# Realtidssystem

## - Introduktion, jämlöpande exekvering -

EDAF85 - Realtidssystem (Helsingborg)
Roger Henriksson

Föreläsning 1

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)

# Om kursen

- Upplägg:
  - 7-8 föreläsningar
  - 4 övningar
    - uppgifter, både teoretiska och praktiska, delvis från gamla tentor, gås igenom, laborationsuppgifterna introduceras
  - 3 obligatoriska laborationer
    - fördelade på 6 veckor, dvs 2 veckor per laboration
    - görs i grupper om två studenter
    - räkna inte med att den schemalagda tiden räcker utan ni måste arbeta med laborationerna på egen hand i förväg
    - på laborationstiderna har ni möjlighet att få hjälp och redovisa era lösningar
  - Zoomhandledning

# Om kursen (2)

- ALL information på http://cs.lth.se/edaf85
  - Allmän information, laborationsanvisningar, ex-tentor
  - Föreläsningsbilder
  - Nyheter (kolla regelbundet)
- "Obligatoriskt" Kursmaterial
  - E-bok: I.C. Bertolotti & G. Manduchi, "Real-time Embedded Systems" (CRC Press, 2012)
  - "Multithreaded programming in Java" (Elin A. Topp)
  - Laborationshäfte
  - Övningsuppgifter

# Kursöversikt

- Realtidssystem - jämlöpande processer under tidsgarantikrav (Föreläsning 1)
  - Jämlöpande exekvering, trådhantering, hantering av delade resurser
    - ömsesidig uteslutning, signalering, datahantering, trådkommunikation

      **semafor**:
      föreläsning 2
      övning 1
      laboration 1

      **monitor**:
      föreläsning 3
      övning 2
      laboration 2

      **mailbox**:
      föreläsning 4

      laboration 3

  - Tidskrav, systemhantering, schemaläggning
    - resursallokering, schemaläggning, prioritetshantering, schemaläggningsanalys

      **dödläge**:
      föreläsning 5
      övning 3

      **schemaläggning + analys**
      föreläsning 6, 7
      övning 4

# Innehåll

- Introduktion till *Jämlöpande exekvering*
  - Sekventiell bearbetning i en parallel värld
  - Hantera olika uppgifter samtidigt - jämlöpande exekvering
- Trådar, processer, parallellitet och Java
  - Pre-emption, kontextbyte
  - Trådar i Java
  - Hantera gemensamma resurser, kapplöpning, ömsesidig uteslutning

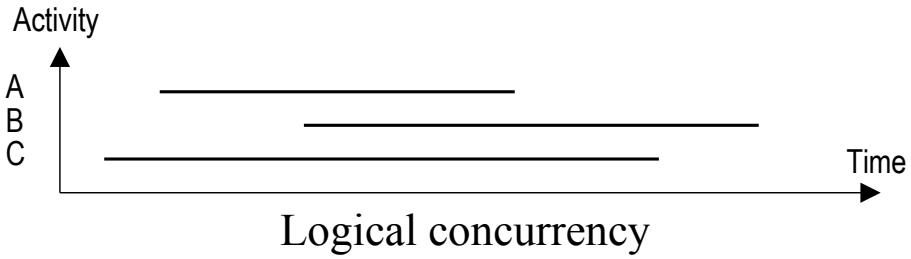
# Innehåll

- Introduktion till *Jämlöpande exekvering*
  - Sekventiell bearbetning i en parallel värld
  - Hantera olika uppgifter samtidigt - jämlöpande exekvering
- Trådar, processer, parallellitet och Java
  - Pre-emption, kontextbyte
  - Trådar i Java
  - Hantera gemensamma resurser, kapplöpning, ömsesidig uteslutning

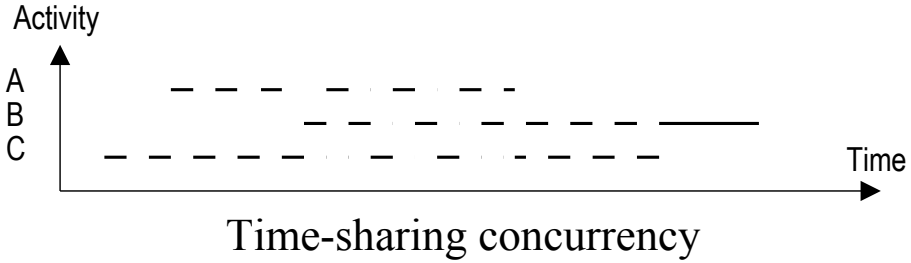
# Den parallella världen

- Flera "processorer":
  - Laga mat medan en annan dukar
  - En kör, en läser kartan
- En "processor":
  - Stryka kläder och lyssna på musik
  - Läsa kartan och gå mot målet i besvärlig terräng

# Jämlöpande aktiviteter



Logical concurrency

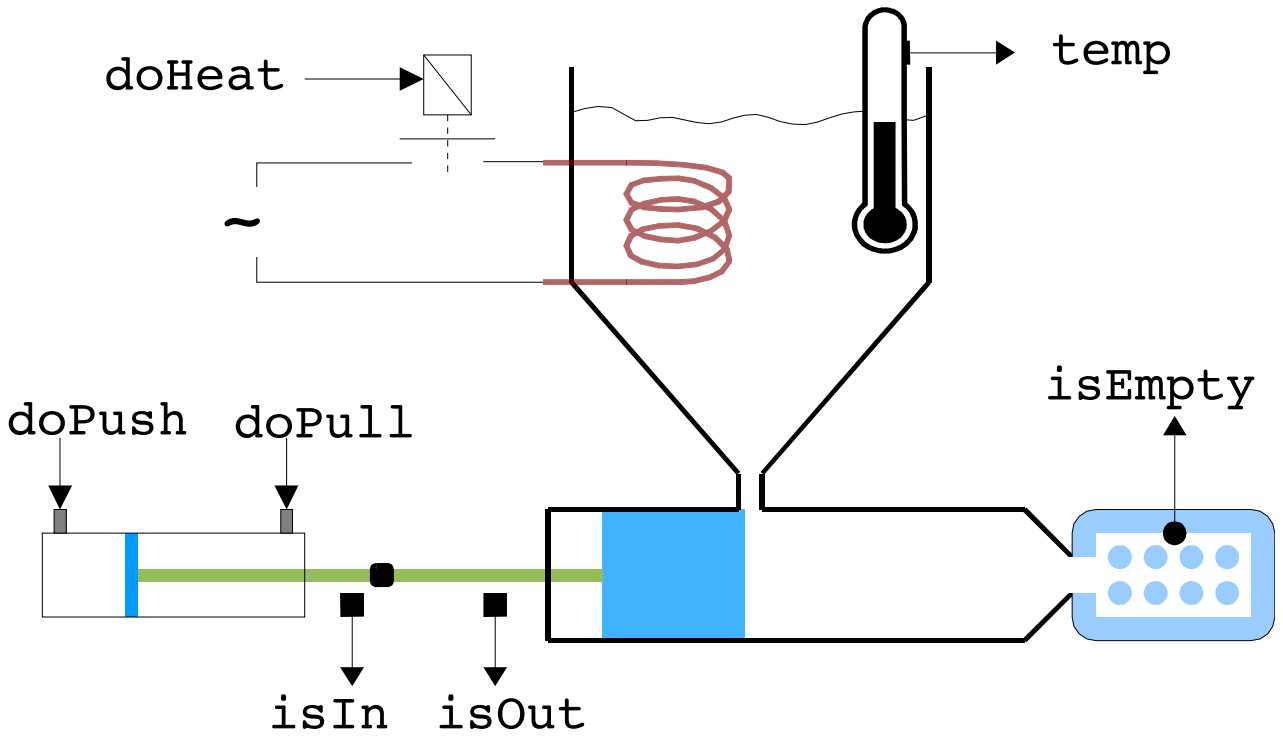


Time-sharing concurrency

- Laga mat medan en annan dukar? ✓
- En kör, en läser kartan? ✓
- Stryka kläder och lyssna på musik? ✓
- Läsa kartan och gå mot målet i besvärlig terräng? ⚠

# Maskinen som gjuter LEGO-klossar

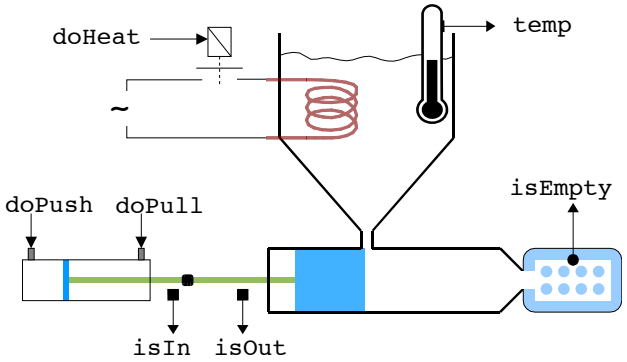
Styra temperaturen i plastmassan (period) och pistongen (evenemang) samtidigt, utan att det blir fel i en av styrningarna.



# Maskinen som gjuter LEGO-klossar

I ett program:

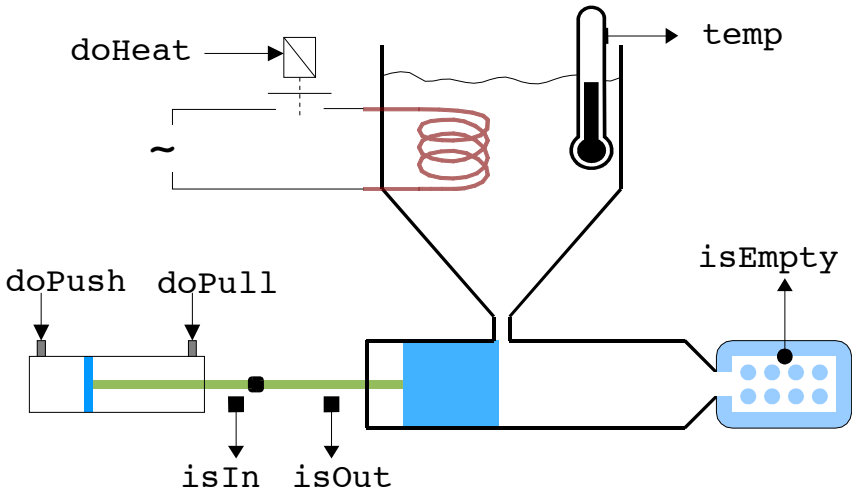
```
// börja med pistongen bak
while( true) {
     while( !isEmpty){
          tempControl();
          sleep( tsamp);
     }
     on( doPush);
     while( !isOut) {
          tempControl();
          sleep( tsamp);
     }
     off( doPush);
     on( doPull);
     while( !isIn) {
          tempControl();
          sleep( tsamp);
     }
     off( doPull);
}
```



# Maskinen som gjuter LEGO-klossar

Mera naturligt: I två program...

```
// Aktivitet 1: Temperatur
while( true) {
  if( temp > max)
    off( doHeat);
  else if( temp < min)
    on( doHeat);
  sleep( tsamp);
}

// Aktivitet 2: Pistong
while( true){
  await( isEmpty);
  on( doPush);
  await( isOut);
  off( doPush);
  on( doPull);
  await( isIn);
  off( doPull);
}
```



# “Real-world” objekt och aktioner

## Inbyggda datorer och deras styrning (mjukvara):

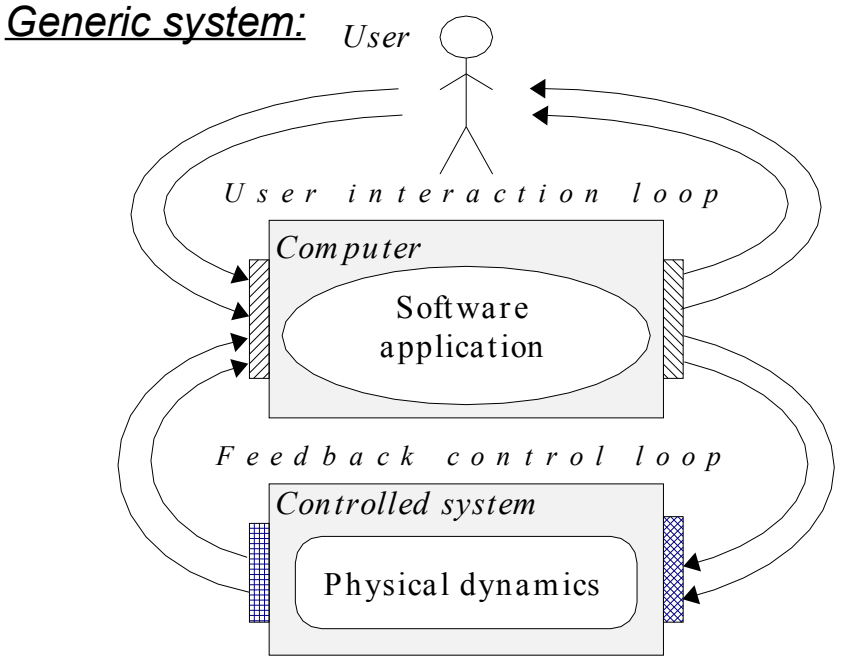
***Sekventiella*** *program som körs jämlöpande (och i realtid)*
*för att styra en* ***parallell*** *omgivning*

Vi måste tänka både sekventiellt och parallellt när vi bygger system

# Realtidskrav

Ett realtidssystem måste

- utföra alla beräkningar *logiskt korrekt*
- reagera på inmatningar *jämlöpande*
- alltid ha *konsistent* data att arbeta med
- producera alla (del-)resultat *i tid*

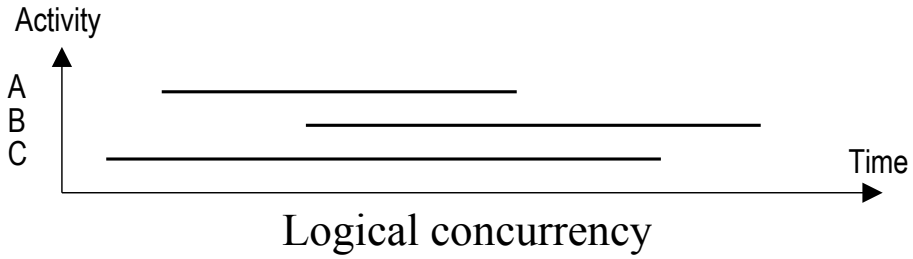



För att uppnå ***"realtids-korrekthet" (real-time correctness)*** måste mjukvaran säkerställa, att det "jämlöpande-korrekta" (concurrency-correct) ***resultatet produceras pålitligt i tid***.

# Innehåll

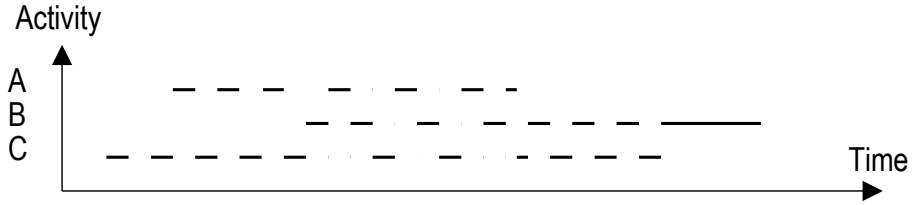
- Introduktion till *Jämlöpande exekvering*
  - Sekventiell bearbetning i en parallel värld
  - Hantera olika uppgifter samtidigt - jämlöpande exekvering
- Trådar, processer, parallellitet och Java
  - Pre-emption, kontextbyte
  - Trådar i Java
  - Hantera gemensamma resurser, kapplöpning, ömsesidig uteslutning

# Jämlöpande exekvering av sekventiella processer



Logical concurrency

En processor måste delas av flera aktiviteter - genom att dela över tid i väldigt små enheter, får man något som ser "samtidigt" ut.



Time-sharing concurrency

# Kontextbyte
## (Context switch)

- *Kontexten* av ett program / en tråd är all specifik data som tråden behöver ha med sig – dess aktuella tillstånd
- När systemet byter från en löpande tråd till den nästa händer det alltså ett *kontextbyte* (*context switch*), dvs all information som gäller den gamla löpande tråden sparas undan för senare vidarebearbetning, och all data tillhörande den nya löpande tråden plockas fram.
- I en typisk s.k. *pre-emptive* ("avbrytbar") OS-kärna kan det se ut så här:

| | |
|---|---|
| Turn off interrupts | } Save |
| Push PC, then CPU registers on stack | |
| Save stack pointer in process record | } Switch |
| Get new process record and restore stack pointer from it | |
| Pop CPU registers, then PC from stack | } Restore |
| Turn on interrupts | |

- Varje tråd har alltså sin egen *stack*, som blir allokerad vid trådens skapande.

# Pre-emption

Det finns olika strategier för tillåtelse av kontextbyten:

- *Non-pre-emptive scheduling* ("*icke-avbrytbar*" schemaläggning): Tråden som "kör" kan inte avbrytas tills den släpper CPUn frivilligt
    - explicit genom att anropa *yield()* eller
    - implicit genom (*synchronized*) operationer som kan blockera.
- *Pre-emption point based scheduling* ("*avbrytningspunktbaserad*" schemaläggning): Tråden som "kör" kan avbrytas vid vissa punkter i programmet (definierad genom språket eller run-time systemet)
- *Pre-emptive scheduling* ("*avbrytbar*" schemaläggning): Tråden som "kör" kan avbrytas när som helst av schemaläggaren (som styrs av hårdvaru-interrupts).

För att det ska bli "rätt" med schemaläggningen och väntetiderna, antar våra program att kärnan är pre-emptive, dvs. trådar kan avbrytas vid behov och då hanterar systemet kontextbytet på ett korrekt sätt.

# class Thread

`java.lang.Thread`

- En tråd är ett *aktivt objekt*, medan ett vanligt objekt med metoder som anropas inom en sekvens kallas för *passivt objekt*.
- Metoden *run()* utför trådens uppgifter (som i ett passivt objekt), men bara metoden *start()* som anropar *run()* ger tråden sitt eget liv.
- Två vägar att implementera en tråd, dvs implementera metoden *run()*:
  - implementera interfacet `Runnable` eller ärva från `class Thread`

# Använda Runnable

```
public interface Runnable {
    void run();
}
...
class MyRunnableObject implements Runnable {
    void run() {
        /* Do loads of stuff the thread is supposed to do */
    }
    /* implement other stuff that is needed by the thread */
}
...
MyRunnableObject myR = new MyRunnableObject(...);
Thread aThread = new Thread( myR);
aThread.start();

/* You can use lambda expressions instead: */
Thread anotherThread = new Thread(() -> { /* Thread code goes here */ });
anotherThread.start();
```

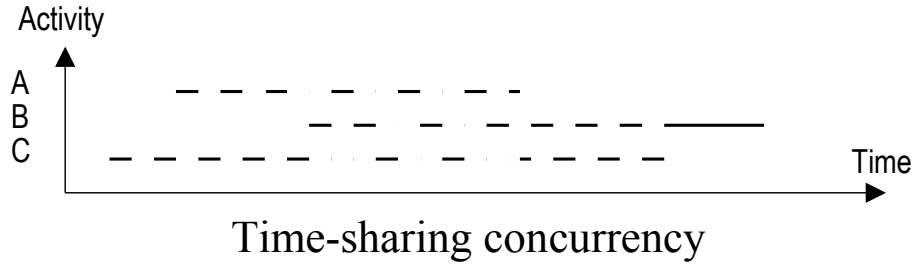
# Använda class Thread

```
class MyActivity extends Thread {

    public MyActivity() {
        // init here, done before `start´ is called
    }

    public void run() {
        while( true) {
            // do whatever the thread should do, once it is set alive by calling
            // start (which calls run, effectively)
        }
    }

}

...

public static void main( String[] args) {
    // do / declare some stuff

    MyActivity myAct1 = new MyActivity();

    // ... do some stuff ...

    myAct1.start();                    // Vad händer här om man "bara" kör run()?

    // ... do some more stuff ...

    myAct1.join();
}
```

# class Thread

```
public class Thread implements Runnable {

        static int MAX_PRIORITY;          // Highest possible priority
        static int MIN_PRIORITY;          // Lowest possible priority
        static int NORM_PRIORITY;         // Default priority

        Thread();                         // Use run in subclass
        Thread( Runnable target);         // Use run in `target´

        void start();                     // Create thread which calls `run´
        void run();                       // Work to be defined by subclass
        static Thread currentThread();    // Get currently executing thread

        void setPriority( int pri);       // Change the priority to `pri´
        int getPriority();                // Returns this thread's priority

        static void sleep( long t);       // Suspend execution at least `t´ ms
        static void yield();              // Reschedule to let others run

        void interrupt();                 // Set interrupt request flag
        boolean isInterrupted();          // Check interrupt flag of "this" thread
        static boolean interrupted();     // Check if interrupted for currently
                                          // running thread

        boolean isAlive();                // true if started and not dead
        void join();                      // Waits for this thread to die
        void join( long t);               // Try to join, but only for `t´ ms
}
```

# Jämlöpande exekvering av sekventiella processer



Time-sharing concurrency

Varje aktivitet (tråd) måste utföras som ett logiskt korrekt, sekventiellt program.

Samtliga parallella aktiviteter tillsammans måste leverera korrekt beteende jämnt och under alla omständigheter (inmatningar, händelser...).

Att testa / verifiera detta är i det närmaste omöjligt, för att pyttesmå ändringar i hur sekvenserna "flätas samman" kan påverka mycket, och det är i princip omöjligt att få till exakt samma flätning två gånger.

# Sekventiell bearbetning

*Situation 1: Utbetalning först*

A: Läs 5000
A: Belopp = 5000 - 1000
A: Skriv 4000

B: Läs 4000
B: Belopp = 4000 + 10000
B: Skriv 14000

*Situation 2: Inbetalning först*

B: Läs 5000
B: Belopp = 5000 + 10000
B: Skriv 15000

A: Läs 15000
A: Belopp = 15000 - 1000
A: Skriv 14000

Två aktiviteter (program, trådar (threads), processer) utförs oberoende och ej samtidigt. I båda situationer blir resultatet korrekt.

# Bankkontot igen

*Situation 1:*

```
A: Läs 5000

   B: Läs 5000

A: Belopp = 5000 - 1000

   B: Belopp = 5000 + 10000

A: Skriv 4000

   B: Skriv 15000
```

*Situation 2:*

```
A: Läs 5000

   B: Läs 5000
   B: Belopp = 5000 + 10000
   B: Skriv 15000

A: Belopp = 5000 - 1000
A: Skriv 4000
```

Två aktiviteter (program, trådar (threads), processer) utförs samtidigt, då de hanterar samma resurser. I båda situationer blir resultatet fel.

Här behövs det alltså någon mekanism för ömsesidig uteslutning (mutual exclusion) för att hantera kritiska sekvenser (critical sections) och odelbara aktioner (atomic actions).

# Kritiska sekvenser
## (Critical sections)

- Delar av ett program (en sekvens) som behöver tillgång till en delad resurs.
- Får inte bli avbruten av en annan programsekvens som använder samma resurs, eller av ett nytt anrop till sig själv.
- Kraven kan uppfyllas med hjälp av *Semaforer*, *Monitorer* eller "Postlådor" (*Mailboxes*)
- På låg nivå (native code) kan man också slå av interrupts (avbrott).

# Sammanfattning

- Infört begreppen *jämlöpande exekvering, tråd, schemaläggning, pre-emption (avbrytbarhet), kontext / kontextbyte*, (lite) *ömsesidig uteslutning*
- Diskuterat fällor såsom fel vid gemensam resurshantering, kapplöpning
- Introducerat trådar i Java

- Lästips:
  - e-bok: Kap 1 ( +2).
  - kompendium: Kap 1 (Introduction) + 2.1 (Threads)