Programació de serveis i processos

TEMA 01 – PART V

Programació de processos i fils

1. Programació multiprocés i paral·lela
   1.4. Sincronització i comunicació entre processos
      1.4.3. Solucions a problemes de sincronització i comunicació
         - Monitors

Unió Europea
Fons social europeu
L'FSE inverteix en el teu futur

IES DR. LLUÍS SIMARRO

1. Programació multiprocés i paral·lela

1.4 Sincronització i comunicació entre processos

1.4.3. Solucions a problemes de sincronització i comunicació

Monitors

Una altra forma de resoldre la **sincronització** de **processos** és l'ús de monitors. Els **monitors** són un conjunt de **procediments encapsulats** que ens proporcionen l'accés a **recursos compartits** a través de **diversos processos** en **exclusió mútua**.

Les **operacions** del **monitor** estan **encapsulades** dins d'un **mòdul** per protegir-les del programador. **Únicament un procés** pot estar en **execució** dins d'aquest **mòdul**.

El grau de seguretat és alt ja que els **processos no saben** com estan **implementats** aquests **mòduls**. El **programador no sap com** i en quin **moment** es **criden** les **operacions** del **mòdul**, així és més robust. Un monitor, una vegada implementat, **si funciona correctament sempre funcionarà bé**.

Un monitor es pot veure com una habitació, tancada amb una porta, que té a dins els recursos. Els **processos** que vulguen **utilitzar** aquests **recursos** han d'entrar a l'habitació, però amb les **condicions** que marca el **monitor** i **únicament un procés** a la vegada. La **resta** que vulga fer ús dels recursos haurà d'**esperar** que isca el que està dins.

Per la seua **encapsulació**, l'única acció que ha de prendre el **programador** del procés que vulga **accedir** al **recurs** protegit és **informar** al **monitor**. L'**exclusió mútua** hi està **implícita**. Els **semàfors** en canvi, s'han d'**implementar** amb una **seqüència correcta** de **senyal** (`sendSignal`) i **esperar** (`sendWait`) per tal de no bloquejar el sistema.

■ Un **monitor** és un **algoritme** que fa una abstracció de dades que ens permet **representar** de **forma abstracta** un **recurs compartit** mitjançant un **conjunt** de **variables** que defineixen el seu estat. L'**accés** a aquestes **variables** únicament és possible des d'uns **mètodes** del **monitor**.

Els **monitors** han de poder incorporar un **mecanisme** de **sincronització**. Per tant, s'han d'implementar. Es pot fer ús de **senyals**. Aquests senyals s'utilitzen per **impedir** els **bloquejos**. Si el **procés** que està en el **monitor** ha d'**esperar** un **senyal**, es posa en estat d'**espera** o **bloquejat fora del monitor**, permetent que un **altre procés utilitze** el **monitor**. Els **processos** que estan **fora** del **monitor**, estan a l'**espera** d'una **condició** o **senyal** per **tornar** a **entrar**.

Aquestes **variables** que s'utilitzen pels **senyals** i són utilitzades pel monitor per la **sincronització** s'anomenen **variables de condició**. Aquestes poden ser **manipulades** amb operacions de `sendSignal` i `sendWait` (com els semàfors).

- `sendWait`: un procés que està esperant a un **esdeveniment** indicat per una **variable** de **condició abandona** de forma temporal el **monitor** i es posa a la **cua** que correspon a la seua **variable** de **condició**.
- `sendSignal`: **desbloqueja** un **procés** de la **cua** de **processos** bloquejats amb la **variable** de **condició** indicada i es posa en estat **preparat** per **entrar** al **monitor**. El procés que entra **no ha de ser el que més temps porta esperant**, però s'ha de **garantir** que el **temps d'espera** d'un procés siga **limitat**. Si no existeix **cap procés** a la **cua**, l'operació `sendSignal` no té efecte, i el **primer procés** que demane l'ús del **monitor entrarà**.

Un monitor consta de 4 elements:

- **Variables o mètodes permanents o privats**: són les **variables** i **mètodes interns** al **monitor** que **només** són **accessibles** des de **dins** del **monitor**. **No** es **modifiquen** entre **dues crides consecutives** al **monitor**.
- **Codi d'inicialització**: **inicialitza** les **variables permanents**, s'**executa** quan el

monitor és **creat**.

- **Mètodes externs o exportats**: són **mètodes** que són **accessibles** des de **fora** del **monitor** pels **processos** que **volen entrar** a utilitzar-lo.
- **Cua de processos**: és la **cua** de **processos bloquejats** a l'**espera** del **senyal** que els **allibere** per **tornar** a **entrar** al **monitor**.

En el camp de la **programació** un **monitor** és un **objecte** en el qual **tots** els seus **mètodes** estan **implementats** sota **exclusió mútua**. En el llenguatge **Java** són **objectes** d'una **classe** en els quals tots els seus **mètodes públics** són `synchronized`.

`wait`, `notify` i `notifyAll`, són operacions que permeten **sincronitzar** el **monitor**.

Un **semàfor** és un **objecte** que permet **sincronitzar** l'**accés** a un **recurs compartit** i un **monitor** és una **interfície** d'**accés** al **recurs compartit**. Constitueix l'encapsulament en un objecte, més segur, robust i escalable.

Sincronitzar a Java

A Java l'**execució d'objectes no està protegida**. Si volem **aplicar exclusió mútua** a Java hem d'utilitzar la **paraula** reservada `synchronized` a la **declaració** d'**atributs**, de **mètodes** o en un **bloc de codi** dins d'un mètode.

Tots els **mètodes** amb el modificador `synchronized` s'executaran en **exclusió mútua**. El modificador assegura que si s'està **executant** un **mètode sincronitzat** **cap altre mètode sincronitzat** es puga **executar**. Però els mètodes no sincronitzats poden estar executant-se i amb més d'un procés a la vegada. A més, el **mètode sincronitzat** pot ser executat **únicament** per **un procés**, la **resta** de **processos** que volen executar el mètode s'hauran d'**esperar que acabe** el procés que l'està executant.

Sincronitzar mètodes a Java:

```
public synchronized void metodeSincronitzat {
    //part de codi sincronitzat
}
```

Si no ens interessa sincronitzar tot el mètode, sinó **únicament una part del codi**, utilitzarem la paraula reservada `synchronized` sobre el mateix objecte que ha cridat el mètode en el qual es troba la part del codi que s'ha de sincronitzar.

Sincronitzar blocs de codi a Java:

```
public void metodeNoSincronitzat {
    //part de codi sense sincronitzar
    synchronized(this){
      //part de codi sincronitzat
    }
     //part de codi sense sincronitzar
}
```

`this` és l'**objecte que ha cridat al mètode**, però també podem utilitzar un altre objecte `synchronzed(objecte)`, si el que volem és realitzar **més d'una operació atòmica sobre un objecte**.

Java, per tant, ens proporciona amb `synchronized` els **recursos** que ens donaria la **implementació** d'un **semàfor** o d'un **monitor**. Les seues llibreries ens proporcionen un **accés** en **exclusió mútua** i **controlen** els errors de **bloqueig** o **interbloqueig** que es puga ocasionar **entre fils** en execució.

Atomicitat d'una operació i classes atòmiques de Java

L'únic àrbitre que tenim en la gestió de processos és el **planificador**. S'encarrega de **decidir** quin **procés** fa **ús** del **processador**. Després les **instruccions** o operacions que formen el **procés** es van **executant** al processador **seguint** un **ordre** concret.

Una **operació** o un **grup d'operacions** s'han de poder **executar com** si foren una **única instrucció**. A més, **no** es poden **executar** de forma **concurrent** amb qualsevol **altra operació** en la qual hi haja **involucrada** una **dada** o recurs que **utilitza la primera operació**.

- L'**atomicitat** d'una operació és poder garantir que no es poden executar dues operacions de forma concurrent si fan ús d'un recurs compartit fins que una de les dues deixa lliure aquest recurs. Una operació atòmica únicament ha d'observar dos **estats**: l'**inicial** i el **resultat**. Una operació atòmica, **o s'executa completament o no ho fa en absolut**.

L'**atomicitat real** són les **instruccions** que executen a **codi màquina**. En canvi, l'**atomicitat model** és un **grup d'instruccions**, a codi màquina, que s'**executa de forma atòmica**.

Classes d'operacions atòmiques

La instrucció `x++;` o `x=x+1;` té la següent seqüència d'**instruccions atòmiques reals**:

1. Llegir valor x
2. Sumar 1 a x
3. Guardar valor x

L'**atomicitat de model** entén la instrucció completa, `x = x + 1` com a **atòmica**.

Dos processos executen en **paral·lel** el següent exemple:

```
package operacionsatomiques;
public class OperacionsAtomiques {
  public static void main(String[] args) {
    int x=0;
    x=x+1;
    System.out.println(x);
  }
}
```

Si tenim en compte que els **dos processos executen** en **paral·lel** la instrucció `x=x+1;` en l'**atomicitat real** una possible ordenació d'execució seria la següent:

1	`int x=0;`	
	Procés 1 (x=x+1)	**Procés 2 (x=x+1)**
2	Llegir valor x	
3		Llegir valor x
4	Afegir a x 1	
5		Afegir a x 1
6	Guardar valor x	
7		Guardar valor x
8	`System.out.printl(x);`	

Depenent de l'ordre d'execució de les operacions atòmiques d'un procés i l'altre el **resultat** o el **valor** d'**x pot variar**.

Classes atòmiques de Java

Les classes atòmiques de Java són una primera solució als **problemes de processos concurrents**. Permeten **executar** algunes **operacions** com si foren **atòmiques reals**.

Una **operació atòmica** té únicament dos **estats**: l'**inicial** i el **resultat**, és a dir, es completa **sense interrupcions** des de l'**inici fins** al **final**.

- `Volatile` és un modificador, utilitzat en la programació concurrent, que es pot aplicar a alguns tipus primitius de variables. Indica que el **valor** de la **variable** quedarà **guardat** a la **memòria** i **no a un registre** del processador. Així, és **accessible** per un **altre procés** o fil **per modificar-la**. Per exemple:

```
volatile int c;
```

A **Java**, l'**accés** a les **variables** de tipus **primitius** (**excepte** `double` i `long`) és **atòmic**. Així, també és **atòmic** l'**accés** a totes les **variables** de tipus **primitiu** en les quals apliquem el **modificador** `volatile`.

Java assegura l'atomicitat de l'accés a les variables volàtils o primitives, però **no a la seues operacions**. Això vol dir que l'operació `x++;` en la qual `x` és un `int`, **no és una operació atòmica** i per tant **no** és un **fil segur**.

Java incorpora, des de la versió 1.5, al paquet `java.util.concurrent.atomic`, les classes que ens permeten **convertir** en **atòmiques** (fils segurs) algunes **operacions**, com ara **augmentar**, **reduir**, **actualitzar** i **afegir** un **valor**. Totes les classes tenen mètodes `get` i `set` que també són atòmics.

Les classes que incorpora són: `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray` i actualitzadors que són bàsicament derivats d'una variable tipus `volatile` i són `AtomicIntegerFieldUpdater`, `AtomicLongFieldUpdater` i `AtomicReferenceFieldUpdater`.

Per tant, podem convertir l'operació `x++` o `x=x+1` en una operació atòmica .

Classe Comptador sense operacions atòmiques

```
public class Comptador {
    private int x = 0;
    public void augmenta() {
        x++;
    }
    public void disminueix() {
        x--;
    }
    public int getValorComptador() {
        return x;
    }
}
```

Per convertir a operacions atòmiques l'exemple anterior, utilitzarem els mètodes de la classe **AtomicInteger** del paquet **java.util.concurrent.atomic.AtomicInteger**.

Classe comptador utilitzant classes atòmiques

```
import java.util.concurrent.atomic.AtomicInteger;

public class ComptadorAtomic {
    private AtomicInteger x =new AtomicInteger(0);
    public void augmenta() {
      x.incrementAndGet();
    }
    public void disminueix() {
      x.decrementAndGet();
    }
    public int getValorComptadorAtomic() {
      return x.get();
    }
}
```

A la documentació oficial podeu trobar més informació sobre aquest paquet. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html

Col·leccions concurrents

Les **col·leccions** són **objectes** que contenen **múltiples dades** d'un **mateix tipus**. Les col·leccions poden estar **involucrades** també en l'execució de **processos concurrents** i ser utilitzades per diferents processos de manera que s'acaben **compartint** també les **dades** contingudes a la **col·lecció**.

A Java per treballar amb col·leccions hem d'utilitzar el Framework `Collections` en el qual es troben els paquets `java.util` i `java.util.concurrent`, que contenen les **classes** i **interfícies** que ens permeten crear **col·leccions**.

A Java les interfícies `List<E>`, `Set<E>` i `Queue<E>` serien les col·leccions més importants. Representen **llistes**, **conjunts** i **cues**. `Map<E>` és també una col·lecció de clau/valor. Cadascuna amb les seues característiques d'**ordenació**, els mètodes d'**inserció**, d'**extracció** i **consulta**, si permeten **elements repetits** o no, etc.

El problema amb aquestes col·leccions és que en la **programació concurrent** en la qual siguen compartides per **diferents processos** o fils de forma **simultània**, podrien donar **resultats inesperats**.

Alguns llenguatges de programació van solucionar aquest problema sincronitzant les classes. A Java s'utilitza, per exemple, `Collections.synchronizedList(List<T> list)` sobre la interfície `List<E>` per poder sincronitzar la classe que la implementa, la classe `ArrayList`. Altra classe que implementa la interfície `List` és `Vector`. Java ja sincronitza aquesta classe.

En la versió 1.5 va aparèixer `java.util.concurrent` que incorpora classes que ens permeten **solucionar problemes** de **concurrència** d'una forma simple.

`ArrayList` és una col·lecció de dades i si volem que siga utilitzada per diferents fils d'execució el que hem de fer és **sincronitzar totes les operacions de lectura i escriptura** per preservar la **integritat** de les **dades**. Si aquest `ArrayList` és molt consultat i poc actualitzat, la **sincronització penalitza** molt l'**accés**. Java crea la classe `CopyOnWriteArrayList`, que és l'execució de fil segur d'una `ArrayList`.

La classe `CopyOnWriteArrayList` cada vegada que es modifica l'array es crea en la memòria un array amb el contingut **sense processos** de **sincronització**. Això fa que les consultes no siguen tan costoses.

La cua **FIFO** (en anglès, First In Firts Out) és una estructura de dades implementada de forma que els primers elements en entrar a l'estructura són els primers en eixir-ne.

- La interfície `BlockingQueue` implementa una cua FIFO, que **bloqueja** el **fil** d'execució si intenta **treure** de la **cua** un **element** i la **cua** està **buida** o si intenta **inserir** un **element** i està **plena**. També conté mètodes que fan que el bloqueig dure un temps determinat abans de donar un error d'inserció o lectura i així evitar una execució infinita.
- La classe `SynchronousQueue` és una cua de bloqueig, implementa una `BlockingQueue`. El que fa és que per **cada** operació **d'inserció** ha **d'esperar** una **d'eliminació** i per cada inserció ha d'esperar una eliminació.
- `ConcurrentMap` és una altra classe que defineix una **taula hash** amb **operacions atòmiques**.

Podeu trobar més informació sobre el paquet a la web oficial de Java. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html