HILOS - Sincronización

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

Los programas multiproceso suelen funcionar erráticamente o producir valores erróneos debido a la falta de sincronización de los subprocesos. La sincronización es el acto de serializar (o ordenar uno a la vez) el acceso de los hilos a aquellas secuencias de código que permiten que varios hilos manipulen las variables de clase e instancia y otros recursos compartidos.

Comenzamos con un ejemplo que ilustra el por qué algunos programas multiproceso deben utilizar la sincronización. A continuación exploramos el mecanismo de sincronización de Java en términos de monitores y bloqueos, y la palabra clave `sincronized`. Debido a que el uso incorrecto del mecanismo de sincronización niega sus beneficios, concluimos investigando dos problemas que resultan de tal uso indebido.

> Sugerencia: A diferencia de las variables de clase e instancia, los subprocesos no pueden compartir variables y parámetros locales. La razón: Las variables y parámetros locales se asignan a la pila de llamadas de método de un hilo. Como resultado, cada subproceso recibe su propia copia de esas variables. Por el contrario, los subprocesos pueden compartir campos de clase y campos de instancia porque esas variables no se asignan en la pila de método de llamada de un subproceso. En su lugar, se asignan en la memoria compartida de montón (heap), como parte de las clases (atributos de clase) u objetos (atributos de instancia).

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