Realtidssystem

- Meddelanden och händelsehantering -

EDAF85 - Realtidssystem (Helsingborg)
Roger Henriksson

Föreläsning 4

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

Innehåll

- Meddelanden
  - Monitorer som brevlådor - mailbox
  - Trådkommunikation via mailbox / meddelanden
- Interrupt() och InterruptedException

Minns du buffertexemplet?

Blockerande köer är mycket användbara.

- Många program kan uttryckas som ett dataflöde, t.ex. producenter och konsumenter
- Trådar kommunicerar genom att sända och ta emot meddelanden (*messages, events, tokens*)
- Meddelanden är objekt som kan förmedla information och/eller användas för synkronisering
- Varje tråd har typiskt en buffert (kö) för inkommande meddelanden. Vi kallar denna buffert för en brevlåda (*mailbox, message queue, event queue*)

```
class MyBlockingQueue<E> {

  private Queue<E> q = new LinkedList<>();

  public synchronized void add(E e)
    throws InterruptedException
  {
    if (q.isEmpty()) {
      notifyAll();
    }
    q.add(e);
  }

  public synchronized E take()
    throws InterruptedException
  {
    while(q.isEmpty()) {
      wait();
    }
    return q.poll();
  }
}
```

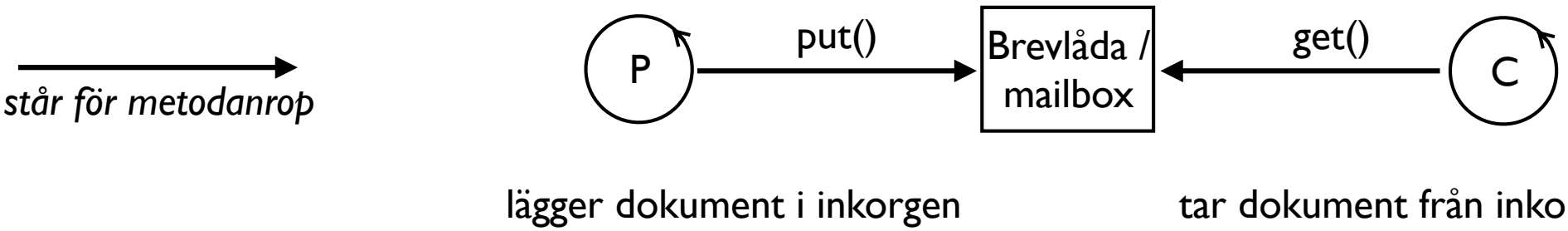
Meddelanden: synkronisering/signalering

- Hittills har vi använt semaforer eller monitorer för synkronisering och signalering. Meddelandesändning är ett tredje, ofta attraktivt, alternativ.
- Meddelandesändning är särskilt lämpligt i distribuerade system i vilka trådar kan tänkas köra på separata datorer i ett nätverk. Meddelandena sänds då via nätverket.
- Exempel: Realtidsoperativsystemet OSE och programmeringsspråket Erlang med rötter i Ericssons telefonisystem bygger båda på meddelandesändning. Ett OSE- eller Erlangbaserat system kan göras distribuerat med inga eller mycket få ändringar i koden.

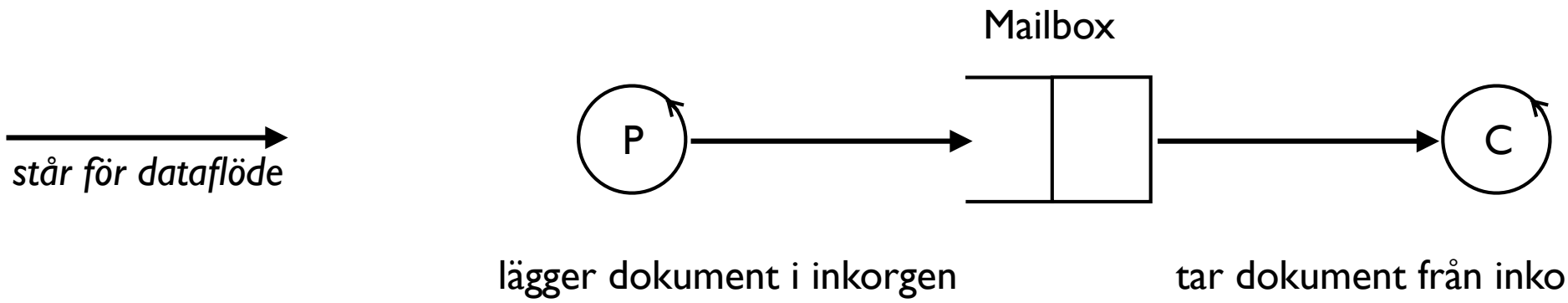
Producent och konsument

Vi vill ha
- asymmetrisk synkronisering - *producer* ska kunna jobba vidare utan att behöva vänta på *consumer*
- överföring av information - ett meddelande (*message*)

Hittills har vi kunnat göra detta med monitorer:

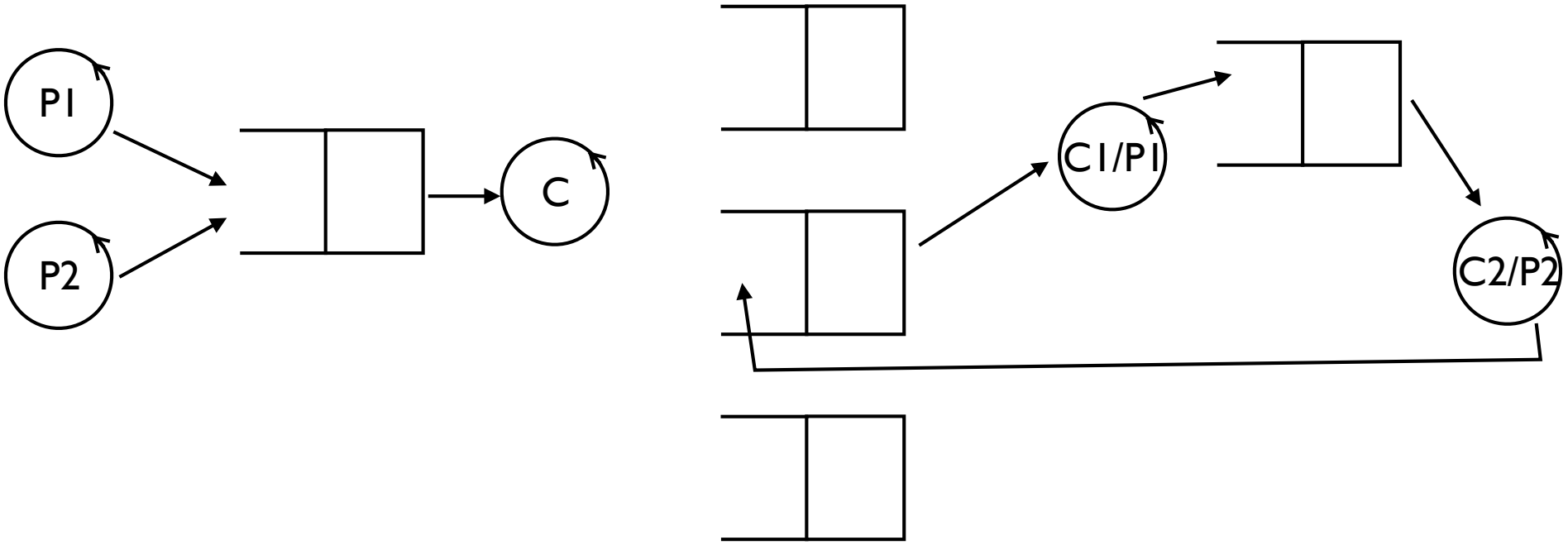


Vi inför begreppet Mailbox (brevlåda) när vi använder monitorn så här och ritar den lite annorlunda



Mailboxar i system

Kommunikation mellan trådar leder ofta till ett helt *mailbox*-nätverk



- I Java använder vi `BlockingQueues` som kan lagra data av godtycklig typ
- En tråd kan lägga meddelanden i flera olika brevlådor
- En tråd läser oftast bara från en brevlåda (krångligt annars – hämta meddelande blockerar)
- Inga delade data
- Asynkron sändning – att skicka meddelanden bör inte blockera

Semafor/monitor/brevlåda ekvivalenta

- En brevlåda (meddelande), liksom en semafor, kan implementeras mha en monitor.
- En kö med tomma meddelandeobjekt är ekvivalent med en semafor
  - Antal meddelanden i kön motsvarar antal körtillstånd
  - Skicka meddelande motsvarar `release()`
  - Ta emot meddelande motsvarar `acquire()`

Alla tre konstruktioner är i teorin lika kraftfulla, men praktiska i olika situationer.

Interface BlockingQueue

```
public interface BlockingQueue<E> extends Queue<E> {
  boolean add(E e);          // Throws exception if full
  E           take();        // Blocks until available
  E           poll(long timeout,
                   TimeUnit unit);   // Blocks for
                                     // limited time
  // … other stuff …
```

- Flera varianter av operationerna – hanterar fulla/tomma köer olika
- Flera implementationer i standardbiblioteket
- Vi vill **blockera** när vi läser från en tom kö
- Om vi använder oändliga (???) köer så behöver vi inte hantera fallet att kön blir full

Design pattern: Actor

- Vi behöver ofta koppla samman en brevlåda med en tråd
- Varje sådan tråd har sin egen meddelande (brevlåda)
- Varje sådan tråd tar emot meddelanden via sin egen brevlåda och sänder meddelanden till andra trådars brevlådor
- En sådan tråd med sin egen brevlåda brukar kallas för en **actor**. Actor i detta sammanhang är ett begrepp som inte ska förväxlas med actors i UML.
- I laboration 3 implementerar vi en särskild trådklass, `ActorThread`, som representerar en actor (tips: läs kompendiet avsnitt 4.3 noga!)

ActorThread: tråd med inbyggd brevlåda

```
public class ActorThread<M> extends Thread {
  /** Send a message to this thread.
      NOTE: Thread-safe; called by other threads */
  public void send(M message);

  /** Returns the first message in the queue,
      or blocks if none available. */
  protected M receive();

  /** Returns the first message in the queue, or blocks
      up to 'timeout' milliseconds if none available.
      Returns null if no message is received. */
  protected M receiveWithTimeout(long timeout);
}
```

Notera: Vi avviker här (med stor försiktighet) från regeln som säger att en tråd bör inte ha andra publika metoder än `run()`.

Exempel: Producer/Consumer

```
class Producer extends Thread {
  private Consumer cons;
  public Producer(Consumer c) { cons = c; }

  public void run() {
    Scanner scan = new Scanner(System.in);
    try {
      while (true) {
        cons.send(scan.nextLine());
      }
    } catch(InterruptedException e ) {
      System.err.println("Producer failure: ”+e);
    }
  }
}

class Consumer extends ActorThread<String> {
  public void run() {
    try {
      while (true) {
        String m = receive();
        System.out.println("Received: "+m);
      }
    } catch(InterruptedException e ) {
      System.err.println("Consumer failure: "+e);
    }
  }
}
```

Brevlådors ”Dos and Don’ts”

Meddelanden är objekt som skickas från en tråd till en annan

- Dela inte några data
- Inkludera all information mottagaren behöver i meddelandeobjektet – skapa din egen klass om du behöver
- Betrakta meddelandeobjekten som oföränderliga (immutable) – det finns ingen anledning att ändra i en annans meddelande
- Behåll inte en referens till meddelandeobjektet när det väl är sänt – då blir det ju ett delat objekt
- Undvik att blanda meddelandesändning med semaforer/monitorer

Gör `receive()` på ett ställe i en loop

Gör så här:

```
while (true) {
  String m = receive();
  // Do work based on m
}
```

Gör inte så här:

```
while (true) {
  String m1 = receive();
  …
  String m2 = receive();
  …
  if (x<3) {
    String m3 = receive();
  }
}
```

I det andra fallet inför man **implicita tillstånd**, vilket kan vara svårt att få rätt.

Hantera tillstånd

En tråd kan ofta befinna sig i olika *tillstånd*. Ofta kan man modellera en tråds beteende med hjälp av en *tillståndsmaskin*. Försök göra dessa tillstånd explicita.

```
private static final int
  DOOR_OPEN    = 1,
  DOOR_OPENING = 2,
  DOOR_CLOSING = 3,
  DOOR_CLOSED  = 4;

public void run() {
  int state = DOOR_CLOSED;
  while (true) {
    DoorEvent m = receive();

    switch (state) {
    case DOOR_CLOSING:
      if m.isButton()) {
        state = DOOR_OPENING;
        io.open();
      }
      break;
    case  DOOR_CLOSED:
```

Fördelar med meddelandesändning

- Tydlig ansvarsfördelning – enkla gränssnitt mellan trådarna
- Enkelt att dela data mellan trådar
- Vi skyddar trådars interna data från andra trådar
- Mindre risk för deadlock (nästa föreläsning)
- Fungerar väl även i distribuerade system

Att stoppa en tråd

- Enkelt att starta tråd… men hur stoppar vi en tråd?
- Betrakta det tidigare exemplet med en producent och en konsument - hur stoppar vi konsumenten när vi inte behöver den längre?
- Vi skulle kunna sända ett speciellt stoppmeddelande som sa åt tråden att avsluta sin exekvering
- Det finns dock en mer generell lösning i Java: `interrupt()`

`InterruptedException` och `interrupt()`

För att förstå `InterruptedException` och `interrupt()` måste vi förstå

1. vad som egentligen händer när vi anropar `interrupt()`
2. hur vi kan använda denna mekanism i ett program

Vad händer i `interrupt()`?

När tråd `t1` anropar `interrupt()` på tråd `t2`

```
t2.interrupt();
```

- Interruptstatusen för `t2` sätts till `true`.
  Vi kan se det som om varje tråd har en liten monitor med ett enkelt booleskt värde i.
- Så fort `t2` utför en blockerande operation (`wait()`, `sleep()`) kastas ett `InterruptedException`
- Motsvarande `catch`-sats utförs (om sådan finns)
- En aktiv tråd kan även testa interruptstatusen med ett anrop av `isInterrupted()`

Hur använder vi `interrupt()`?

```
t.interrupt(); // instruct t to stop
```

Ett anrop av `interrupt()` instruerar en tråd att:

- avsluta sin exekvering ”så fort som möjligt”
- köra eventuell uppstädningskod som kan vara nödvändig (t.ex. stänga filer)
- avsluta exekveringen (typiskt genom att lämna `run()`)

Att avsluta en tråd är en asynkron process. Tråden slutar inte exekvera omedelbart.

Först ska tråden få tillfälle att köra, sedan ska ovanstående arbete utföras vilket kan ta en viss (kort) tid.

`interrupt()` är asynkront

Vid behov kan vi vänta på att tråden faktiskt slutar att exekvera:

```
t.interrupt();
t.join();
```

Det är upp till den avbrutna tråden att bestämma vad som ska hända.

Tråden är inte förpliktigad att avsluta, men det är vad `interrupt()` normalt används till.

Ta hand om InterruptedException

När vi får ett `InterruptedException` måste vi avgöra om vi ska fånga det (med en `try-catch`-sats) eller kasta det vidare upp längs anropskedjan (med en `throws`-deklaration).

En bra princip är att vi ska fånga det på den nivå där vi vet vad vi ska göra med det. Det beror på applikationen.

- I en monitormetod vet vi ofta inte vad vi ska göra – använd `throws`
- I en `run()`-metod brukar vi däremot veta – använd `try-catch`

Sammanfattning

- Meddelandesändning med brevlådor/mailbox
- Kunna lösa laboration 3
- Läs- och tittips:
  - e-bok: Kapitel 6, sid 141-157
  - kompendium: sid 36-38 samt kapitel 4
  - Kortfilmer på kurshemsidan (”Föreläsningar och övningar”)
    Mycket stor hjälp inför laboration 3!