Realtidssystem

- Monitorer, synkroniserade metoder -

EDAF85 - Realtidssystem (Helsingborg)
Roger Henriksson

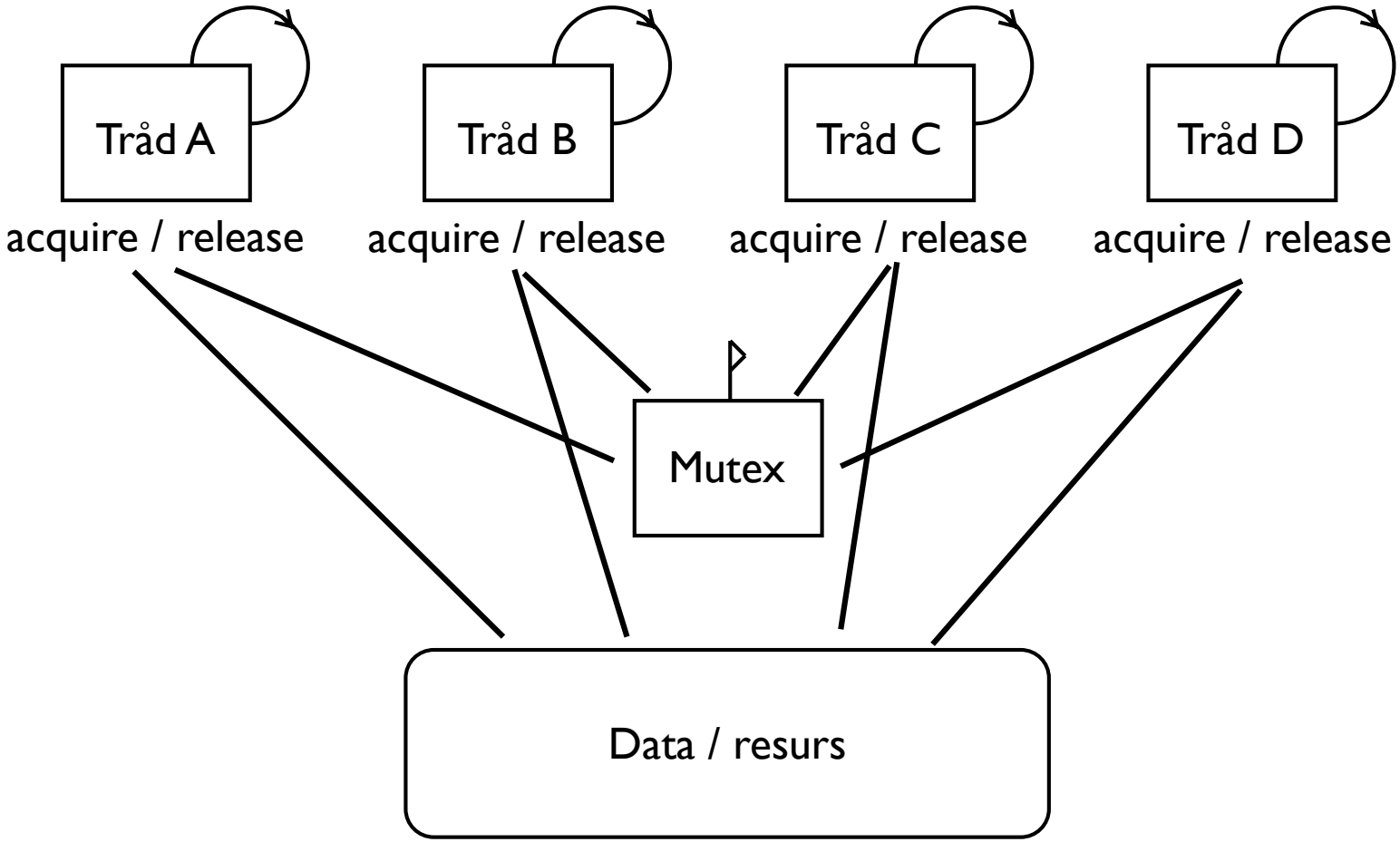
Föreläsning 3

Kursens innehåll motsvarar tidigare omgångar under beteckning EDA698
Föreläsningsbilder av Elin A. Topp
Stora delar baserad på: Föreläsningsmaterial EDA040 (Klas Nilsson, Mathias Haage) samt EDA698 (Mats Lilja)

Innehåll

- Monitorer
  - Monitorprincipen - objekt med inbyggd ömsesidig uteslutning
  - Monitorer i Java - `synchronized, wait & notify`

Semaforer och komplexa program



Acquire / *Release* sprids över alla trådar, beroendena blir många, det hela blir komplext och svårt att genomskåda för att hitta fel eller verifiera. (Jfr övning 1, uppgift 1 + 3).

Från semafor till monitor

```
class Buffer {
  Semaphore mutex;        // Mutual exclusion
  Semaphore free;         // For buffer full blocking.
  Semaphore avail;        // For blocking when no data is available.
  String buffData;        // The actual buffer.

  Buffer(){
    mutex = new Semaphore(1);
    free = new Semaphore(1);
    avail = new Semaphore();
  }

  void  putLine( String input) {
    free.acquire();                    // Wait for buffer empty.
    mutex.acquire();                   // Mutual exclusion start
    buffData = new String(input);      // Store copy of object.
    mutex.release();                   // Mutual exclusion end
    avail.release();                   // Allow others to get line.
  }

  String getLine(){
    avail.acquire();                   // Wait for data available.
    mutex.acquire();                   // Mutual exclusion start
    String line = buffData;            // Store copy of object.
    mutex.release();                   // Mutual exclusion end
    free.release();                    // Allow others to put line.
    return line;
  }
}
```

"Buffer" är ett passivt objekt (ingen tråd), som skyddar sina metoder / attribut för hantering genom olika trådar - i princip *en monitor* (jfr övning 1, uppgift 2, och labb1)

Monitor - koncept

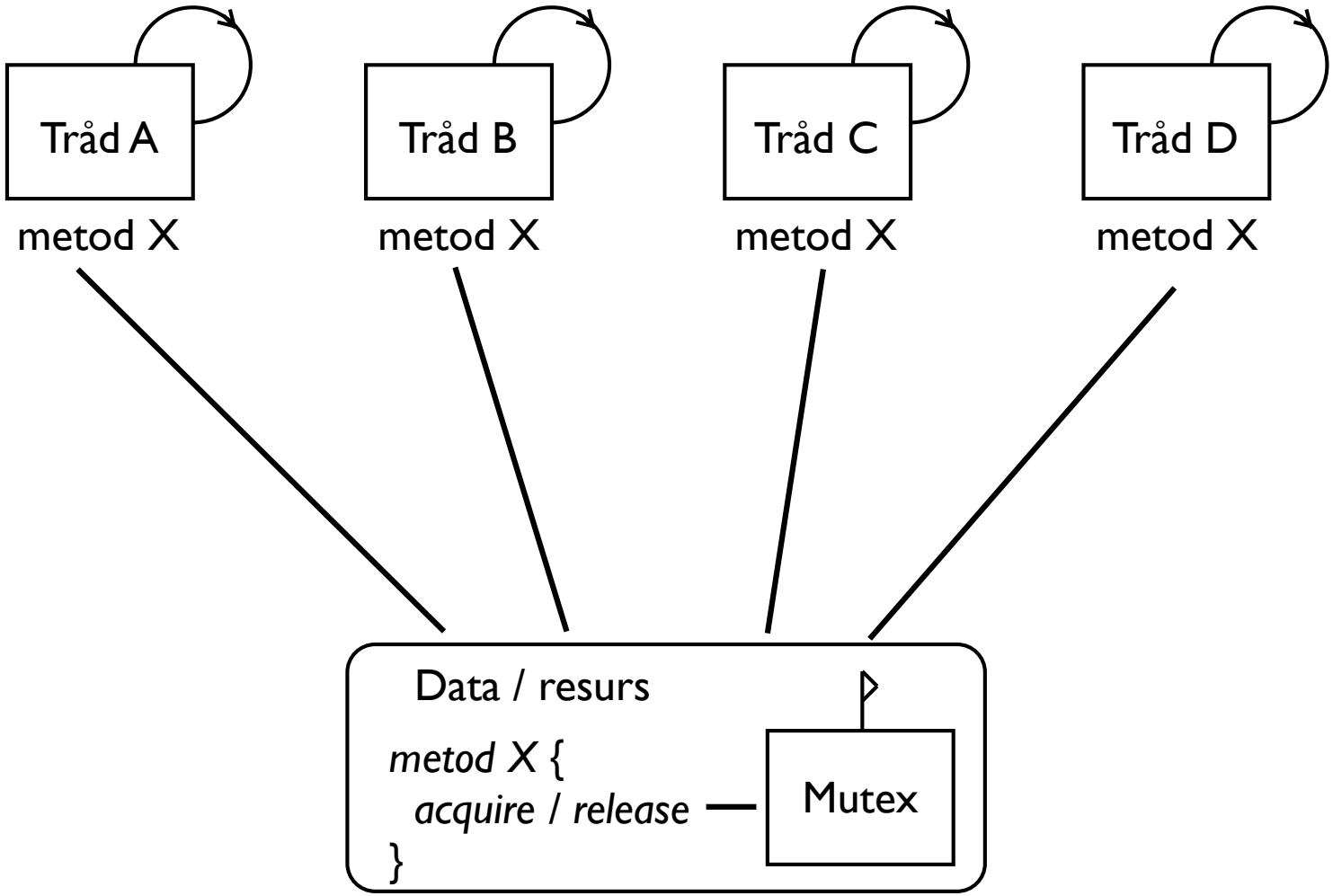
- I OOP använder vi oss av klasser, dvs datastrukturer som tillhandahåller egna metoder för hanteringen av interna data-attributer, för att garanterar kapsling och korrekt hantering.
- Klasser, som inte bara har metoder, men implementerar metoder så att attributer / kritiska sekvenser av kod blir skyddade vid samtidiga anrop från olika trådar, kallas för monitorer

- Metoder som

```
class Account {
  // ...
  void deposit(int a){
    mutex.acquire();
    balance += a;
    mutex.release();
  }
}
```

implementerar monitor-konceptet genom användning av semaforer

Monitorer och komplexa program



Acquire / *Release* hanteras inuti det passiva objektet. Trådarna anropar metoden som de behöver utan att behöva bry sig om den gemensamma hanteringen av data / resurser .

Programmeringsspråk - stöd för monitorer

Problem: Att använda semaforer i en monitor kräver fortfarande (för mycket) disciplin.

- Idé: Erbjuda stöd genom speciella språk-konstruktioner
- Olika grader av stöd i olika språk:
  - **Inget stöd** (C / C++): Direkta anrop till biblioteksfunktioner. Objekt-orientering kan förenkla användningen.
  - **Explicit** per metod (Java): Deklarerade egenskaper av metoder (med både språk- och run-time stöd)
  - **Implicit** per task (Ada): Deklarerade egenskaper av klasser, som överförs implicit till alla attributer och metoder i klassen
- **Inget stöd** resulterar i mera komplicerade programmering. **Implicit** språk-stöd är säkrast och enklast, men kan leda till begränsningar. Java-ansatsen (**explicit** och möjlighet till deklaration av metoder med "inbyggd" ömsesidig uteslutning) är en pragmatisk lösning till problemet.

Monitorer i Java - `synchronized`

- Kritisk sekvens / region / kod-block / metod är markerad med nyckelordet `synchronized`
- En liten nackdel: Vare sig klasser eller attribut kan deklareras `synchronized`, det krävs alltså lite disciplin när man implementerar metoderna i en klass!
- Monitor-konceptet uttryckt med Javas monitor-mekanism:

```
class Account {
  // ...
  public synchronized void deposit( int a){
    balance += a;
  }
}
```

- Betyder att ingen annan tråd kan köra samma, eller annan metod deklarerad `synchronized`, i samma ***objekt***.

Objektkategorier

- "Enkelt" passivt objekt
  - *Trådsäker* genom att ha metoder som är *reentrant* (dvs, som inte hanterar klass-attribut, t ex `java.lang.Math`)
  - Explicit *icke trådsäker;* får bara användas av en enda tråd (`java.util.HashSet`)
  - Implicit *icke trådsäker*; måste antas om inte dokumenterad
- Monitor-objekt
  - Metoder som implementerar ömsesidig uteslutning, t ex genom `synchronized`
  - Borde vara passivt; blanda INTE monitorer och trådar!
- Tråd-objekt
  - Aktivt objekt (efter start(), innan det avslutas); driver exekveringen av metoder i passiva objekt
  - "Don't call me, I'll call you!" (bortsett från "meta-metoder" som t ex *start*() och *join*())

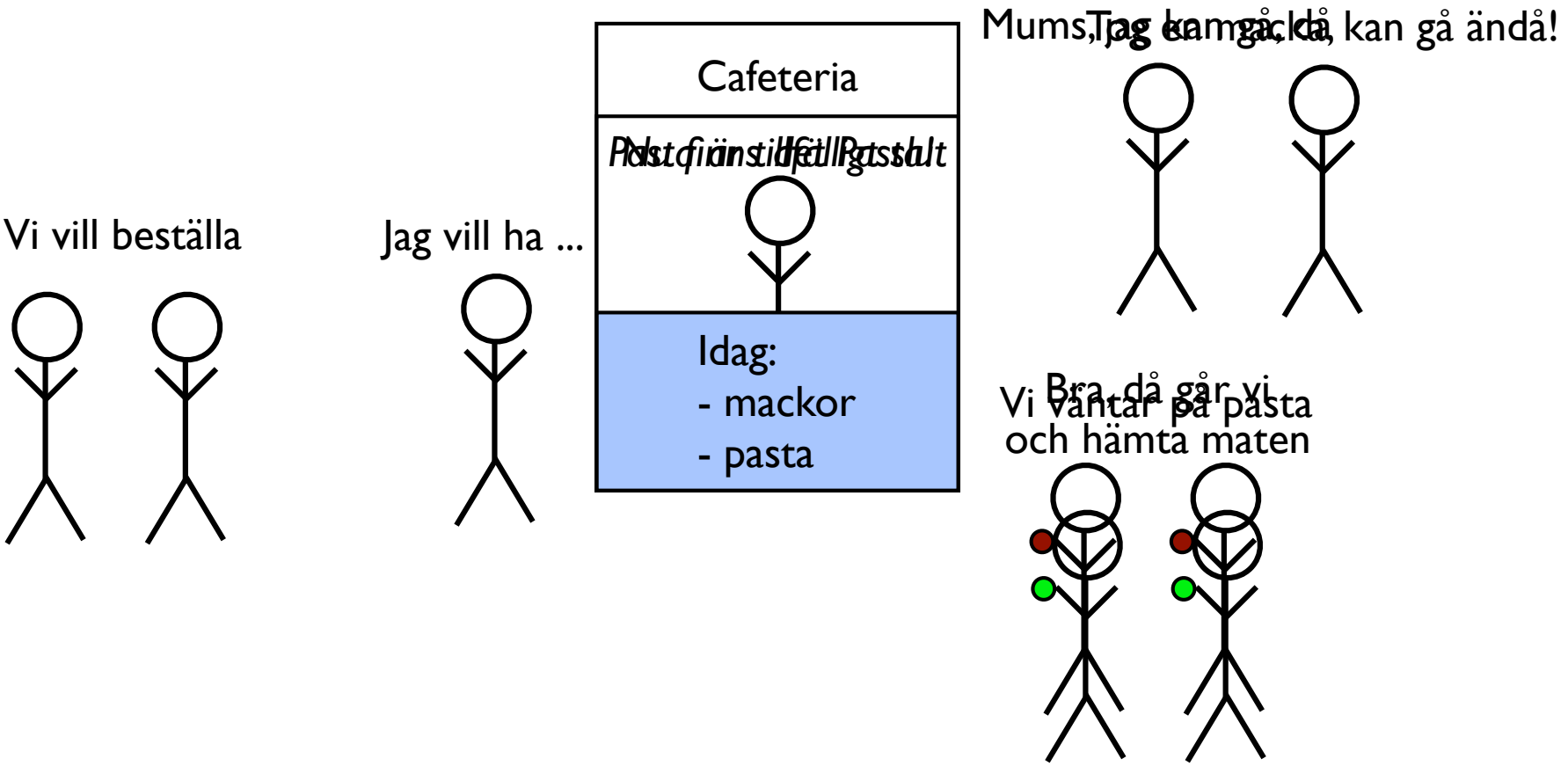
Användningsregler

- Blanda INTE tråd och monitor i samma klass!!!
- I en monitor: ALLA metoder som deklareras *public* borde normalt vara `synchronized`
- Tråd-osäkra klasser kan och BORDE paketeras i en monitor (wrapper class)
  - Använd INTE (utspridda) `synchronized`-kodblock (detaljer i kompendiet!):

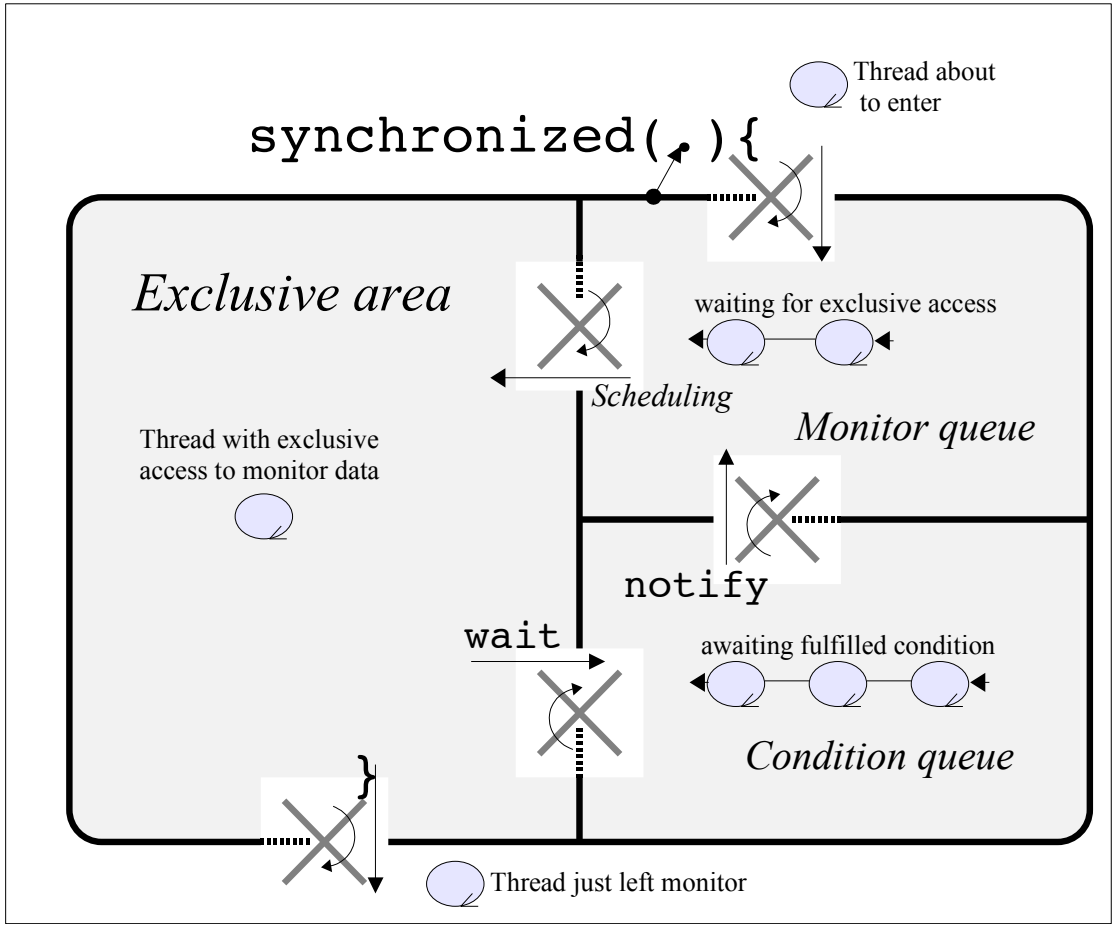
```
…
synchronized(object) {
    …
}
```

Samtidigt, men med olika krav

`synchronized` erbjuder ömsesidig uteslutning - räcker det för resurshantering?

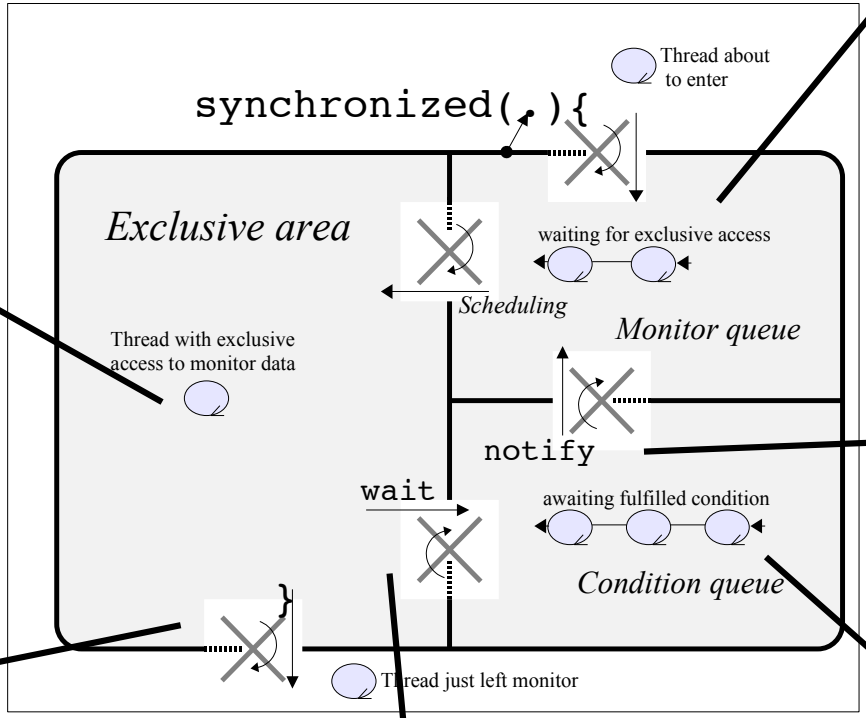


synchronized - wait - notify



Cafeteria med `synchronized, wait och notify`

Kö för att kunna beställa eller hämta beställt mat - `synchronized` ordnar med ömsesidig uteslutning, bara en kan prata med personalen.



Den som står vid disken frågar om maten hen vill ha finns, dvs beställer eller hämtar mat

När brickorna blir gröna (`notify()`) kan en, eller flera (`notifyAll()`), ställa sig i kö igen för att hämta ut maten

Den som vill ha mat som är tillgängligt får det och kan gå

Pasta finns inte just nu och personen får en röd "väntebricka" (`wait()`).

De med röda brickor väntar på att brickorna blir gröna

Exempel: Hiss

Personer kommer till hissen slumpmässigt

med knapptryck meddelar de hissen, att de vill åka från ett ställe

hissen rymmer enbart ett visst antal personer

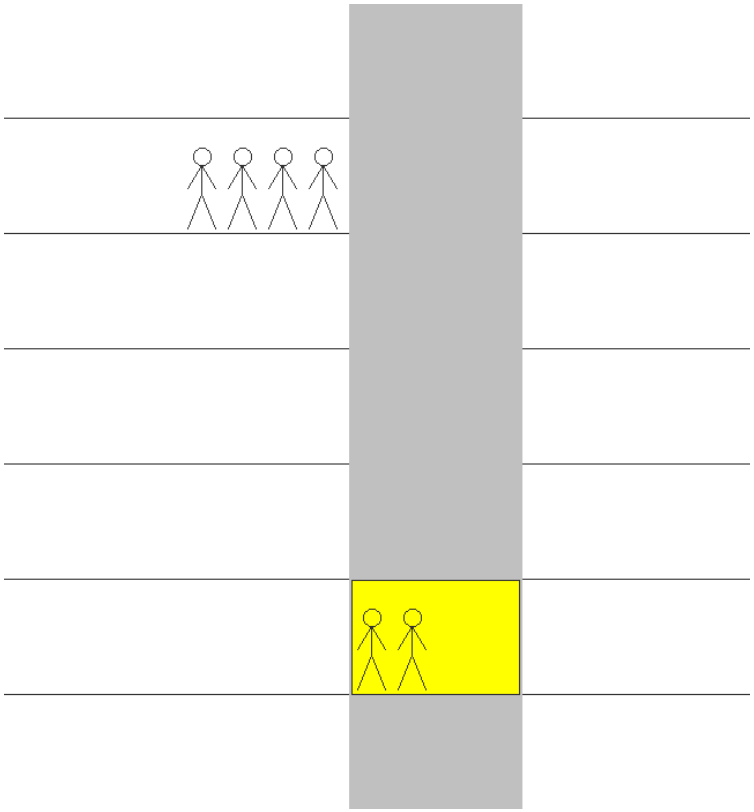
personer väntar på att hissen kommer, och har plats

i hissen trycker man på en knapp för att visar vart man vill åka

personer i hissen väntar på att hissen når rätt våning

den enkla hissen åker från våning 0 till våning 6 och tillbaka hela tiden, med uppehåll vid varje våning

hissen väntar vid varje våning på att alla som vill kan gå av och (i mån av plats) på.



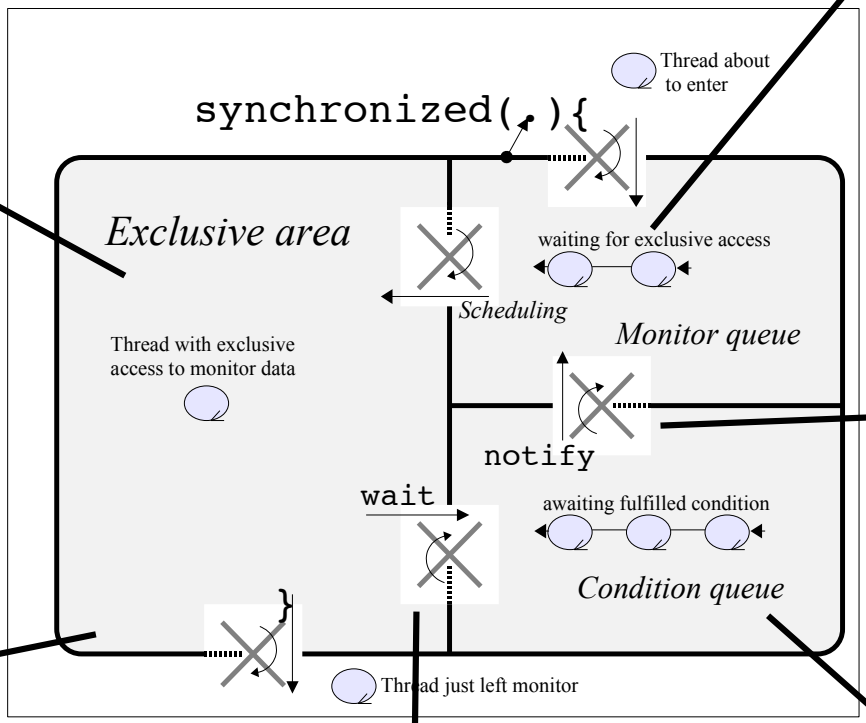
Exempel: Hiss
- många med olika krav, en resurs, en motor -

På våningen, framme vid hissen: Trycka på knappen, kolla om hissen är där och det finns plats i den

På våningarna: kö för att kunna trycka på knappen
+ kö för att kunna kliva på när hissen är på rätt våning

I hissen: vänta tills man kan lämna (dörren är öppen)

I hissen: trycka på knappen för våningen man vill åka till, kolla om hissen är där man vill åka till



Hissen: När den kommer till en ny våning, meddelas detta till de som väntar

I hissen: har man nått våningen man vill åka till, kliver man av (när dörren är öppen)

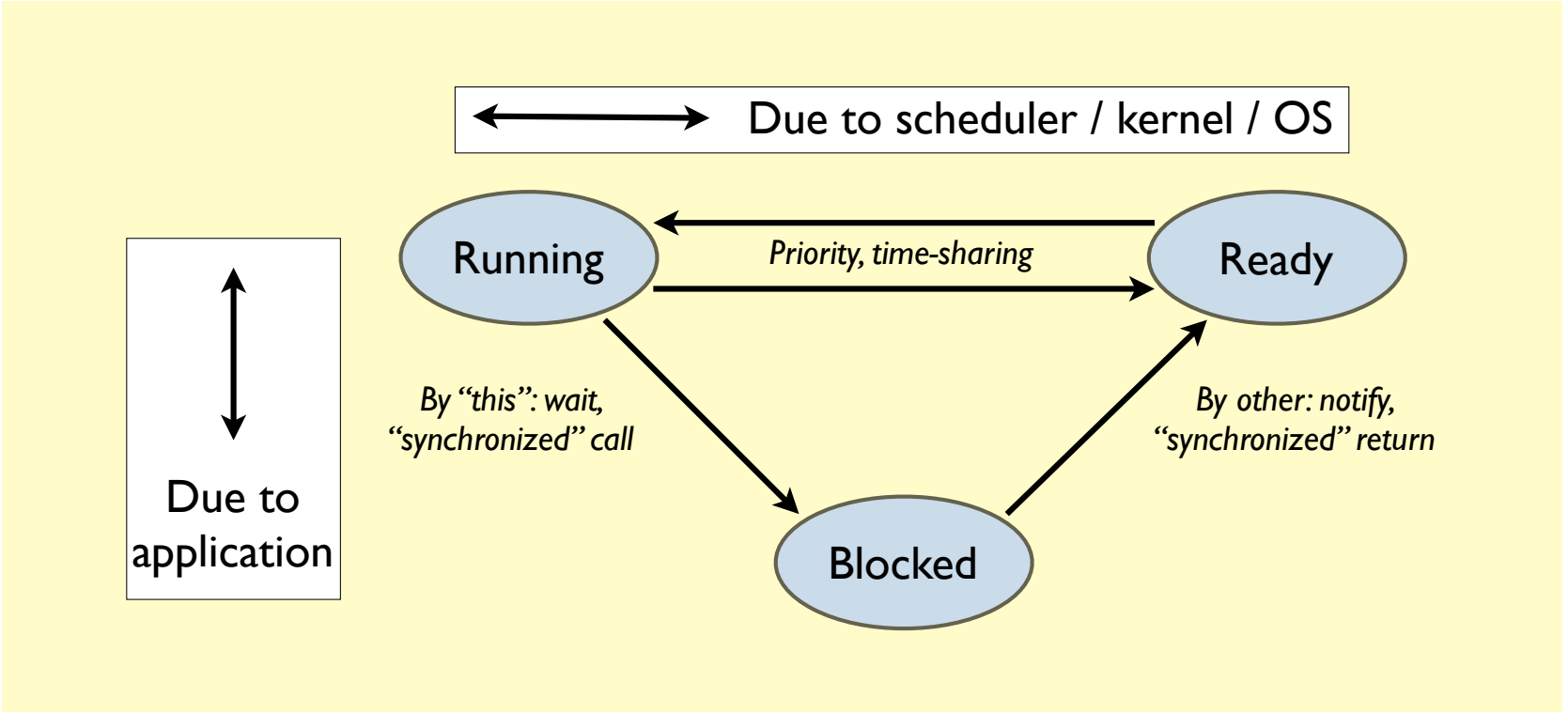
På våningen: Hissen är inte där, eller har inte plats

I hissen: rätt våning är inte nått

På våningen: vänta tills hissen kommer

I hissen: vänta på att hissen når rätt våning

Exekveringstillstånd



Schemaläggningstillstånd (scheduling state)

- Running
- Ready (annan tråd returnerar från *synchronized* metod, eller anropar *notify() / notifyAll()*)
- Blocked (tråden själv anropar *synchronized* metod, eller *wait()* i metoden)

acquire/release och wait/notify

Semaforer har tillstånd

Jämför

P1	P2
*	*
s.acquire();	*
*	s.release();
*	*

med

P1	P2
*	*
*	s.release()
s.acquire();	*
*	*

P1 fortsätter i båda fallen, då semaforens interna tillstånd förändras med release

Notification har INGET tillstånd

Jämför

P1	P2
*	*
wait();	*
*	notify();
*	*

med

P1	P2
*	*
*	notify();
wait();	*
*	*

P1 väntar tills nästa "notify" ... och väntar… Tillstånd måste implementeras med egna attribut och villkor för "wait".

Vem ska väckas? `notify()` eller `notifyAll()`

- `notify()`
  - Väcker upp en tråd åt gången, dvs den som är nästa tråd att köra
  - "Nästa tråd att köra" avgörs från schemaläggaren efter prioriteter, turordning, osv beroende på schemaläggningspolicyn
  - Svårt att avgöra om man faktiskt väcker "rätt" tråd, om man vet att en viss sådan väntar på en viss uppdatering i monitorn
  - Borde användas enbart om man verkligen behöver det för att dra upp prestanda
- `notifyAll()`
  - Väcker *alla* trådar som väntar på att ett krav uppfylls
  - Trådarna köas upp för CPU-tillgång och får sen tävla om tillgång till monitorn, som ju också är skyddad med `synchronized`
  - Enkelt och säkert att använda, alla som behöver det får en "notify"
  - Borde väljas som "standard", framförallt i den här kursen!

En (riktigt) dålig *buffer*

```
class MyBuffer {

  // ...
  public synchronized void post( String text){
    if( bufferFull) wait();

    buffData[nextToPut] = new String( text);
    bufferEmpty = false;
    if( ++numOfTextBits >= maxNumOfTexts) bufferFull = true;
    if( ++nextToPut == maxNumOfTexts) nextToPut = 0;

    notifyAll();
  }

  public synchronized String fetch(){
    if( bufferEmpty) wait();

    String res = buffData[nextToGet];
    buffData[nextToGet] = null;
    bufferFull = false;
    if( --numOfTextBits < 1) bufferEmpty = true;
    if( ++nextToGet == maxNumOfTexts) nextToGet = 0;

    notifyAll();
    return res;
  }
}
```

`if ( ...) wait();` gör det hela väldigt instabilt, om någon annan tråd kommer emellan kan eventuellt kravet inte vara uppfyllt efter `wait()` ändå, eftersom alla (producers) väcks! Skulle notify() lösa problemet? NEJ!!!

Dessutom: Spontan väckning ur **`wait()`** kan förekomma.

Mycket bättre

```
class MyBuffer {

  // ...
  public synchronized void post( String text){
    while( bufferFull) wait();

    buffData[nextToPut] = new String( text);
    bufferEmpty = false;
    if( ++numOfTextBits >= maxNumOfTexts) bufferFull = true;
    if( ++nextToPut == maxNumOfTexts) nextToPut = 0;
    notifyAll();
  }

  public synchronized String fetch(){
    while( bufferEmpty) wait();

     String res = buffData[nextToGet];
     buffData[nextToGet] = null;
     bufferFull = false;
     if( --numOfTextBits < 1) bufferEmpty = true;
     if( ++nextToGet == maxNumOfTexts) nextToGet = 0;

     notifyAll();
     return res;
   }
}
```

`while ( ...) wait();` gör buffer robust ifall någon annan tråd kommer emellan och ändrar tillståndet i monitorn, men det är lite onödigt ofta som det notifieras!

Riktigt bra (och effektivt)

```
class MyBuffer {

  // ...
  public synchronized void post( String text){
    while( bufferFull) wait();

    buffData[nextToPut] = new String( text);
    if( bufferEmpty) {
      notifyAll();
      bufferEmpty = false;
    }
    if( ++numOfTextBits >= maxNumOfTexts) bufferFull = true;
    if( ++nextToPut == maxNumOfTexts) nextToPut = 0;
  }

  public synchronized String fetch(){
    while( bufferEmpty) wait();

    String res = buffData[nextToGet];
    buffData[nextToGet] = null;
    if( bufferFull) {
      notifyAll();
      bufferFull = false;
    }
    if( --numOfTextBits < 1) bufferEmpty = true;
    if( ++nextToGet == maxNumOfTexts) nextToGet = 0;
  }
}
```

`while ( ...) wait();` gör buffer robust ifall någon annan tråd kommer emellan och ändrar tillståndet i monitorn, samt är effektiv - signalerar bara vid faktiska ändringar i tillståndet

Monitor == Semafor (?)

Semaforer och monitorer är teoretiskt likvärdiga!

- Semaforerna vi använder kan implementeras genom att använda en monitor med metoderna acquire() och `release()` och en heltalsvariabel.

- Monitorer kan implementeras med hjälp av semaforer, åtminstone för ett känt antal trådar, genom att använda en mutexsemafor på monitor-objektet och en räknande semafor för varje tråd som vill ha tillgång till monitor-objektet.

Därmed kan man alltid överföra den ena mekanism i den andra (om än möjligen på ett ganska krångligt sätt).

Beroende på problemet man vill lösa, ska man välja rätt ansats.

Pragmatiskt här i kursen:
Väckarklockelabben - använd semaforer,
Hisslabben - använd monitor (synchronized - wait - notify)

Enkel semafor

```
class SimpleSemaphore {
  private int counter;

  public SimpleSemaphore(int initialValue){
    counter=initialValue;
  }

  public synchronized void acquire(){
    while( counter==0 ) wait();
    counter—;
  }

  public synchronized void release(){
    counter++;
    notifyAll();
  }
}
```

Sammanfattning

- Semaforer på återseende
- Monitor-begreppet
- monitorer i Java: `synchronized – wait – notify`
- Exempel för användning av monitorer
- Regler och tips för monitorer
  och mekanismerna `synchronized, wait, notify`

- Kunna lösa övning 2 (monitorer) och laboration 2
- Kunna använda både semaforer och monitorer i enkla program

- Läs- och tittips:
  - e-bok: Kap5 (avsnitt 5.6 och 5.7, s 130 -140, Monitors / Summary)
  - kompendium: Kap 2.3 (Monitors)
  - Kortfilmer på kurshemsidan (”Föreläsningar och övningar”)