HILOS - Planificación

Objetivo

El objetivo de la guía es el de afianzar el conocimiento acerca de los hilos para resolver problemas reales con el uso del lenguaje de programación Java. Además le presentará las diferentes variantes de dicho mecanismo y los pros y contras de cada una.

Competencias

- Entender el concepto de Hilo
- Utilizar el lenguaje Java para implementar Hilos
- Resolver problemas reales mediante el uso de Hilos

Introducción

En un mundo idealizado, todos los subprocesos del programa tendrán sus propios procesadores para ejecutarlos. Hasta que llegue el momento en que las computadoras tengan miles o millones de procesadores, los hilos a menudo deben compartir uno o más procesadores. El sistema operativo de la JVM o de la plataforma subyacente descifra cómo compartir el recurso del procesador entre los subprocesos, una tarea conocida como programación de subprocesos. La parte de la JVM o del sistema operativo que realiza la programación de subprocesos es un planificador de subprocesos.

Recuerde dos puntos importantes sobre la programación de hilos:

Java no obliga a una VM a planificar los subprocesos de una manera específica o a tener un planificador de subprocesos. Esto implica una programación de hilos dependiente de la plataforma. Por lo tanto, debe tener cuidado al escribir un programa Java cuyo comportamiento depende de cómo se programan los subprocesos y debe operar de forma coherente en diferentes plataformas.

Afortunadamente, cuando se escriben programas Java, es necesario pensar en cómo Java planifica los subprocesos sólo cuando al menos uno de los subprocesos del programa utiliza mucho el procesador durante largos períodos de tiempo y los resultados intermedios de la ejecución de este subproceso son importantes. Por ejemplo, un applet contiene un subproceso que crea dinámicamente una imagen. Periódicamente, desea que el hilo de pintura dibuje el contenido actual de esa imagen para que el usuario pueda ver cómo avanza la imagen. Para asegurarse de que el hilo de cálculo no monopoliza el procesador, considere pensar en la planificación de subprocesos.

Examine el programa que crea dos subprocesos intensivos en procesador:

Listing 1. SchedDemo.java

```
// SchedDemo.java
class SchedDemo
{
   public static void main (String [] args)
   {
      new CalcThread("CalcThread A").start();
      new CalcThread("CalcThread B").start();
   }
}

class CalcThread extends Thread
{
   CalcThread(String name)
   {
      // Pass name to Thread layer.
      super (name);
   }

   double calcPI()
   {
      boolean negative = true;
      double pi = 0.0;
      for (int i = 3; i < 100000; i += 2)
      {
            if (negative)
            pi -= (1.0 / i);
            else
            pi += (1.0 / i);
            negative = !negative;
      }
      pi += 1.0;
      pi *= 4.0;
      return pi;
   }
   public void run ()
   {
      for (int i = 0; i < 5; i++)
      System.out.println (getName () + ": " + calcPI ());
   }
}
```

SchedDemo crea dos subprocesos que cada uno calcula el valor de pi (cinco veces) e imprime cada resultado. Dependiendo de cómo la implementación de la JVM planifique los subprocesos, es posible que aparezca una salida similar a la siguiente:

```
CalcThread A: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
```

De acuerdo con la salida anterior, el planificador de hilos comparte el procesador entre ambos hilos. Sin embargo, se puede ver salida similar a esto:

```
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread A: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
CalcThread B: 3.1415726535897894
```

La salida anterior muestra al prlanificador de hilos favoreciendo un hilo sobre otro. Las dos salidas anteriores ilustran dos categorías generales de planificadores de hilo: verde y nativo. Exploramos sus diferencias de comportamiento luego. Al discutir cada categoría, me refiero a los estados del hilo, de los cuales hay cuatro:

Estado inicial: un programa ha creado el objeto de subproceso de un subproceso, pero el subproceso todavía no existe porque el método start() del objeto de subproceso todavía no se ha llamado.
Estado Runnable: Este es el estado predeterminado de un thread. Una vez completada la llamada a start(), un subproceso se convierte en runnable si el subproceso está ejecutándose, es decir, utilizando el procesador. Aunque muchos subprocesos pueden ser ejecutables, sólo se ejecuta uno. Los planificadores de subprocesos determinan qué subprograma ejecutable asignar al procesador.
Estado bloqueado: cuando un subproceso ejecuta los métodos sleep(), wait() ó join(), cuando un subproceso intenta leer datos aún no disponibles de una red y cuando un hilo espera para adquirir un bloqueo, este subproceso está en estado bloqueado: no está en funcionamiento ni en posición de correr. (Probablemente puede pensar en otras veces cuando un hilo esperaría a que algo suceda). Cuando un subproceso bloqueado se desbloquea, este subproceso se desplaza al estado ejecutable.
Estado de finalización: Una vez que la ejecución deja el método run() de un subproceso, ese subproceso está en el estado de finalización. En otras palabras, el hilo deja de existir.

Planificación de hilos verdes

No todos los sistemas operativos admiten subprocesos, por ejemplo el antiguo sistema de Microsoft Windows 3.1. Para estos sistemas, Sun Microsystems puede diseñar una JVM que divida su único hilo de ejecución en múltiples subprocesos. La JVM (no el sistema operativo de la plataforma subyacente) suministra la lógica de subprocesamiento y contiene el planificador de subprocesos. Los subprocesos de JVM son subprocesos verdes o subprocesos de usuario.

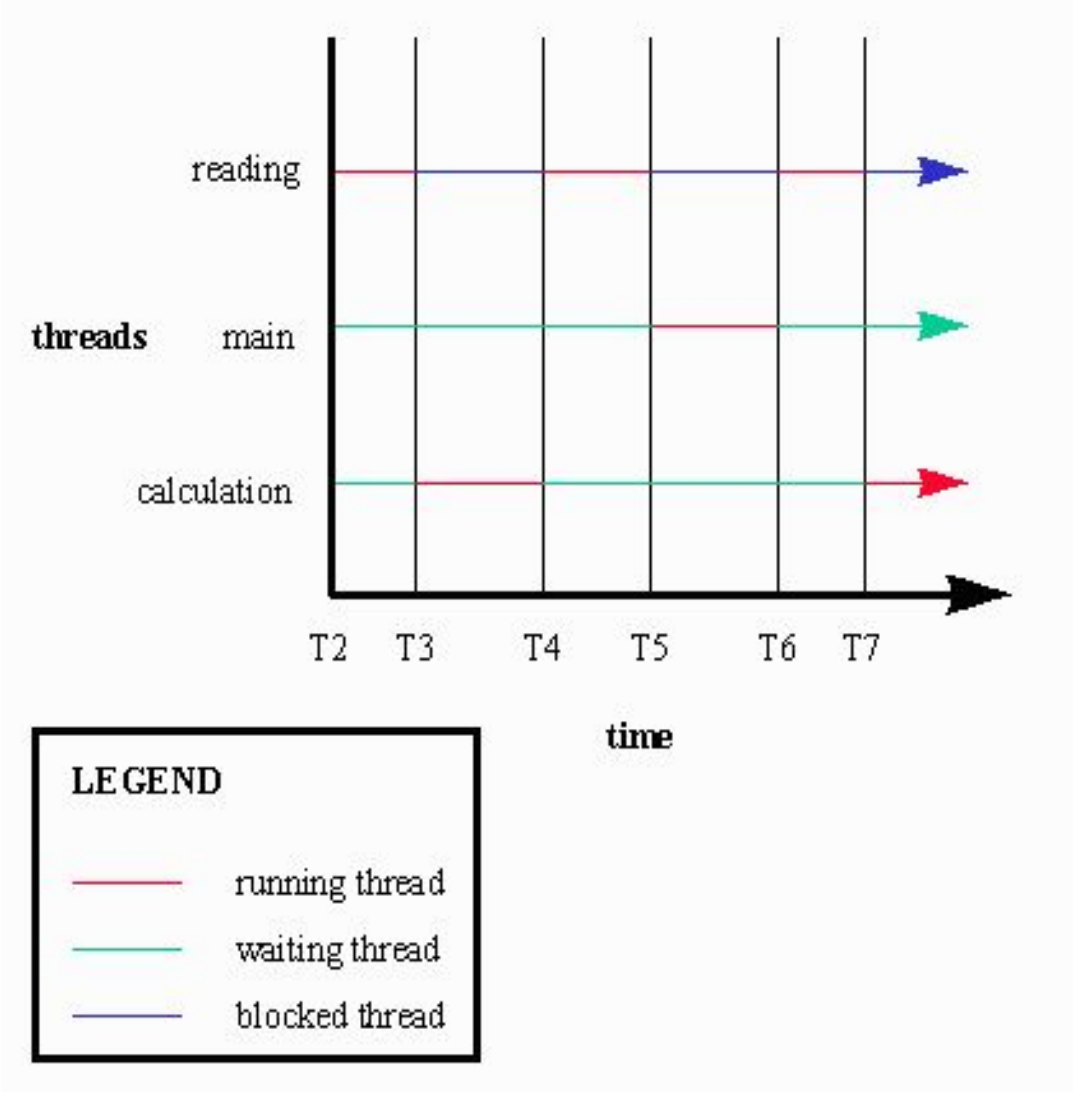
El planificador de subprocesos de una JVM planifica los subprocesos verdes de acuerdo con la prioridad -la importancia relativa de un subproceso, que se expresa como un entero de un intervalo de valores bien definido. Normalmente, el programador de subprocesos de JVM elige el subproceso de prioridad más alta y permite que dicho subproceso se ejecute hasta que se termina o bloquea. En ese momento, el programador de subproceso elige un subproceso de la siguiente prioridad más alta. Ese hilo (usualmente) se ejecuta hasta que se termina o bloquea. Si, mientras se ejecuta un subproceso, un subproceso de prioridad más alta se desbloquea (quizás el tiempo de suspensión del subproceso de prioridad más alta expiró), el planificador de subprocesos previene o interrumpe el subproceso de prioridad inferior y asigna el subproceso desbloqueado de alta prioridad al procesador.

> Nota: Un subproceso ejecutable con la prioridad más alta no siempre se ejecutará. Aquí está la especificación de la prioridad del lenguaje Java:
>
> Cada hilo tiene una prioridad. Cuando hay competencia para procesar recursos, los subprocesos con mayor prioridad se ejecutan generalmente en preferencia a los subprocesos con menor prioridad. Esta preferencia no es, sin embargo, una garantía de que el hilo de prioridad más alta siempre estará en ejecución, y las prioridades de hilo no pueden utilizarse para implementar de forma confiable la exclusión mutua.

Esa nota dice mucho sobre la implementación de JVM de los hilos verdes. Esas JVMs no pueden darse el lujo de dejar que los hilos se bloqueen porque eso ataría el único hilo de ejecución de la JVM. Por lo tanto, cuando un subproceso debe bloquearse, como cuando el subproceso está leyendo los datos que tardan en llegar desde un archivo, la JVM puede detener la ejecución del subproceso y usar un mecanismo de sondeo para determinar cuándo llega el dato. Mientras el subproceso permanece detenido, el planificador de subprocesos de la JVM podría programar un subproceso de prioridad inferior para ejecutarse. Suponga que los datos llegan mientras se ejecuta el subproceso de prioridad inferior. Aunque el hilo de mayor prioridad debe ejecutarse tan pronto como los datos lleguen, eso no sucede hasta que la JVM le pregunte a el sistema operativo y descubra la llegada. Por lo tanto, el subproceso de prioridad inferior se ejecuta aunque el subproceso de mayor prioridad deba ejecutarse. Sólo necesita preocuparse por esta situación sólo cuando

requiera un comportamiento en tiempo real desde Java. Pero Java no es un sistema operativo en tiempo real, así que ¿por qué preocuparse?

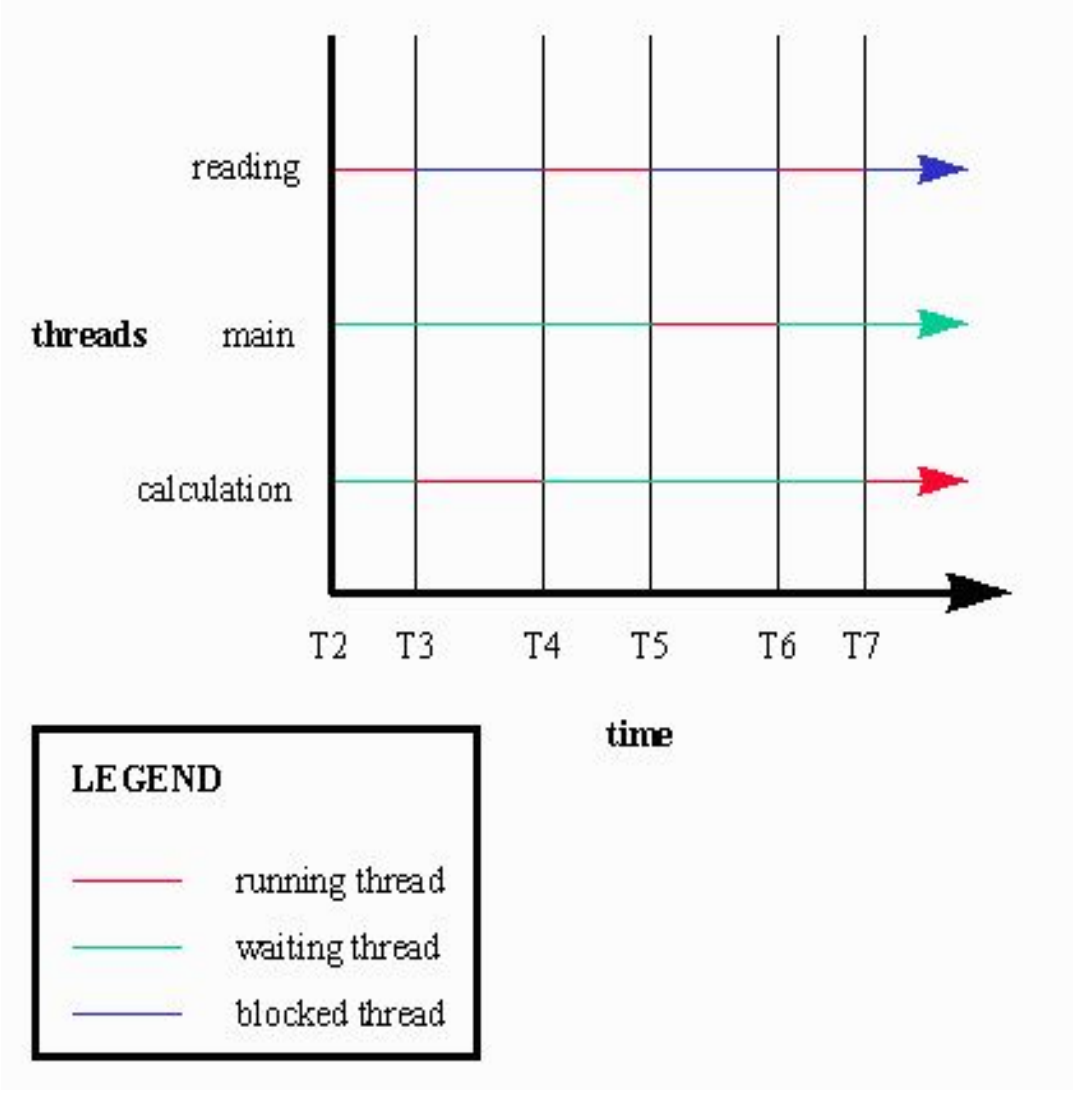
Para entender qué hilo verde ejecutable se convierte en el hilo verde actualmente en ejecución, considere lo siguiente. Suponga que su aplicación consta de tres subprocesos: el subproceso principal que ejecuta el método main(), un subproceso de cálculo y un subproceso que lee la entrada del teclado. Cuando no hay entrada de teclado, el hilo de lectura se bloquea. Suponga que el hilo de lectura tiene la prioridad más alta y el hilo de cálculo tiene la prioridad más baja. (Por simplicidad, asuma también que no hay otros hilos JVM internos disponibles). La Figura 1 ilustra la ejecución de estos tres subprocesos.



En el instante T0, el hilo principal comienza a funcionar. En el instante T1, el hilo principal inicia el hilo de cálculo. Debido a que el hilo de cálculo tiene una prioridad menor que el hilo principal, el hilo de cálculo espera al procesador. En el instante T2, el hilo principal inicia el hilo de lectura. Debido a que el hilo de lectura tiene una prioridad más alta que el hilo principal, el hilo principal espera al procesador mientras se ejecuta el hilo de lectura. En el

instante T3, los bloques de hilos de lectura y el hilo principal corren. En el instante T4, el hilo de lectura se desbloquea y se desplaza; El hilo principal espera. Finalmente, en el instante T5, los bloques de hilos de lectura y el hilo principal corren. Esta alternancia en ejecución entre la lectura y los hilos principales continúa mientras el programa se ejecuta. El hilo de cálculo nunca se ejecuta porque tiene la prioridad más baja, una situación conocida como inanición.

Podemos alterar este escenario dando al hilo de cálculo la misma prioridad que el hilo principal. La figura 2 muestra el resultado, comenzando con el tiempo T2. (Antes de T2, la Figura 2 es idéntica a la Figura 1.)



En el instante T2, el hilo de lectura se ejecuta mientras los hilos principal y de cálculo esperan el procesador. En el instante T3, se ejecutan los bloques de hilos de lectura y el hilo de cálculo, porque el hilo principal se encontraba justo antes del hilo de lectura. En el instante T4, el hilo de lectura se desbloquea y se desplaza; Los hilos principales y de cálculo esperan. En el instante T5, los bloques de hilos de lectura y el hilo principal se

ejecutan, porque el hilo de cálculo se ejecuta justo antes del hilo de lectura. Esta alternancia en la ejecución entre los subprocesos principal y de cálculo continúa mientras el programa funcione y dependa de la ejecución y bloqueo de la hebra de mayor prioridad.

Debemos considerar un último elemento en la programación de hilo verde. ¿Qué sucede cuando un hilo de menor prioridad tiene un bloqueo que un hilo de mayor prioridad requiere? Esto bloquea el subproceso de mayor prioridad porque no puede obtener el bloqueo, lo que implica que el hilo de prioridad más alta tiene efectivamente la misma prioridad que el subproceso de prioridad inferior. Por ejemplo, un hilo de prioridad 6 intenta adquirir un bloqueo que mantiene un hilo de prioridad 3. Debido a que el subproceso de prioridad 6 debe esperar hasta que pueda adquirir el bloqueo, el subproceso de prioridad 6 termina con una prioridad 3, un fenómeno conocido como inversión de prioridad.

La inversión de prioridad puede retardar en gran medida la ejecución de un hilo de mayor prioridad. Por ejemplo, suponga que tiene tres subprocesos con prioridades de 3, 4 y 9. El subproceso de Prioridad 3 se está ejecutando y los otros subprocesos están bloqueados. Suponga que el hilo de prioridad 3 toma un bloqueo y el subproceso de prioridad 4 se desbloquea. El subproceso de prioridad 4 se convierte en el subproceso actualmente en ejecución. Debido a que el hilo de prioridad 9 requiere el bloqueo, continúa esperando hasta que el hilo de prioridad 3 libere el bloqueo. Sin embargo, el hilo de prioridad 3 no puede liberar el bloqueo hasta que el bloqueo de prioridad 4 se bloquee o termine. Como resultado, el hilo de prioridad 9 retarda su ejecución.

El planificador de hilos de una JVM normalmente resuelve el problema de inversión de prioridad a través de la herencia de prioridad: El planificador de hilos aumenta en silencio la prioridad del hilo que sostiene el bloqueo cuando un hilo de mayor prioridad solicita el bloqueo. Como resultado, tanto el hilo que sostiene el bloqueo como el hilo que espera el bloqueo tienen temporalmente prioridades iguales. Utilizando el ejemplo anterior, el hilo de prioridad 3 (que sostiene el bloqueo) se convertiría temporalmente en un hilo de prioridad 9 tan pronto como el hilo de prioridad 9 intente adquirir el bloqueo y se bloquee. Como resultado, el hilo de prioridad 9 (que sostiene el bloqueo) se convertiría en el subproceso actualmente en ejecución (incluso cuando el subproceso de prioridad 4 se desbloquee). El hilo de prioridad 9 terminaría su ejecución y liberaría el bloqueo, permitiendo que el hilo de prioridad de espera 9 adquiera el bloqueo y continue la ejecución. El hilo de prioridad 4 carecería de la oportunidad de convertirse en el subproceso actualmente en ejecución. Una vez que el subproceso con prioridad elevada libera el bloqueo, el planificador de subprocesos restaura la prioridad del subproceso a su prioridad original. Por lo tanto, el planificador de hilos restablecerá el hilo de prioridad 9 que previamente sostenía el bloqueo a la prioridad 3, una vez que libere el bloqueo. Por lo tanto, la herencia de prioridad garantiza que un hilo de menor prioridad que sostiene un bloqueo no sea interrumpido por un hilo cuya prioridad exceda la prioridad del hilo que retiene el bloqueo, pero es menor que la prioridad del hilo esperando que se libere el bloqueo.

Planificación de hilos nativos

La mayoría de las JVM dependen del sistema operativo subyacente (como Linux o Microsoft Windows XP) para proporcionar un planificador de subprocesos. Cuando un sistema operativo gestiona la planificación de subprocesos, los subprocesos son subprocesos nativos. Al igual que con la planificación de hilo verde, la prioridad resulta importante para la planificación de hilos nativos: los hilos de prioridad más alta normalmente interrumpen subprocesos de prioridad baja. Sin embargo, los programadores de hilos nativos a menudo introducen un detalle adicional: time-slicing (quantum).

> Nota: Algunos planificadores de hilo verde también admiten el time-slicing. Y muchos planificadores de hilos nativos admiten la herencia de prioridad. Como resultado, los planificadores de hilo verde y los planificadores de hilos nativos normalmente difieren sólo en el origen del planificador de hilos: JVM o sistema operativo.

Los planificadores de subprocesos nativos suelen introducir el time-slicing para evitar que la inanición del procesador en hilos de prioridad igual. La idea es dar a cada hilo de igual prioridad la misma cantidad de tiempo, conocido como un quantum. Un temporizador registra el tiempo restante de cada quantum y alerta al planificador de hilos cuando el quantum expira. El planificador de subprocesos planifica entonces otro subproceso de prioridad igual para ejecutarse, a menos que se desbloquee un subproceso de mayor prioridad.

El Time-slicing complica la escritura de los planificadores multiproceso independientes de la plataforma que dependen de la planificación coherente de subprocesos, porque no todos los planificadores de subprocesos implementan el tiem-slice. Sin cortar el tiempo, un subproceso ejecutable de prioridad igual se mantendrá en ejecución (suponiendo que es el subproceso actualmente en ejecución) hasta que el subproceso finalice, se bloquee o se reemplace por un subproceso de mayor prioridad. Por lo tanto, el planificador de subprocesos no puede dar a todos los subprocesos ejecutables de igual prioridad la posibilidad de ejecutarse. Aunque complicado, el time-slicing no le impide escribir programas multiproceso independientes de la plataforma. Los métodos setPriority(int priority) y yield() influyen en la planificación de subprocesos para que un programa se comporte de manera bastante consistente (en lo que respecta a la planificación de subprocesos) en todas las plataformas.

> Nota: para evitar que los hilos de menor prioridad mueran de hambre, algunos planificadores de subprocesos, como los de Windows, otorgan aumentos temporales de prioridad a subprocesos que no se han ejecutado en mucho tiempo. Cuando se ejecuta el hilo, ese incremento de prioridad disminuye. Los planificadores siguen dando preferencia a subprocesos de

> prioridad superior a los subprocesos de prioridad más baja, pero al menos todos los subprocesos tienen la oportunidad de ejecutarse.

Programación con el método setPriority(prioridad int)

¡Suficiente teoría! Aprendamos cómo influir en la planificación de subprocesos en el nivel del código fuente. Una forma es usar el método `Thread void setPriority(int priority);`. Cuando se le llama, `setPriority(int priority)` establece la prioridad de un subproceso asociado al objeto de hilo especificado (como en `thd.setPriority (7);`). Si la prioridad no está dentro del rango de prioridades que las constantes MIN_PRIORITY y MAX_PRIORITY de Thread especifican, `setPriority(int priority)` lanza un objeto `IllegalArgumentException`.

> Nota: Si llama a setPriority(prioridad int) con un valor de prioridad que excede la prioridad máxima permitida para el grupo de subprocesos del subproceso respectivo, este método reduce silenciosamente el valor de prioridad para que coincida con la prioridad máxima del grupo de subprocesos.

Cuando necesite determinar la prioridad actual de un subproceso, llame al método int getPriority() del Thread, a través del objeto thread de ese subproceso. El método `getPriority()` devuelve un valor entre MIN_PRIORITY (1) y MAX_PRIORITY (10). Uno de esos valores puede ser 5: el valor que asigna a la constante NORM_PRIORITY, que representa la prioridad predeterminada de un subproceso.

El método `setPriority(int priority)` resulta útil para evitar la inanición. Por ejemplo, suponga que su programa consta de un subproceso que bloquea y un subproceso de cálculo que no bloquea. Al asignar una prioridad más alta al subproceso que bloquea, asegúrese de que el hilo de cálculo no suprime el hilo de bloqueo. Debido a que el hilo de bloqueo periódicamente se bloquea, el hilo de cálculo no se muere de hambre. Por supuesto, suponemos que el planificador de subprocesos no admite time-slicing. Si el programador de subprocesos admite temporalmente el corte, es probable que no vea ninguna diferencia entre las llamadas a `setPriority(prioridad int)` y si no hay llamadas a ese método, dependiendo de lo que están haciendo los subprocesos afectados. Sin embargo, al menos se asegurará de su código será portable alrededor de los planificadores de subprocesos.

Listing 2. PriorityDemo.java

```
// PriorityDemo.java
class PriorityDemo
```

```
{
   public static void main (String [] args)
   {
      BlockingThread bt = new BlockingThread ();
      bt.setPriority (Thread.NORM_PRIORITY + 1);
      CalculatingThread ct = new CalculatingThread ();
      bt.start ();
      ct.start ();
      try
      {
      Thread.sleep (10000);
      }
      catch (InterruptedException e)
      {
      }
      bt.setFinished (true);
      ct.setFinished (true);
   }
}

class BlockingThread extends Thread
{
   private boolean finished = false;
   public void run ()
   {
      while (!finished)
      {
      try
      {
            int i;
            do
            {
            i = System.in.read ();
            System.out.print (i + " ");
            }
            while (i != '\n');
            System.out.print ('\n');
      }
      catch (java.io.IOException e)
      {
      }
      }
   }
   public void setFinished (boolean f)
   {
      finished = f;
   }
}
class CalculatingThread extends Thread
{
```

```
   private boolean finished = false;
   public void run ()
   {
      int sum = 0;
      while (!finished)
      sum++;
   }
   public void setFinished (boolean f)
   {
      finished = f;
   }
}
```

PriorityDemo tiene un hilo que se bloquea y un hilo de cálculo además del hilo principal. Supongamos que ha ejecutado este programa en una plataforma en la que el planificador de subprocesos no admite el time-slice. ¿Qué pasaría? Considere dos escenarios:

Suponga que no hay `bt.setPriority(Thread.NORM_PRIORITY + 1);`: El hilo principal se ejecuta hasta que duerme. En ese momento, suponga que el planificador de subprocesos inicia el subproceso de bloqueo. Ese hilo se ejecuta hasta que llama a System.in.read (), lo que hace es bloquearse. El planificador de hilos asigna entonces el hilo de cálculo al procesador (asumiendo que el hilo principal no se ha desbloqueado de su estado de suspensión). Debido a que los hilos de bloqueo, principal y de cálculo tienen la misma prioridad, el hilo de cálculo continúa ejecutándose en un bucle infinito.

Planificar con el método yield()

Muchos desarrolladores prefieren la alternativa al método setPriority(prioridad int), el método de Thread `static void yield();`, debido a su simplicidad. Cuando el subproceso actualmente en ejecución llama a `Thread.yield ();`, el planificador de subprocesos mantiene el subproceso actualmente en ejecución en el estado ejecutable, pero (usualmente) elige otro subproceso de igual prioridad para ser el subproceso actual, a menos que haya un subproceso de mayor prioridad. en cuyo caso el hilo de mayor prioridad se convierte en el hilo actualmente en ejecución. Si no hay ningún subproceso de prioridad más alta ni otros subprocesos de prioridad igual, el planificador de subprocesos reprograma inmediatamente el subproceso que llamo a yield() como el subproceso en ejecución. Además, cuando el planificador de hilos selecciona un hilo de prioridad igual, el hilo escogido puede ser el hilo que llamo a yield(), lo que significa que yield() no realiza nada excepto el retraso. Este comportamiento normalmente ocurre bajo un planificador de subprocesos time-slicing. El Listado 3 muestra el método yield():

Listing 3. YieldDemo.java

```
// YieldDemo.java
class YieldDemo extends Thread
{
```

```
   static boolean finished = false;
   static int sum = 0;
   public static void main (String [] args)
   {
      new YieldDemo ().start ();
      for (int i = 1; i <= 50000; i++)
      {
            sum++;
            if (args.length == 0)
            Thread.yield ();
      }
      finished = true;
   }
   public void run ()
   {
      while (!finished)
      System.out.println ("sum = " + sum);
   }
}
```

Desde una perspectiva lógica, el hilo principal de YieldDemo inicia un nuevo subproceso (de la misma prioridad NORM_PRIORITY) que emite repetidamente el valor de una suma hasta que el valor del campo de instancia `finished` es verdadero. Después de iniciar el hilo, el hilo principal entra en un bucle que incrementa repetidamente el valor de la suma. Si ningún argumento pasa a YieldDemo en la línea de comandos, el hilo principal llama a `Thread.yield();` después de cada incremento. De lo contrario, no se hace ninguna llamada a ese método. Una vez que el bucle termina, el hilo principal asigna true a finished, por lo que el otro hilo terminará. Después de eso, el hilo principal termina.

Ahora que sabes lo que YieldDemo debe lograr, ¿qué tipo de comportamiento puedes esperar? Esa respuesta depende de si el planificador de hilos usa el time-slice y si se hacen llamadas a yield(). Tenemos cuatro escenarios a considerar:

No se corta el tiempo y no hay llamadas a yield(): El hilo principal se ejecuta hasta su finalización. El planificador de subprocesos no planificará el subproceso de salida una vez que termine el subproceso principal. Por lo tanto, no ve ninguna salida.

No time-slicing y yield (): Después de la primera llamada yield (), el thread de salida se ejecuta para siempre porque finished contiene false. Debería ver el mismo valor de suma impresa repetidamente en un bucle sin fin (porque el hilo principal no se ejecuta e incrementa la suma). Para contrarrestar este problema, el subproceso de salida también debería llamar a yield () durante cada iteración del bucle while.

Time-slicing y no yield(): ambos hilos tienen aproximadamente igual cantidad de tiempo para ejecutar. Sin embargo, probablemente verá muy pocas líneas de salida porque cada System.out.println ("sum =" + sum); La llamada al método ocupa una mayor porción de un quantum que una suma ++;. (Se requieren muchos ciclos de procesador para enviar salida

al dispositivo de salida estándar, mientras que (relativamente) pocos ciclos de procesador son necesarios para incrementar una variable entera). Se observan menos líneas de salida.

Time-slicing y yield (): Debido a que el thread principal produce cada vez que incrementa la suma, el hilo principal completa menos trabajo durante un quantum. Debido a eso, y debido a que el hilo de salida recibe más cuántos, se ven muchas más líneas de salida.

> Nota: ¿Debería llamar a setPriority(int priority) o yield()? Ambos métodos afectan los hilos de forma similar. Sin embargo, setPriority(int priority) ofrece flexibilidad, mientras que yield() ofrece simplicidad. Además, yield() podría reprogramar inmediatamente el hilo que la produce, lo cual no logra nada.

El mecanismo de espera / notificación

El bloqueo asociado y el área de espera de cada objeto permiten a la JVM sincronizar el acceso a las secciones de código crítico. Por ejemplo: Cuando el hilo X intenta obtener un bloqueo antes de entrar en un contexto sincronizado que protege una sección de código crítico del acceso concurrente al hilo, y el hilo Y está ejecutándose dentro de ese contexto (y manteniendo el bloqueo), la JVM coloca a X en un área de espera. Cuando Y sale del contexto sincronizado (y libera el bloqueo), la JVM elimina X del área de espera, asigna el bloqueo a X y permite que dicho hilo entre en el contexto sincronizado. Además de su uso en sincronización, el área de espera cumple un segundo objetivo: forma parte del mecanismo de espera / notificación, el mecanismo que coordina las actividades de múltiples hilos.

La idea detrás del mecanismo de espera/notificación es la siguiente: Un hilo se obliga a esperar algún tipo de condición, un requisito previo para la ejecución, debe existir antes de que continúe. El hilo en espera supone que algún otro hilo creará esa condición y, a continuación, notificará al hilo en espera para continuar la ejecución. Normalmente, un hilo examina el contenido de una variable de condición -una variable booleana que determina si un hilo espera- para confirmar que no existe una condición. Si no existe una condición, el hilo espera en el área de espera de un objeto. Posteriormente, otro hilo establecerá la condición modificando el contenido de la variable de condición y luego notificando al hilo de espera que la condición ya existe y el hilo de espera puede continuar la ejecución.

Para soportar el mecanismo wait/notify, Object declara el método `void wait();` (para forzar un hilo a esperar) y el método `void notify();` (para notificar un hilo en espera que puede continuar la ejecución). Debido a que cada objeto hereda los métodos de Object, wait() y notify() están disponibles para todos los objetos. Ambos métodos comparten una característica común: están sincronizados. Un subproceso debe llamar a wait () o notify() desde dentro de un contexto sincronizado debido a una condición de competencia inherente al mecanismo wait/notify. Aquí es cómo funciona esa condición de carrera:

1. El hilo A prueba una condición y descubre que debe esperar.El hilo B establece la condición y llama a notify() para informar a A para reanudar la ejecución. Porque A todavía no está esperando, no pasa nada.
2. El hilo A espera, llamando a wait().
3. Debido a la notificación previa () llamada, A espera indefinidamente.

Para resolver la condición de carrera, Java requiere un subproceso para introducir un contexto sincronizado antes de llamar a wait() o notify(). Además, el subproceso que llama wait() (el subproceso de espera) y el subproceso que llama notify() (el subproceso de notificación) deben competir por el mismo bloqueo. Cualquier hilo debe llamar a wait() o notify() a través del mismo objeto en el que entran en sus contextos sincronizados porque wait() se integra firmemente con el bloqueo. Antes de esperar, un hilo que ejecuta wait() libera el bloqueo, lo que permite que el hilo de notificación entre en su contexto sincronizado para establecer la condición y notificar el hilo en espera. Una vez que llega la notificación, la JVM despierta el hilo de espera, que luego intenta volver a adquirir el bloqueo. Cuando se reactiva con éxito la cerradura, el hilo de espera anterior regresa de wait(). ¿Confuso? El fragmento de código siguiente ofrece una aclaración:

```
// Condition variable initialized to false
// to indicate condition has not occurred.
boolean conditionVar = false;
// Object whose lock threads synchronize on.
Object lockObject = new Object();

// Thread A waiting for condition to occur...
synchronized (lockObject)
{
   while (!conditionVar)
     try
     {
          lockObject.wait();
     }
     catch (InterruptedException e) {}
}

// ... some other method
// Thread B notifying waiting thread that condition has now
occurred...
synchronized (lockObject)
{
   conditionVar = true;
   lockObject.notify ();
}
```

El fragmento de código introduce la variable de condición conditionVar, que los hilos A y B usan para probar y establecer una condición y bloquear la variable lockObject, que ambos hilos utilizan para propósitos de sincronización. La variable de condición se inicializa en false porque la condición no existe cuando el código inicia la ejecución. Cuando A necesita esperar una condición, entra en un contexto sincronizado (siempre que B no esté en su contexto sincronizado). Una vez dentro de su contexto, A ejecuta una sentencia while loop cuya expresión booleana prueba el valor de conditionVar y espera (si el valor es false) llamando a wait(). Observe que lockObject aparece como parte de synchronized (lockObject) y lockObject.wait(); - que no es coincidencia. Desde dentro de wait(), A libera el bloqueo asociado con el objeto en el que se realiza la llamada a wait(): el objeto asociado con lockObject en lockObject.wait();. Esto permite que B ingrese a su contexto sincronized(lockObject), establezca conditionVar a true y llame a lockObject.notify(); Para notificar a A que la condición ya existe. Al recibir la notificación, A intenta volver a adquirir su cerradura. Eso no ocurre hasta que B deja su contexto sincronizado. Una vez que A retoma su bloqueo, regresa de wait() y vuelve a probar la variable de condición. Si el valor de esta variable es true, A deja su contexto sincronizado.

> Precaución: Si se realiza una llamada a wait () o notify () desde fuera de un contexto sincronizado, cualquiera de las llamadas produce una IllegalMonitorStateException.

Aplicar wait / notify a la relación productor - consumidor

Para demostrar la practicidad de wait / notify, te presento la relación productor-consumidor, que es común entre los programas multiproceso donde dos o más subprocesos deben coordinar sus actividades. La relación productor-consumidor demuestra la coordinación entre un par de hilos: un hilo productor (productor) y un hilo consumidor (consumidor). El productor produce algún artículo que consume un consumidor. Por ejemplo, un productor lee los elementos de un archivo y los pasa a un consumidor para su procesamiento. El productor no puede producir un artículo si no hay espacio disponible para almacenar ese artículo porque el consumidor no ha terminado de consumir su artículo (s). Además, un consumidor no puede consumir un elemento que no existe. Esas restricciones impiden que un productor produzca artículos que un consumidor nunca recibe para consumo y evita que un consumidor intente consumir más artículos de los que están disponibles. El Listado 4 muestra la arquitectura de un programa orientado al productor / consumidor:

Listing 4. ProdCons1.java

```
// ProdCons1.java
class ProdCons1
{
```

```
   public static void main (String [] args)
   {
      Shared s = new Shared ();
      new Producer (s).start ();
      new Consumer (s).start ();
   }
}
class Shared
{
   private char c = '\u0000';
   void setSharedChar (char c) { this.c = c; }
   char getSharedChar () { return c; }
}
class Producer extends Thread
{
   private Shared s;
   Producer (Shared s)
   {
      this.s = s;
   }
   public void run ()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
           try
           {
              Thread.sleep ((int) (Math.random () * 4000));
           }
           catch (InterruptedException e) {}
           s.setSharedChar (ch);
           System.out.println (ch + " produced by producer.");
      }
   }
}
class Consumer extends Thread
{
   private Shared s;
   Consumer (Shared s)
   {
      this.s = s;
   }
   public void run ()
   {
      char ch;
      do
      {
         try
         {
            Thread.sleep ((int) (Math.random () * 4000));
         }
```

```
            catch (InterruptedException e) {}
            ch = s.getSharedChar ();
            System.out.println (ch + " consumed by consumer.");
         }
         while (ch != 'Z');
      }
}
```

ProdCons1 crea hilos productores y consumidores. El productor pasa las letras mayúsculas individualmente al consumidor llamando a s.setSharedChar(ch) ;. Una vez que el productor termina, ese hilo termina. El consumidor recibe caracteres en mayúsculas, desde dentro de un bucle, llamando a s.getSharedChar (). La duración del bucle depende del valor de retorno de ese método. Cuando se devuelve Z, el bucle termina, y, por lo tanto, el productor informa al consumidor cuando terminar. Para que el código sea más representativo de los programas del mundo real, cada hilo duerme durante un período de tiempo aleatorio (hasta cuatro segundos) antes de producir o consumir un elemento.

Dado que el código no contiene condiciones de carrera, la palabra clave sincronizada está ausente. Todo parece bien: el consumidor consume todos los caracteres que produce el productor. En realidad, existen algunos problemas que la siguiente salida parcial de una invocación de este programa muestra:

```
consumed by consumer.
A produced by producer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
E produced by producer.
F produced by producer.
F consumed by consumer.
```

La primera línea de salida, consumida por el consumidor., Muestra al consumidor que intenta consumir una letra mayúscula no existente. La salida también muestra al productor que produce una letra (A) que el consumidor no consume. Esos problemas no resultan de la falta de sincronización. En su lugar, los problemas resultan de la falta de coordinación entre el productor y el consumidor. El productor debe ejecutar primero, producir un solo elemento, y luego esperar hasta que reciba la notificación de que el consumidor ha consumido el artículo. El consumidor debe esperar hasta que el productor produzca un artículo. Si ambos hilos coordinan sus actividades de esa manera, los problemas antes mencionados desaparecerán. El Listado 5 demuestra que la coordinación, que el mecanismo wait / notify inicia:

Listing 5. ProdCons2.java

```
// ProdCons2.java

class ProdCons2
{
   public static void main (String [] args)
   {
      Shared s = new Shared ();
      new Producer (s).start ();
      new Consumer (s).start ();
   }
}

class Shared
{
   private char c = '\u0000';
   private boolean writeable = true;

   synchronized void setSharedChar (char c)
   {
      while (!writeable)
         try
         {
            wait ();
         }
         catch (InterruptedException e) {}

      this.c = c;
      writeable = false;
      notify ();
   }

   synchronized char getSharedChar ()
   {
      while (writeable)
         try
         {
            wait ();
         }
         catch (InterruptedException e) { }

      writeable = true;
      notify ();

      return c;
   }
}

class Producer extends Thread
```

```
{
   private Shared s;

   Producer (Shared s)
   {
      this.s = s;
   }

   public void run ()
   {
      for (char ch = 'A'; ch <= 'Z'; ch++)
      {
           try
           {
              Thread.sleep ((int) (Math.random () * 4000));
           }
           catch (InterruptedException e) {}

           s.setSharedChar (ch);
           System.out.println (ch + " produced by producer.");
      }
   }
}

class Consumer extends Thread
{
   private Shared s;

   Consumer (Shared s)
   {
      this.s = s;
   }

   public void run ()
   {
      char ch;

      do
      {
         try
         {
            Thread.sleep ((int) (Math.random () * 4000));
         }
         catch (InterruptedException e) {}

         ch = s.getSharedChar ();
         System.out.println (ch + " consumed by consumer.");
      }
      while (ch != 'Z');
   }
```

```
}
```

Cuando ejecuta ProdCons2, debería ver el siguiente resultado:

```
A produced by producer.
A consumed by consumer.
B produced by producer.
B consumed by consumer.
C produced by producer.
C consumed by consumer.
D produced by producer.
D consumed by consumer.
E produced by producer.
E consumed by consumer.
F produced by producer.
F consumed by consumer.
G produced by producer.
G consumed by consumer.
```

Los problemas desaparecieron. El productor siempre ejecuta ante el consumidor y nunca produce un artículo antes de que el consumidor tenga la oportunidad de consumirlo. Para producir esta salida, ProdCons2 utiliza el mecanismo wait/notify.

El mecanismo wait / notify aparece en la clase Shared. Específicamente, wait () y notify () aparecen en los métodos setSharedChar (char c) y getSharedChar () de Shared. Shared también introduce un campo de instancia de escritura, la variable de condición que funciona con wait () y notify () para coordinar la ejecución del productor y del consumidor. Aquí es cómo funciona esa coordinación, asumiendo que el consumidor ejecuta primero:

El consumidor ejecuta s.getSharedChar ().
Dentro de ese método sincronizado, el consumidor llama wait () (porque writable contiene true). El consumidor espera hasta recibir la notificación.
En algún momento, el productor llama a s.setSharedChar (ch) ;.
Cuando el productor introduce ese método sincronizado (posible porque el consumidor liberó el bloqueo dentro del método wait () justo antes de esperar), el productor descubre el valor de writeable como verdadero y no llama a wait ().
El productor guarda el carácter, establece la posibilidad de escribir en false (por lo que el productor debe esperar si el consumidor no ha consumido el carácter en el momento en que el productor invoca setSharedChar (char c)) y llama a notify () para despertar al consumidor Consumidor está esperando).
El productor sale de setSharedChar (char c).
El consumidor se despierta, establece la posibilidad de escribir en true (por lo que el consumidor debe esperar si el productor no ha producido un carácter en el momento en que el consumidor invoca getSharedChar ()), notifica al productor que despierte ese hilo (suponiendo que el productor esté esperando) Devuelve el carácter compartido.

Nota: Para escribir programas más confiables que utilicen wait / notify, piense en qué condiciones existen en su programa. Por ejemplo, ¿qué condiciones existen en ProdCons2? Aunque ProdCons2 contiene sólo una variable de condición, existen dos condiciones. La primera condición es que el productor espera que el consumidor consuma un carácter y el consumidor notifique al productor cuando consuma el carácter. La segunda condición representa que el consumidor espera que el productor produzca un carácter y el productor notifique al consumidor cuando produce el carácter.

La necesidad de sincronizar

¿Por qué necesitamos la sincronización? Para obtener una respuesta, considere este ejemplo: Escribe un programa Java que usa un par de subprocesos para simular el retiro/depósito de transacciones financieras. En ese programa, un hilo realiza depósitos mientras que el otro realiza retiros. Cada subproceso manipula un par de variables compartidas, variables de clase e instancia, que identifican el nombre y la cantidad de la transacción financiera. Para que una transacción financiera sea correcta, cada subproceso debe finalizar la asignación de valores a `name` y `amount` (e imprimir esos valores, para simular el almacenamiento de la transacción) antes de que el otro hilo comience a asignar valores a `name` y `amount`. Después de algo de trabajo, termina con un código fuente similar al listado 1:

Listing 1. NeedForSynchronizationDemo.java

```
// NeedForSynchronizationDemo.java
class NeedForSynchronizationDemo
{
   public static void main (String [] args)
   {
      FinTrans ft = new FinTrans();
      TransThread tt1 = new TransThread(ft, "Deposit Thread");
      TransThread tt2 = new TransThread(ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}

class FinTrans
```

```
{
   public static String transName;
   public static double amount;
}

class TransThread extends Thread
{
   private FinTrans ft;

   TransThread(FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }

   public void run ()
   {
      for (int i = 0; i < 100; i++)
      {
           if (getName ().equals ("Deposit Thread"))
           {
               // Start of deposit thread's critical code section
               ft.transName = "Deposit";
               try
               {
                  Thread.sleep ((int) (Math.random () * 1000));
               }
               catch (InterruptedException e)
               {
               }
               ft.amount = 2000.0;
               System.out.println (ft.transName + " " + ft.amount);
               // End of deposit thread's critical code section
           }
           else
           {
               // Start of withdrawal thread's critical code section
               ft.transName = "Withdrawal";
               try
               {
                  Thread.sleep ((int) (Math.random () * 1000));
               }
               catch (InterruptedException e)
               {
               }
               ft.amount = 250.0;
               System.out.println (ft.transName + " " + ft.amount);
               // End of withdrawal thread's critical code section
           }
      }
```

```
    }
}
```

El código fuente de NeedForSynchronizationDemo tiene dos secciones de código críticas: una accesible al subproceso de depósito y la otra accesible al subproceso de retiro. En la sección de código crítico del subproceso de depósito, ese subproceso asigna la referencia del objeto String de depósito a la variable compartida `transName` y asigna 2000.0 a la variable compartida `amount`. De manera similar, dentro de la sección de código crítico del subproceso de retiro, ese hilo asigna la referencia del objeto de cadena de retiro a `transName` y asigna 250.0 a `amount`. Después de las asignaciones de cada hilo, se imprimen los contenidos de esas variables. Cuando se ejecuta `NeedForSynchronizationDemo`, es posible que espere salida similar a una lista intercalada de líneas Retiro 250.0 y Depósito 2000.0. En su lugar, recibe una salida similar a la siguiente:

```
Withdrawal 250.0
Withdrawal 2000.0
Deposit 2000.0
Deposit 2000.0
Deposit 250.0
```

El programa definitivamente tiene un problema. El hilo de retiro no debe simular retiros de $ 2000 y el hilo de depósito no debe simular depósitos de $ 250. Cada subproceso produce salida incoherente. ¿Qué causa esas inconsistencias? Considera lo siguiente:

- En una máquina con un único procesador, los hilos comparten el procesador. Como resultado, un subproceso sólo se puede ejecutar durante un cierto período de tiempo. En ese momento, el sistema operativo JVM hace una pausa en la ejecución de este subproceso y permite que otro subproceso ejecute una manifestación de programación de subprocesos. En una máquina multiprocesador, dependiendo del número de subprocesos y procesadores, cada subproceso Puede tener su propio procesador.
- En un equipo de un solo procesador, el período de ejecución de un subproceso podría no durar el tiempo suficiente para que el subproceso termine de ejecutar su sección de código crítico antes de que otro subproceso comience a ejecutar su propia sección de código crítico. En una máquina multiprocesador, los subprocesos pueden ejecutar código simultáneamente en sus secciones de código crítico. Sin embargo, pueden entrar en sus secciones de código crítico en diferentes momentos.
- En máquinas con un solo procesador o multiprocesador, puede ocurrir el siguiente escenario: El subproceso A asigna un valor a la variable compartida X en su sección de código crítico y decide realizar una operación de entrada / salida que requiere 100 milisegundos. El hilo B entra en su sección de código crítico, asigna un valor diferente a X, realiza una operación de entrada / salida de 50 milisegundos y asigna valores a las variables compartidas Y y Z. La operación de entrada / salida del hilo A se completa y ese hilo asigna su propia Valores a Y y Z. Debido a que X contiene un

valor B-asignado, mientras que Y y Z contienen A-valores asignados, se porduce una inconsistencia resultados.

¿Cómo surge una inconsistencia en NeedForSynchronizationDemo? Supongamos que el hilo de depósito ejecuta `ft.transName = "Depósito";` Y luego llama a `Thread.sleep()`. En ese punto, el hilo de depósito entrega el control del procesador durante el período de tiempo que debe dormir, y el hilo de retiro se ejecuta. Supongamos que el hilo de depósito duerme durante 500 milisegundos (un valor seleccionado aleatoriamente, gracias a Math.random(), desde el rango inclusivo de 0 a 999 milisegundos). Durante el tiempo de suspensión del hilo de depósito, el hilo de extracción ejecuta `ft.transName = "Retiro";` duerme durante 50 milisegundos (el valor de reposo del hilo de extracción seleccionado aleatoriamente), despierta, `ejecuta ft.amount = 250.0;` y ejecuta `System.out.Println(ft.transName + "" + ft.amount);` - todo antes de que el hilo de depósito se despierte. Como resultado, el hilo de extracción imprime Retiro 250.0, que es correcto. Cuando el hilo de depósito se despierta, ejecuta ft.amount = 2000.0 ;, seguido por System.out.println (ft.transName + "" + ft.amount) ;. Esta vez, se imprime retiro 2000.0, que no es correcto. Aunque el hilo de depósito previamente ha asignado la cadena "Depósito" a transName, esa referencia desapareció posteriormente cuando el hilo de retiro asignó la cadena "Retirado" a esa variable compartida. Cuando se despertó el subproceso de depósito, no pudo restaurar la referencia correcta a transName, pero continuó su ejecución asignando 2000.0 a cantidad. Aunque ninguna variable tiene un valor no válido, los valores combinados de ambas variables representan una inconsistencia. En este caso, sus valores representan un intento de retirar, 2000.

Hace mucho tiempo, los científicos de la computación inventaron un término para describir los comportamientos combinados de múltiples hilos que conducen a inconsistencias. Ese término es condición de carrera: el acto de cada hilo que compite por completar su sección crítica de código antes de que otro hilo entre en esa misma sección de código crítico. Como demuestra NeedForSynchronizationDemo, los pedidos de ejecución de threads son impredecibles. No hay garantía de que un hilo pueda completar su sección de código crítico antes de que otro hilo entre en esa sección. Por lo tanto, tenemos una condición de carrera, que causa inconsistencias. Para evitar condiciones de carrera, cada subproceso debe completar su sección de código crítico antes de que otro subproceso entre en la misma sección de código crítico u otra sección de código crítico relacionada que manipule las mismas variables o recursos compartidos. Sin medios para serializar el acceso, es decir, permitir el acceso a un solo hilo a la vez, a una sección de código crítico no se pueden evitar condiciones de carrera o inconsistencias. Afortunadamente, Java proporciona una manera de serializar el acceso a los hilos: a través de su mecanismo de sincronización.

Nota: De los tipos de Java, sólo las variables de punto flotante de largo y doble precisión son propensas a inconsistencias. ¿Por qué? Una JVM de 32 bits normalmente accede a una variable entera de 64 bits o una variable de punto flotante de doble precisión de 64 bits en dos pasos adyacentes de 32 bits. Un hilo podría completar el primer paso y luego esperar mientras otro subproceso ejecuta ambos pasos. Entonces, el primer hilo podría despertar y

completar el segundo paso, produciendo una variable con un valor diferente del valor del primer o segundo hilo. Como resultado, si al menos un subproceso puede modificar una variable entera larga o una variable de coma flotante de doble precisión, todos los subprocesos que leen y / o modifican esa variable deben utilizar la sincronización para serializar el acceso a la variable.

El mecanismo de sincronización de Java

Java proporciona un mecanismo de sincronización para evitar que más de un hilo ejecute código en una o más secciones de código crítico en cualquier momento. Este mecanismo se basa en los conceptos de monitores y locks (bloqueos). Piense en un monitor como un envoltorio protector alrededor de una sección de código crítico y un lock (bloqueo) como una entidad de software que utiliza un monitor para evitar que varios hilos entren en el monitor. La idea es la siguiente: cuando un hilo desea entrar en una sección de código crítico vigilada por el monitor, ese hilo debe adquirir el bloqueo asociado con un objeto que se asocia con el monitor. (Cada objeto tiene su propio bloqueo.) Si algún otro hilo lo mantiene bloqueado, la JVM obliga al hilo solicitante a esperar en un área de espera asociada con el monitor / bloqueo. Cuando el hilo en el monitor libera el bloqueo, la JVM elimina el hilo en espera del área de espera del monitor y permite que dicho hilo adquiera el bloqueo y continúe con la sección de código crítico del monitor.

Para trabajar con monitores / bloqueos, la JVM proporciona las instrucciones `monitorenter` y `monitorexit`. Afortunadamente, no es necesario trabajar a un nivel tan bajo. En su lugar, puede utilizar la palabra clave `sincronized` de Java en el contexto de la sentencia sincronizada y los métodos sincronizados.

La sentencia sincronized

Algunas secciones de código crítico ocupan pequeñas porciones de código de los métodos que los contienen. Para proteger el acceso de varios hilos a estas secciones de código crítico, utilice la sentencia sincronized. Esta declaración tiene la siguiente sintaxis:

```
'sincronized' '(' iddeobjeto ')'
'{'
   // Sección de código crítico
‘}’
```

La sentencia sincronized comienza con la palabra clave `sincronized` y continúa con un iddeobjeto, que aparece entre un par de paréntesis. El iddeobjeto hace referencia a un objeto cuyo bloqueo se asocia con el monitor que representa la sentencia sincronizada. Finalmente, la sección de código crítico de las sentencias de Java aparece entre un par de llaves. ¿Cómo se interpreta la sentencia sincronized? Considere el siguiente fragmento de código:

```
sincronized ("objeto de sincronización")
{
   // Acceso a variables compartidas y otros recursos compartidos
}
```

Desde una perspectiva de código fuente, un subproceso intenta ingresar a la sección de código crítico que protege la sentencia sincronized. Internamente, la JVM comprueba si algún otro hilo contiene el bloqueo asociado con el objeto "objeto de sincronización". Si no hay otro hilo que contiene el bloqueo, la JVM da el bloqueo al hilo solicitante y permite que el hilo entre en la sección de código crítico entre las llaves. Sin embargo, si algún otro hilo tiene el bloqueo, la JVM obliga al hilo solicitante a esperar en una zona de espera privada hasta que el hilo actualmente dentro de la sección de código crítico termine de ejecutar la sentencia final y avance más allá del carácter de llave final.
Puede utilizar la sentencia sincronized para eliminar la condición de carrera de NeedForSynchronizationDemo. Para ver cómo, examine el Listado 2

Listing 2. SynchronizationDemo1.java

```
// SynchronizationDemo1.java
class SynchronizationDemo1
{
   public static void main (String [] args)
   {
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}

class FinTrans
{
   public static String transName;
   public static double amount;
}

class TransThread extends Thread
{
   private FinTrans ft;

   TransThread(FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }
```

```
   public void run()
   {
      for (int i = 0; i < 100; i++)
      {
           if (getName().equals("Deposit Thread"))
           {
               synchronized(ft)
               {
                  ft.transName = "Deposit";
                  try
                  {
                     Thread.sleep((int)(Math.random () * 1000));
                  }
                  catch (InterruptedException e)
                  {
                  }
                  ft.amount = 2000.0;
                  System.out.println(ft.transName + " " + ft.amount);
               }
           }
           else
           {
               synchronized(ft)
               {
                  ft.transName = "Withdrawal";
                  try
                  {
                     Thread.sleep((int) (Math.random () * 1000));
                  }
                  catch (InterruptedException e)
                  {
                  }
                  ft.amount = 250.0;
                  System.out.println(ft.transName + " " + ft.amount);
               }
           }
      }
   }
}
```

Observe cuidadosamente SynchronizationDemo1; El método run() contiene dos secciones de código crítico intercaladas entre sincronized(ft) {y}. Cada uno de los subprocesos de depósito y retiro debe adquirir el bloqueo que se asocia con el objeto FinTrans que referencia ft antes de que cualquiera de los hilos pueda ingresar a su sección de código crítico. Si, por ejemplo, el hilo de depósito está en su sección de código crítico y el hilo de extracción desea entrar en su propia sección de código crítico, el hilo de extracción intenta adquirir el bloqueo. Debido a que el hilo de depósito sostiene que se bloquea mientras se ejecuta dentro de su sección de código crítico, la JVM obliga al hilo de extracción a esperar

hasta que el hilo de depósito ejecute esa sección de código crítico y libere el bloqueo. (Cuando la ejecución abandona la sección de código crítico, el bloqueo se libera automáticamente.)

> Sugerencia: Cuando necesite determinar si un subproceso contiene el bloqueo asociado de un objeto determinado, llame al método holdsLock(Object o) booleano estático de Thread. Este método devuelve un valor verdadero booleano si el subproceso que llama a ese método contiene el bloqueo asociado al objeto al que o hace referencia; De lo contrario, devuelve falso. Por ejemplo, si va a colocar System.out.println (Thread.holdsLock(ft)); Al final del método main() de SynchronizationDemo1, holdsLock () devolvería false. Esto porque el subproceso principal que ejecuta el método main() no utiliza el mecanismo de sincronización para adquirir ningún bloqueo. Sin embargo, si fuera a colocar System.out.println (Thread.holdsLock (ft)); En cualquiera de las sentencias sincronizadas (ft) de run(), holdsLock() devolvería true porque el thread de depósito o el thread de retiro tenían que adquirir el bloqueo asociado al objeto FinTrans al que ft hace referencia antes de que el hilo pudiera entrar en su código crítico.

Métodos sincronizados

Puede utilizar sentencias sincronizadas a lo largo del código fuente de su programa. Sin embargo, usted podría encontrarse con situaciones en las que el uso excesivo de tales declaraciones conduce a un código ineficiente. Por ejemplo, supongamos que su programa contiene un método con dos sentencias sincronizadas sucesivas que intentan adquirir el bloqueo asociado del mismo objeto común. Dado que la adquisición y la liberación del bloqueo del objeto consume tiempo, las repetidas llamadas (en un bucle) a ese método pueden degradar el rendimiento del programa. Cada vez que se hace una llamada a ese método, debe adquirir y liberar dos bloqueos. Cuanto mayor sea el número de adquisiciones y liberaciones de bloqueo, más tiempo pasará el programa para adquirir y liberar los bloqueos. Para evitar este problema, puede considerar el uso de un método sincronizado.

Un método sincronizado es un método de instancia o clase cuyo encabezado incluye la palabra clave `sincronized`. Por ejemplo: `sincronized void print(String s)`. Al sincronizar un método de instancia entero, un hilo debe adquirir el bloqueo asociado al objeto en el que se produce la llamada al método. Por ejemplo, dado un ft.update("Depósito", 2000.0); y suponiendo que update() está sincronizada, un subproceso debe adquirir el bloqueo asociado al objeto que referencia ft. Para ver una versión de método sincronizado del código fuente SynchronizationDemo1, consulte el Listado 3:

Listing 3. SynchronizationDemo2.java

```
// SynchronizationDemo2.java
class SynchronizationDemo2
{
   public static void main (String [] args)
```

```
   {
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}

class FinTrans
{
   private String transName;
   private double amount;

   synchronized void update(String transName, double amount)
   {
      this.transName = transName;
      this.amount = amount;
      System.out.println (this.transName + " " + this.amount);
   }
}

class TransThread extends Thread
{
   private FinTrans ft;
   TransThread (FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }
   public void run ()
   {
      for (int i = 0; i < 100; i++)
          if (getName ().equals ("Deposit Thread"))
             ft.update ("Deposit", 2000.0);
          else
             ft.update ("Withdrawal", 250.0);
   }
}
```

Aunque ligeramente más compacto que el Listado 2, el Listado 3 logra el mismo propósito. Si el subproceso de depósito llama al método update(), la JVM comprueba si el hilo de retiro ha adquirido el bloqueo asociado con el objeto al que hace referencia. Si es así, el hilo de depósito espera. De lo contrario, ese hilo entra en la sección de código crítico.

SynchronizationDemo2 muestra un método de instancia sincronizo. Sin embargo, también puede sincronizar métodos de clase. Por ejemplo, la clase `java.util.Calendar` declara un método `public static sincronized Locale[] getAvailableLocales()`. Dado que los métodos de clase no tienen concepto de una referencia, ¿de dónde obtiene el

método de clase su bloqueo? Los métodos de clase adquieren sus bloqueos de los objetos de clase: cada clase cargada se asocia con un objeto de clase, de la cual los métodos de clase de la clase cargada obtienen sus bloqueos. Me refiero a tales bloqueos como bloqueos de clase.

> Precaución: No sincronice el método run() de un objeto thread debido a situaciones en las que varios threads necesitan ejecutar run(). Debido a que esos subprocesos intentan sincronizarse en el mismo objeto, sólo un subproceso a la vez puede ejecutar run (). Como resultado, cada subproceso debe esperar a que el subproceso anterior termine antes de que pueda acceder a run().

Algunos programas mezclan métodos sincronizados de instancia y métodos de clase sincronizados. Para ayudarle a entender lo que sucede en los programas donde los métodos de clase sincronizados llaman a métodos de instancia sincronizados y viceversa (a través de referencias a objetos), tenga en cuenta los dos puntos siguientes:

- Los bloqueos de objetos y los bloqueos de clase no se relacionan entre sí. Son entidades diferentes. Usted adquiere y libera cada bloqueo independientemente. Un método de instancia sincronizado que llama a un método de clase sincronizado adquiere ambos bloqueos. En primer lugar, el método de instancia sincronizado adquiere el bloqueo de objetos de su objeto. En segundo lugar, ese método adquiere el bloqueo de clase del método de clase sincronizada.
- Los métodos de clase sincronizada pueden llamar a los métodos sincronizados de un objeto o utilizar el objeto para bloquear un bloque sincronizado. En ese escenario, un subproceso adquiere inicialmente el bloqueo de clase del método de clase sincronizada y posteriormente adquiere el bloqueo de objeto del objeto. Por lo tanto, un método de clase sincronizada que llama a un método de instancia sincronizada también adquiere dos bloqueos.

El siguiente fragmento de código ilustra el segundo punto:

```
class LockTypes
{
   // Object lock acquired just before
   // execution passes into instanceMethod()
   synchronized void instanceMethod()
   {
      // Object lock released as thread exits instanceMethod()
   }

   // Class lock acquired just before
   // execution passes into classMethod()
   synchronized static void classMethod(LockTypes lt)
   {
      lt.instanceMethod ();
      // Object lock acquired just before
```

```
      // critical code section executes

      synchronized (lt)
      {
         // Critical code section
         // Object lock released as
         // thread exits critical code section
      }
      // Class lock released as thread exits classMethod()
   }
}
```

El fragmento de código demuestra que el método de clase sincronizado classMethod() llama al método de instancia sincronizado instanceMethod(). Al leer los comentarios, verá que classMethod () adquiere primero su bloqueo de clase y luego adquiere el bloqueo de objeto asociado con el objeto LockTypes que lt hace referencia.

Dos problemas con el mecanismo de sincronización

A pesar de su simplicidad, los desarrolladores a menudo usan mal el mecanismo de sincronización de Java, lo que provoca problemas que van desde no sincronización a interbloqueo. Esta sección examina estos problemas y proporciona un par de recomendaciones para evitarlos.

> Nota: Un tercer problema relacionado con el mecanismo de sincronización es el costo de tiempo asociado con la adquisición y liberación del bloqueo. En otras palabras, toma tiempo para que un hilo adquiera o libere un bloqueo. Al adquirir / liberar un bloqueo en un bucle, los costos de tiempo individuales se suman, lo que puede degradar el rendimiento. Para JVMs más viejas, el costo de tiempo de adquisición de bloqueo a menudo resulta en penalizaciones significativas de rendimiento. Afortunadamente, la JVM HotSpot de Sun Microsystems (que se entrega con el SDK de Java 2 de Sun, Standard Edition (J2SE) SDK) ofrece una rápida adquisición y liberación de bloqueos, reduciendo en gran medida el impacto de este problema.

Sin sincronización

Después de que un hilo de forma voluntaria o involuntaria (a través de una excepción) sale de una sección de código crítico, libera un bloqueo para que otro hilo pueda obtener la entrada. Supongamos que dos subprocesos quieren entrar en la misma sección de código crítico. Para evitar que ambos subprocesos entren simultáneamente en esa sección de código crítico, cada subproceso debe intentar adquirir el mismo bloqueo. Si cada hilo intenta adquirir un bloqueo diferente y tiene éxito, ambos hilos entran en la sección de código crítico; Ningún hilo tiene que esperar a l otro hilo para liberar su bloqueo porque el otro hilo

adquiere un bloqueo diferente. El resultado final: No hay sincronización, como se demuestra en el Listado 4:

Listing 4. NoSynchronizationDemo.java

```
// NoSynchronizationDemo.java
class NoSynchronizationDemo
{
   public static void main (String [] args)
   {
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}
class FinTrans
{
   public static String transName;
   public static double amount;
}
class TransThread extends Thread
{
   private FinTrans ft;
   TransThread (FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }
   public void run ()
   {
      for (int i = 0; i < 100; i++)
      {
           if (getName ().equals ("Deposit Thread"))
           {
               synchronized (this)
               {
                  ft.transName = "Deposit";
                  try
                  {
                     Thread.sleep ((int) (Math.random () * 1000));
                  }
                  catch (InterruptedException e)
                  {
                  }
                  ft.amount = 2000.0;
                  System.out.println (ft.transName + " " + ft.amount);
```

```
            }
        }
        else
        {
            synchronized (this)
            {
               ft.transName = "Withdrawal";
               try
               {
                  Thread.sleep ((int) (Math.random () * 1000));
               }
               catch (InterruptedException e)
               {
               }
               ft.amount = 250.0;
               System.out.println (ft.transName + " " + ft.amount);
            }
        }
    }
  }
}
```

Cuando ejecuta NoSynchronizationDemo, verá una salida que se asemeja al siguiente extracto:

```
Withdrawal 250.0
Withdrawal 2000.0
Deposit 250.0
Withdrawal 2000.0
Deposit 2000.0
```

A pesar del uso de sentencias sincronizadas, no se produce ninguna sincronización. ¿Por qué? Examine `sincronized(this)`. Dado que la palabra clave se refiere al objeto actual, el hilo de depósito intenta adquirir el bloqueo asociado al objeto TransThread cuya referencia asigna inicialmente a tt1 (en el método main()). De manera similar, el hilo de extracción intenta adquirir el bloqueo asociado con el objeto TransThread cuya referencia asigna inicialmente a tt2. Tenemos dos objetos TransThread diferentes y cada hilo intenta adquirir el bloqueo asociado con su objeto TransThread respectivo antes de entrar en su propia sección de código crítico. Debido a que los subprocesos adquieren bloqueos diferentes, ambos hilos pueden estar en sus propias secciones de código crítico al mismo tiempo. El resultado es que no hay sincronización.

> Sugerencia: Para evitar un escenario sin sincronización, elija un objeto común a todos los subprocesos relevantes. De esta manera, esos hilos competirán para obtener el bloqueo del mismo objeto y sólo un hilo a la vez puede entrar en la sección de código crítico asociada.

Punto muerto (Abrazo Mortal, Dead Lock)

En algunos programas, puede producirse el siguiente escenario: El subproceso A adquiere un bloqueo que el subproceso B necesita antes de que el subproceso B pueda introducir la sección de código crítico de B. Del mismo modo, el hilo B adquiere un bloqueo que el hilo A necesita antes de que el hilo A pueda entrar en la sección de código crítico de A. Debido a que ninguno de los hilos tiene el bloqueo que necesita, cada hilo debe esperar para adquirir su bloqueo. Además, debido a que no puede continuar el hilo, ningún hilo puede liberar el bloqueo del otro hilo, y la ejecución del programa se bloquea. Este comportamiento se conoce como abrazo moral, que el Listado 5 demuestra:

Listing 5. DeadlockDemo.java

```
// DeadlockDemo.java
class DeadlockDemo
{
   public static void main (String [] args)
   {
      FinTrans ft = new FinTrans ();
      TransThread tt1 = new TransThread (ft, "Deposit Thread");
      TransThread tt2 = new TransThread (ft, "Withdrawal Thread");
      tt1.start ();
      tt2.start ();
   }
}
class FinTrans
{
   public static String transName;
   public static double amount;
}
class TransThread extends Thread
{
   private FinTrans ft;
   private static String anotherSharedLock = "";
   TransThread (FinTrans ft, String name)
   {
      super (name); // Save thread's name
      this.ft = ft; // Save reference to financial transaction object
   }
   public void run ()
   {
      for (int i = 0; i < 100; i++)
      {
           if (getName ().equals ("Deposit Thread"))
           {
               synchronized (ft)
               {
```

```
                synchronized (anotherSharedLock)
                {
                   ft.transName = "Deposit";
                   try
                   {
                      Thread.sleep ((int) (Math.random () * 1000));
                   }
                   catch (InterruptedException e)
                   {
                   }
                   ft.amount = 2000.0;
                   System.out.println (ft.transName + " " + ft.amount);
                }
             }
          }
          else
          {
             synchronized (anotherSharedLock)
             {
                synchronized (ft)
                {
                   ft.transName = "Withdrawal";
                   try
                   {
                      Thread.sleep ((int) (Math.random () * 1000));
                   }
                   catch (InterruptedException e)
                   {
                   }
                   ft.amount = 250.0;
                   System.out.println (ft.transName + " " + ft.amount);
                }
             }
          }
       }
    }
}
```

Si ejecuta DeadlockDemo, probablemente verá sólo una sola línea de salida antes de que la aplicación se bloquee. Para descongelar DeadlockDemo, presione Ctrl-C (asumiendo que está utilizando el kit de herramientas SDK 1.4 de Sun en un indicador de comandos de Windows).

¿Qué causa el estancamiento? Mire atentamente el código fuente; El hilo de depósito debe adquirir dos bloqueos antes de que pueda entrar en su sección de código crítico más interna. El bloqueo externo se asocia con el objeto FinTrans que referencia ft y el bloqueo interno se asocia con el objeto String que otroSharedLock hace referencia. De manera similar, el hilo de extracción debe adquirir dos bloqueos antes de que pueda entrar en su propia sección de código crítico más interior. El bloqueo externo se asocia con el objeto

String que otroSharedLock hace referencia, y el bloqueo interno se asocia con el objeto FinTrans que referencia ft. Supongamos que los órdenes de ejecución de ambos hilos son tales que cada hilo adquiere su bloqueo externo. Así, el hilo de depósito adquiere su bloqueo FinTrans, y el hilo de extracción adquiere su bloqueo de cadena. Ahora que ambos hilos poseen sus cerraduras exteriores, están en su sección de código crítico exterior apropiada. Ambos hilos intentan entonces adquirir las cerraduras internas, para que puedan entrar en las secciones de código crítico interno apropiado.

El hilo de depósito intenta adquirir el bloqueo asociado con el objeto otherSharedLock - referenciado. Sin embargo, el hilo de depósito debe esperar porque el hilo de extracción tiene ese bloqueo. De manera similar, el hilo de extracción intenta adquirir el bloqueo asociado con el objeto referenciado por ft. Pero el hilo de retiro no puede adquirir ese bloqueo porque el hilo de depósito (que está esperando) lo mantiene. Por lo tanto, el hilo de extracción también debe esperar. Ningún hilo puede proceder porque ninguno de los hilos suelta el bloqueo que contiene. Y ni el hilo puede liberar el bloqueo que tiene porque cada hilo está esperando. Cada hilo se bloquea y el programa se congela.

> Sugerencia: Para evitar el bloqueo, analice cuidadosamente su código fuente para situaciones en las que los subprocesos podrían intentar adquirir los bloqueos de otros, como cuando un método sincronizado llama a otro método sincronizado. Debe hacerlo porque una JVM no puede detectar o impedir el bloqueo.

Revisión

Para lograr un rendimiento sólido con subprocesos, encontrará situaciones en las que sus programas multiproceso necesitan serializar el acceso a secciones de código crítico. Conocida como sincronización, esta actividad evita inconsistencias que resultan en un extraño comportamiento del programa. Puede utilizar las sentencias sincronizadas para proteger partes de un método o sincronizar todo el método. Sin embargo, peine cuidadosamente su código para ver si hay fallos que pueden resultar en sincronización o bloqueos fallidos.

Bibliografía

1. Java 101: Comprensión de los subprocesos Java, parte 2: sincronización de subprocesos
   http://www.javaworld.com/article/2074318/java-concurrency/java-101--understanding-java-threads--part-2--thread-synchronization.html?page=3