Föreläsning 14

Trådar
Trådsäkerhet

Aktiva objekt

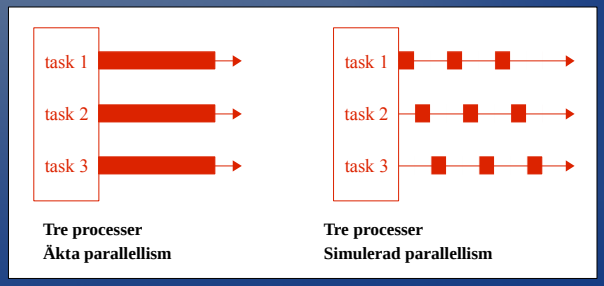
Det är välkänt från vardagslivet att saker händer samtidigt.

Aktiva objekt gör saker på eget initiativ, medan *passiva objekt* endast gör saker när de blir ombedda.

Det är viktigt att det underliggande systemet stöder parallellism så att man kan modellera verkligheten på ett bra sätt.

Det är onaturligt att modellera parallellism som ett sekventiellt förlopp.

I datorer med en processor *simuleras parallellismen*.



Motivering

Många program måste kunna hålla på med flera saker samtidigt:

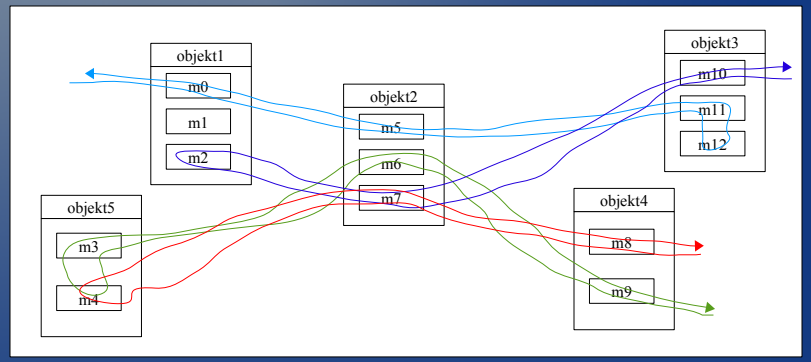
- bokningssystem av olika slag
- en webbserver som måste kunna leverera flera webbsidor samtidigt
- en grafisk applikation som ritar samtidigt som den arbetar med nästa bild
- ett grafiskt användargränssnitt som utför beräkningar och samtidigt är redo att ha dialog med användaren
- en webbläsare som börjar visa ett dokument innan hela dokumentet är nedladdat

Denna typ av program kallas samverkande program (*concurrent programs*).

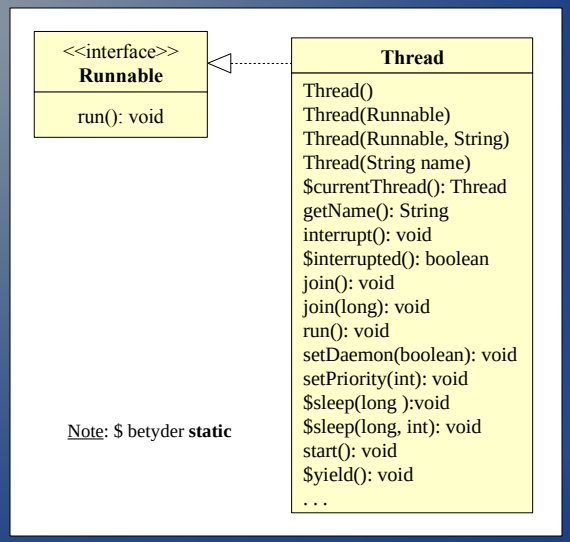
Vi skall här endast ge en introduktion till programmering med samverkande program, för mer djupgående studier hänvisas till kursen Parallellprogrammering.

Trådar

I Java beskriver man aktiva objekt (och parallellism) med hjälp av standardklassen Thread. Aktiviteter som pågår samtidigt kallas därför i Java för *trådar*.



Klassen Thread



Trådar

I ett parallellt program beskrivs varje aktivitet som en instans av klassen Thread. Det tråden skall utföra definieras genom att överskugga metoden run().

```
public class SupporterThread extends Thread {
   private String team;
   public SupporterThread(String team) {
      this.team = team;
   }//constructor
   public void run() {
      for(int i = 0; i <= 10; i = i + 1)
         System.out.print(team + " ");
   }//run
}//SupporterThread
```

Metoden run() definierar vad tråden gör

Trådar

För att starta exekveringen av en tråd anropar man metoden start()!

Anropas run() direkt skapas ingen tråd.

```
public class SMFinal {
   public static void main(String[] arg) {
      Thread ifkFan = new SupporterThread("IFK");
      Thread aikFan = new SupporterThread("AIK");
      ifkFan.start();
      aikFan.start();
   }//main
}//SMFinal
```

En tråd startas genom att anropa metoden start()

Utskriften kan t.ex. bli:

IFK IFK IFK IFK IFK IFK AIK AIK AIK AIK
IFK IFK IFK IFK AIK AIK AIK AIK AIK AIK

Icke-deterministiskt beteende

Sekventiella program säges vara *deterministiska* vilket betyder att:

- vilka operationer som utförs i programmet är en konsekvens av föregående operationer
- det är förutsägbart i vilken ordning operationerna i ett sekventiellt program kommer att utföras
- om ett program körs flera gånger med samma indata kommer samma resultat att erhållas varje gång

Program med parallellism uppvisar ett *icke-deterministiskt beteende*:

- i vilken ordning operationerna sker mellan de olika programflödena vet vi inte
- två programkörningar med samma indata kan ge olika resultat

Metoden sleep()

En tråd kan temporärt avbryta sin exekvering med metoden sleep(), som är en *klassmetod* i klassen Thread.

Metoden sleep() finns i följande två versioner:

public static sleep(**long** mills) **throws** InterruptedException

public static sleep(**long** mills, **int** nanos) **throws** InterruptedException

Tiden som man vill avbryta trådens exekvering anges alltså i millisekunder respektive millisekunder och nanosekunder.

Eftersom sleep() kan kasta en kontrollerad exceptionell händelse måste anropet av metoden ligga i ett **try-catch**-block.

Exempel: SleepingSupporterThread

Nedan har vi skrivit om föregående exempel på så sätt att vi låter tråden ”sova” med ett slumpmässigt tidsintervall mellan 0 och 1 sekund mellan varje utskrift.

Vi nyttjar också att klassen Thread har en konstruktor

public Thread(String name)

som ger tråden namnet name, samt att detta namn kan avläsas med metoden getName().

```
public class SleepingSupporter extends Thread {
   public SleepingSupporter(String team) {
      super(team);      //Ger tråden ett namn
   }
   public void run() {
      for(int i = 1; i <= 10; i = i + 1) {
         try{
            sleep((int)(Math.random() * 1000));
         }
         catch(InterruptedException e) {}
         System.out.println(i + " " + getName());
      }
   }//run
}//SleepingSupporter
```

Exempel: fortsättning

Vi skriver ett testprogram som innehåller tre objekt av typen SleepingSupporterThread:

```
public class CupFinal {
   public static void main(String[] arg) {
      Thread ifkFan = new SleepingSupporter("IFK");
      Thread aikFan = new SleepingSupporter("AIK");
      Thread hulligan = new SleepingSupporter("Ut med domaren");
      ifkFan.start();
      aikFan.start();
      hulligan.start();
   }//main
}//CupFinal
```

En körning kan se ut enligt nedan:

```
1 AIK
1 IFK
1 Ut med domaren
2 IFK
3 IFK
2 AIK
3 AIK
2 Ut med domaren
4 AIK
. . .
```

Interfacet Runnable

Java tillåter inte multipla implementationsarv, därför är det omöjligt för en klass som är subklass till Thread att ärva från någon annan klass. Aktiva objekt har många gånger behov av att ärva från andra klasser.

Lösningen är att använda *interfacet* Runnable:

```
public interface Runnable {
   public void run();
}
```

Interfacet Runnable

- har endast en metod run()
- varje klass som implementerar Runnable måste implementera metoden run()
- metoden run() i klassen som implementerar Runnable är utgångspunkten för exekveringen av en tråd.

Interfacet Runnable

Ett Runnable-objekt abstraherar begreppet *ett jobb* (t.ex. ett recept).

Ett Thread-objekt abstraherar begreppet *en arbetare* (t.ex. en kock).

Hur kan vi associera ett jobb med en arbetare?

Genom att klassen Thread har en konstruktor

```
public Thread(Runnable target)
```

Konstruktorn skapar ett Thread-objekt som när det startas exekverar metoden run() i objekt target.

Exempel:

```
Runnable recipe = . . .;
Thread chef = new Thread(recipe);
chef.start();
```

Exempel Runnable

Vi skriver en klass Writer som har två instansvariabler text och interval. När ett objekt av klassen Writer exekveras skall texten text skrivas ut gång på gång med ett tidsintervall av interval sekunder mellan utskrifterna.

Exempel Runnable

```
public class Writer implements Runnable {
   private String text;
   private long interval;
   public Writer(String text, long time) {
      this.text = text;
      interval = time * 1000;
   }//constructor
   public void run() {
      while(!Thread.interrupted()) { //Kontrollera om aktuell tråd blivit avbruten
         try {
            Thread.sleep(interval);  //Låt den aktuella tråden sova
         }
         catch (InterruptedException e) {
            break;
         }
         System.out.print(text + " ");
      }
   }//run
}//Writer
```

Exempel Runnable

För att demonstrera hur flera objekt av klassen Writer kan exekvera parallellt ges nedan ett program som under en minut skriver ut "IFK" var 3:e sekund, "AIK" var 4:e sekund och "Ut med domaren" var 5:e sekund.

Exempel Runnable

```
public class TestWriter {
   public static void main(String[] arg){
      Thread ifkFan = new Thread(new Writer("IFK", 3));
      Thread aikFan = new Thread(new Writer("AIK", 4));
      Thread hulligan = new Thread(new Writer("Ut med domaren", 5));
      ifkFan.start();
      aikFan.start();
      hulligan.start();
      try {
         Thread.sleep(60000);
         ifkFan.interrupt();
         aikFan.interrupt();
         hulligan.interrupt();
      }
      catch(InterruptedException e) {}
   }//main
}//TestWriter
```

Avbryt trådarna efter 60 sekunder

Exempel Runnable

```
public class TestWriter2 {
   public static void main(String[] arg){
      Thread ifkFan = new Thread(new Writer("IFK", 3));
      Thread aikFan = new Thread(new Writer("AIK", 4));
      Thread hulligan = new Thread(new Writer("Ut med domaren", 5));
      ifkFan.setDaemon(true);
      aikFan.setDaemon(true);
      hulligan.setDaemon(true);
      ifkFan.start();
      aikFan.start();
      hulligan.start();
      try {
         Thread.sleep(60000);
      }
      catch(InterruptedException e) {}
   }//main
}//TestWriter2
```

Sätt till demon-trådar. Dödas av JVM när det inte finns andra icke demon-trådar som exekverar.

Exempel Runnable

Utskriften från programmet blir:

IFK AIK Ut med domaren IFK AIK IFK Ut med domaren AIK IFK Ut med domaren IFK AIK IFK Ut med domaren AIK IFK AIK IFK Ut med domaren IFK AIK Ut med domaren IFK AIK IFK Ut med domaren AIK IFK IFK Ut med domaren AIK IFK AIK Ut med domaren IFK AIK IFK Ut med domaren IFK AIK IFK Ut med domaren AIK IFK

Vänta på att tråden skall avsluta – metoden join()

Vi utökar vår tidigare klass SMFinal enligt:

```
public class SMFinal2 {
   public static void main(String[] arg) {
      Thread ifkFan = new SupporterThread("IFK");
      Thread aikFan = new SupporterThread("AIK");
      ifkFan.start();
      aikFan.start();
      System.out.println("Storma plan!");
   }//main
}//SMFinal2
```

När stormas planen?

Vänta på att tråden skall avsluta – metoden join()

För att vänta på att en tråd avslutas används metoden join():

```
public class SMFinal3 {
   public static void main(String[] arg) {
      Thread ifkFan = new SupporterThread("IFK");
      Thread aikFan = new SupporterThread("AIK");
      ifkFan.start();
      aikFan.start();
      try {
         ifkFan.join();
         aikFan.join();
      }
      catch (InterruptedException e) { }
      System.out.println("Slutsignal! Storma plan!");
   }//main
}//SMFinal3
```

Väntar på att trådarna avbryts!

Bakgrundsjobb

```
public class BackgroundJob extends Thread {
   private int maxSteps = 0;
   private long sum = 0;    // Calculate sum = 1+2+3+...
   private long interval;
   public BackgroundJob(long interval, int maxSteps) {
      this.maxSteps = maxSteps;
      this.interval = interval;
   }//constructor

   public long getResult() {
      return sum;
   }//getResult
}//BackgroundJob
```

```
public void run() {
   int i = 1;
   while ( i <= maxSteps && ! Thread.interrupted() ) {
      try { Thread.sleep(interval); }
      catch (InterruptedException e) { break;}
      System.out.println("Background work... " + i++);
      sum += i;
   }
   System.out.println("Background work finished");
}//run
```

Bakgrundsjobb (fortsättning)

```
public class Main {
   private BackgroundJob backgroundJob = new BackgroundJob(1000,10);
   public Main() {
      backgroundJob.start();  // Start background thread
      foregroundJob();        // Do foreground job in parallel
   } //constructor
```

```
   public static void main(String[] arg) {
      Main m = new Main();
   }//main
} //Main
```

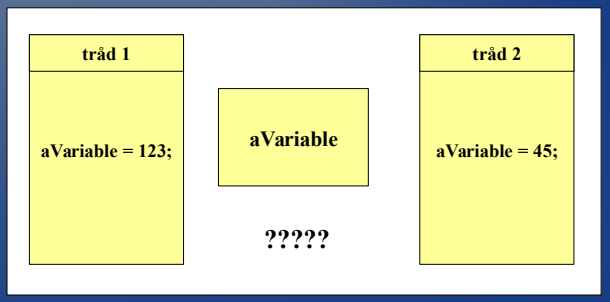
```
   public void foregroundJob() {
      int i = 0;
      while (i <= 20 ) {
         try { Thread.sleep(200); }
         catch (InterruptedException e) { break; }
         System.out.println("Foreground: " + i++);
      }
      try { backgroundJob.join(); }
      catch (InterruptedException e) { }
      System.out.println("Background result: " +
      backgroundJob.getResult() );
      System.out.println("Foreground work continues...");
   }//foregroundJob
```

Trådsäkerhet

Då ett objekt modifieras kan det inta ett antal tillfälliga tillstånd som inte är konsistenta (d.v.s. ogiltiga).

Om en tråd avbryts under en modifikation av ett objekt, så kan det lämna objektet i ett ogiltigt tillstånd.

En klass säges vara trådsäker om den garanterar konsistens för sina objekt, även om det finns multipla trådar närvarande.



Kritiska sektioner

Kodsegment som har access till samma objekt från olika separata trådar utgör en s.k. *kritisk sektion* (eng: *critical section*), och måste synkroniseras på så sätt att endast en tråd i taget får tillgång till objektet, annars kan objektet hamna i ett *inkonsistent tillstånd*.

Kritiska sektioner

Exempel:

Antag att vi har en klass för att handha bankkonton enligt nedan:

```
public class Account {
   private double balance;
   . . .
   public boolean withdraw(double amount) {
      if (amount <= balance) {
         double newBalance = balance – amount;
         balance = newBalance;
         return true;
      }
      else
         return false;
   }//withdraw
   public void deposite(double amount) {
      balance = balance + amount;
   }//deposite
   . . .
}//Account
```

Inkonsistent tillstånd

Antag vidare att balansen på ett konto är 100000 kronor och att två uttag görs samtidigt vardera på 100000 kronor. Följande kan hända:

Balans	Uttag 1	Uttag 2
100000	amount<= balance	
100000		amount<= balance
100000	newBalance = balance - amount	
100000		newBalance = balance - amount
0	balance = newBalance	
0		balance = newBalance
0	**return true**	
0		**return true**

Ett inkonsistent tillstånd på kontot har inträffat!!

Balansen borde vara -100000 men är 0!

Vi har fått ***race condition***. En tråd försöker läsa data medan en annan tråd uppdaterar densamma.

Synkronisering

För att kontot inte skall hamna i ett inkonsistent tillstånd måste metoden withdraw synkroniseras, d.v.s. endast en tråd i taget skall kunna få tillgång till metoden.

Java's lösning för att åstadkomma synkronisering:

- varje objekt definieras som en *monitor*, vilken har ett singulärt lås
- för att en tråd skall få tillträde till en kritisk sektion måste tråden först få tillgång till låset
- att ha tillgång till låset innebär att ingen annan tråd kan ha tillgång till låset
- låset återlämnas när tråden lämnar den kritiska sektionen.

Allt detta görs automatiskt med konstruktionen **synchronized**. Programmerarens uppgift är att identifiera de kritiska sektionerna.

Synkroniserad version av withdraw

En synkroniserad version av withdraw får följande utseende:

```
public class Account {
   private double balance;
   . . .
   public synchronized boolean withdraw(double amount) {
      if (amount <= balance) {
         double newBalance = balance – amount;
         balance = newBalance;
         return true;
      }
      else
         return false;
   }//withdraw
   public void deposite(double amount) {
      balance = balance + amount;
   }//deposite
   . . .
}//Account
```

Konsistent tillstånd

Om vi nu återigen antar att balansen på kontot är 100000 kronor och att två uttag görs samtidigt på 100000 kronor händer följande:

Balans	Uttag 1	Uttag 2
100000	amount<= balance	
100000	newBalance = balance - amount	
0	balance = newBalance	
0	**return true**	
0		amount<= balance
0		**return false**

Ömsesidig uteslutning

För att göra klassen Account trådsäker räcker det inte att enbart metoden withdraw är synkroniserad, utan även metoden deposite måste synkroniseras.

Detta på grund av att båda metoderna konkurrerar om att förändra tillståndet på variabeln `balance`.

Ömsesidig uteslutning

```
public class Account {
   private double balance;
   . . .
   public synchronized boolean withdraw(double amount) {
      if (amount <= balance) {
         double newBalance = balance - amount;
         balance = newBalance;
         return true;
      }
      else
         return false;
   }//withdraw
   public synchronized void deposite(double amount) {
      balance = balance + amount;
   }//deposite
   . . .
}//Account
```

`ac.withdraw` stänger ute ytterligare en `ac.withdraw`
`ac.withdraw` stänger ute `ac.deposit`
`ac.deposit` stänger ute ytterligare en `ac.deposit`
`ac.deposit` stänger ute `ac.withdraw`
men `ac1.withdraw` stänger inte ute `ac2.withdraw`, etc.

Låset ägs av tråden

Låset ägs av tråden, vilket förhindrar att en tråd blir blockerad av ett lås som tråden redan har:

```
public class ChainExample {
   public synchronized void methodA() {
      //do something
      methodB();
      //do something more
   }
   public synchronized void methodB() {
      //do something
   }
}
```

I och med att tråden är i besittning av låset när metoden methodA börjar exekvera och låset släpps först när exekveringen av metoden är klar är tråden i besittning av låset när den synkroniserade metoden methodB anropas.

Synkronisering av satser (1)

Om alla metoder är synkroniserade i ett objekt kan endast en tråd åt gången använda objektet.

För lite synkronisering

- konflikter mellan trådar
- inkonsistens på grund av tillgång till samma resurs (race condition)

För mycket synkronisering

- trådarna får vänta på varandra, ingen parallellism

Synkronisering av satser (2)

Det är möjligt att synkronisera enskilda satser med konstruktionen:

```
synchronized (obj) {
    <statements>
}
```

där obj är en godtycklig objektreferens.

För att att kunna exekvera satserna i <statements> måste tråden äga låset till objektet obj.

```
public static void aMethod(int[] n) {
   . . .
   synchronized(n) {
      for (int = 0; i < n.length; i = i + 1) {      // objektet är låst för
         if(n[i] > 100) {                           //andra trådar medan
            save(n[i]);                             //dessa satser utförs
         }
      }
   }
    . . .
}//aMethod
```

Synkronisering av satser (3)

Om satser som berör en primitiv variabel skall synkroniseras, associeras den primitiva variabeln med ett lås-objekt.

```
public class PrimaryTypeSynchronization {
   private int valueA;
   private double valueB;
   private Object lockA = new Object();
   private Object lockB = new Object();
   public void setA(int value) {
      synchronized(lockA) {
         valueA = value;
      }
   }//setA
   public double getA() {
      synchronized(lockA) {
         return valueA;
      }
   }//getA
```

låsobjekt

```
   public void setB(double value) {
      synchronized(lockB) {
         valueB = value;
      }
   }//setB
   public double getB() {
      synchronized(lockB) {
         return valueB;
      }
   }//getB
}//PrimaryTypeSynchronization
```

Samarbete mellan trådar

Synkronisering handlar om *uteslutning* inte samarbete.

Samarbete innebär:

vänta: när en tråd inte kan fortsätta, så låt andra trådar fortsätta

meddela: väck upp sovande trådar om det händer något som dessa kanske väntar på

- Metoden wait()
  - suspenderar tråden och öppnar låset för objektet
  - andra trådar får möjlighet att exekvera synkroniserade metoder
  - tråden väcks när någon tråd anropar notifyAll för objektet
- Metoden notifyAll()
  - signalerar till andra trådar som väntar på att objektet skall vakna

Anm: Det finns även en metod notify(), men den bör inte användas.

Producent-konsument exempel

Vi skall titta på ett mycket enkelt exempel på samarbete mellan två trådar. Vi har en klass CubbyHole som har en instansvariabel content, vilken kan förändras med metoden put och som kan läsas av med metoden get.

```
public class CubbyHole {
   private int content;
   public  void put(int value) {
      content = value;
   }//put
   public int get() {
      return content;
   }//get
}//CubbyHole
```

WRONG! **Inte trådsäker**

Vi har vidare en producent som sätter nytt värde på instansvariabeln och en konsument som avläser instansvariabeln. Vad vi vill åstadkomma är att konsumenten avläser instansvariabeln i *samma takt* som producenten förändrar dess värde.

Producent

```
public class Producer extends Thread {
   private CubbyHole cubbyhole;
   public Producer(CubbyHole c) {
      cubbyhole = c;
   }//constructor
   public void run() {
      for (int i = 0; i < 10; i++) {
         cubbyhole.put(i);
         System.out.println("Producer put " + i);
         try {
            Thread.sleep((int) (Math.random()*100));
         }
         catch (InterruptedException e) {}
      }
   }//run
}//Producer
```

Konsument

```
public class Consumer extends Thread {
   private CubbyHole cubbyhole;
   public Consumer(CubbyHole c) {
      cubbyhole = c;
   }//constructor
   public void run() {
      int value = 0;
      for (int i = 0; i < 10; i++) {
         value = cubbyhole.get();
         System.out.println("Consumer gets " + value);
         try {
            Thread.sleep((int) (Math.random() * 100));
         }
         catch (InterruptedException e) {}
      }
   }//run
}//Consumer
```

Testprogram

Vi skriver ett litet test program:

```
public class ProducerConsumerTest {
   public static void main(String[] args) {
      CubbyHole c = new CubbyHole();
      Producer p1 = new Producer(c);
      Consumer c1 = new Consumer(c);
      p1.start();
      c1.start();
   }//main
}//ProducerConsumerTest
```

Exempel på utskrift:

```
Producer put 0
Consumer gets 0
Consumer gets 0
Consumer gets 0
Producer put 1
Consumer gets 1
Producer put 2
Consumer gets 2
Producer put 3
Producer put 4
Consumer gets 4
Producer put 5
Consumer gets 5
Consumer gets 5
```

Problem

Producer och Consumer inverterar när de använder CubbyHole.

- Producer kan vara långsammare än Consumer.

```
Producer put 0
Consumer gets 0
Consumer gets 0
Consumer gets 0
Producer put 1
```

- Producer kan vara snabbare än Consumer.

```
Consumer gets 2
Producer put 3
Producer put 4
Consumer gets 4
```

Förslag på lösning: Synkronisera metoderna set och get i klassen CubbyHole

Ett första (och misslyckat) försök att lösa problemet

```
public class CubbyHole {
   private int content;
   private boolean empty = true;
   public synchronized void put(int value) {
      while(!empty) { } //vänta till empty blir true
      content = value;
      empty = false;
   }//put
   public synchronized int get() {
      while(empty) { } //vänta tills empty blir false
      empty = true;
      return content;
   }//get
}//CubbyHole
```

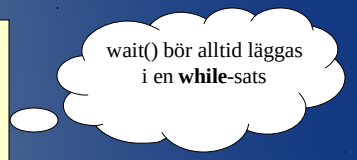
Exempel på utskrift från programmet:

```
Producer put 0
Consumer gets 0
Producer put 1
Consumer gets 1
Inget mer händer.
```

Vi har fått ***deadlock***!

En korrekt lösning

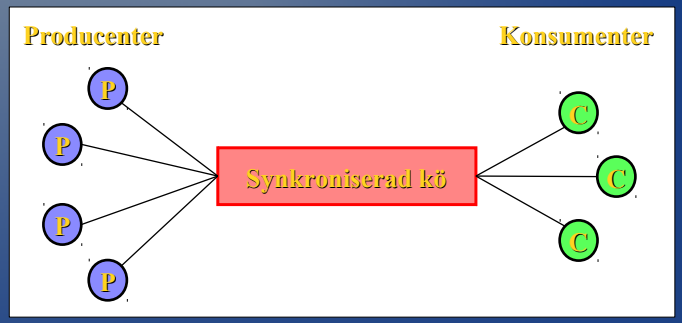
```
public class CubbyHole {
   private int content;
   private boolean empty = true;
   public synchronized void put(int value) {
      while (!empty) {
         try {
            wait(); //släpp låset och vänta på att väckas
         }
         catch (InterruptedException e) {}
      }
      content = value;
      empty = false;
      notifyAll(); //väck alla som väntar
   }//put
```




```
   public synchronized int get() {
      while(empty) {
         try {
            wait(); //släpp låset och vänta på att väckas
         }
         catch (InterruptedException e) {}
      }
      empty = true;
      notifyAll(); //väck alla som väntar på låset
      return content;
   }//get
}//CubbyHole
```

Flera producenter och flera konsumenter

Vanligt är att man har ett system med många producenter och många konsumenter. I ett sådant används en synkroniserad kö för att lagra de producerade enheterna.



Klassen Queue

```
import java.util.ArrayList;
public class Queue<T> {
   private ArrayList<T> queue = new ArrayList<T>();
   public int size() {
      return queue.size();
   }//size
   public synchronized void put(T obj) {
      queue.add(obj);
      notifyAll();
   }//put
      public synchronized T take() {
         while (queue.isEmpty()) {
            try {
               wait();
            }
            catch (InterruptedException e) {
               return null;
            }
         }
         T obj = queue.get(0);
         queue.remove(0);
         return obj;
      }//take
}//Queue
```

Klassen Producer

```
public class Producer extends Thread {
   private String name;
   private long interval;
   private Queue<String> queue;
   private int nr = 0;
   public Producer(String name, long interval, Queue<String> queue) {
      this.name = name;
      this. interval = interval;
      this.queue = queue;
   }//constructor
      public void run() {
         while (!Thread.interrupted()) {
            try {
               Thread.sleep(interval);
            }
            catch (InterruptedException e ) {
               break;
            }
            queue.put(name + ": " + nr++);
         }
      }//run
}//Producer
```

Klassen Consumer

```
public class Consumer extends Thread {
   private String name;
   private long interval;
   private Queue<String> queue;
   public Consumer(String name, long interval, Queue<String> queue) {
      this.name = name;
      this.interval = interval;
      this.queue = queue;
   }//constructor
      public void run() {
         while (!Thread.interrupted()) {
            try {
               Thread.sleep(interval);
            }
            catch (InterruptedException e ) {
               break;
            }
            System.out.println(name + " received job from "
                               + queue.take());
         }
      }//run
}//Consumer
```

Klassen JobShop

```
import java.util.ArrayList;
public class JobShop {
   private Queue<String> theQueue = new Queue<String>();
   private ArrayList<Thread> threads = new ArrayList<Thread>();
   public JobShop() throws InterruptedException {
      createAndStartThreads();
      Thread.sleep(20000);
      stopThreads();
      System.out.println("Jobs left in queue: " + theQueue.size());
      System.exit(0);
   }//constructor
  public void stopThreads() {
      for (Thread t : threads)
         t.interrupt();
   }//stopThread
```

```
   public void createAndStartThreads() {
      threads.add( new Producer("Lamm", 6237, theQueue));
      threads.add(new Producer("Alto", 2846, theQueue));
      threads.add(new Producer("Holm", 1239, theQueue));
      threads.add(new Consumer("Erik", 982, theQueue));
      for (Thread t : threads)
         t.start();
   }//cerateAndStartThreads
   public static void main(String[] arg) throws InterruptedException {
      JobShop j = new JobShop();
   }//main
}//JobShop
```