas.

Al programar utilizando más de una hebra, estamos programado de forma concurrente, note que ya mencionamos que concurrente es un término más general que paralelo, así que, sin importar por ahora si hay paralelismo real, ese necesario aprender a escribir programas concurrentes para poder luego explotar las facilidades del paralelismo real..

En una programación concurrente es común que las hebras que ejecutan necesiten en un momento dado **sincronizar** e intercambiar información. Esta comunicación que suele hacerse a través de alguna "memoria compartida" (en lo adelante le llamaremos simplemente variable) puede dar lugar a diversos problemas cuando más de una hebra estén haciendo modificaciones concurrentemente sobre la misma memoria. Existen diversas técnicas y patrones de diseño para disminuir la probabilidad de riesgo y garantizar la robustez de la comunicación, pero no es posible anticiparse del todo a la posibilidad de caer en alguno de estos problemas. A continuación se verán algunos de los casos más comunes y los recursos que hay para solucionarlos.

17.2.1 CONDICIÓN DE CARRERA

Cuando dos hebras acceden a una misma variable, están "compitiendo" por el acceso al recurso (la variable X en la Tabla 17-1), como la ejecución concurrente no garantiza un orden determinista en el tiempo para la ejecución de las instrucciones de las dos hebras se tiene una **condición de carrera** (*race condition*).

Hebra A	Hebra B
1. Leer valor de X	1. Leer valor de X

2. Adicionar 1 a X	2. Restar 1 a X
3. Guardarr valor de X	3. Guardar valor de X

Tabla 17-1 Ejemplo de condición de carrera

Entonces es posible que las líneas de código de B 1-3, se ejecuten todas después de A2 y antes de A3, dando lugar a resultados inesperados. Podemos formalizar entonces que:

Múltiples hebras se encuentran en una condición de carrera si el resultado de su ejecución depende del orden en que se ejecutan las instrucciones que componen cada hebra.

El código del
Listado 17.1 muestra un ejemplo en C# de la condición de carrera. Las hebras A y B descritas en la tabla, se reducen en C# a las líneas "`x=x+1`" y "`x=x-1`". Esta es nuestra condición de carrera.

```
static void Main(string[] args)
{
  Thread hebraA = new Thread(AdicionaX);
  Thread hebraB = new Thread(RestaX);
  hebraA.Start();
  hebraB.Start();
  hebraA.Join();
  hebraB.Join();
  Console.WriteLine("Valor final: " + x);
}

static int x = 0;
static void AdicionaX()
{
  for (int i = 0; i < 100000; i++)
    x = x + 1;
}
static void RestaX()
{
  for (int i = 0; i < 100000; i++)
    x = x - 1;
}
```

Listado 17.1 Ejemplo de condición de carrera

Al aplicar el método `Start()` a un objeto de tipo `Thread` se inicia la ejecución del código que se le asoció en la creación de la hebra. Es decir, según el Listado 17.1 al hacer `hebraA.Start()` y `hebraB.Start()` se está iniciando la ejecución de los métodos `AdicionaX()` y `RestaX()`, respectivamente.

Al hacer `hebraA.Join()`, se le está indicando a la hebra desde la que se hace esta invocación (el método `Main` en este ejemplo) esperar por la terminación de `hebraA` (es decir por la terminación del método `AdicionaX`. Así la ejecución de esta aplicación de consola no alcanza la línea final, en la que escribe el valor de "x", hasta que `hebraA` y `hebraB` hayan concluido su ejecución.

```
Valor final: -33733
Press any key to continue
```

Figura 17.2 Ejemplo de salida afectado por condición de carrera

Como se puede notar de la Figura 17.2, el resultado obtenido no es el correcto. El valor final debería ser cero, ya que sumamos y restamos igual número de veces. Sin embargo, debido a la existencia de una condición de carrera la variable `x` no ha sido modificada de forma correcta por ambas hebras.

17.2.2 SECCIONES CRÍTICAS Y BLOQUEOS

Las secciones de código de una hebra que hacen uso de una o mas variables compartidas y que pueden entrar en una condición de carrera se denominan **secciones críticas** por lo que éstas deben ser **mutuamente exclusivas**[4]; así que mientras una hebra esté ejecutando una sección crítica las otras no podrán hacerlo. Existen diversos recursos tanto en hardware como en software para lidiar con una situación de exclusión mutua, los cuales de modo general son denominados ***mutex***[5]. Los recursos más comunes son los **semáforos**, **monitores** y **candados** (*locks*).

C# tiene la palabra reservada `lock` para definir un candado con el cual especificar que el control sobre una sección crítica se hará de forma exclusiva. La sincronización en este caso se realiza "poniéndole el candado a una variable".

Hebra A	Hebra B
1. Poner candado a X 2. Leer valor de X 3. Adicionar 1 a X 4. Guardar valor de X 5. Quitar el candado a X	1. Poner candado a X 2. Leer valor de X 3. Restar 1 a X 4. Guardar valor de X 5. Quitar el candado a X

Tabla 17-2. Uso de candados de sincronización sobre una variable en una sección crítica

La Tabla 17-2 nos muestra cómo habría que poner y quitar un candado a la variable compartida para obtener derechos exclusivos sobre ésta. De este modo cualquier otra hebra que intente acceder, tendrá que esperar a que sea desbloqueado (se le quite el candado) el recurso. Esto resuelve la condición de carrera ya que las instrucciones de código A 2-4 y B 2-4 nunca se ejecutarían de forma concurrente. En código C# esto se expresa como se muestra en el Listado 17.2 utilizando la palabra reservada `lock` para marcar una zona de exclusión mutua.

```
static void Main(string[] args)
{
  Thread hebraA = new Thread(AdicionaX);
  Thread hebraB = new Thread(RestaX);
  hebraA.Start();
  hebraB.Start();
  hebraA.Join();
  HebraB.Join();
  Console.WriteLine("Valor final: " + x);
}
```

[4] Dijkstra, E. W. (1965). "Solution of a problem in concurrent programming control". Communications of the ACM 8 (9): 569. doi:10.1145/365559.365617.

[5] Contracción del inglés "Mutual Exclusion"

```
static int x = 0;
static object candado = new object();
static void AdicionaX()
{
  for (int i = 0; i < 100000; i++)
    lock (candado)
    {
      x = x + 1;
    }
}
static void RestaX()
{
  for (int i = 0; i < 100000; i++)
    lock (candado)
    {
      x = x - 1;
    }
}
```

Listado 17.2 Ejemplo de uso del candado "lock" en C#.

Si la totalidad del código a ejecutar por las hebras se encuentra sincronizado sobre la misma variable, entonces no habrá realmente paralelismo ni concurrencia. Se recomienda reservar el uso de bloqueos solo a áreas que deben ser mutualmente exclusivas.

```
Valor final: 0
Press any key to continue
```

Figura 17.3 Salida de nuestro primer programa concurrente utilizando candados

La salida del programa en este caso es la deseada (Figura 17.3) pues al finalizar la ejecución del programa (sin importar que hebra termina primero) el valor final de `x` debe ser cero.

Este sencillo ejemplo ha servido para introducir varios elementos para hacer programación concurrente en C#. En primer lugar la clase `Thread`, mediante la cual se definen y manipulan hebras (en esta caso con los ejemplos de sus métodos `Start` y `Join`). El acceso sincronizado de las hebras a una región crítica se ha solucionado utilizando el candado `lock`, evitando así una condición de carrera. Pero como se verá más adelante, las secciones críticas no siempre son tan simples, sino que lo más usual es que tengan un código mucho más complejo en ellas.

En situaciones tan sencillas se puede prescindir del uso de candados y usar algunos métodos estáticos de la clase `System.Threading.Interlocked`, con los que se pueden hacer de modo sincronizado operaciones tan simples como son en este caso sumas y restas (Listado 17.3).

```
static void AdicionaX()
{
  for (int i = 0; i < 100000; i++)
    Interlocked.Increment(ref x);
```

```
}
static void RestaX()
{
  for (int i = 0; i < 100000; i++)
    Interlocked.Decrement(ref x);
}
```

Listado 17.3 Uso de la clase Interlocked para realizar operaciones atómicas entre variables compartidas por hebras

17.3 LA CLASE MONITOR

El código del Listado 17.2 se introduce la palabra reservada `lock` para controlar el acceso exclusivo a una sección crítica. Pero esta controla toda una sección de inicio a fin, sin dejarnos libertad para adquirir o liberar un recurso basado en una lógica más propia del problema que se esté solucionando.

Para este propósito se dispone de la clase `System.Threading.Monitor`, que tiene los métodos `Enter` y `Exit` mediante los cuales una hebra puede adquirir y liberar un variable de una forma más fluida dentro del código. Un bloque `lock` puede lograrse con un bloque `try-finally` que haga uso de `Enter` y `Exit` como se muestra en el Listado 17.4

```
object x = new object();
public void UsandoLock()
{
  lock (x)
  {
    //código a ejecutar
  }
}
public void UsandoMonitor()
{
  try
  {
    Monitor.Enter(x);
    //código a ejecutar
  }
  finally
  {
    Monitor.Exit(x);
  }
}
```

Listado 17.4 Sincronización utilizando la clase Monitor o la primitiva Lock.

El método `Enter(object x)` esperará indefinidamente hasta que logre obtener acceso a la variable `x`, y el método `Enter(object x)` liberará a la variable `x`.

A diferencia de usar `lock(x){...}`, donde las llaves indican sintácticamente un único lugar donde el código se apropia de `x` y un único lugar donde el código libera a `x`; usando `Enter` y `Exit` apropiar y liberar se podrá hacer desde disímiles posiciones en el

código, incluso dependiendo de condicionales; por eso se dice que es una notación más fluida.

Otros métodos de la clase `Monitor` nos brindan más facilidades. Como no es posible analizarlos todos a continuación se describirán a tres de los más utilizados `Wait`, `Pulse` y su variante `PulseAll`.

17.3.1 PRODUCTOR-CONSUMIDOR

Para comprender mejor los métodos `Wait`, `Pulse` y `PulseAll` veremos a continuación un patrón de programación concurrente muy frecuente que se conoce como **Productor-Consumidor**.

Este patrón se presenta cuando se quiere comunicar una hebra que "produce" datos (productor) con una hebra que de "consume" dichos datos (consumidor) comunicándose los datos mediante una zona común (*buffer*), que usualmente funcionará como una cola (ver Capítulo 13). El trabajo del *productor* consiste en colocar datos en la cola y el del *consumidor* en extraer los datos de ésta. El problema es entonces garantizar que un productor no pueda colocar datos en la cola si ya está llena, y que el consumidor no intente extraer datos de la cola si está vacía.

Desde el punto de vista del productor, la solución sería poner a "dormir" la hebra de producción si la cola se ha llenado. Una vez que la hebra consumidor consuma un elemento, se "despertará" al productor para que pueda seguir "produciendo" (colocando datos en la cola). Por otro lado la hebra consumidor se pondrá a dormir si la cola está vacía y se le hará "despertar" cuando se coloquen nuevos datos en la cola.

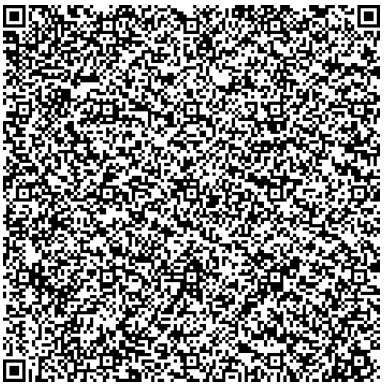
El método `Wait(object x)` es el que permite poner a dormir una hebra, haciéndola esperar por una notificación sobre el objeto `x`, mientras que el método `Pulse(object x)` nos permite despertar una hebra que esté esperando por una notificación en el mismo objeto `x`. Ambos métodos reciben como parámetro el objeto sobre el cual se realiza la sincronización (por el que se ponen a dormir y por el que se despiertan).

El **Error! Reference source not found.** muestra una implementación sencilla de la incronización productor-consumidor usando una cola y los métodos `Wait` y `Pulse` en dicho código. Para simplificar, se considerará que la cola no está acotada de tamaño pero sí que hay múltiples productores y consumidores.

```
class Dato
{
  public long Valor { get; set; }
  public static long Contador;
}

class Program
{
  static Queue<Dato> cola = new Queue<Dato>();
  static Random r = new Random();

  private static Dato Produce()
  {
    Thread.Sleep(r.Next(200, 500));
    var dato = new Dato();
```

```
    dato.Valor = Interlocked.Increment(ref Dato.Contador);
    return dato;
  }

  private static void Consume(Dato dato)
  {
    Thread.Sleep(r.Next(200, 400));
  }

  public static void Productor(object nombre)
  {
    while (true)
    {
      var dato = Produce();
      lock (cola)
      {
        cola.Enqueue(dato);
        Console.WriteLine("{0} produce : {1}",
           nombre, dato.Valor);
        Monitor.Pulse(cola);
      }
    }
  }
```

```
public static void Consumidor(object nombre)
  {
    while (true)
    {
      Dato dato = null;
      lock (cola)
      {
        if (cola.Count == 0)
        {
          Console.WriteLine("{0} esperando ...", nombre);
          Monitor.Wait(cola);
        }
        dato = cola.Dequeue();
      }
      Console.WriteLine("{0} consume : {1}", nombre,
       dato.Valor);
      Consume(dato);
    }
  }
```



```
static void Main(string[] args)
  {
    var productor1 = new Thread(new ParameterizedThreadStart(Productor));
    productor1.Start("Productor_1");
    var productor2 = new Thread(new ParameterizedThreadStart(Productor));
    productor2.Start("Productor_2");
    var productor3 = new Thread(new ParameterizedThreadStart(Productor));
    productor3.Start("Productor_3");

    var consumidor1 = new Thread(new ParameterizedThreadStart(Consumidor));
    consumidor1.Start("Consumidor_1");
    var consumidor2 = new Thread(new ParameterizedThreadStart(Consumidor));
```

```
    consumidor2.Start("Consumidor_2");
    var consumidor3 = new Thread(new ParameterizedThreadStart(Consumidor));
    consumidor3.Start("Consumidor_3");
  }
```

Listado 17.5 Implementación de productor-consumidor utilizando Wait/Pulse

Note que se ha dado un tiempo aleatorio a la demora en producir o consumir cada producto. Parte de una posible salida se visualiza en la Figura 17.4

¿Cómo funciona? Tanto productor como consumidor le ponen un candado a la `cola` (objeto estático global que todos comparten) antes de poner un sacar un dato de la misma, de modo de evitar que pueda ocurrir que se esté poniendo y quitando a la vez. El método `Monitor.Wait` lo usa el consumidor en el caso de la cola vacía para hacer esperar (detiene la ejecución de la hebra) al consumidor, pero al hacerlo libera el candado que se ha puesto sobre la variable `cola`. Esto permite que otra hebra pueda adquirir la variable `cola` y eventualmente poner un dato en la ella y realizar una llamada al método `Pulse`. Cuando el método `Productor` produce un dato invoca al método `Pulse` para notificar a alguna hebra del *pool* de hebras que están esperando por notificación (a través en este caso del objeto `cola` que comparten) de que puede continuar su ejecución. La decisión de cuál hebra se selecciona cae entre las responsabilidades del sistema operativo; el programador no debe asumir ningún conocimiento al respecto.

```
Consumidor_1 esperando ...
Consumidor_3 esperando ...
Consumidor_2 esperando ...
Productor_1 produce : 1
Consumidor_1 consume : 1
Productor_2 produce : 2
Consumidor_3 consume : 2
Productor_3 produce : 3
Consumidor_2 consume : 3
Consumidor_1 esperando ...
Consumidor_3 esperando ...
Productor_1 produce : 4
Consumidor_1 consume : 4
Productor_2 produce : 5
Consumidor_3 consume : 5
Consumidor_2 esperando ...
Productor_3 produce : 6
Consumidor_2 consume : 6
Consumidor_1 esperando ...
Productor_1 produce : 7
```

Figura 17.4 Salida parcial de ejecución de un par de hebras productor-consumidor.

Si se utilizase el método `PulseAll` en lugar de `Pulse`, el sistema operativo despierta todas las hebras en espera del recurso de forma simultánea. Hacer esto crearía una condición de carrera por lo que habría que modificar el código, para que las hebras notificadas que

no logran obtener un dato a consumir vuelvan al estado de espera. Así que si cambiamos `Pulse` por `PuseAll`, entonces en lugar de utilizar la línea `if (cola.Count==0)`, hay que utilizar un ciclo `while`, como se muestra en el Listado 17.6. Adicionalmente se ha considerado en este código que la cola tiene una capacidad limitada, es decir que puede llenarse, de modo que los productores esperan por notificación de que la cola no está llena para poder producir y notifican cuando producen para indicar que no está vacía; mientras que los consumidores esperan porque la cola no esté vacía para poder consumir y notifican que han consumido para indicar que no está llena.

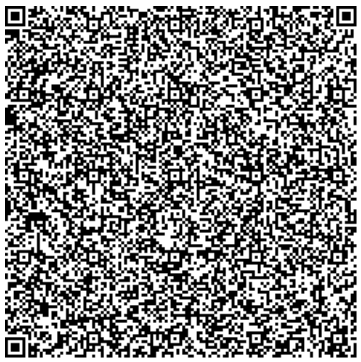
```
public static MAX = ...
void Productor(object nombre)
{
  while (true)
  {
    var dato = Produce();
    lock (cola)
    {
      cola.Enqueue(dato);
      Console.WriteLine("{0} produce : {1}",
                        nombre, dato.Valor);
      Monitor.PulseAll(cola);
      while (cola.Count == MAX)
      {
        Console.WriteLine("Esperando. Cola LLENA.");
        Monitor.Wait(cola);
      }
      cola.Enqueue(dato);
      Console.WriteLine("{0} Notificando a las hebras.",
                         nombre);
      Monitor.PulseAll(cola);
    }
   }
  }

  public static void Consumidor(object nombre)
  {
    while (true)
    {
      Dato dato = null;
      lock (cola)
      {
        while (cola.Count == 0)
        {
          Console.WriteLine("{0} esperando ...",
                            nombre);
          Monitor.Wait(cola);
        }
        dato = cola.Dequeue();
      }
      Console.WriteLine("{0} consume : {1}",
                        nombre, dato.Valor);
      Consume(dato);
    }
  }
```

Listado 17.6 Solución al productor-consumidor utilizando PulseAll

En este punto, la solución al problema del productor-consumidor está completa. Sin importar el número de hebras de ambos tipos que utilicemos, no habrá problemas de sincronización y hemos acotado la cantidad máxima de elementos de la cola.

Sin embargo, esto aún puede mejorarse al utilizar `PulseAll` se está notificando a todas las hebras, es decir tanto a productores como a consumidores. Es fácil notar que esto no es necesariamente eficiente. ¿Para qué notificar a los productores si solo se quiere avisar a los consumidores? Podemos escribir una solución que resuelva el problema de forma más eficiente, notificando solo al grupo de hebras que corresponda. Para hacerlo debemos recurrir a un uso más directo de la clase `Monitor` y no utilizar el candado `lock`. En el Error! Reference source not found. se ha resuelto el problema utilizando monitores adicionales sobre dos nuevas variables `lleno` y `vacio`, y también sobra la variable `cola`, ya que a su vez el acceso a ella debe también ser sincronizado (no vaya alguien a estar sacando mientras otro está poniendo).

```
static object noLlena = new object();
static object noVacia = new object();

public static void Productor(object nombre)
{
  while (true)
  {
    var dato = Produce();
    Monitor.Enter(cola);
    while (cola.Count == MAX)
    {
      Console.WriteLine("{0} Cola LLENA esperar ...",
                        nombre);
      Monitor.Exit(cola);
      lock (noLlena) Monitor.Wait(noLlena);
      Monitor.Enter(cola);
    }
    cola.Enqueue(dato);
    Monitor.Exit(cola);
    Console.WriteLine("{0} notificando a consumidores.",
                      nombre);
    lock (noVacia) Monitor.PulseAll(noVacia);
  }
}

public static void Consumidor(object nombre)
{
  while (true)
  {
    Dato dato = null;
    Monitor.Enter(cola);
    while (cola.Count == 0)
    {
      Console.WriteLine(("{0} Cola VACÍA esperar ...",
                         nombre);
      Monitor.Exit(cola);
      lock (noVacia) Monitor.Wait(noVacia);
```



```
      Monitor.Enter(cola);
      Console.WriteLine("{0} notificando a productores.",
                        nombre);
    }
    dato = cola.Dequeue();
    Monitor.Exit(cola);
    lock (noLlena) Monitor.PulseAll(noLlena);
    Console.WriteLine("{0} consume : {1}", nombre,
dato.Valor);
    Consume(dato);
  }
}
```

Listado 17.7 Solución más eficiente utilizando la clase Monitor

De este modo se notifica solo al grupo de hebras que lo necesitan, si se colocan elementos en la cola, los consumidores son notificados, y si se consume un recurso, entonces se avisa a los productores de que hay espacio en la cola para más datos. Esta solución además alcanza mayor nivel de paralelismo, pues reduce el tiempo en que toda la cola se encuentra en posesión de una hebra.

17.4 LA CLASE MUTEX

La clase `Monitor` y la palabra reservada `lock`, nos permitieron realizar sincronización entre varias hebras. Estas hebras están ejecutando dentro de un mismo programa, compartiendo su espacio de memoria. En este contexto una variable puede ser compartida entre varias hebras de una misma aplicación. ¿Pero cómo podríamos realizar algo semejante entre varias aplicaciones? A nivel del lenguaje de programación no se puede declarar una variable que sea visible en varios programas. Para lograr esto se dispone de la clase `System.Threading.Mutex`.

Esta clase nos permite trabajar con secciones críticas que traspasen la frontera de un programa. Será posible realizar sincronización dentro de las hebras de un mismo programa (que llamaremos proceso) o comunicarse con hebras de otro proceso en la misma computadora.

Un mutex nos permitirá utilizar sus métodos `WaitOne()` y `ReleaseMutex()` en manera similar a los métodos `Enter()` y `Exit()` de la clase `Monitor`. Solo que no es necesario pasar una variable sobre la cual sincronizar, el propio mutex hace de "zona de memoria" para este propósito.

Un mutex será visible a nivel del sistema operativo ejecutando en una computadora; así todos los procesos que están ejecutando sobre el sistema puede acceder a un mutex. De esta manera se puede realizar comunicación y sincronización entre hebras que se están ejecutando en diferentes espacios de memorias (cada proceso con la suya).

Para crear un mutex global (o un mutex de sistema) el mutex debe crearse con un nombre (de crearse sin nombre, el mutex será solo visible dentro del mismo programa, pero para esto ya contamos con otras herramientas). En el Listado 17.8 se ha creado un mutex pasando dos parámetros al constructor. El primer parámetro de tipo `bool` servirá para especificar si la hebra actual será la que comenzará siendo dueña del mutex; el

segundo parámetro es el nombre que se le da al mutex para que de esta forma sea accesible a otros procesos ejecutando en el sistema operativo.

```
class Program
{
  private static Mutex mutex = new Mutex(false,
                                         "MutexJugadores");

  static void Main(string[] args)
  {
    Thread hebra = new Thread(new ThreadStart(Jugando));
    hebra.Name = args[0];
    hebra.Start();
    Console.Clear();
  }

  private static void Jugando()
  {
    while (true)
      Juega();
  }

  private static void Juega()
  {
    Console.WriteLine("{0} esperando para jugar ...",
                      Thread.CurrentThread.Name);
    mutex.WaitOne();
    Console.WriteLine("{0} jugando ...",
                      Thread.CurrentThread.Name);
    Thread.Sleep(2000);  // Simular haciendo jugada
    Console.WriteLine("{0} hizo jugada (pulse Enter)!",
                      Thread.CurrentThread.Name);
    Console.ReadLine();
    mutex.ReleaseMutex();
  }
}
```

Listado 17.8. Exclusión mutua entre procesos utilizando utex

En el Listado 17.8 se ha utilizado un mutex de sistema para simular la exclusión mutua en el uso de un recurso compartido por más de un proceso. La simulación consiste en dos jugadores jugando entre ellos. El recurso en este caso, será el derecho a jugar y utilizar el procesador para pensar, que solo puede ser utilizado por un jugador a la vez.

En este caso lo que vamos a hacer es ejecutar simultáneamente dos veces el mismo programa para obtener dos procesos. La Figura 17.5 nos da la salida si se ejecutó el programa pasando cada vez el nombre "Kasparov" y "DeepBlue" respectivamente.

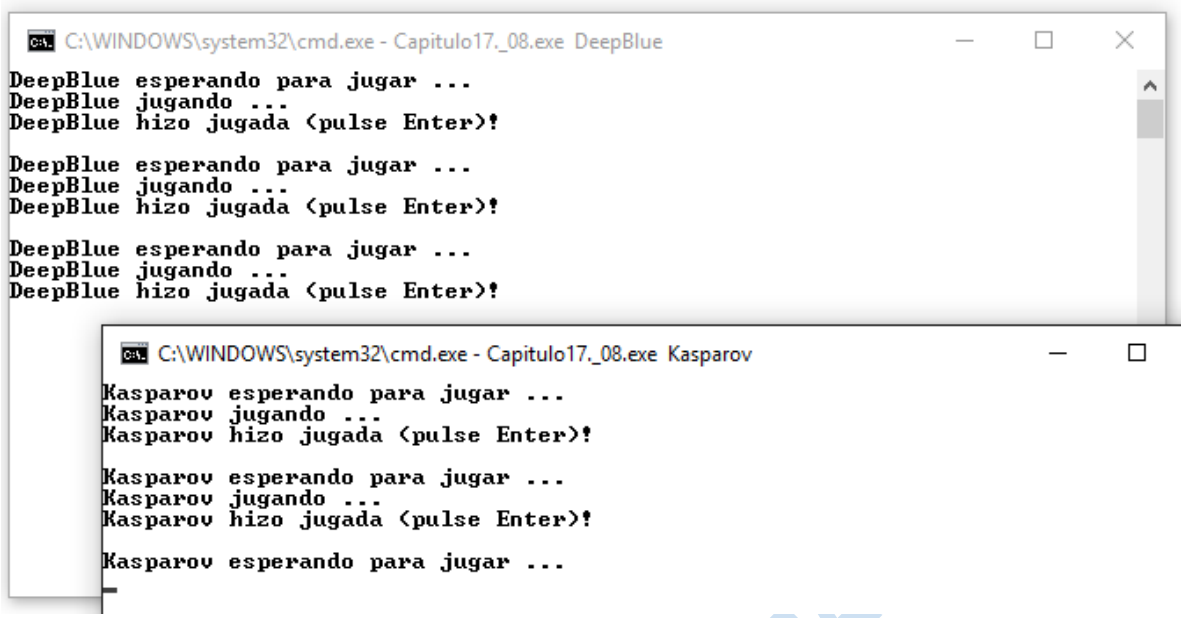


Figura 17.5. Ejemplo comunicación entre procesos utilizando la clase Mutex

17.5 LA CLASE SEMAPHORE

Los semáforos se representan con la clase `System.Threading.Semaphore`. Éstos permiten declarar zonas de exclusivdad en las que ms de una hebra puede entrar a la vez. Es posible considerar un Mutex, como un semáforo que solo permite entrar una única hebra en la zona de exclusividad.

El constructor `Semaphore(int inicial, int máximo)` permite especificar el valor inicial disponible en el semáforo y la cantidad máxima de hebras que podrán acceder a él. Un tercer parámetro es el nombre del semáforo; al igual que la clase `Mutex`, darle un nombre al semáforo lo hará visible a nivel de sistema.

Para ejemplificar su uso simulemos varios lectores (hebras) que intentarán compartir un numero limitado de copias de un libro (semáforo). La implementación se muestra en el Listado 17.9 donde hemos utilizado un semáforo para simular que solo se tienen 3 copias un libro. Por ahora por simplicidad hemos de ejecutar una sola vez el programa aunque se ha declarado el semáforo como global

```
private static Semaphore semaforo =
                new Semaphore(3, 3, "libro");

static void Main(string[] args)
{
  Console.Clear();
  for (int i = 1; i <= 5; i++)
  {
    Thread hebra =
       new Thread(new ThreadStart(Lector));
    hebra.Name = "Lector[" + i + "]";
    hebra.Start();
  }
}
```

```
private static void Lector()
{
  while (true)
    Leyendo();
}

private static void Leyendo()
{
  Console.WriteLine("{0} esperando por el libro ....",
    Thread.CurrentThread.Name);
  semaforo.WaitOne();

  Console.WriteLine("{0} está leyendo ...",
                    Thread.CurrentThread.Name);
  Thread.Sleep(2000);  // Simular tiempo de lectura

  Console.WriteLine("{0} terminó. Devuelve el libro.",
                    Thread.CurrentThread.Name);
  semaforo.Release();
}
```

Listado 17.9 Utilización de semáforos

En este ejemplo, hemos permitido que de las 5 hebras (lectores) de nuestro programa solo un máximo de 3 puedan entrar en la zona crítica a la vez. Una ejecuión del programa daría una salida similar a la Figura 17.6.

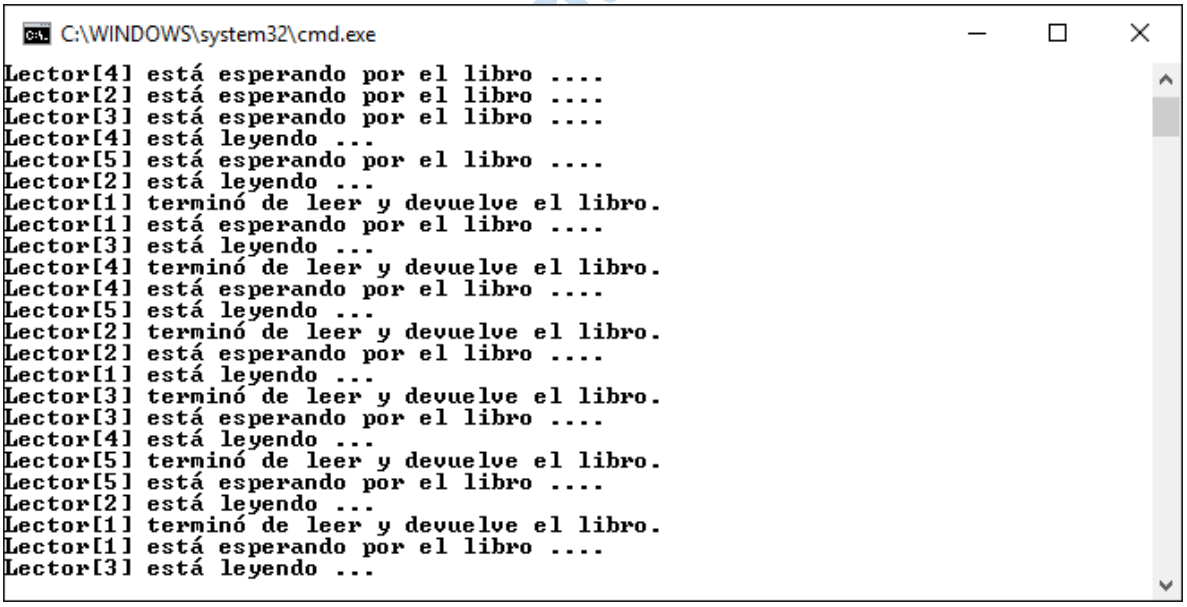


Figura 17.6 Salida de la simulación utilizando semáforos

En este ejemplo se utilizaron varias hebras dentro de un único programa pero si ejecutamos dicho programa varias veces de forma simultánea se puede confirmar que el resultado es similar, y es que como hemos nombrado al semáforo, este estará disponible para todos los procesos del sistema. Así si ejecutamos el código dos veces, tendríamos

10 lectores (hebras) intentando acceder al máximo de 3 copias del libro (el máximo con el que fue creado nuestro semáforo).

La utilización de semáforos, permite que más de una hebra puedan entrar a la vez en una zona crítica. Esto, aunque es claramente conveniente en muchos casos, también puede abrir la puerta a problemas si la modificación de algún recurso ocurre dentro de la zona critica. Es responsabilidad del programador realizar tal operación de forma segura, ya sea usando candados o monitores.

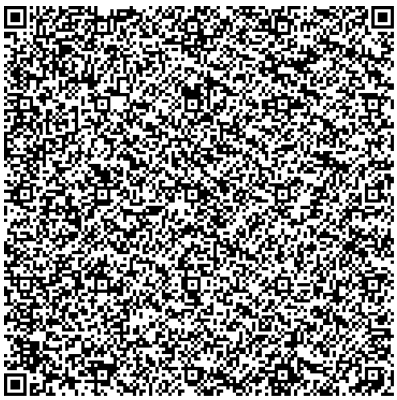
Todos estos recursos: candados, monitores, semáforos y mutex, ofrecen un buen arsenal para atacar variados problemas que impliquen concurrencia y sincronización entre hebras y entre procesos.

17.6 PRIORIDADES

Suele ser responsabilidad del sistema operativo realizar la ejecución de las hebras, así como seleccionar cuál hebra es la próxima que tendrá acceso a un recurso (del conjunto de hebras que esperan por él). Aunque el programador puede tomar el control de ésta actividad, lo recomendado es dejarlo así. Pero hay algo que se puede cambiar con relativa facilidad para modificar este comportamiento, y es la **prioridad** que se le puede asociar a una hebra.

La prioridad, es un valor discreto (expresado con el tipo `ThreadPriority`) que define "cuán importante es una hebra". Así la ejecución de una hebra de alta prioridad, será planificada con *cierta* preferencia sobre las de más baja prioridad. La prioridad de una hebra está definida en la propiedad `Priority` cuyo valor es del tipo enumerativo `ThreadPriority`. Los posibles valores son `Lowest, BelowNormal, Normal, AboveNormal, Highest` que corresponden con valores enteros de 0 a 4 respectivamente.

```
class Programa
{
  static void Main()
  {
    PruebaPrioridad prueba =
      new PruebaPrioridad();
    Thread hebra1 = new Thread(prueba.Cuenta);
    hebra1.Name = "Hebra Uno";
    hebra1.Priority = ThreadPriority.Highest;
    Thread hebra2 = new Thread(prueba.Cuenta);
    hebra2.Name = "Hebra Dos";
    hebra2.Priority = ThreadPriority.Normal;
    Thread hebra3 = new Thread(prueba.Cuenta);
    hebra3.Name = "Hebra Tres";
    hebra3.Priority = ThreadPriority.Lowest;

    hebra1.Start();
    hebra2.Start();
    hebra3.Start();
    // contar por 10 segundos.
    Thread.Sleep(10000);
    prueba.SeguirContando = false;
  }
```



```
}

class PruebaPrioridad
{
  static bool _SeguirContando;

  public PruebaPrioridad()
  {
    _SeguirContando = true;
  }
  public bool SeguirContando
  {
    set { _SeguirContando = value; }
  }
  public void Cuenta()
  {
    long Contador = 0;
    while (_SeguirContando)
    {
      Contador++;
    }
    Console.WriteLine("{0,-11} con prioridad
       {1,11} contó hasta {2,13}",
       Thread.CurrentThread.Name,
       Thread.CurrentThread.Priority.ToString(),
       Contador.ToString("N0"));
  }
}
```

Listado 17.10 Ejemplo de prioridades en hebras

El código del Listado 17.10 (modificado de un ejemplo de la documentación de C#) muestra un caso en que tres hebras con diferentes prioridades realizan un simple conteo numérico por aproximadamente 10 segundos. Como puede notar el lector en la salida de la Figura 17.7 la diferencia no es tan grande, aunque palpable, esto en parte se debe a que el sistema operativo debe garantizar que, aunque se cumplan las prioridades, ninguna hebra se quede permanentemente sin tener tiempo de procesador.

```
Hebra Tres  con prioridad      Lowest contó hasta 3,068,011,110
Hebra Uno   con prioridad     Highest contó hasta 3,099,242,225
Hebra Dos   con prioridad      Normal contó hasta 3,090,301,437
Press any key to continue . . .
```

Figura 17.7 Ejemplo de salida con prioridades. Muestra una diferencia sutil.

Las prioridades permiten indicar la preferencia en la planificación de la ejecución de las hebras; éstas, no son en ninguna forma, una manera directa de especificar orden de ejecución. El sistema operativo es el único responsable de esto. ¿Qué pasaría si se modificasen las prioridades en un productor-consumidor de forma incorrecta y uno de ellos jamás se ejecutase? En la próxima sección se analizarán éste y otros problemas de sincronización en el trabajo con hebras que pueden tener consecuencias en el funcionamiento global de una aplicación.

17.7 PROBLEMAS DE SINCRONIZACIÓN

Los mecanismos anteriores de exclusión mutua ayudan a mejorar la robustez de la comunicación entre hebras, pero lamentablemente por sí solos no solucionan todos los problemas de sincronización que se pueden presentar. Los problemas más conocidos son la **muerte por inanición** (*starvation*), el **interbloqueo** (*deadlock*), y el **bloqueo activo** (*livelock*).

17.7.1 MUERTE POR INANICIÓN Y BLOQUEO ACTIVO

La **muerte por inanición** significa que una hebra jamás logra acceder a un recurso que necesita para trabajar mientras que otras hebras (tal vez trabajando de modo "goloso") ocupan dicho recurso la mayor parte (o la totalidad) del tiempo. Esto puede ser posible por un mal diseño de la sincronización o por una asignación inapropiada de las prioridades.

Para ilustrarlo tomaremos el código del Listado 17.10 e insertaremos la línea de código "`Process.GetCurrentProcess().ProcessorAffinity = (System.IntPtr)1;`" al inicio del método main. Esto forzará a nuestro programa a solo ubicar hebras en un único procesador. La salida mostrada en la Figura 17.8 muestra una diferencia mucho mayor. La hebra con más alta prioridad tuvo mucho más acceso al recurso que las demás (unas 260 veces más). En este caso, es posible que las otras dos hebras al tener tan poco (o ningún) acceso al recurso se considere que sufren de muerte por inanición. Dependiendo del diseño del programa, un mal manejo de prioridades, es sin duda una de las formas mas comunes de caer en este problema, aunque no la única.

```
Hebra Uno   con prioridad     Highest contó hasta 11,379,246,338
Hebra Dos   con prioridad      Normal contó hasta    64,070,515
Hebra Tres  con prioridad      Lowest contó hasta    43,176,768
Press any key to continue . . .
```

Figura 17.8 Salida que muestra una posible muerte por inanición

El **bloqueo activo** (*livelock*) es aquel en el que dos hebras continuamente cambian de estado sin lograr avanzar. Considere por ejemplo dos hebras que dependen cada una de eventos de la otra para progresar. Si la continuación de una necesita de respuesta de la otra, entonces es posible que ocurra un bloqueo activo.

Imagine a dos pintores que disponen de una sola paleta y un solo pincel para pintar entre ambos una misma pintura uno toma la paleta y si cuando intenta tomar el pincel éste está ocupado entonces por cortesía libera la paleta para que el otro pintor pueda pintar; sin embargo el otro ha intentado hacer lo mismo pero a la inversa. Pudiera darse entonces la situación en que ambos están tomando y liberando paleta y pincel, sin que cada uno logre tener los dos para poder pintar, provocando un ciclo interminable. En el Listado 17.11 se muestra dos métodos "pintores" que hacen sincronización sobre un par de recursos `paleta` y `pincel`, pero lo hacen de forma inversa uno al otro. Un pintor A toma la `paleta` y luego intenta tomar el `pincel`, si no puede, espera un tiempo antes de volver a intentarlo, pero por cortesía libera la `paleta` mientras tanto. A su vez un pintor B intenta pintar también con la misma paleta y pincel pero toma primero el `pincel`, para

luego tomar la `paleta` de la misma forma que lo hace el pintor A. Es posible que pueda darse el caso en que ambos pintores con buenas intenciones estén liberando y ocupando inversamente el recurso que el otro necesita, intercambiándose el estado, sin finalmente avanzar. En este caso la ejecución del código jamás se detiene pero tampoco llega al punto deseado en que pueden pintar, a esta situación se le denomina bloqueo activo.

```
static void PintorA(object paleta, object pincel)
{
  Monitor.Enter(paleta);
  while (!Monitor.TryEnter(pincel))
  {
    Console.WriteLine("Libera paleta y espera pincel");
    Monitor.Exit(paleta);
    Thread.Sleep(10);
    Monitor.Enter(paleta);
  }
  Console.WriteLine("A pintando con paleta y pincel");
  Monitor.Exit(pincel);
  Monitor.Exit(paleta);
}

static void PintorB(object paleta, object pincel)
{
  Monitor.Enter(pincel);
  while (!Monitor.TryEnter(paleta))
  {
    Console.WriteLine("Libera pincel y espera paleta...");
    Monitor.Exit(pincel);
    Thread.Sleep(10);
    Monitor.Enter(pincel);
  }
  Console.WriteLine("B pintando con paleta y pincel");

  Monitor.Exit(paleta);
  Monitor.Exit(pincel);
}
```

Listado 17.11 Código que puede generar un bloqueo activo

Como con todos los problemas de concurrencia (desde la elemental condición de carrera), es posible que el código mencionado se ejecute muchas de veces sin que cause problemas, sin embargo, la situación discutida que causa el problema está presente y un simple análisis del código lo demuestra. Ármese de paciencia y ejecute la aplicación hasta que esto ocurra.

Aunque el bloqueo activo le puede resultar parecido a una muerte por inanición, éste no es el caso porque aquí ambas hebras se mantienen activas tomando y liberando pincel y paleta, pero lamentablemente sin lograr pintar.

Cuando se trabaja con concurrencia, es necesario eliminar el análisis empírico y casuístico de "funcionó cuando lo ejecuté" y proceder a un mejor y más profundo diseño de la solución.

La máquina de radiación médica Therac-25 causó al menos 6 accidentes entre 1985 y 1987 porque varios pacientes recibieron dosis masivas de radiación

producto de un error de concurrencia.

17.7.2 INTERBLOQUEO O BLOQUEO MUTUO

El bloqueo mutuo o interbloqueo, conocido popularmente por su término en inglés *deadlock*, consiste en una situación donde dos o más hebras se quedan a la espera de recursos en posesión de otras que a la vaz están en espera de recursos en posición de otras y así sucesivamente formando una circularidad. Para entrar en estado de bloqueo mutuo deben darse *todas* las siguientes condiciones que se conocen como **condiciones de Coffman**[6].

1. **Condición de exclusión mutua**: existe al menos un recurso que es compartido pero al cual solo puede acceder una hebra a la vez.
2. **Condición de retención y espera**: Hay al menos una hebra que tiene un recurso, pero que espera por otro recurso en posesión de otra hebra.
3. **Condición de no expropiación:** Los recursos solo pueden ser liberados de forma voluntaria (es decir ningún código externo a la hebra puede forzar hacerlo).
4. **Condición de espera circular:** Dada m hebras $P_1...P_m$, la hebra P_1 está esperando un recurso en posesión de P_2, y así sucesivamente de manera circular porque P_m está esperando por P_1.

El problema a continuación conocido como el de ***los 5 filósofos*** se ha convertido en un clásico [7] para ejemplificar problemas de sincronización.

17.7.2.1 Problema de los 5 filósofos

Cinco filósofos se pasan la vida comiendo y pensando, sentados en una mesa circular con un tazón común en el centro que contiene "infinitos" espaguetis y con un tenedor colocado entre cada par de filósofos. Cada filósofo alterna a su antojo el comer y el pensar, sin comunicarse con los demás. Para comer cada filósofo necesita de los dos tenedores (el que tiene a su izquierda y el que tiene a su derecha) por lo que puede ocurrir que tome uno de los dos tenedores y tenga que esperar por el otro que está ocupado por un filósofo vecino. Al terminar de comer, el filósofo debe colocar de vuelta ambos tenedores en la mesa[8]. Puede ocurrir una situación en que cada filósofo tenga ocupado el tenedor a su izquierda y al querer tomar el tenedor a su derecha éste esté ocupado por el filósofo a su derecha quien también está en la misma situación y así de modo circular y entonces todos están bloqueados queriendo comer pero sin poder comer por falta de un tenedor.

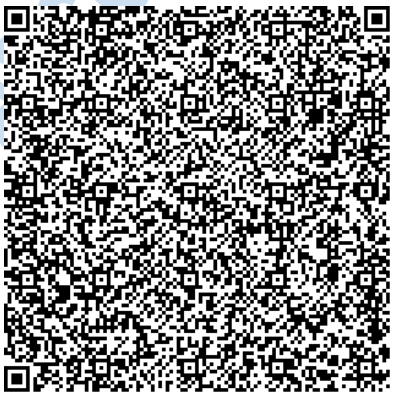
Problema: *Diseñar un método con el cual los filósofos pueden alternar entre comer y pensar eternamente, sin previo conocimiento de cuándo los otros filósofos desean comer*

[6]Coffman, Edward G., Jr.; Elphick, Michael J.; Shoshani, Arie (1971). "System Deadlocks". ACM Computing Surveys 3 (2): 67–78. DOI:10.1145/356586.356588.
[7] Dijkstra, Edsger (1965) W. EWD-1000. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. http://www.cs.utexas.edu/users/EWD/transcriptions/EWD10xx/EWD1000.html
[8] Realmente el ejemplo no es nada higiénico pero así es como se ha presentado históricamente en la literatura.

y claro sin tener que aumentar la cantidad de tenedores porque es obvio que si cada uno tiene dos tenedores no habría problema.

En el Listado 17.12 se presenta una implementación trivial basada en el enunciado del problema[9]. Esta implementación tiene el problema de cumplir con las cuatro condiciones de Coffman anteriormente planteadas; por lo que enfrentará problemas de bloqueo mutuo.

```
class Program
{
  static Random r = new Random();
  class Tenedor
  {
  }
  class Filosofo
  {
    Tenedor izq, der;
    string nombre;
    public Filosofo(Tenedor izq, Tenedor der,
                    string nombre)
    {
      this.izq = izq; this.der = der;
      this.nombre = nombre;
    }
    public void Vive()
    {
      while (true)
      {
        Console.WriteLine("{0, -30}{1}",
                          "Pensando ...", nombre);
        Thread.Sleep(r.Next(40));
        Console.WriteLine("{0,-30}{1}",
                          "Queriendo comer", nombre);
        lock (izq)
        {
          //Demora en tomar un tenedor
          Thread.Sleep(10);
          lock (der)
          {
            Console.WriteLine("{0,-30}{1}",
                          "Comiendo ...", nombre);
            Thread.Sleep(r.Next(20));
          }
        }
      }
    }
  }
```



[9] Disculpe el lector la incongruencia histórica porque no todos estos personajes fueron contenporáneos

```
static void Main(string[] args)
{
  Tenedor[] tenedores = new Tenedor[5] {
        new Tenedor(), new Tenedor(),
        new Tenedor(), new Tenedor(),
        new Tenedor() };
  Filosofo socrates = new Filosofo(tenedores[0],
                        tenedores[1], "Socrates");
  Filosofo platon = new Filosofo(tenedores[1],
                        tenedores[2], "Platon");
  Filosofo seneca = new Filosofo(tenedores[2],
                        tenedores[3], "Séneca");
  Filosofo diogenes = new Filosofo(tenedores[3],
                        tenedores[4], "Diógenes");
  Filosofo aristoteles = new Filosofo(tenedores[4],
                        tenedores[0],"Aristóteles");
  new Thread(socrates.Vive).Start();
  new Thread(platon.Vive).Start();
  new Thread(seneca.Vive).Start();
  new Thread(diogenes.Vive).Start();
  new Thread(aristoteles.Vive).Start();
  }
}
```

Listado 17.12. Implementación de la cena de los filósofos que causa bloqueo mutuo

Ejecute el código del Listado 17.12, tal vez con un poco de paciencia, hasta que la ejecución queda bloqueada porque los filósofos quedan en un estado de bloqueo mutuo (cada uno habiendo listado el mensaje de `Queriendo comer` pero sin poder haber empezado a comer por falta de tenedores.) Note que aunque está garantizado que sucederá un interbloqueo, no es determinista en qué momento preciso esto ocurrirá dado que el tiempo en que tardan los filósofos en comer y pensar es aleatorio.

La muerte por inanición y el bloqueo activo, son problemas menos comunes que el interbloqueo,; pero son más difícies de detectar ya que pueden no apreciarse de manera evidente en el comportamiento de la aplicación. Sin embargo, un interbloqueo es apreciable porque se "congela" la ejecución de la aplicación (interface que no responde, salida de datos que no ocurre).

Veamos a continuación cómo podría resolverse este problema de interbloqueo.

17.7.2.2 SOLUCION CON UN ARBITRO/MEDIADOR

Donde quiera que haya varias partes que no se ponen de acuerdo puede ayudar un mediador que los aconseje y organice. En este caso considere que hay un mesero al cual los filósofos le deben pedir permiso para comer, el problema consistiría entonces en ponerle un candado al mesero para que solo pueda atender a un filósofo a la vez. El

```
static Random r = new Random();
static object mesero = new object();
class Tenedor
{
}
class Filosofo
```

```
{
  Tenedor izq, der;
  string nombre;
  public Filosofo(Tenedor izq, Tenedor der,
                  string nombre)
  {
    this.izq = izq; this.der = der;
    this.nombre = nombre;
  }
  public void Vive(){
    while (true)
    {
      Console.WriteLine("{0, -30}{1}",
                        "Pensando ...", nombre);
      Thread.Sleep(r.Next(40));
      Console.WriteLine("{0,-30}{1}",
                        "Queriendo comer", nombre);
      Monitor.Enter(mesero);
      lock (izq)
      {
        Thread.Sleep(10);
        //Demora en tomar un tenedor
        lock (der)
        {
          Monitor.Exit(mesero);
          Console.WriteLine("{0,-30}{1}",
                        "Comiendo ...", nombre);
          Thread.Sleep(r.Next(20));
        }
      }
    }
  }
}
```

Listado 17.13 muestra semejante implementación.

```
static Random r = new Random();
static object mesero = new object();
class Tenedor
{
}
class Filosofo
{
  Tenedor izq, der;
  string nombre;
  public Filosofo(Tenedor izq, Tenedor der,
                  string nombre)
  {
    this.izq = izq; this.der = der;
    this.nombre = nombre;
  }
  public void Vive(){
    while (true)
    {
      Console.WriteLine("{0, -30}{1}",
                        "Pensando ...", nombre);
      Thread.Sleep(r.Next(40));
      Console.WriteLine("{0,-30}{1}",
                        "Queriendo comer", nombre);
```

```
            Monitor.Enter(mesero);
            lock (izq)
            {
              Thread.Sleep(10);
              //Demora en tomar un tenedor
              lock (der)
              {
                Monitor.Exit(mesero);
                Console.WriteLine("{0,-30}{1}",
                                  "Comiendo ...", nombre);
                Thread.Sleep(r.Next(20));
              }
            }
          }
        }
      }
```

Listado 17.13. Implementación utilizando un mediador (mesero)

En la solución se ha utilizado la clase `Monitor` y sus métodos estáticos `Enter` y `Exit`. Note que en este caso se libera al mesero una vez el filósofo tiene ambos tenedores para que eventualmente pueda darle (si no están ocupados) los dos tenedores a otro filósofo y alcanzar mayor paralelismo, de lo contrario solamente un filósofo podría comer a la vez.

Si el lector ejecuta el código, notará que la ejecución nunca se bloquea, siempre habrá al menos alguien comiendo o pensando. ¿Significa esto que la solución es completamente correcta? Pues la respuesta es "depende". En este caso depende de cómo el sistema operativo donde se ejecuta el programa se encargue de ejecutar las tareas que esperan por un recurso. Si cuando una hebra (un filósofo) espera por un recurso (el mesero) sobre el que otra hebra le hizo `Enter` entonces la hebra se pone en una cola que se comporta con el funcionamiento de "el primero que entra es el primero que sale"[10] (Capítulo 13) entonces no habrá problema, en algún momento a esa hebra le llegará su turno de poder apropiarse del recurso. Pero es *posible* que el sistema operativo no se comporte así (¿acaso no ha estado ustedes jamás en un restaurante donde aunque cree que le corresponde el mesero no le atiende y atiende a otra mesa?). En este caso, aunque no hay interbloqueo (porque siempre podrá haber algún filósofo comiendo o pensando), es posible que haya alguno que esté siempre esperando para comer(**muerte por inanición**).

Para eliminar completamente la posibilidad de inanición basta con que se garantice que la forma en que se procesan los pedidos sea FIFO. Se deja como ejercicio al lector crear una implementación correcta de la solución con mediador que utilice una cola (`System.Collections.Generic.Queue<T>`) y que no tenga que depender de cómo funciona el sistema operativo.

17.7.2.3 SOLUCION MEDIANTE JERARQUÍA DE RECURSOS

La solución original propuesta por Dijkstra, consiste en asignar un orden parcial a los recursos, en este caso enumerando los tenedores de 1 a 5. Entonces cada filósofo al

[10] FIFO del inglés First In First Out

momento de comer, toma primero el tenedor de menor orden luego el de mayor, de modo que si no puede tomar el de menor orden no toma ninguno.

```
class Program
{
  static Random r = new Random();
  class Tenedor
  {
    public int Orden { get; private set; }
    public Tenedor(int orden) { Orden = orden; }
  }
  class Filosofo
  {
    Tenedor izq, der;
    string nombre;
    public Filosofo(Tenedor izq, Tenedor der,
                    string nombre)
    {
      this.izq = izq; this.der = der;
      this.nombre = nombre;
    }
    public void Vive()
    {
      while (true)
      {
        Console.WriteLine("{0, -30}{1}",
                          "Pensando ...", nombre);
        Thread.Sleep(r.Next(40));
        Console.WriteLine("{0,-30}{1}",
                          "Queriendo comer", nombre);
        var primero = izq;
        var segundo = der;
        if (izq.Orden > der.Orden)
        {
          primero = der;
          segundo = izq;
        }
        lock (primero)
        {
          Thread.Sleep(10);
          lock (segundo)
          {
            Console.WriteLine("{0,-30}{1}",
                              "Comiendo ...", nombre);
            Thread.Sleep(r.Next(20));
          }
        }
      }
    }
  }
static void Main(string[] args)
  {
    Tenedor[] tenedores = new Tenedor[5] {
      new Tenedor(0), new Tenedor(1),
      new Tenedor(2), new Tenedor(3),
      new Tenedor(4) };
    Filosofo socrates = new Filosofo(tenedores[0],
                         tenedores[1], "Socrates");
```

```
    Filosofo platon = new Filosofo(tenedores[1],
                          tenedores[2], "Platon");
    Filosofo seneca = new Filosofo(tenedores[2],
                          tenedores[3], "Séneca");
    Filosofo diogenes = new Filosofo(tenedores[3],
                         tenedores[4], "Diógenes");
    Filosofo aristoteles =
           new Filosofo(tenedores[4],
                        tenedores[0], "Aristóteles");
    new Thread(socrates.Vive).Start();
    new Thread(platon.Vive).Start();
    new Thread(seneca.Vive).Start();
    new Thread(diogenes.Vive).Start();
    new Thread(aristoteles.Vive).Start();
  }
}
```

Listado 17.14. Implementación con jerarquía de recursos

En la implementación de la solución de Dijkstra en el Listado 17.14 se ha modificado la clase `Tenedor` para contener la propiedad `Orden` y los filósofos ordenan ahora los tenedores a la hora de comer. Esta solución es libre de bloqueo mutuo y libre de muerte por inanición.

Libre de muerte por inanición, es un criterio de mayor peso que *libre de bloqueo mutuo* ya que una solución libre de bloqueo mutuo puede incurrir en inanición, pero no viceversa.

Lo que ocurre es que un bloqueo mutuo en general es percibido por el usuario al ver el "congelamiento" de la aplicación; sin embargo, una muerte por inanición puede no ser percibida en tanto se estén viendo resultados y el usuario no se dé cuenta que hay alguien que no está trabajando.

17.7.2.4 SOLUCION USANDO MONITOR.TRYENTER

El método `Monitor.TryEnter`, como su nombre sugiere, intenta apropiarse de un recurso (poner un candado) y nos informa del resultado, siendo posible especificarle un tope de tiempo para completar la acción, para de este modo no esperar indefinidamente por el recurso. De manera que no es difícil entonces escribir un programa concurrente que no incurra en interbloqueo[11] haciendo uso de este método. Tomemos por ejemplo ahora la implementación inicial al problema de los filósofos y realicemos la sincronización utilizando `TryEnter/Exit`. La implementación se muestra en el Listado 17.15. Note que si un filósofo, habiéndose apropiado del tenedor izquierdo luego no se puede apropiar del tenedor derecho, entonces libera al tenedor izquierdo, por lo que no hace retención y espera. De modo que no se cumple una de las condiciones de Coffman; lo que es suficiente para garantizar que no ocurrirá interbloqueo.

[11] Claro que, aún con las herramientas adecuadas, siempre es posible que algún programador se empeñe y se las arregle para escribir un programa incorrecto.

```
public void Vive()
{
  while (true)
  {
    Console.WriteLine("{0, -30}{1}",
                      "Pensando ...", nombre);
    Thread.Sleep(r.Next(40));
    Console.WriteLine("{0,-30}{1}",
                      "Queriendo comer", nombre);
    if (Monitor.TryEnter(izq, 100))
    {
      Thread.Sleep(10); //Demora en tomar un tenedor
      if (Monitor.TryEnter(der, 100))
      {
        Console.WriteLine("{0,-30}{1}",
                      "Comiendo ...", nombre);
        Thread.Sleep(r.Next(20));
        Monitor.Exit(der);
      }
      Monitor.Exit(izq);
    }
  }
}
```

Listado 17.15 Solución al problema de los filósofos sin interbloqueo utilizando Monitor

El lector puede pensar que para qué tanta complicación si bastaría con tener suficiente cantidad de tenedores para no tener que compartirlos. Lo que sucede en los problemas reales es que hay recursos a compartir que pueden ser costosos como para reproducirlos en cualquier cantidad, o que por la naturaleza propia del problema son una cantidad fija, o limitada.

17.8 PROGRAMACIÓN PARALELA

Los recursos vistos hasta ahora en este capítulo permiten crear y manipular hebras, compartir recursos y realizar sincronización sobre estos, así como también poner hebras en espera, despertarlas y darles prioridad. Para terminar en esta sección presentaremos abstracciones de programación más avanzadas para expresar el paralelismo y la sincronización consecuente, que en su implementación de los mismos encapsulan a muchos de los recursos vistos anteriormente.

Cuando se desea procesar datos en paralelo, estén estos en una colección física como un array, en una base de datos u obtenidos de un `IQueryable` (en caso de LINQ), el .NET Framework presenta una forma elegante de procesarlos en paralelo, abstrayendo al usuario de la tarea de tener crear y manipular directamente hebras. Este mecanismo de paralelismo, es denominado **paralelismo de datos**. Es decir, se quiere hacer un mismo procesamiento a una gran cantidad de datos y se pretende "paralelizar" esta tarea haciendo "a la vez" el mismo procesamiento a diferentes agrupaciones de los datos (varios cocineros haciendo una gran cantidad de patatas fritas en un restaurante de comida rápida). Por otra parte, cuando se quieren realizar varias tareas diferentes en

paralelo para combinar los resultados de forma diferida en un único resultado (varios cocineros preparando los diferentes ingredientes de un mismo plato en un restaurante de diseño) se habla de **paralelismo de tareas**.

En la Figura 17.9 (tomada de MSDN) se muestra la arquitectura existente para ambas técnicas dentro del .NET Framework. Como puede observar el lector, el marco de ejecución; es capaz de traducir a hebras, particionar los datos y realizar sincronización de forma transparente al usuario. Aunque eventualmente el programador podrá realizar un control más estricto de estas operaciones, la API que se ofrece es lo suficiente simple y poderosa como para usarse directamente en la mayoría de los casos.

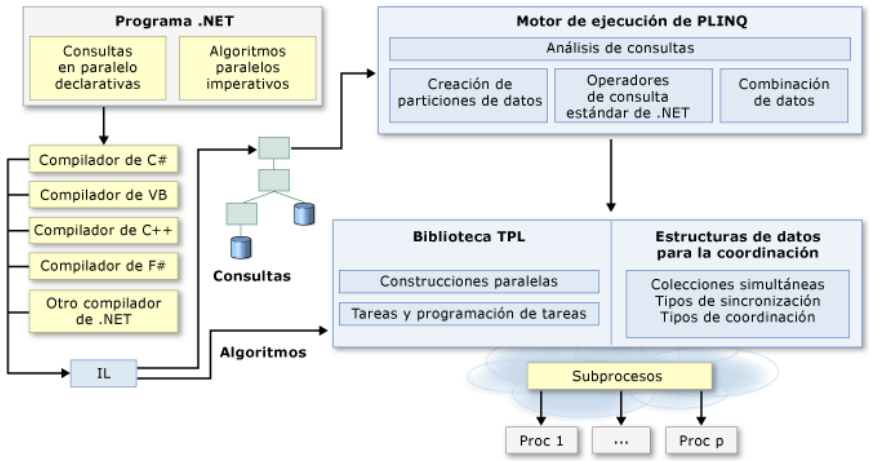


Figura 17.9 Arquitectura del .NET Framework para la programación paralela

17.8.1 PARALELISMO DE DATOS

El paralelismo de datos se enfoca en dividir **los datos** en grupos apropiados para su procesamiento en paralelo por diferentes procesadores de forma simultánea (SIMD[12]). Básicamente consiste en asignar una misma secuencia de código para ejecutar en cada uno de los procesadores[13], cada uno procesando datos diferentes. Esto implica que hay que DIVIDIR los datos antes de enviarlos a cada procesador y AGRUPAR los resultados provenientes de cada uno de estos.

En el .NET Framework el paralelismo de datos se puede lograr de diferentes formas. La biblioteca **TPL** (del inglés Task Parallel Library) ofrece una clase `Parallel` que tiene versiones paralelas para ciclos del tipo `For` y `Foreach` y las extensiones al lenguaje de consulta `LINQ` para paralelismo, denominadas **PLINQ** (Parallel LINQ) que permiten paralelizar el procesamiento de una fuente de datos.

17.8.1.1 PARALLEL.FOR

El método estático `Parallel.For` permite realizar iteraciones al estilo de un ciclo `For` tradicional pero en paralelo. Con este ciclo se puede iterar de forma paralela

[12] Single Instruction Multiple Data es una clasificación de arquitectura de computadoras dentro de la taxonomía de Flynn.
[13] Si los procesadores son reales por el hardware disponible, o virtuales, es algo de lo que se encarga el Framework

especificando la cota inferior y superior de un intervalo de números enteros y una acción (expresada por un delegado) a ejecutar por cada iteración.

El código del Listado 17.16 calcula cuántos primos hay entre 1 y 10 millones, preguntando a cada número si es primo. Se ha hecho de forma secuencial, y luego en paralelo para que el lector pueda observar la diferencia en rendimiento.

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Encontrando primos");
    int maximo = 10000000; int k = 0;
    Stopwatch reloj = new Stopwatch();
    //sin paralelismo
    reloj.Start();
    for(long i = 1; i <= maximo; i++)
      if (EsPrimo(i)) k++;
    reloj.Stop();
    Console.WriteLine(
     "{0} encontrados SIN PARALELISMO en {1} ms",
      k, reloj.Elapsed);
    //con paralelismo
    k = 0;
    reloj.Restart();
    Parallel.For(1, maximo + 1, (i) =>
      {
        if (i.EsPrimo())
          Interlocked.Increment(ref k);
      });
    reloj.Stop();
    Console.WriteLine(
    "{0} encontrados CON PARALELISMO en {1} ms",
     k, reloj.Elapsed);
  }
  public static bool EsPrimo(long num)
  {
    if ((num & 1) == 0)
      return (num == 2);
    if (num == 1) return false;
    for (long i = 3; i * i <= num; i += 2)
      if (num % i == 0)
        return false;
    return true;
  }
}
```

Listado 17.16 Utilización de Parallel.For

La implementación interna del paralelismo de la biblioteca TPL aprovechará la existencia de varios núcleos de procesamiento para ejecutar verdaderamente en forma paralela diferentes invocaciones al delegado que determina si un número es primo. El uso de la clase `Interlocked` es necesario al incrementar el contador con los resultados

pues, aunque no hemos creado una hebra de forma explícita, la ejecución en paralelo por parte de la TPL implica que habrá una cantidad variable de hebras operando.

```
Espere mientras encuentran primos
664579 primos encontrados SIN PARALELISMO en 00:00:14.6133902
664579 primos encontrados CON PARALELISMO en 00:00:03.5291499
Press any key to continue . . .
```

Figura 17.10 Salida para 10 millones

La salida de la Figura 17.10 muestra una aceleración de poco más de 4x. Quizás no sea coincidencia que el ejemplo fue ejecutado para esta demostración en una computadora con 4 núcleos físicos (8 virtuales con Hyper-Threading). Note que los núcleos virtuales ofrecen un cierto incremento en el rendimiento, pero no es tan grande como el de los núcleos físicos.

La mejora (aceleración) obtenida es excelente, sin embargo, esto no debe crear falsas expectativas en el lector, pues aún en casos como éste, donde es posible paralelizar 100% del problema[14], la propia división de las tareas, creación de hebras, sincronización de resultados y demás crean una sobrecarga adicional (*overhead*) que hace que la expectativa real de aceleración siempre sea S < N, donde N es el número de procesadores. El hecho de tener procesadores virtuales ayudó a pasar esa barrera, pero no permitió tampoco acercarse a una aceleración de 8x.

En el código de este ejemplo el método de encontrar los números primos carece de estado, pues se pretende usar para cada número sin conocimiento de ningún cálculo anterior. Un lector con conocimientos en matemáticas podría observar que quizás esto no es lo más eficiente y es mejor usar las cribas de Eratóstenes, Euler o Atkin. Sin embargo, el problema a resolver es saber si un numero K es primo. No es obtener todos los primos entre 1-K aunque es lo que hemos hecho en este ejemplo para ilustrar el paralelismo.

17.8.1.2 PARALLEL.FOREACH

Otra extensión presente en la TPL es `Parallel.ForEach`, y lo utilizaremos para realizar una iteración en paralelo sobre una colección. Este sería el equivalente en paralelo de un ciclo `foreach` con el cual podemos recorrer cualquier `IEnumerable`.

La sobrecarga[15] más simple es la que recibe dos parámetros: la colección y el delegado a ejecutar por cada elemento. Es la TPL la encargada de solicitar los elementos necesarios, dividirlos entre los procesadores y agrupar los resultados.

```
static void Main(string[] args)
{
  int maximo = 10000000; int k = 0;
  int[] elems = new int[maximo];
  Console.WriteLine(
```

[14] Estos problemas son llamados "embarazosamente paralelos" y esto significa que con solo adicionar más potencia de computo es posible resolverlos más rápido.

[15] Los métodos For y ForEach cuentan con una variedad de sobrecargas que dejamos al lector consultar en la ayuda, pues son muchas como para listarlas todas en este capítulo.

```
   "Generando {0} numeros aleatorios...", maximo);
  Random r = new Random(DateTime.Now.Millisecond);
  for (int i = 0; i < maximo; i++)
    elems[i] = r.Next();

  Console.WriteLine("Encontrando primos ...");

  Parallel.ForEach(elems, (num) =>
  {
    if (EsPrimo(num))
      Interlocked.Increment(ref k);
  });

  Console.WriteLine(
   " {0} encontrados CON PARALELISMO.", k);
}
```

Listado 17.17. Usando Parallel.ForEach para iterar una colección en paralelo

En el ejemplo del Listado 17.17 se ha generado un array con 10 millones de valores aleatorios y se utilizado `Parallel.ForEach` para determinar si cada uno de ellos es o no primo. En el ejemplo del Listado 17.18 se utiliza el `Parallel.ForEach` para crear versiones reducidas de todas las imágenes en una carpeta procurando mantener las proporciones de la imagen original. Se ha realizado la misma operación secuencialmente y en paralelo para mostrar la diferencia en rendimiento.

```
static void Main(string[] args)
{
  var origen = @"C:\demo\origen\";
  var salida1 = @"C:\demo\salida1\";
  var salida2 = @"C:\demo\salida2\";
  var imagenes = Directory.GetFiles(origen, "*.jpg");
  Image.GetThumbnailImageAbort callback =
    new Image.GetThumbnailImageAbort(ThumbnailCallback);
  Stopwatch reloj = new Stopwatch();
  reloj.Start();
  foreach (var archivo in imagenes)
  {
    var imagen = Image.FromFile(archivo);
    double proporcion = 120.0 / imagen.Width;
    var reducida = imagen.GetThumbnailImage(120,
      (int)(imagen.Height * proporcion), callback, IntPtr.Zero);
    reducida.Save(Path.Combine(salida1, Path.GetFileName(archivo)));
    reducida.Dispose();
    imagen.Dispose();
  }
  reloj.Stop();
  Console.WriteLine(reloj.Elapsed);
  reloj.Restart();
  Parallel.ForEach(imagenes, (archivo) =>
  {
    var imagen = Image.FromFile(archivo);
    double proporcion = 120.0 / imagen.Width;
```

```
    var reducida = imagen.GetThumbnailImage(120,
      (int)(imagen.Height * proporcion), callback, IntPtr.Zero);
    reducida.Save(Path.Combine(salida2, Path.GetFileName(archivo)));
    reducida.Dispose();
    imagen.Dispose();
  });
  Console.WriteLine(reloj.Elapsed);
}

public static bool ThumbnailCallback()
{
  return false;
}
```

Listado 17.18 Creado versiones reducidas de imágenes utilizando Parallel.ForEach

El método `Parallel.Foreach` pasa cada elemento de la colección como parámetro al delegado que se ejecuta en paralelo, lo cual resulta muy similar a la utilización de un `foreach` común y corriente, con la salvedad que el tiempo de ejecución es mucho menor como muestra la Figura 17.11. En este caso la aceleración fue más cercana a 3x aunque la computadora tiene 4 núcleos. En adición a la sobrecarga normal que ocurre cuando se paraleliza algo, también el acceso a disco pone cierto cuello de botella en el tiempo total de ejecución. El programador debe prestar atención a que elementos del sistema pueden poner trabas o demoras a la ejecución en paralelo de su programa.

```
Tiempo sin paralelismo: 00:00:20.6210589
Tiempo con paralelismo: 00:00:06.9393025
Press any key to continue . . .
```

Figura 17.11 Muestra de mejoramiento de rendimiento con Parallel.ForEach

17.8.1.3 PARALLEL LINQ

La extensión de paralelismo para LINQ (conocida como PLINQ) es quizás la variante más sencilla. Simplemente lo que hay que hacer es adicionar la extensión `AsParallel()` a cualquier consulta LINQ y el procesamiento de la consulta se hará en paralelo.

Usando el ejemplo anterior de reducción de las imágenes, ahora utilizando LINQ se procesan todos todos los archivos de una carpeta, pero adicionando `AsParallel()` en la consulta (Listado 17.19). La biblioteca TPL se encarga de dividir el trabajo entre los procesadores y combinar los resultados. El rendimiento es similar al del ejemplo utilizando `ForEach`.

```
static void Main(string[] args)
{
```

```
  var origen = @"C:\demo\origen\";
  var salida = @"C:\demo\salida\";
  var imagenes = Directory.GetFiles(origen, "*.jpg");
  var resultados = from archivo in imagenes.AsParallel()
                   select new
                   {
                     Nombre = Path.GetFileName(archivo),
                     Imagen = Reducida(archivo)
                   };
  Stopwatch reloj = new Stopwatch();
  reloj.Start();

  //en paralelo
  foreach (var r in resultados.AsParallel())
    File.WriteAllBytes(Path.Combine(salida, r.Nombre), r.Imagen);
  Console.WriteLine("Tiempo con paralelismo: " + reloj.Elapsed);
}

private static byte[] Reducida(string archivo)
{
  Image.GetThumbnailImageAbort callback =
    new Image.GetThumbnailImageAbort(ThumbnailCallback);
  var imagen = Image.FromFile(archivo);
  double proporcion = 120.0 / imagen.Width;
  var result = imagen.GetThumbnailImage(120,
    (int)(imagen.Height * proporcion), callback, IntPtr.Zero);
  var memStream = new MemoryStream();
  result.Save(memStream, ImageFormat.Jpeg);
  result.Dispose();
  imagen.Dispose();
  return memStream.ToArray();
}
```

Listado 17.19 Consulta con AsParallel()

Al igual que con el cálculo de números primos, se puede observar una buena mejoría en el tiempo total de ejecución. Como este programa hace acceso a disco, la velocidad de lectura y escritura al mismo interfieren en la velocidad general del proceso. Tal vez una mejor solución sería tener una sola hebra dedicada a salvar a disco [consumidor], mientras que otras hebras reducen las imágenes [productoras], se invita al lector a programar esto.

En este ejemplo el orden en que salen los resultados no es necesariamente el mismo en el que se van obteniendo en la consulta, esto se debe a que existen varias hebras

trabajando al unísono para procesar todos los datos. Si el orden es importante, se puede utilizar otra extensión llamada `AsOrdered()` en adición a `AsParallel()`. Claro, ordenar los resultados en la salida vendrá en sacrificio de rendimiento. Es una decisión del programador determinar si el orden de salida de los resultados es crucial en la solución de un problema.

17.8.2 PARALELISMO DE TAREAS

El paralelismo de tareas es quizás la forma mas tradicional de paralelismo; poder realizar varias tareas a la vez (los cocineros preparando diferentes platos para diferentes comensales o los cocineros preparando los diferentes ingredientes de un plato para un mismo comensal). En términos de código estas tareas se expresan como métodos o funciones (secuencia de instrucciones) que se ejecutan en diferentes hebras.

Desde que se presentó la clase `Thread`, se podría haber dicho que eso era una suerte de paralelismo de tareas, lo que técnicamente sería correcto. De modo que todo podría realizarse utilizando la clase `Thread` y los recursos de sincronización. Sin embargo, al igual que como se vio anteriormente con el `Parallel.For` y `Parallel.ForEach`, hay mejores recursos de abstracción en la TPL para expresar de modo más simple el trabajo con tareas.

17.8.2.1 TASKS

En este caso la clase `Task` y su variante `Task<T>` representan métodos o funciones, respectivamente, que se ejecutarán en otra hebra diferente a la que invoca su ejecución. Para construir una tarea hay que invocar su constructor pasando como parámetro un delegado o expresión lamba que especifica el código a ejecutar por la tarea. En el caso de `Task<T>`, la expresión lambda, o el delegado, deben retornar un valor del tipo genérico `T`.

Definir una tarea no significa que comience su ejecución; para eso se utiliza el método `Start()`, que iniciará la tarea en una hebra distinta de la que llama a `Start`. Varias propiedades de la clase, tales como `IsCompleted`, `IsFaulted`, `IsCancelled` y `Status`, brindan información sobre el estado de ejecución de la tarea.

Se puede esperar por la finalización de una tarea en la hebra actual mediante el método `Wait()` o una de sus variantes. Esto es similar al método `Join()` de la clase `Thread`. En el Listado 17.20 se ponen a calcular números primos mientras se realiza la ejecución de otro método.

```
static void Main(string[] args)
{
  Task tarea = new Task(() => CalculaPrimos(10000000));
  //iniciar la tarea en otra hebra
  tarea.Start();
  //realizar otras operaciones
  OtroMetodo();
  //esperar por que termine la tarea
  tarea.Wait();
}
```

```
private static void OtroMetodo()
{
  Thread.Sleep(2000);
}
```

Listado 17.20 Instanciando y esperando por una tarea

Si la tarea es de tipo `Task<T>` entonces otra forma de esperar por la terminación de la tarea es consultando la propiedad `Task<T>.Result`, que es una propiedad de tipo `T` que devuelve el valor calculado por la expresión lambda (o el delegado) asociado a la tarea. Esto es conocido como invocación "diferida" o "promesa"[16] pues la propiedad no tiene ningún valor hasta que la tarea termine. Una conveniencia de este tipo de recurso es que el programador puede iniciar una tarea que realiza un cálculo asíncrono y continuar realizando otras operaciones, para solo solicitar el valor de ese cálculo cuando lo requiera más adelante. Solo en ese momento es que la hebra esperará por el resultado si la tarea no ha terminado (Listado 17.21). El código muestra otra forma de crear e iniciar una tarea e iniciarla en la misma invocación utilizando el método estático `Task.Factory.StartNew<T>(Func<T>)`.

```
static void Main(string[] args)
{
  //iniciar la tarea en otra hebra
  var tarea = Task.Factory.StartNew(() =>
              { return CalculaPrimos(10000000); });
  //...realizar otras operaciones
  OtroMetodo();
  //...esperar por fin de la tarea y obtener el valor
  var total = tarea.Result;
  Console.WriteLine("Total de primos: " + total);
}
```

Listado 17.21 Utilizando la propiedad Task<T>.Result

Cuando se han invocado varias tareas puede que se quiera esperar solo porque termine la ejecución de una cualquiera de ellas o por la ejecución de todas. Para ello se dispone de los métodos `WaitAny` y `WaitAll` respectivamente (Listado 17.22).

```
void ProbandoWaitAll()
{
  var t1 = new Task(buscando1);
  var t2 = new Task(buscando2);
  var t3 = new Task(buscando3);

  t1.Start();
  t2.Start();
  t3.Start();

  Task.WaitAny(t1, t2, t3);
  //alguna de las tres terminó en este punto
  Task.WaitAll(t1, t2, t3);
```

[16] El término **promesa** es el que ha sido adoptado por la comunidad, también podríamos llamarle "variable futura"

```
  //todas terminaron en este punto
}
```

Listado 17.22 Uso de WaitAny/Waitall para esperar por fin de tareas

Es posible también especificar una secuencia de tareas para que se ejecuten una a continuación de la otra, pasándose los resultados de su ejecución. Para ello se dispone del método `ContinueWith`, como se hace en el

```
static void ProbandoContinueWith()
{
  var t1 = new Task<int>(metodo1);
  var t2 = new Task<string>(metodo2);
  var t3 = new Task<object>(metodo3);

  t1.ContinueWith((resultadoT1) => t2.Start())
    .ContinueWith((resultadoT2) => t3.Start());
  t1.Start();
  //Como t3 es la ultima, esperamos por ella..
  t3.Wait();
}
```

Listado 17.23. Aquí se han creado tres tareas que devuelven tipos específicos, y se usa `ContinueWith` para especificar cuál tarea será la próxima pasándole además los resultados de ejecución de la tarea previa, así en este caso `resultadoT1` es un `int`, y `resultadoT2` es un `string`. La TPL ejecutará las tareas `t1`,`t2` y `t3` de forma ordenada y finalmente esperará por la finalización de `t3`.

Listado 17.23 Creando cadena de tareas con ContinueWith

17.8.2.2 ASYNC/AWAIT

En C#[17] se tienen las palabras reservadas `async`/`await`, que facilitan aún más el trabajo con tareas y la programación de forma más transparente de la asincronía. Con este recurso es posible especificar que un método será asíncrono y cuándo esperar por la terminación de éste.

El significado de la palabra reservada `await` es contextual, ésta solo se puede utilizar dentro de un método (o delegado) que a su vez haya sido declarado como asíncrono (precediendo la definición del método con la palabra `async`). Excepto el método de entrada `Main`, cualquier método (expresión lambda o método anónimo) puede ser

[17] Desde la version 5.0

declarado como asíncrono siempre que retorne `Task`, `Task<T>` o `void`, siendo este último utilizado principalmente en los manejadores de eventos.

```
async Task<string> MedirPagina()
{
   HttpClient client = new HttpClient();
   //descargar una página de forma asíncrona
   Task<string> tDescarga =
     client.GetStringAsync("http://www.matcom.uh.cu");

   //ejecutar otro método independientemente
   OtroMetodo();

   //esperar por el resultado si aún no está listo
   string contenido = await tDescarga;

   //retornar el resultado
   return contenido;
 }
```

Listado 17.24 Uso del par async/await para crear y consumir métodos asíncronos

El Listado 17.24 muestra este par en uso, mientras se va descargando una página de internet se va realizando otra operación, cuando se ha concluido esta otra operación, se procede a disponer del contenido de la descarga.

Sea una hebra "A" desde la que se invoca a `MedirPagina()`, que a su vez invoca al método `HttpClient.GetStringAsync()` que es asíncrono, y por tanto comenzará la ejecución de la tarea en otra hebra "B" quedando en `tDescarga` una referencia a dicha tarea (ya iniciada). El método `OtroMetodo()` o cualquier código que se coloque dentro de esta sección justo antes del `await`, será ejecutado en la misma hebra "A" desde la que se invocó a `MedirPagina()`, concurrentemente con la descarga de la página que se está ejecutando en "B".

Al llegar a la línea donde se utiliza `await`, entonces la ejecución de "A" esperará que la hebra "B" termine la ejecución de la llamada `client.GetStringAsync` que se supone devuelva el valor `string` (la página descargada). Una vez que se tiene este valor asignado a la variable `contenido`, se retorna a quien llamó (dentro de la hebra "A").

El uso del par `async`/`await` permite crear código mucho más legible en el que se utilizan tareas facilitando el desarrollo de APIs asíncronas.

Es posible escribir código que no haga uso de `async`/`await`, solo utilizando los métodos `Start`, `Wait`, `ContinueWith` y otros de la clase `Task`, de la misma forma que también es posible hacerlo utilizando directamente la clase `Thread`; pero como se habrá dado cuenta el lector usando `async`/`await` se simplifica el trabajo y se logra un código más legible.

Uno de los beneficios principales de este recurso, es la facilidad que ofrece para definir APIs compleamente asíncronas. Será el compilador quien se encargue de realizar la traducción del código. Todo el código que sigue a una línea `await`, se coloca a continuación de la tarea por la que se espera, pero en la misma hebra, lo

que no es lo mismo que utilizar `ContinueWith`. Se propone al lector hacer esta programación por su cuenta para tener mejor noción de la complejidad involucrada.

Este recurso es muy útil para facilitar la escritura de interfaces gráficas de usuario (GUI) fluidas (siempre dispuestas a atender al usuario). Es decir, que una acción desencadenada con una componente visual no congele la interacción con la interfaz mientras esta acción se está ejecutando (por ejemplo un clic sobre un botón desencadena una búsqueda que accede a la red mientras se puede seguir interactuando con otras partes de la interfaz que no dependan del resultado de la búsqueda). Pero las ventajas no solo tienen que ver con componentes visuales, en las versiones recientes del .NET Framework una buena cantidad de APIs se están reinventando como asíncronas.

17.9 APOSTILLA

No queríamos concluir este libro sin dar una panorámica sobre temas tan importantes en la programación moderna como la concurrencia, el paralelismo, y la sincronización que esto implica.

En este capítulo se han presentado algunos de los principales conceptos y problemas de la programación concurrente y paralela, y mostrado algunas de las ventajas del .NET Framework y el lenguaje C# que ayudan a expresar de modo más simple este tipo de aplicaciones.

Estos temas por lo general no suelen ser objeto de muchos primeros cursos y entrenamientos de programación. En sí mismos son suficientes para dedicarles libros y cursos completos, de modo que es imposible pretender resumirlos en un simple capítulo. Sin embargo, esperamos que al menos lo presentado aquí le permita familiarizarse con estos temas y le sirvan de base e incentivo para profundizar por su cuenta en la gran cantidad de aportes y documentación existente. En tal caso no olvide buscar nombres de pioneros como Amdahl, Gustafson, Flynn y Dijkstra entre otros.