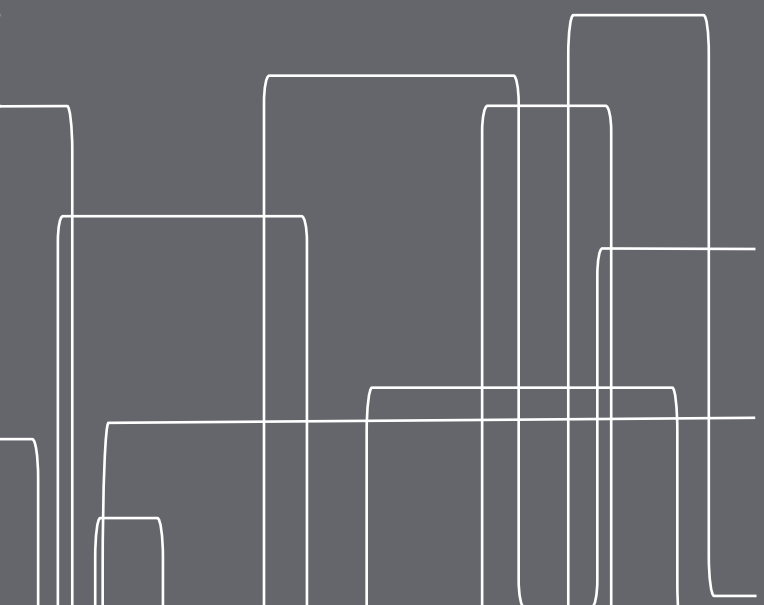
Objektorienterad Programkonstruktion

Föreläsning 12

12 dec 2016



Parallella Problem

- I program med flera parallella exekveringstrådar kan det uppstå problem, fel och andra fenomen som inte förekommer i enkeltrådade program:
  - **Thread interference** - två (eller fler) trådar stör varandra
  - **Deadlock** - två (eller fler) trådar blockerar varandra. Detta feluppstår ofta när man försöker lösa ovanstående fel
  - **Starvation** - trådar tillbringar all sin tid med att vänta på samma begränsade resurs i stället för at göra nytta
  - **Livelock** - trådar reagerar på varandras handlingar på ett sätt som skapar nya reaktioner i oändliga loopar

Thread Interference

- När två trådar arbetar med samma variabel, kan det hända att den ena tråden påbörjar en operation innan den andra är färdig. Om operationernas inbördes ordning spelar roll kommer resultatet att påverkas av hur de olika trådarna har trasslat in sig i varandra, och variera mellan olika körningar.
- Det är inte alltid uppenbart vilka operationer som kan trassla ihop sig!
- En operation som alltid körs som en sammanhållen enhet kallas **atomic** (atomär)

Thread Interference, exempel

- Antag att det finns en gemensam `int i`, som två olika trådar har tillgång till
- Då kan den enkla operationen `++i` resultera i interferens
- Detta beror på att `++i` egentligen består av tre steg:
  - 1. Läs `i` från minnet
  - 2. Öka `i` med 1
  - 3. Skriv tillbaka värdet till minnet
- Två trådar (**A** och **B**) kan nu göra så här:

alternativ 1:	alternativ 2:
A: läs i (värde 0)	A: läs i (värde 0)
B: läs i (värde 0)	A: öka i (värde 1)
A: öka i (värde 1)	A: skriv i (värde 1)
A: skriv i (värde 1)	B: läs i (värde 1)
B: öka i (värde 1)	B: öka i (värde 2)
B: skriv i (värde 1)	B: skriv i (värde 2)

Synkronisering

- För att undvika att två trådar utför olika (eller samma) delar av samma metod samtidigt
- Man tillåter bara en tråd i taget att anropa en metod
- Andra trådar som vill anropa metoden ställs på kö tills metoden blir "ledig"
- I Java åstadkommer man detta med nyckelordet `synchronized`, tex:

```
public synchronized void incrementCounter(){
      counter ++;
}
```

Lock

- *Concurrency Pattern*
- Synkroniseringen görs med hjälp av ett så kallat **lås** (**lock**)
- För att kunna anropa en synkroniserad metod måste den anropande tråden ha låset till metodens objekt
- Låset kan bara innehas av en tråd åt gången, och andra trådar som vill anropa en metod låst med samma lås (dvs i samma objekt) ställs på kö tills låset är ledigt
- Eftersom varje objekt bara har ett lås, kommer en tråds anrop av en synkroniserad metod att blockera alla andra anrop av denna och andra synkroniserade metoder i detta objekt
- Eftersom alla synkroniserade metoder i ett objekt är låsta med samma lås, kan en tråd anropa flera olika synkroniserade metoder - den har ju redan låset. Detta kallas **reentrant**

Exempel

```
public class SynchronizedCounter{

    private int counter;

    public SynchronizedCounter(){
        counter = 0;
    }

    public synchronized void increment(){
        counter++;
    }

    public synchronized void decrement(){
        counter--;
    }

    public synchronized int getValue(){
        return counter;
    }
}
```

Synkroniserad Konstruktor?

- Det går **inte** att synkronisera konstruktorn
- Det bör inte behövas, eftersom objektet inte går att referera till förrän konstruktorn har returnerat
- Man bör dock se till att man inte lämnar ut referenser till objektet (från konstruktorn) innan objektet är färdigt
- Man vill undvika detta:

```
public MyObject(LinkedList publicList){
    someVariable = Math.random();
    publicList.add(this);
    otherVariable = safeVariableGenerator();
}
```

- I exemplet ovan kan andra trådar hämta instanser av `MyObject` från `publicList` innan värdet på `otherVariable` sätts

Deadlock

- Betrakta följande kod:

```
public synchronized void syncMethod(){
    i++;
    otherObject.syncMethod();
}
```

- I exemplet ovan kan ju två trådar samtidigt anropa `syncMethod` på två olika objekt, och alltså ta låsen i respektive objekt. Anta nu att dessa två objekt finns lagrade som otherObject i varandra. Då kommer båda trådarna att vänta på att den andra ska släppa sitt lås, och de kommer aldrig tills slutet av `syncMethod`, och släpper aldrig sina lås - detta kallas **deadlock**

Synkroniserade kodblock

- Man kan synkronisera en del av ett metodanrop, i stället för hela
- Detta är viktigt om man anropar metoder från andra objekt inne i metoden man vill synkronisera
- I Java går det att synkronisera ett begränsat kodblock, med hjälp av `synchronized(Object lock)`
- `lock` anger i detta fall vilket objekt som skall tillhandahålla låset, typiskt kan det vara `this`
- Vi kan till exempel skriva så här:

```
public void syncMethod(){
    synchronized(this){
        i++;
    }
    otherObject.syncMethod();
}
```

- Nu släpps låset innan vi anropar metoden i `otherObject`

Olika lås

- Det går att låsa olika metoder med olika lås, om vi tycker att det är OK om olika trådar anropar dem:

```
public class SynchronizedCounterPair {

    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {c1++; }
    }

    public void inc2() {
        synchronized(lock2) {c2++;}
    }
}
```

Starvation

- Om man har ett vanligt förekommande metodanrop som skyddas av ett lås, kan man riskera att de flesta trådar tillbringar all sin tid med att vänta på lås i stället för att utföra arbete. Programmet låser sig inte, men kan gå väldigt långsamt trots att inga tunga uppifter utförs.

- Problemet kan lösas genom att stora synkroniserade metoder om möjligt delas upp i flera små, med separata lås.

Livelock

- Om två trådar reagerar på varandras handlingar kan de också hamna i en evighetsloop. De är inte i deadlock, men de kommer ändå inte vidare.
- Exempel: tråd **A** och tråd **B** försöker båda se till att summan av `A.x` och `B.x` är udda:

```
while((this.x + otherObject.x)%2 == 1 ){
    this.x++;
}
```

- I detta fall kan vi hamna i ett läge där båda trådarna anpassar sig efter den andra, och aldrig löser problemet, likt två personer som försöker undvika en krock i en korridor men hela tiden kliver åt sidan åt samma håll

Mer Lock i Java

- Java erbjuder också ett gränsnitt `Lock,` för låsklasser
- För dessa kan man prova att ta ett lås, och om det inte går, så kan man släppa låset utan att anropa metoden
- Detta är lite krångligare än att använda synchronized, och man måste hantera låsen själv explicit
- Detta kan ge upphov till ytterligare l**ivelock**
- Om man använder `try/catch` måste man komma ihåg att släppa låset i `finally`

```
private final Lock myLock = new ReentrantLock();
...
if(myLock.tryLock()){
    doSomeSynchronizedStuff();
    myLock.unlock();
}else{
    throw(new Exception("Lock was taken!"));
}
```

Exempel

```
public class SynchronizedCounter{

    private int counter;

    public SynchronizedCounter(){
        counter = 0;
    }

    public synchronized void increment(){
        counter++;
    }

    public synchronized void decrement(){
        counter--;
    }

    public synchronized int getValue(){
        return counter;
    }
}
```

Exempel

```
public class SynchronizedCounter{

    private int counter;

    public SynchronizedCounter(){
        counter = 0;
    }

    public synchronized void increment(){
        counter++;
    }

    public synchronized void decrement(){
        counter--;
    }

    public synchronized int getValue(){
        return counter;
    }
}
```