Programmering Fortsättning

Föreläsning 7

Joachim von Hacht

Innehåll

1 Processer

En process är ett program under körning (programmet självt är bara en samling bytes). En process behöver olika resurser t.ex.;

- Minne (heap, anropsstack, m.m.)
- Processorkraft.
- IO resurser, in och ut strömmar.

En process har också vissa säkerhetsattribut och ett tillstånd (kan t.ex. flyttas ut från arbetsminnet till virtuellt minne (disk)).

På ett system som kan köra flera processer samtidigt (multitasking) måste processerna fördelas antingen på flera processorer eller så får processerna dela på processorns tid (time sharing).

Hur tiden fördelas mellan processer sköts av en schemaläggare (scheduler). Finns många olika strategier (processer kan ha olika prioritet). Schemaläggare är helt

plattformsberoende. För att kunna flytta program får de inte vara beroende av schemaläggaren funktion (timing).

Olika processer är normalt helt separerade. Om två processer vill kommunicera sker detta med speciella tekniker, kallas "inter process communication". I Unix/Linux finns t. ex. pipes som möjliggör detta.

Att starta en process kräver en del arbete, en process är "tung" att starta.

1.1 Blockerande anrop

En process följer en sekvens av instruktioner. Om sekvensen stöter på ett anrop till en metod innebär detta normalt att resten av programmet (processen) får vänta tills metoden exekverat klart, kallas blockerande anrop.

- Om metoden tar lång tid måste "allt annat" vänta.

Kan vara irriterande t.ex. då man använder GUI:n. Om metoden tar lång tid (ladda ner fil) kommer hela gränssnittet att låsa sig tills filen är nerladdad (processen kan fortsätta med instruktionerna för att uppdatera GUI:et först efter att nedladdningen är klar).

KOD blocking

2 Trådar

En lösning på blockerande anrop är trådar[1]. Trådar gör att processen kan göra flera saker samtidigt (eller time sharing). Trådade program kan upplevas som mer responsiva[2].

- En tråd är en fristående "thread of execution" *inom en process*[3].
- En tråd är en lättviktsprocess det går snabbt att byta mellan trådar (jämfört med processer).
- Processen har flera (samtidigt) exekverande förlopp (parallell, concurrent-progam)

Trådar påminner om processer ovan men;

- Olika trådar kommer åt den gemensamma processens minne (variabler), dessa delas alltså mellan trådarna (shared resource). Ingen interprocesskommunikation behövs.
- Varje tråd har en egen anropsstack, lokala variabler delas inte!
- Programmets instruktioner (objektens metoder) kan köras av olika trådar, i ena stunden exekveras en metod av en tråd, i nästa körs de av en annan tråd (multi-threading). Hur detta växlar (context switch) kan vi som programmerare inte styra! Schemaläggaren byter trådar utifrån någon strategi (vi uppfattar det som slumpmässigt, vi vet aldrig vilken tråd som kommer att köras efter nästa byte).

Observation Trådade program ger en kraftigt ökad komplexitet. Flera parallella förlopp som kan påverka tillståndet.

D.v.s. undvik om möjligt trådade program.

3 Trådar i Java

(JLS 17)

[1] Finns även non-blocking IO i paketet nio (new IO). Används t.ex. om man skall skriva högkapacitetsservrar.

[2] Trådade program exekverar normalt inte snabbare, det tar tid att byta mellan trådar.

[3] Finns typer, native/green-threads, daemon-threads,..

- Alla Javaprogram har minst en tråd, den som börjar exekvera main-metoden(), kallas "main thread"[4].
- Använder man Swing skapas automatiskt ett antal trådar t.ex. händelsetråden.
- I Lab 2 kommer vi att se att RMI skapar trådar.
- Ett Java-program avslutas då samtliga trådar exekverat klart (main kan vara klar innan andra trådar! Ibland dyker utskrifter upp fast man tror man avslutat programmet...:-))[5].

3.1 Runnable

Alla klasser som är tänkta att exekveras i en egen tråd måste implementera interfacet Runnable med enda metod;

```
public void run();
```

All kod tråden skall exekvera måste finnas i implementering av run (vanligen en loop). Vissa standarklasser implementerar interfacet t.ex ...

3.2 ...Thread

- Klassen Thread representerar en tråd (dock är en tråd inte ett objekt! ..det är en thread of execution, blanda inte ihop...).
- Klassen implementerar som sagt gränssnittet Runnable.
- Ett antal metoder t.ex. start() som startar tråden.
- Metoden stop() skall aldrig användas. För att stoppa tråden (döda den) används boolesk variabel som bryter loopen i run.metoden. Därmed avslutas run och tråden dör, se vidare nedan.
- Thread ärver några grundläggande metoder från Object t.ex. wait, och notify (se vidare kanonisk form).

3.2.1 Statiska metoder i Thread

Finns ett antal användbara;

- Thread.currentThread() ger (namnet) på den tråd som exekverar koden.
- Thread.sleep(), söver metoden en viss tid (d.v.s. tråden exekveras inte under denna tid).

3.3 Skapa trådar

Kan skapas på flera sätt (vi kommer dock att använda färdig klasser på högre abstraktionsnivå).

- Skapa klass som implementerar Runnable. Skapa en instans av Thread och skicka klassen som argument till Threads kontruktor. Medför att koden i vår run-metod kommer att exekveras i tråden.
- Låt klassen ärva Thread (mindre bra).

KOD thread.basic

DEMO Eclipse debugger/Threads

3.4 Tillstånd

En tråd kan befinna sig i olika tillstånd;

New En ny oinitierad tråd.

[4] Finns trådar i bakgrunden t.ex. minnehanteringen men dessa räknar vi inte med.

[5] Förenklat, finns även s.k. daemon threads m.m.

Alive...

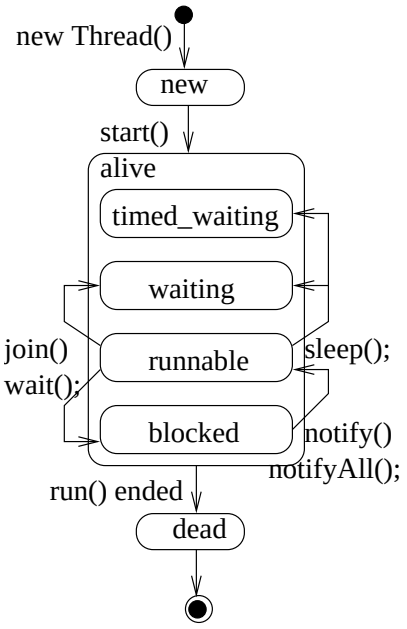
Runnable Tråden är initierad och klart att exekvera, detta tillstånd nås *efter det att metoden start() har anropats*. När exekveringen börjar/sker avgörs av schemaläggaren. Man skall alltså aldrig själv anropa run(), ger IllegalThreadStateException.

Blocked Inga instruktioner exekveras, tråden är tillfälligt stoppad. Vissa metoder försätter tråden i detta tillstånd. För att bli exekverbar igen finns åter andra metoder. Ofta pendlar tråden mellan runnable och blocked.

Waiting/Timed waiting Väntar på att en annan tråd skall bli klar eller att en viss tid gått (millisec.). Sköts också med metodanrop.

OBS! Att vi aldrig kan styra vilken tråd som skall exekvera, vi kan bara välja mellan runnable, blocked och waiting (finns inte thread[i] eller likande).

Dead run-metoden har exekverat klart. Tråden dör och kan aldrig återupplivas (måste skapas på nytt).



Bilden visar metodanrop och tillståndsövergångar. Finns fler varianter, wait kan gå till timed_waiting o.s.v. visas inte här.

3.5 Trådsäkerhet

Eftersom trådar delar tillstånd får de inte skriva/läsa hur som helst! Betrakta följande;

```
// Not thread safe
int x = 10;
x = x + 1;
```

Tilldelningen skall utföras av två trådar d.v.s. efter detta skall x vara 12.

1. Tråd A läser x värde från minnet och blir avbruten...
2. Tråd B läser x och blir avbruten...
3. Tråd A ökar x med 1 och skriver sedan till x (x = 11 alltså);
4. Tråd B gör som A d.v.s. x = 11, ...!

3.5.1 Race conditions

- Det ovan kallas "race-conditions", d.v.s. resultatet är beroende på hur schemaläggaren hanterar trådarna[6].

- Operationer (metoder) som *inte* leder till race conditions kallas trådsäkra (thread safe).

KOD threads.racecondition

3.5.2 Trådar och lokala variabler

Det tillstånd vi talar om rör som vanligt inte lokala variabler (vilka inte räknas med i tillståndet). Lokala variabler "lever" på stacken. För att detta skall fungera med trådar har varje tråd en egen stack.

3.6 Atomära operationer

Vissa operationer är garanterade att inte bli avbrutna (är trådsäkra). Dock får man se upp med cachening, se nedan;

- en enstaka läsning eller skrivning av en variabel, frånsett typerna long och double, är en atomär operation[7].

Följande är inte trådsäkert (inte atomärt);

```
// Not thread safe (3 operations)
x++;
```

[6] Ett vanlig fel kallas "check and act" d.v.s. man kontrollerar ett villkor och skall därefter utföra något. Precis när man kontrollerat villkoret bli tråden avbruten och en annan tråd ändrar villkoret.

[7] Finns klass AtomicLong.

3.7 Synkronisering

- Innebär att man "bakar ihop" ett kodstycke till en atomär operation. Nödvändigt då flera trådar skall anropa koden och denna använder delad resurser. Åstadkomms i Java med det reserverade ordet synchronized (ingår inte i metodsignaturen).

```
class A {
  // Synchronized method
  synchronized void doIt(){
     // Critical section
     // Shard resource used
  }
}

// Synchronized statement
synchronized(exp){
  // Critical section
  // Shard resource used
}
```

  Exp måste vara av referenstyp (vanligen ett enkelt objekt).

- Området mellan { och } kallas en kritisk sektion[8]. För metoden är alltså hela metoden en kritisk region.

- Om en tråd påbörjer en kritisk sektion är den garanterad att kunna exekvera klart sektionen. Byte mellan tråd sker före eller efter.

- Konstruktorer kan inte vara synchronized.

- OBS! Sammansättning av atomära operationer inte blir atomär.

[8] In concurrent programming a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

3.7.1 Teknik

- Alla Java-objekt är associerade med en monitor[9]. En tråd kan låsa eller låsa upp monitorn. Vi kan förenklat säga att alla objekt i Java har ett implicit "lås" (lock = intrinsic locks = built in locks[10]).
- En tråd måste "exklusivt äga" objektets lås innan den kan exekvera den kritisk regionen (d.v.s. om tråden lyckats så låser den objektet (och därmed regionen för andra trådar). Vem som äger ett lås bestämms av systemet [11].
  - För synkroniserade metoder är det låset för this tråden äger. Alla *synkroniserade* metoder i this kommer att vara låsta för andra trådar. Icke-synkroniserade kan köras.
  - En synkroniserad metod kan anropa en annan synkroniserad metod på objektet (det är ju samma tråd, samma lås, kallas "reentrant").
  - För synkroniserade satser är det låset för objeket i "parentesen" man äger.
- Låset släpps då tråden lämnar den kritiska regionen (bara den tråd som har låset kan släppa det).
- Om en region är låst av en tråd och en annan tråd försöker komma åt låset tvingas den att vänta tills låset släpps (läggs i ett s.k. wait set, eventuellt tillsammans med flera andra trådar,
  - OBS! Inte säkert att just den tråden får låset först när det släpps, någon annan tråd som väntar kan få det... Finns ingen turordning.
- En synkroniserad statisk metod innebär att man låser alla statiska klassmetoder (det finns ett klasslås).
- Synkronisering tar viss tid!
  - Behållare i Java finns ofta i två versioner en trådsäker lite långsammare och en icke-trådsäker.
- Det ovan kallas ofta "mutual exclusion", ömsesidig uteslutning.

KOD threads._synchronized

3.8 Synlighet

Ingen annan tråd kan se tillståndsförändringarna då en tråd exekverar en kritisk region (tråden har en cache).

- Först då tråden lämnar regionen blir resultatet synliga för andra[12]!
- Innebär att man ibland måste använda synchronized-metoder trots att man bara har en atomär skriv/läs operation.

3.8.1 Volatile

Ett nyckelord;

[9] Monitor en språkmekanism (kompilatorm, JVM) som automatisk hanterar kritiska regioner m.h.a. lås (mutex).

[10] Finns andra varianter med bibliotek.

[11] JLS 17.1 "The synchronized statement computes a reference to an object; it then attempts to perform a lock action on that object's monitor"

[12] "When an object acquires a lock, it first invalidates its cache, so that it is guaranteed to load variables directly from main memory. Similarly, before an object releases a lock, it flushes its cache, forcing any changes made to appear in main memory."

> "Volatile variables share the visibility features of synchronized, but none of the atomicity features."[13]

Kan användas i vissa situationer istället för lock's

3.9 Deadlock

> "deadlock refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain"

BILD

KOD threads.deadlock

4 Swing

Swing är inte trådsäkert!

> "The Swing API was designed to be powerful, flexible, and easy to use. In particular, we wanted to make it easy for programmers to build new Swing components, whether from scratch or by extending components that we provide.
>
> For this reason, we do not require Swing components to support access from multiple threads.[14]"// Muller Walrath (from Sun)

4.1 The Single thread rule

> "Once a Swing component has been realized (painted or to be painted on screen), all code that might affect or depend on the state of that component should be executed in the event-dispatching thread"

Alltså allt som påverkar GUI:et skall exekveras i "event-dispatching tråden" (EDT, som startas automatisk då man har en Swing application).

Vi går ett steg längre och konstruerar dessuotm hela GUI:et i EDT (se nedan).

4.2 SwingUtilities

För att hantera interaktionen mellan EDT och andra trådar kan man använda klassen SwingUtilities.

Om vi har en tråd som skall påverka GUI:et kan vi låta denna tråd "lämna över" till EDT genom att använda metoderna.

- SwingUtilities.invokeLater() (icke-blockerade anrop)
- SwingUtilities.invokeAndWait() (blockerade anrop)

KOD swingutilities

4.3 SwingWorker

Problemet ovan med att GUI:et låster sig vid tidsödande operationer kan lösas m.h.a. klassen SwingWorker (som kommer

[13] http://www.ibm.com/developerworks/java/library/j-jtp06197.html

[14] Single-threaded GUI frameworks are not unique to Java; Qt, NextStep, MacOS Cocoa, X Windows, and many others are also single-threaded. This is not for lack of trying; there have been many attempts to write multithreaded GUI frameworks, but because of persistent problems with race conditions and deadlock, they all eventually arrived at the single-threaded event queue model in which a dedicated thread fetches events off a queue and dispatches them to applicationdefined event handlers. //Java Concurrency in Practice

att starta en egen tråd, bakgrundstråden). Klassen är generisk och lite "mystisk"

```
public abstract class SwingWorker<T, V>
  extends Object implements RunnableFuture<T>
```

Typparametrarna står för:

- T returtypen för metoden doInBackground(), som körd i bakgrundstråden.
- V står för typen på mellanliggande resultat (innan doInBackground är klar, kan t.ex. ha en progress indicator)

Metoden done() kommer att exekveras i EDT. I denna kan man alltså uppdatera GUI:et. De värden man kan behöva (resultatet av doInBackground) får man genom att anropa metoden get()

5 Trådsäkra behållare

Metoder som lägger till/tar bort saker ut behållare behöver inte vara trådsäkra om behållaren är det. Följande gäller;

	syncronized	non-syncronized
List	java.util.Vector	java.util.ArrayList
	java.util.Stack	
		java.util.LinkList
	java.util.concurrent.CopyOnWriteArrayList	
Set		java.util.TreeSet
		java.util.HashSet
		java.util.LinkHashSet
	java.util.concurrent.CopyOnWriteArraySet	
Map		java.util.TreeMap
	java.util.Hashtable java.util.concurrent.ConcurrentHashMap	java.util.HashMap
		java.util.LinkHashMap
		java.util.IdentityHashMap
		java.util.EnumMap

Om man vill skapa en trådsäker variant av den behållara men har kan man använda;

```
Collection mySynchCollection =
  Collections.
   synchronizedCollection(myCollection);
```

6 Schemalagda händelser

Ibland har man behov av att vissa saker utförs med jämna mellanrum (i bakgrunden). Man kan m.h.a klasserna Timer och TimerTask schemalägga körningar.

6.1 util.Timer och TimerTask

(java.util.Timer, inte swing.Timer)

- Timer, A facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals (Timer är trådsäker).
- TimerTask, A task that can be scheduled for one-time or repeated execution by a Timer (implements Runnable).

Schemalagda saker bör gå relativt snabbt, annars tar den övriga schemalagdas task's tid.

KOD timertask

6.2 swing.Timer

Det finns en timer i Swing, används för animationer (sänder en ActionEvent enlig schemaläggningen). Eventen kan fångas i en vanlig Swing lyssnare. OBS! Att denna kod kommer att köras i EDT.

7 Design

- Undvik trådar om möjligt
- Trådsäkerhet åstadkomms genom;
  - Icke muterbara klasser.
  - Tillståndslösa klasser.
  - Användning av synkronisering;

  - Trådsäkerhet garanteras inte bara för att man använder synchronized, måste ske med insikt (variabler kan bero på varann, m.m.)
  - Synkronisering kostar i effektivitet.

- Deadlock
  - Man får se upp med synchronized, begränsa användning så mycket som möjligt. Eftersom vi låser objekt kan överdriven användning leda till deadlock.
  - Anrop "ut" från en synkroniserad metod måste ske med stor försiktighet, eventuellt kan det leda till en callback, som i sin tur leder till deadlock (objektet är ju redan låst av oss).

- Kan delegera trådsäkerhet, om man t.ex. använder trådsäkra behållare kanske inte metoden behöver vara trådsäker.