Aplicaciones para Ambientes Distribuidos

Laboratorio # 4 – Prof. Regis Rivera

Objetivo: Crear aplicaciones aplicando sincronización para concurrencia
Nota: Los nombres de archivos de cada aplicación están dados en el icono 

Marco de bloqueo frente a sincronización de subprocesos en Java

El mecanismo de sincronización de subprocesos se puede lograr utilizando el marco Lock, que está presente en el paquete java.util.concurrent. El marco de bloqueo funciona como bloques sincronizados, excepto que los bloqueos pueden ser más sofisticados que los bloques sincronizados de Java. Los bloqueos permiten una estructuración más flexible del código sincronizado. Este nuevo enfoque se introdujo en Java 5 para abordar el problema de sincronización que se menciona a continuación.

Veamos una clase Vector, que tiene muchos métodos sincronizados. Cuando hay 100 métodos sincronizados en una clase, solo se puede ejecutar un subproceso de estos 100 métodos en un momento dado. Solo un hilo puede acceder a un solo método en un momento dado utilizando un bloque sincronizado. Esta es una operación muy costosa. Las cerraduras evitan esto al permitir la configuración de varias cerraduras para diferentes propósitos. Se pueden tener un par de métodos sincronizados bajo un bloqueo y otros métodos bajo un bloqueo diferente. Esto permite una mayor concurrencia y también aumenta el rendimiento general.

```
Bloquear bloqueo = nuevo ReentrantLock();
lock.lock();

// Sección crítica
bloqueo y desbloqueo();
```

Un bloqueo se adquiere mediante el método lock() y se libera mediante el método unlock(). Invocar un unlock() sin lock() generará una excepción. Como ya se mencionó, la interfaz Lock está presente en el paquete java.util.concurrent.locks y ReentrantLock implementa la interfaz Lock.

Nota: El número de llamadas a lock() siempre debe ser igual al número de llamadas a unlock().

En el siguiente código, el usuario ha creado un recurso llamado "TestResource" que tiene dos métodos y dos bloqueos diferentes para cada uno respectivamente. Hay dos trabajos llamados "DisplayJob" y "ReadJob". La clase LockTest crea 5 subprocesos para realizar 'DisplayJob' y 5 subprocesos para realizar 'ReadJob'. Los 10 subprocesos comparten un único recurso "TestResource".

Laboratorio 4.1: Compartir un único recurso en una concurrencia

Lab41.java

```java
import java.util.Date;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockTest
{
    public static void main(String[] args)
    {
        TestResource test = new TestResource();
        Thread thread[] = new Thread[10];
        for (int i = 0; i < 5; i++)
        {
            thread[i] = new Thread(new DisplayJob(test),
            "Hilo " + i);
        }
        for (int i = 5; i < 10; i++)
        {
            thread[i] = new Thread(new ReadJob(test),
            "Hilo " + i);
        }
        for (int i = 0; i < 10; i++)
        {
            thread[i].start();
        }
    }
}

class DisplayJob implements Runnable
{

    private TestResource test;
    DisplayJob(TestResource tr)
    {
        test = tr;
    }
    @Override
    public void run()
    {
        System.out.println("mostrar Job");
        test.displayRecord(new Object());
    }
}

class ReadJob implements Runnable
{

    private TestResource test;

    ReadJob(TestResource tr)
    {
        test = tr;
    }
    @Override
    public void run()
    {
        System.out.println("Job leido");
        test.readRecord(new Object());
    }
}

class TestResource
{

    private final Lock
    displayQueueLock = new ReentrantLock();
    private final Lock
    readQueueLock = new ReentrantLock();

    public void displayRecord(Object document)
    {
        final Lock displayLock = this.displayQueueLock;
        displayLock.lock();
```

```
73          try
74          {
75              Long duration =
76                      (long) (Math.random() * 10000);
77              System.out.println(Thread.currentThread().
78              getName() + ": Probando Resource: mostrando un Job"+
79              " durante " + (duration / 1000) + " segundos ::"+
80              " Fecha - " + new Date());
81              Thread.sleep(duration);
82          }
83          catch (InterruptedException e)
84          {
85              e.printStackTrace();
86          }
87          finally
88          {
89              System.out.printf("%s: El documento ha sido"+
90              " displayed\n", Thread.currentThread().getName());
91              displayLock.unlock();
92          }
93      }
94
95      public void readRecord(Object document)
96      {
97          final Lock readQueueLock = this.readQueueLock;
98          readQueueLock.lock();
99          try
100         {
101             Long duration =
102                     (long) (Math.random() * 10000);
103             System.out.println
104             (Thread.currentThread().getName()
105             + ": Probando Resource: leyendo un Job durante " +
106             (duration / 1000) + " segundos :: Fecha - " +
107             new Date());
108             Thread.sleep(duration);
109         }
110         catch (InterruptedException e)
111         {
112             e.printStackTrace();
113         }
114         finally
115         {
116             System.out.printf("%s: El documento ha "+
117             " sido leido\n", Thread.currentThread().getName());
118             readQueueLock.unlock();
119         }
120     }
```

En el ejemplo anterior,
Comente lo siguiente:

- ¿Es o no es necesario que DisplayJob espere a que los subprocesos de ReadJob completen la tarea?
- ¿ReadJob y Display job utilizan dos bloqueos diferentes?
- ¿Esto no puede ser posible con "sincronizado"?

Esto como respuesta con sus comentarios respecto al laboratorio 4.1

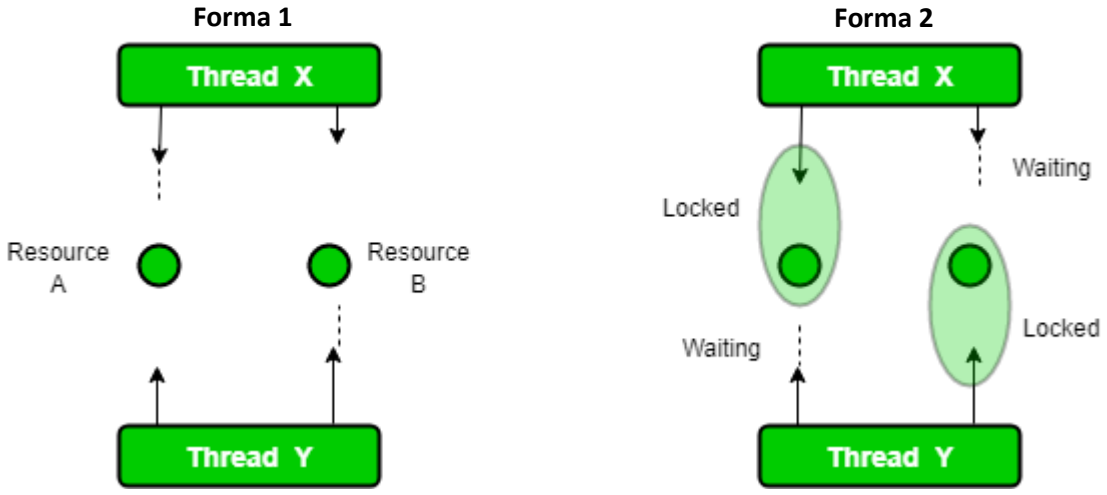
Con respecto a los bloqueos respecto a la sincronización, las diferencias son las siguientes:

Parámetros	Marco de bloqueo	sincronizado
A través de métodos	Sí, los bloqueos se pueden implementar en todos los métodos, puede invocar lock() en el método1 e invocar unlock() en el método2.	Imposible
intenta adquirir el bloqueo	sí, el método trylock(timeout) es compatible con el marco Lock, que bloqueará el recurso si está disponible; de lo contrario, devuelve falso y Thread no se bloqueará.	No es posible con sincronizado
Gestión de bloqueo justo	Sí, la gestión de bloqueo justo está disponible en el caso de un marco de bloqueo. Entrega el candado al hilo que lleva mucho tiempo esperando. Incluso en el modo de equidad establecido en verdadero, si trylock está codificado, se entrega primero.	No es posible con sincronizado
Lista de hilos en espera	Sí, la lista de subprocesos en espera se puede ver utilizando el marco Lock.	No es posible con sincronizado
Liberación de bloqueo en excepciones	`Bloquear.lock(); miMétodo();Lock.unlock();` unlock() no se puede ejecutar en este código si se lanza alguna excepción desde myMethod().	Sincronizado funciona claramente en este caso. Libera el bloqueo

Punto muerto en Java Multithreading

La palabra clave sincronized se usa para hacer que la clase o método sea seguro para subprocesos, lo que significa que solo un subproceso puede tener el bloqueo del método sincronizado y usarlo, otros subprocesos tienen que esperar hasta que se libere el bloqueo y cualquiera de ellos adquiera ese bloqueo.

Es importante usarlo si nuestro programa se ejecuta en un entorno de subprocesos múltiples donde se ejecutan dos o más subprocesos simultáneamente. Pero a veces también causa un problema que se llama Deadlock . A continuación, se muestra un ejemplo sencillo de la condición de interbloqueo.



Laboratorio 4.2: Prueba de DeckLock

Lab42.java

```java
public class Deadlock
{
    public static void main(String[] args)
    {

        Shared s1 = new Shared();
        Shared s2 = new Shared();

        Thread1 t1 = new Thread1(s1, s2);
        t1.setName("Thread1");
        t1.start();

        Thread2 t2 = new Thread2(s1, s2);
        t2.setName("Thread2");
        t2.start();
        Util.sleep(2000);
    }
}

class Util
{
    static void sleep(long millis)
    {
        try
        {
            Thread.sleep(millis);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}

class Shared
{
    synchronized void test1(Shared s2)
    {
        System.out.println(Thread.currentThread().getName() + " ingresa a test1 de " + this);
        Util.sleep(1000);

        s2.test2();
        System.out.println(Thread.currentThread().getName() + " sale test1 de " + this);
    }
    synchronized void test2()
    {
        System.out.println(Thread.currentThread().getName() + " ingresa a test2 of " + this);
        Util.sleep(1000);

        System.out.println(Thread.currentThread().getName() + " sale test2 de " + this);
    }
}
class Thread1 extends Thread
{
    private Shared s1;
    private Shared s2;

    public Thread1(Shared s1, Shared s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    @Override
    public void run()
    {
        s1.test1(s2);
    }
}
class Thread2 extends Thread
{
    private Shared s1;
    private Shared s2;

    public Thread2(Shared s1, Shared s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    @Override
    public void run()
    {
```

```
84              s2.test1(s1);
85          }
86      }
```

No se recomienda ejecutar el programa anterior con IDE en línea. Podemos copiar el código fuente y ejecutarlo en nuestra máquina local. Podemos ver que se ejecuta por tiempo indefinido, porque los subprocesos están en condición de punto muerto y no permiten que el código se ejecute. Ahora veamos paso a paso qué está pasando allí.

1. El subproceso t1 comienza adquiriendo un bloqueo en s1 e ingresa al método test1() de s1.
2. Mientras tanto, el subproceso t2 comienza adquiriendo un bloqueo en s2 e ingresa al método test1() de s2.
3. Dentro del método test1(), ambos subprocesos intentan adquirir bloqueos en los objetos de cada uno, pero los bloqueos ya están retenidos por otro subproceso, lo que hace que ambos subprocesos esperen indefinidamente a que el otro libere el bloqueo.
4. Por lo tanto, ni los métodos test1() ni test2() completan la ejecución y el programa permanece bloqueado en el estado de punto muerto.

Detectar condición de bloqueo muerto

También podemos detectar un punto muerto ejecutando este programa en cmd. Tenemos que recolectar Thread Dump. El comando para recopilar depende del tipo de sistema operativo. Si usamos Windows y Java 8, el comando es jcmd $PID Thread.print

Podemos obtener PID ejecutando el comando jps. El volcado de subprocesos para el programa anterior se encuentra a continuación:

```
jcmd 18692 Thread.print
18692:
2020-06-08 19:03:10
Volcado completo de subprocesos OpenJDK 64-Bit Server VM (11.0.4+10-b304.69 modo mixto, uso compartido):

clase de subprocesos Información SMR:
_java_thread_list=0x0000017f44b69f20 , length=13, elements={
0x0000017f43f77000, 0x0000017f43f79800, 0x0000017f43f90000, 0x0000017f43f91000,
0x0000017f43f95000, 0x0000017f43fa5000, 0x0000017f43fb0800, 0x0000017f43f5b800,
0x0000017f44bc9000, 0x0000017f44afb000, 0x0000017f44bd7800, 0x0000017f44bd8800,
0x0000017f298c9000
}

"Reference Handler" #2 daemon prio=10 os_prio=2 cpu=0.00 ms transcurrido=57.48s tid=0x0000017f43f77000
nid=0x6050 esperando en condición [0x0000005f800ff000]
   java.lang.Thread.State: RUNNABLE
        en java.lang.ref.Reference.waitForReferencePendingList(java.base@11.0.4/Método nativo)
        en java .lang.ref.Reference.processPendingReferences(java.base@11.0.4/Reference.java:241)
        en java.lang.ref.Reference$ReferenceHandler.run(java.base@11.0.4/Reference.java:213)
```

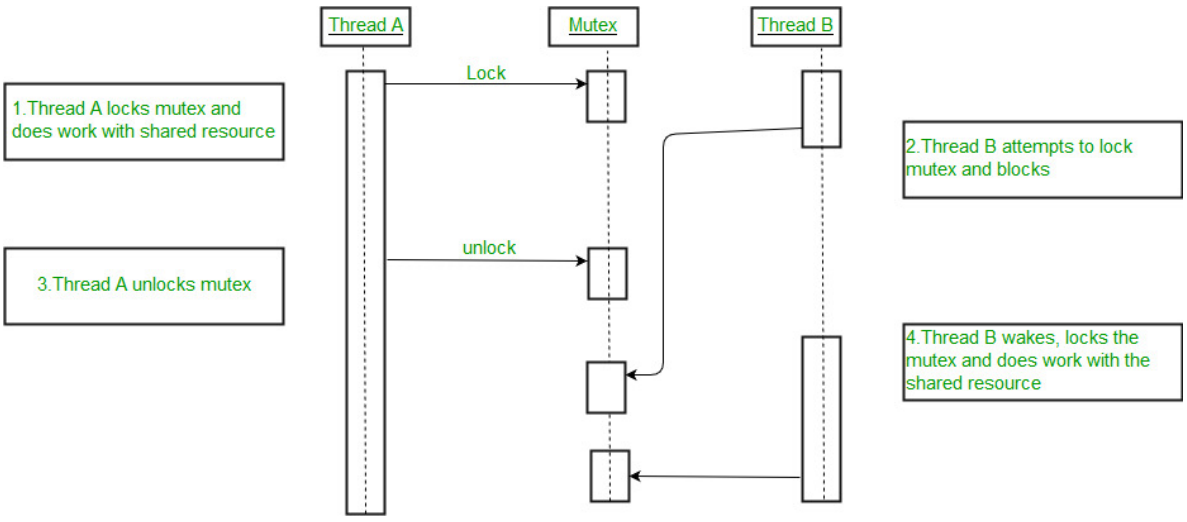
Evitar la condición de bloqueo muerto

Podemos evitar la condición de punto muerto si conocemos sus posibilidades. Es un proceso muy complejo y no fácil de detectar. Pero aún así, si lo intentamos, podemos evitarlo. Existen algunos métodos mediante los cuales podemos evitar esta condición. No podemos eliminar por completo su posibilidad, pero podemos reducirla.

- Evite los bloqueos anidados: esta es la razón principal del bloqueo mutuo. Dead Lock ocurre principalmente cuando damos bloqueos a múltiples subprocesos. Evite dar bloqueo a varios hilos si ya se lo hemos dado a uno.
- Evite bloqueos innecesarios: deberíamos bloquear solo aquellos miembros que sean necesarios. Tener un bloqueo innecesario puede provocar un bloqueo mutuo.
- Uso de unión de subprocesos: la condición de bloqueo muerto aparece cuando un subproceso está esperando que otro termine. Si se produce esta condición, podemos usar Thread.join con el tiempo máximo que creamos que llevará la ejecución.

Diferencia entre bloqueo y monitor en concurrencia de Java

Java Concurrency básicamente trata conceptos como subprocesos múltiples y otras operaciones concurrentes. Esto se hace para garantizar una utilización máxima y eficiente del rendimiento de la CPU, reduciendo así su tiempo de inactividad en general. Han existido bloqueos para implementar subprocesos múltiples mucho antes de que se empezaran a utilizar los monitores. En aquel entonces, los bloqueos (o mutex) eran partes de subprocesos dentro del programa que funcionaban en mecanismos de bandera para sincronizarse con otros subprocesos. Siempre han estado trabajando como una herramienta para proporcionar control de acceso sincrónico sobre recursos y objetos compartidos. Con mayores avances, el uso de monitores comenzó como un mecanismo para manejar el acceso y coordinar subprocesos que demostraron ser más eficientes, libres de errores y compatibles en programas orientados a objetos. Antes de continuar para encontrar las diferencias entre los dos, echemos un vistazo más de cerca a cada uno de ellos.



Descripción general de Lock (o Mutex)

Lock originalmente se usaba en la sección lógica de los subprocesos que se usaban para proporcionar control de acceso sincronizado entre los subprocesos. Los subprocesos verificaron la

disponibilidad del control de acceso sobre objetos compartidos a través de indicadores adjuntos al objeto que indicaban si el recurso compartido está libre (desbloqueado) u ocupado (bloqueado). Ahora la API de concurrencia brinda soporte para el uso de bloqueos explícitamente usando la interfaz de bloqueo en Java. El método explícito tiene un mecanismo de control más preciso en comparación con la implementación implícita de bloqueos mediante monitores. Antes de pasar a hablar de los monitores, veamos una ilustración que demuestra el funcionamiento de las cerraduras básicas.

Monitorear – Descripción general

Monitor en Java Concurrency es un mecanismo de sincronización que proporciona los requisitos fundamentales del subproceso múltiple, es decir, la exclusión mutua entre varios subprocesos y la cooperación entre subprocesos que trabajan en tareas comunes. Los monitores básicamente 'monitorean' el control de acceso de recursos y objetos compartidos entre subprocesos. Al utilizar esta construcción, solo un subproceso a la vez obtiene control de acceso sobre la sección crítica del recurso, mientras que otros subprocesos se bloquean y se hacen esperar hasta que se cumplan ciertas condiciones. En Java, los monitores se implementan utilizando la palabra clave sincronizada (bloques sincronizados, métodos o clases sincronizados). Por ejemplo, veamos cómo se sincronizan dos subprocesos t1 y t2 para usar un objeto de impresora de datos compartido.

Laboratorio 4.3: Monitoreo de Java Concurrency

Lab43.java

```java
import java.io.*;

class GFG {

    public static void main(String[] args)
    {
        SharedDataPrinter printer = new SharedDataPrinter();

        Thread1 t1 = new Thread1(printer);
        Thread2 t2 = new Thread2(printer);

        t1.start();
        t2.start();
    }
}

class SharedDataPrinter {

    synchronized public void display(String str)
    {

        for (int i = 0; i < str.length(); i++) {
            System.out.print(str.charAt(i));

            try {
                Thread.sleep(100);
            }
            catch (Exception e) {
            }
        }
    }
}

class Thread1 extends Thread {

    SharedDataPrinter p;

    public Thread1(SharedDataPrinter p)
    {

```

```
41          this.p = p;
42      }
43
44      public void run()
45      {
46
47          p.display("Aplicaciones para");
48      }
49  }
50
51  class Thread2 extends Thread {
52
53      SharedDataPrinter p;
54
55      public Thread2(SharedDataPrinter p) { this.p = p; }
56
57      public void run()
58      {
59
60          p.display(" ambientes distribuidos");
61      }
62  }
```

Analicemos las principales diferencias entre Lock y Monitor en concurrencia en Java , que se muestra gráficamente en el siguiente cuadro:

Bloqueo (Mutex)	Monitor
Se han utilizado desde la acuñación de los conceptos de subprocesos múltiples.	Nació con desarrollos posteriores en el campo.
Generalmente en forma de campo de datos o bandera que ayuda a implementar la coordinación.	La sincronicidad se implementa a través de un mecanismo de construcción.
La sección crítica (las funciones de bloqueo/desbloqueo y otras operaciones en el objeto compartido) es parte del hilo mismo.	Un mecanismo similar de bloqueo/desbloqueo para sincronización junto con funciones operativas (como lectura/escritura) está presente únicamente con el objeto compartido.
La implementación de la exclusión mutua (ejecución de un hilo que impide la ejecución de otros) y la cooperación (hilos que trabajan en una tarea común) es responsabilidad de los hilos.	La exclusión mutua entre diferentes conjuntos de subprocesos y la cooperación (si es necesario) son manejadas por el propio recurso compartido.
Mecanismo débilmente vinculado ya que todos los subprocesos son independientes y manejan ellos mismos su sincronización en el control de acceso.	El mecanismo es bastante robusto y confiable ya que todo se gestiona únicamente en el lado de los recursos.
Este método es muy propenso a errores cuando el tiempo de bloqueo y el mecanismo construido utilizan el intervalo de tiempo de operación de sincronización de subprocesos son comparables. Existe una buena posibilidad de que, mientras un hilo coloca un bloqueo, su intervalo de tiempo termine y el otro hilo comience a trabajar en el recurso.	Los monitores están bien diseñados para trabajar con grupos de subprocesos pequeños y funcionan de manera muy eficiente a menos que la comunicación entre subprocesos se convierta en una necesidad.
La cola lista o los grupos de subprocesos no están presentes o son manejados por el sistema operativo.	Los subprocesos esperan en colas administradas por el objeto compartido al que todos intentan acceder.
Las cerraduras por sí solas no se utilizan mucho y se implementan mucho menos ampliamente.	Los monitores utilizan intrínsecamente solo bloqueos entre subprocesos y se utilizan mucho más.

Nota:
Como vemos los propios monitores se implementan con el necesario soporte de cerraduras, se suele decir que no son diferentes sino complementarios en la naturaleza de su existencia y funcionamiento.