Uso de recursos compartidos

Cada proceso o hebra se ejecuta de forma independiente. Sin embargo, cuando varias hebras (o procesos) han de acceder a un mismo recurso, se ha de coordinar el acceso a ese recurso.

Ejemplos

Una impresora compartida en red debe imprimir los documentos enviados por distintos usuarios uno detrás de otro, sin mezclar partes de uno con partes de otro.

Cuando varias hebras (o procesos) acceden a los mismos datos (ya sea en memoria o en un fichero), hemos de tener mucho cuidado a la hora de actualizarlos.

Proceso 1	*Proceso 2*
Ingreso en una cuenta	*Transferencia a otra cuenta*
`P1.A` Obtener saldo actual	`P2.A` Obtener saldo actual
… `++`	… `--`
`P1.B` Guardar saldo modificado	`P2.B` Guardar saldo modificado

Si el saldo inicial de la cuenta es de 1000€, realizamos un ingreso de 500€ y una transferencia de 750€, esperamos que el saldo final de la cuenta sea de 750€.

Ahora bien, en función de cómo se ordenen las operaciones de las dos tareas independientes (el ingreso y la transferencia), el resultado final puede ser muy distinto:

- P1.A (lee 1000€)… P2.A (lee 1000€)… P1.B (escribe 1500€)… P2.B (escribe 250€), por lo que nos quedan 250 €.
- P2.A (lee 1000€)… P1.A (lee 1000€)… P2.B (almacena 250€)… P1.B (almacena 1500€), por lo que tenemos 1500 € al final ;-)

El problema proviene de que no sabemos en qué momento se le asignará la CPU a cada hebra/proceso.

Si lo único que nos interesa es leer datos desde distintas tareas independientes, podemos hacerlo en paralelo sin problemas.

Sin embargo, en el momento en el que queramos realizar alguna modificación, hemos de evitar que otras tareas accedan al recurso compartido mientras nosotros estamos usándolo (para que no lean valores incorrectos ni interfieran con lo que estemos haciendo).

Mecanismos de exclusión mutua

Una solución para este problema consiste en que, para acceder a un recurso compartido, haya antes que "adquirirlo".

La hebra que adquiere el recurso bloquea el acceso a él.

Las demás hebras, para acceder al recurso, intentarán adquirirlo y se quedarán bloqueadas hasta que lo "libere" la hebra que lo posee.

De esta forma, el acceso al recurso compartido se realiza secuencialmente (por lo que deberemos tener cuidado para no eliminar completamente el paralelismo de nuestra aplicación).

IMPORTANTE

Un programa debe avanzar hasta proporcionar una respuesta.

En el caso de los programas secuenciales, la existencia de un bucle infinito es la única causa posible de que el programa no proporcione respuesta alguna.

Cuando trabajamos con aplicaciones multihebra, pueden aparecer bloqueos [*deadlocks*] que se ocasionan cuando cada hebra se queda esperando a que otra de las hebras bloqueadas libere un recurso compartido al que quiere acceder.

synchronized

Java permite definir secciones críticas (fragmentos de código que sólo puede estar ejecutando una hebra) mediante la palabra reservada `synchronized`:

```
synchronized (objeto) {

  // A este bloque de código
  // sólo puede acceder una hebra
  // en cada momento

}
```

Si tenemos un objeto cuyo estado queremos actualizar desde distintas hebras de forma coordinada, podemos etiquetar con `synchronized` todos aquellos métodos cuya ejecución concurrente podría causar algún tipo de error:

```
public class Cuenta
{
  private int saldo;

  public synchronized void ingreso (int cantidad)
  {
    saldo += cantidad;
  }

  public synchronized void retirada (int cantidad)
  {
   saldo -= cantidad;
  }
}
```

Cuando se está ejecutando un método `synchronized` asociado a un objeto, el objeto se bloquea y no se puede ejecutar ningún otro método `synchronized` del objeto hasta que termine la ejecución del método que bloqueó el acceso al objeto.

Ejemplo: Realización de reservas

```
if (asiento.disponible())
    asiento.reservar();
```

Si justo después de comprobar que el asiento está disponible y antes de hacer la reserva, se asigna la CPU a otra hebra, ésta podría comprobar que el asiento está libre y reservarlo.

Cuando el control de la CPU vuelva a la primera hebra (que ya había comprobado la disponibilidad de asientos), ésta también hará también la reserva (con lo que tendremos *overbooking*).

Para eliminar el problema, podemos:

1. Incluir el bloque de código en una sección crítica:

```
synchronized (asiento) {

  if (asiento.disponible())
     asiento.reservar();
}
```

2. Marcar con `synchronized` los métodos `disponible()` y `reservar()` de la clase `Asiento`, además de cualquier otro método que pueda modificar el estado del asiento y de asegurarnos de que las operaciones se completan correctamente:

```
public class Asiento
{
  …

  public synchronized boolean disponible ()
  {…}

  public synchronized void reservar (…)
         throws AsientoYaReservadoException
  {…}
}
```

Sobre el uso de mecanismos de exclusión mutua

No hay que abusar del uso de mecanismos secciones críticas. Cada cerrojo que se usa impide potencialmente el progreso de la ejecución de otras hebras de la aplicación. Las secciones críticas eliminan el paralelismo y, con él, muchas de las ventajas que nos llevaron a utilizar hebras en primer lugar. En un caso extremo, la aplicación puede carecer por completo de paralelismo y convertirse en una aplicación secuencial pese a que utilicemos hebras en nuestra implementación.

La ausencia de paralelismo no es, ni mucho menos, el peor problema que se nos puede presentar durante el desarrollo de una aplicación multihebra. El uso de mecanismos de sincronización como cerrojos o monitores puede conducir a otras situaciones poco deseables que también hemos de tener en cuenta:

- **Interbloqueos [deadlocks]**: Una hebra se bloquea cuando intenta bloquear el acceso a un objeto que ya está bloqueado. Si dicho objeto está bloqueado por una hebra que, a su vez, está bloqueada esperando que se libere un objeto bloqueado por la primera hebra, ninguna de las dos hebras avanzará. Ambas se quedarán esperando indefinidamente. El interbloqueo es consecuencia de que el orden de adquisición de los cerrojos no sea siempre el mismo y la forma más evidente de evitarlo es asegurar que los cerrojos se adquieran siempre en el mismo orden.
- **Inanición [starvation]**, cuando una hebra nunca avanza porque siempre hay otras a las que se les da preferencia. Por ejemplo, si le damos siempre prioridad a un “lector” frente a un “escritor”, el escritor nunca obtendrá el cerrojo para modificar el recurso compartido mientras haya lectores. Los escritores "se morirán de inanición".
- **Inversión de prioridad:** El planificador de CPU decide en cada momento qué hebra, de entre todas las no bloqueadas, ha de disponer de tiempo de CPU. Usualmente, esta decisión viene influida por la prioridad de las hebras. Sin embargo, al usar cerrojos se puede dar el caso de que una hebra de prioridad alta no avance nunca, justo lo contrario de lo que su prioridad nos podría hacer pensar. Imaginemos que tenemos tres hebras H1, H2 y H3, de mayor a menor prioridad. H3 está ejecutándose y bloquea el acceso al objeto O. H2 adquiere la CPU al tener más prioridad que H3 y comienza un cálculo muy largo. Ahora, H1 le quita la CPU a H2 y, al intentar adquirir el cerrojo de O, queda bloqueada. Entonces, H2 pasa a disponer de la CPU para proseguir su largo cómputo. Mientras, H1 queda bloqueada porque H3 no llega a liberar el cerrojo de O. Mientras que H3 no disponga de algo de tiempo de CPU, H1 no avanzará y H2, pese a tener menor prioridad que H1, sí que lo hará. El planificador de CPU puede solucionar el problema si todas las hebras disponibles avanzan en su ejecución, aunque sea más lentamente cuando tienen menor prioridad. De ahí la importancia de darle una prioridad alta a las hebras cuyo tiempo de respuesta haya de ser bajo, y una prioridad baja a las hebras que realicen tareas muy largas.

Todos estos problemas ponen de manifiesto la dificultad que supone trabajar con aplicaciones multihebra. Hay que asegurarse de asignar los recursos cuidadosamente para minimizar las restricciones de acceso en exclusiva a recursos compartidos. En muchas ocasiones, por ejemplo, se puede simplificar notablemente la implementación de las aplicaciones multihebra si hacemos que cada una de las hebras trabaje de forma independiente, incluso con copias distintas de los mismos datos si hace falta. Sin datos comunes, los mecanismos de sincronización se hacen innecesarios.