Realtidssystem

- Semaforer, trådsynkronisering -

EDAF85 - Realtidssystem (Helsingborg)
Roger Henriksson

Föreläsning 2

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

- Hur åstadkomma ömsesidig uteslutning?
- Semaforer
- Semaforer i Java
- Ömsesidig uteslutning – signalering – rendez-vous

Bankkontot – kapplöpning

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. Kallas för *kapplöpning* (eng. race condition).

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

Kan man få till *ömsesidig uteslutning* utan systemanrop?

```
class T extends Thread {
   public void run() {
      while( true) {
         nonCriticalSection();
         preProtocol();
         criticalSection();
         postProtocol();
      }
   }
}
```

```
class T2 extends Thread {
   public void run() {
      while( true) {
         nonCriticalSection();
         preProtocol();
         criticalSection();
         postProtocol();
      }
   }
}
```

Critical Section (CS)

- Jämför Bankkonto-exemplet
- Vi kommer titta på konstruktionen av "pre-/postProtocol"
- Antagandet: En tråd får inte att blockeras inom dess kritiska sekvens (critical section).
- Krav: Ömsesidig uteslutning, inget dödläge, ingen "svält", hög verkningsgrad (inte slösa på CPU-tid).

Ömsesidig uteslutning, tråden

```
class T extends Thread {
 public void run() {
  while( true) {
    nonCriticalSection();
    preProtocol();
    criticalSection();
    postProtocol();
  }
 }
}
```

Ömsesidig uteslutning, kraven

Det här måste uppfyllas:

1. Ömsesidig uteslutning (**Mutual exclusion**): Exekvering av en kod i en kritisk sekvens får inte flätas ihop med någon annan tråds kodsekvens.

2. Inget dödläge (**No deadlock**): Om en eller fler tråd(ar) försöker starta exekvering av en kritisk sekvens, måste det finnas någon av dem som faktiskt kan göra det.

3. Ingen "svält" (**No starvation**): En tråd måste få möjlighet att någon gång påbörja exekveringen av dess kritiska sekvens.

4. Effektivitet (**Efficiency**): Liten overhead när det finns enbart en tråd, det hela måste fungera bra även om det bara finns en tråd (det får inte finnas ett obligatorisk "väntemoment" på en annan tråd när en sådan inte kan garanteras finnas).

Kan de här kraven uppfyllas med "vanlig" Java-kod?

Ömsesidig uteslutning
- version 1 -

```
volatile int turn = 1;
```

```
class T1 extends Thread {
   public void run() {
      while( true) {
         nonCS1();
         while( turn != 1);
         cS1();
         turn = 2;
      }
   }
}
```

```
class T2 extends Thread {
   public void run() {
      while( true) {
         nonCS2();
         while( turn != 2);
         cS2();
         turn = 1;
      }
   }
}
```

- Ömsesidig uteslutning: OK
- Inget dödläge: OK, en av dem kan alltid köra.
- Inget svält: OK, omväxlande protokoll. Fast se nedan…
- Effektivitet / en tråd?: NEJ (fallet en tråd kommer hänga sig efter max en omgång), plus vi har en busy-wait (ineffektiv/kan orsaka svält). För många trådar dessutom rörig lösning.

Ej acceptabelt!

Ömsesidig uteslutning
- version 2 -

```
volatile int c1, c2; c1 = c2 = 1;
```

```
class T1 extends Thread {
   public void run() {
      while( true) {
         nonCS1();
         while( c2 != 1);
         c1 = 0;
         cS1();
         c1 = 1;
      }
   }
}
```

```
class T2 extends Thread {
   public void run() {
      while( true) {
         nonCS2();
         while( c1 != 1);
         c2 = 0;
         cS1();
         c2 = 1;
      }
   }
}
```

- Ömsesidig uteslutning: NEJ... (testa sammanflätning)

```
c1 = 1;
   c2 = 1;
while( c2 != 1);
   while( c1 != 1);
c1 = 0;
   c2 = 0;
cS1();
   cS2();
```

Ingen lösning, även om det kan fungera länge innan det kraschar. Dessutom fortfarande busy-wait!

Ömsesidig uteslutning
- version 3 -

volatile int c1, c2; c1 = c2 = 1;

```
class T1 extends Thread {
   public void run() {
      while( true) {
         nonCS1();
         c1 = 0;
         while( c2 != 1);
         cS1();
         c1 = 1;
      }
   }
}
```

```
class T2 extends Thread {
   public void run() {
      while( true) {
         nonCS2();
         c2 = 0;
         while( c1 != 1);
         cS1();
         c2 = 1;
      }
   }
}
```

- Ömsesidig uteslutning: OK
- Inget dödläge: Nej (alltså ja):

```
c1 = 0;
   c2 = 0;
while( c2 != 1);          // I all evighet ...
   while( c1 != 1);       //  ... och lite till
...
```

Ömsesidig uteslutning
- version 4 -

```
volatile int c1, c2; c1 = c2 = 1;
```

```
class T1 extends Thread {
   //...
      nonCS1();
      c1 = 0;
      while( c2 != 1){
         c1 = 1; //**
         c1 = 0;
      }
      cS1();
      c1 = 1; //..
}
```

```
class T2 extends Thread {
   //...
      nonCS2();
      c2 = 0;
      while( c1 != 1){
         c2 = 1; //**
         c2 = 0;
      }
      cS2();
      c2 = 1; //..
}
```

- Ömsesidig uteslutning: OK
- Inget dödläge: OK (//** är yield)
- Inget svält: NEJ, det kan hända att en tråd får köra, men kommer aldrig så långt att den får verkligen utföra CS (kallas "livelock" om det händer att flera trådar hela tiden jobbar utan att göra något nyttigt). Bygger på trådbyte vid exakt rätt tidpunkt.

Ej acceptabelt!

```
c1 = 0;
   c2 = 0;
   while( c1 != 1) {
   c2 = 1; ...
while( c2 != 1);
cS1();
c1 = 1;
nonCS1();
c1 = 0;
   ... c2 = 0; }
   while( c1 != 1) {
   c2 = 1;
while( c2 != 1);
cS1();
c1 = 1; ...
```

Dekkers algoritm

```
volatile int c1, c2, turn; c1 = c2 = turn = 1;
```

```
class DA1 extends Thread {
   //...
      nonCS1();
      c1 = 0;
      while( c2 != 1){
         if( turn == 2) {
            c1 = 1;
            while( turn == 2);
            c1 = 0;
         }
      }
      cS1();
      c1 = 1;
      turn = 2;
}
```

```
class DA2 extends Thread {
   //...
      nonCS2();
      c2 = 0;
      while( c1 != 1){
         if( turn == 1) {
            c2 = 1;
            while( turn == 1);
            c2 = 0;
         }
      }
      cS2();
      c2 = 1;
      turn = 1;
}
```

- Ömsesidig uteslutning: OK
- Inget dödläge: OK
- Ingen svält: OK
- Effektivitet / en tråd?: Inte bra.

Dekkers algoritm fungerar bra för många trådar, men blir då komplex, den löser problemet "ömsesidig uteslutning", MEN med busy-wait. Kan vara användbar i vissa system med flera processorer.

Ömsesidig uteslutning
- Semafor -

```
Semaphore mutex = new Semaphore(1);
```

```
class T1 extends Thread {
   public void run() {
      while( true) {
         nonCS1();
         mutex.acquire();
         cS1();
         mutex.release();
      }
   }
}
```

```
class T2 extends Thread {
   public void run() {
      while( true) {
         nonCS2();
         mutex.acquire();
         cS2();
         mutex.release();
      }
   }
}
```

- Ömsesidig uteslutning: OK
- Inget dödläge: OK
- Inget svält: OK (*release()* startar upp den blockerade tråden direkt)
- Effektivitet / en tråd: OK. Blockerade trådar använder inte CPUn.

Acceptabelt!

Semafor - basfakta

Semaforer utgör en *minimal mekanism* för ömsesidig uteslutning

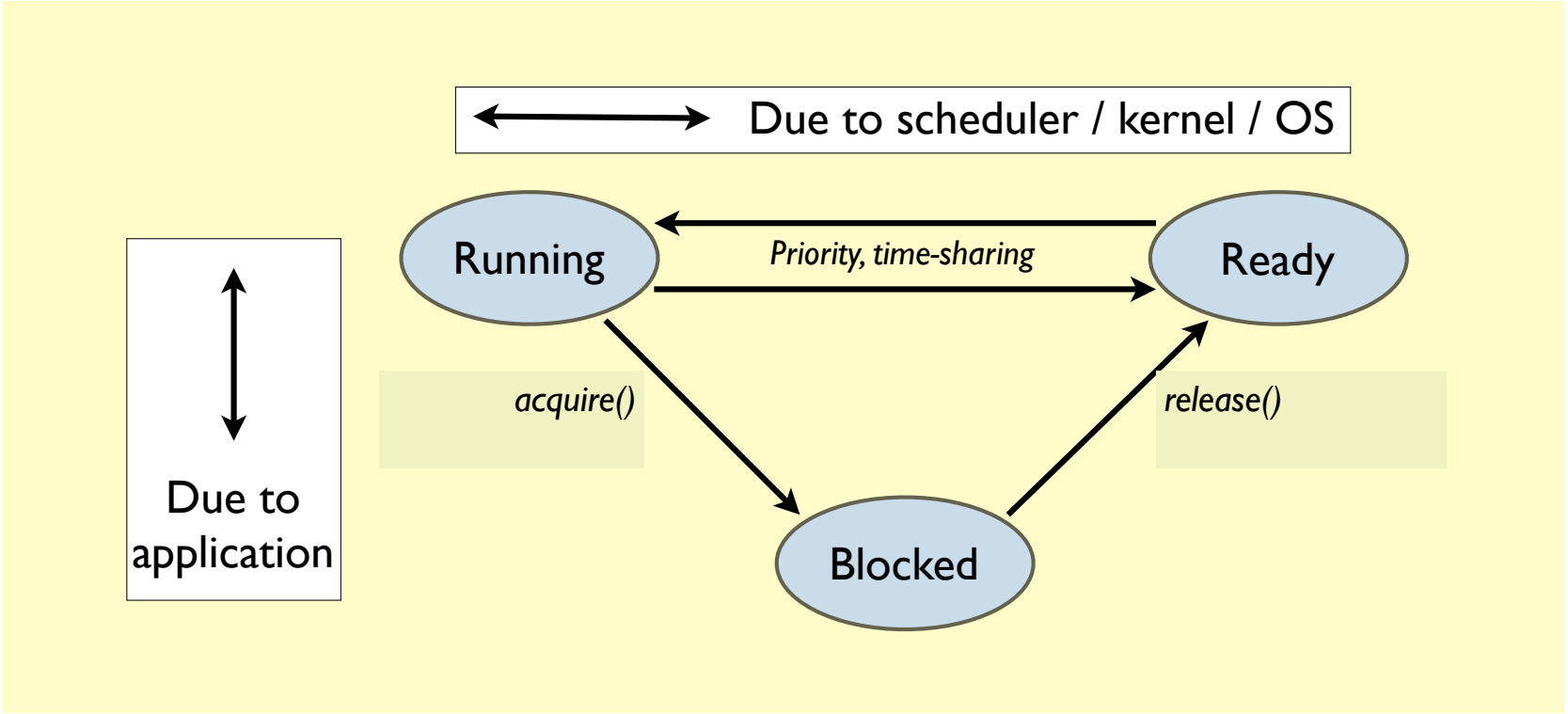
En semafor kan ses som en heltalsvariabel kopplad till två metoder, *acquire*() och *release*():

```
class SemaphorePrinciple {
   int count;
   public void acquire() {
      while( count < 1)
         "suspend executing thread"  // System call required
      --count;
   }

   public void release() {
      if( "any thread suspended")    // System call required
         "resume the first one in queue"  // System call required
      count++;
   }
}
```

OBS 1: *acquire* och *release* är odelbara (atomära) operationer, som kräver systemsupport när de ska implementeras (t ex deaktivera hårdvaruinterrupts).

OBS 2: *acquire* **blockerar** den anropande tråden så länge count == 0. Detta kan **inte** implementeras i vanlig Java-kod.

Exekveringstillstånd



Schemaläggningstillstånd (scheduling state)

- Running
- Ready
- Blocked

Kontext

- PC (Program Counter), SP (stack pointer)
- Data (beroende på tillämpning)
- Registerinnehåll

Semaforer - användning i kod

	Tråd A:	Tråd B:

Mutual exclusion (”mutex”)
(ömsesidig uteslutning)
Startvärde: 1
	...
	mutex.acquire();
	***
	mutex.release();
	...
		...
		mutex.acquire();
		***
		mutex.release();
		...

Signaling (signalering) Startvärde: 0	... buffer.release(); ...	... buffer.acquire(); ...

Rendezvous
Startvärden: 0
	***
	entry.release();
	exit.acquire();
	***
		...
		entry.acquire();
		***
		exit.release();
		...

Typer av semaforer i Java

Semaphore - Generell *räknande* semafor. Både för att åstadkomma ömsesidig uteslutning och för signalering mellan trådar. Den som gör plats / en resurs tillgänglig, signalerar genom att lägga en flagga på högen, den som vill utnyttja en "resursplats" tar en flagga från högen. Finns inga "fria flaggor", måste den som vill ha en vänta.

Lock - Semafor endast för ömsesidig uteslutning. Den som tar flaggan får jobba med gemensamma resurser, den som vill ha den samtidigt, får vänta (blockeras).

Mutexsemafor i Java

Deklarera:

```
import java.util.concurrent.*;
Semaphore mutex;
```

Skapa / initialisera:

```
mutex = new Semaphore(1);
```

Tillämpning:

```
mutex.acquire();      // försök räkna ner semaforen till 0
amount += change;     // kritisk region
mutex.release();      // räkna upp semaforen igen - frigör resursen
```

Lock: Alternativ till mutexsemafor

Deklarera:

```
import java.util.concurrent.*;
Lock mutex;
```

Skapa / initialisera:

```
mutex = new ReentrantLock();
```

Tillämpning:

```
mutex.lock();          // räknar upp låsräknaren (om ej låst eller vi är ägare)
amount += change;      // kritisk region
mutex.unlock();        // räknar ner låsräknaren - frigör resursen om 0
```

Kommentar: En tråd kan låsa en resurs flera gånger, men måste då låsa upp den igen lika många gånger. Jämför med Semaphore: Vad händer om vi försöker ta semaforen flera gånger?

Mutex i trådar

```
import java.util.concurrent.*;

class ThreadTest {
  public static void main(String[] args) {

    Thread t1,t2;
    Semaphore s;
    s = new Semaphore(1);

    t1 = new RogersThread("Thread one",s);
    t2 = new RogersThread("Thread two",s);
    t1.start();
    t2.start();
  }
}
```

```
class RogersThread extends Thread
{
  String theName;
  Semaphore theSem;

  public RogersThread( String n, Semaphore s){
    theName = n;
    theSem = s;
  }

  public void run() {
    theSem.acquire();
    for(int t=1;t<=100;t++) {
      System.out.println(theName + ":" + t);
      sleep(1);
    }
    theSem.release();
  }
}
```

Signal i trådar

```
import java.util.concurrent.*;

class ThreadTest {
  public static void main(String[] args) {

    Thread t1,t2;
    Semaphore s1, s2;
    s1 = new Semaphore(1);
    s2 = new Semaphore(0);

    t1 = new RogersThread("One",s1, s2);
    t1.start();
    t2 = new RogersThread("Two",s2, s1);
    t2.start();
  }
}
```

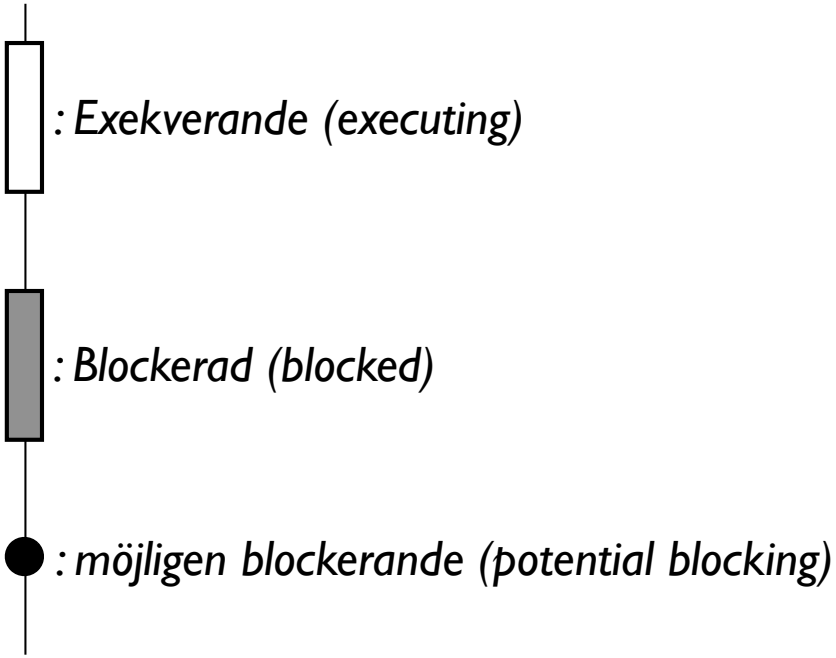
```
class RogersThread extends Thread
{
  String theName;
  Semaphore mySem, otherSem;

  public RogersThread( String n,
    Semaphore s1, Semaphore s2){

    theName = n;
    mySem = s1;
    otherSem = s2;
  }

  public void run() {
    for(int t=1;t<=100;t++) {
      mySem.acquire();
      System.out.println(theName + ":" + t);
      otherSem.release();
      sleep(1);
    }
  }
}
```

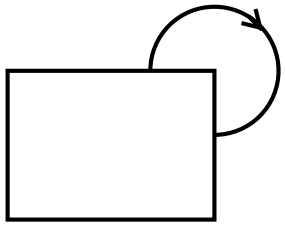
Objekt och blockering

Trådobjekt (som alltså kan referera till en exekverande tråd) kallas för *aktiva objekt*.

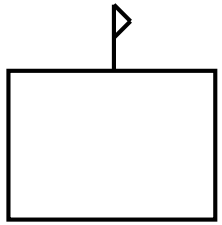
Andra objekt (beskriven genom ”vanliga” Java-klasser), som blir kallade eller drivna genom aktiva objekt (trådar) kallas för *passiva objekt*. T ex en semafor ;-)

Semaforer erbjuder blockerande operationer; hur blir de representerade?



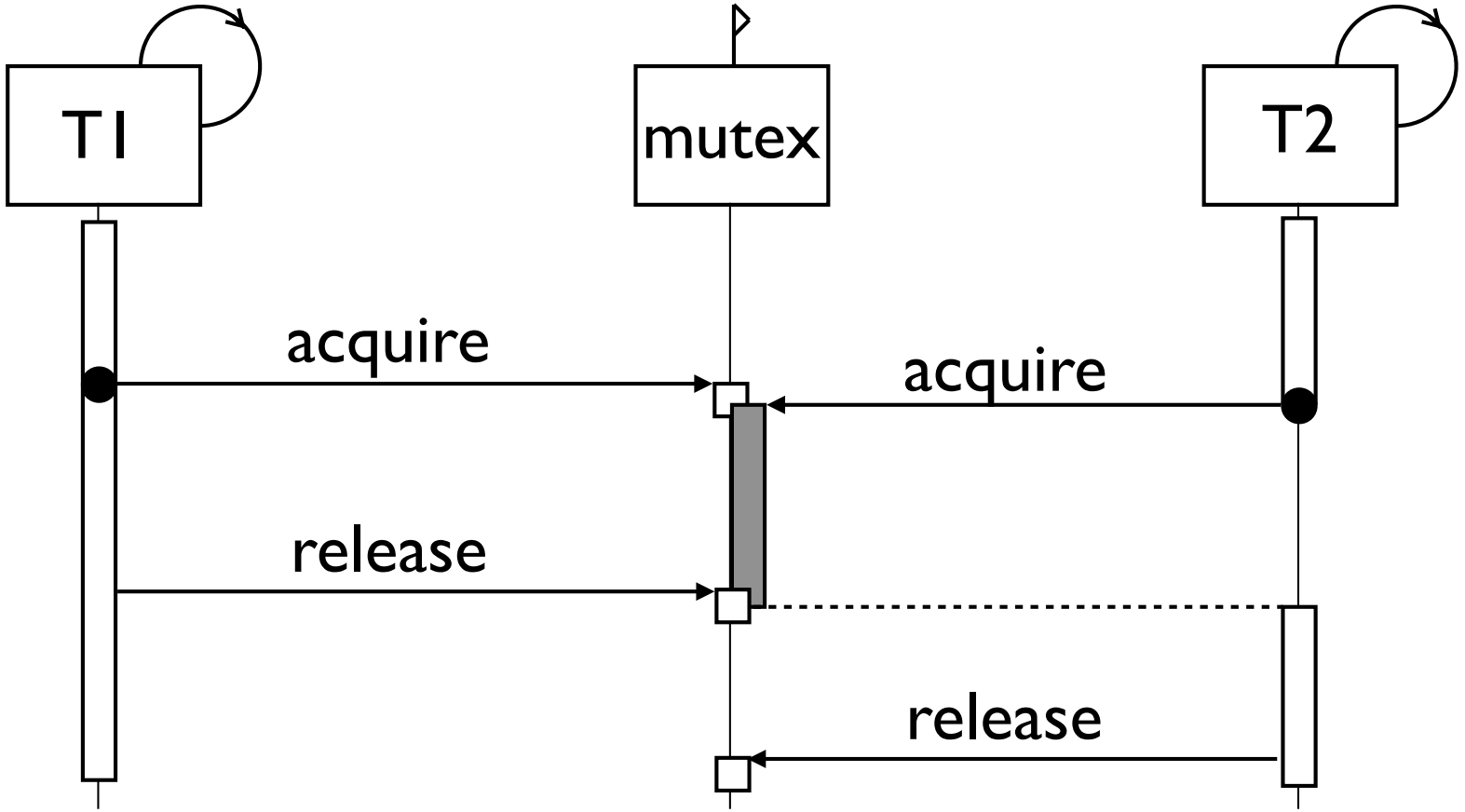


Tråd
(aktivt objekt)



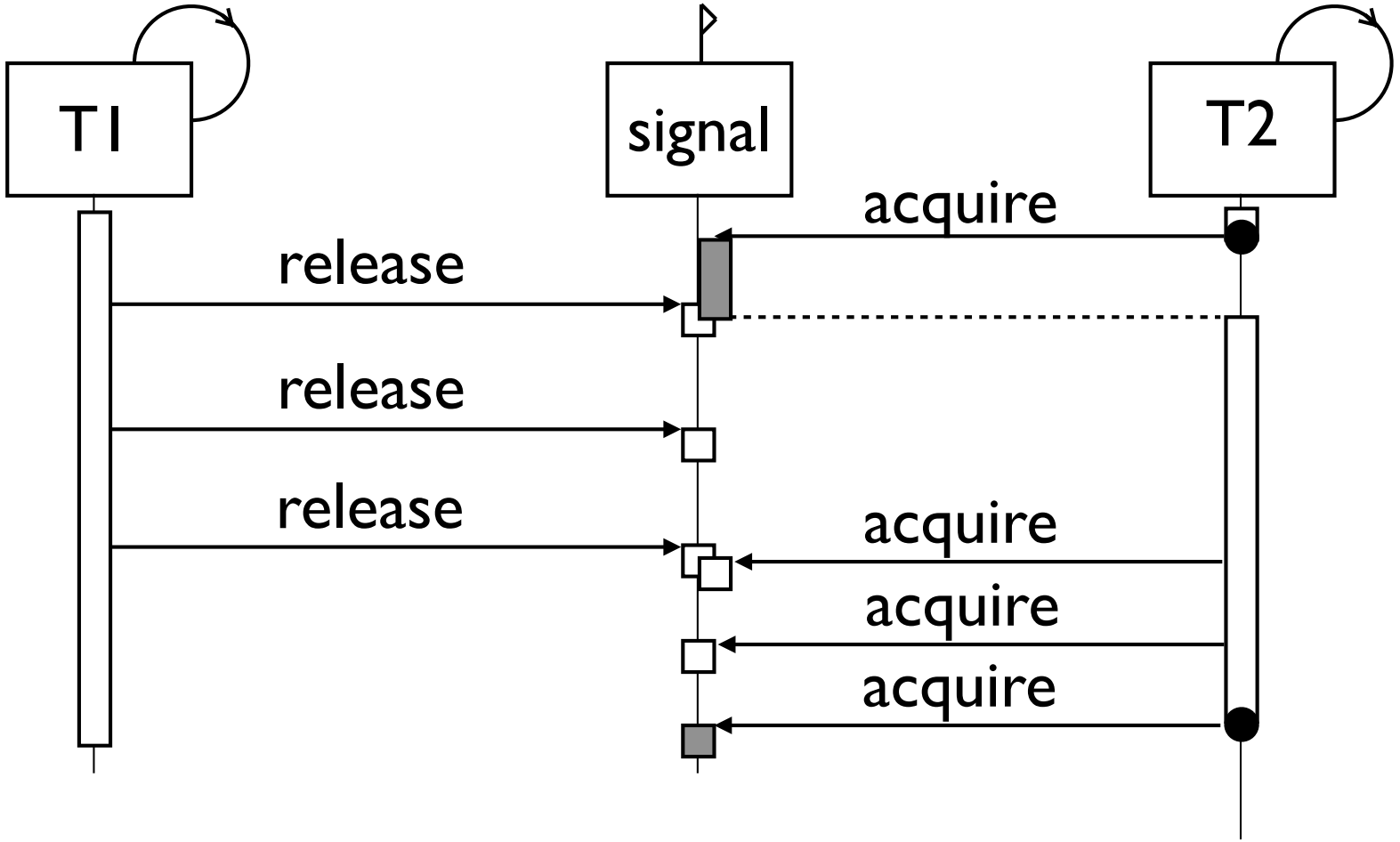
Semafor
(passivt objekt)

Sekvens med mutex / blockerade trådar



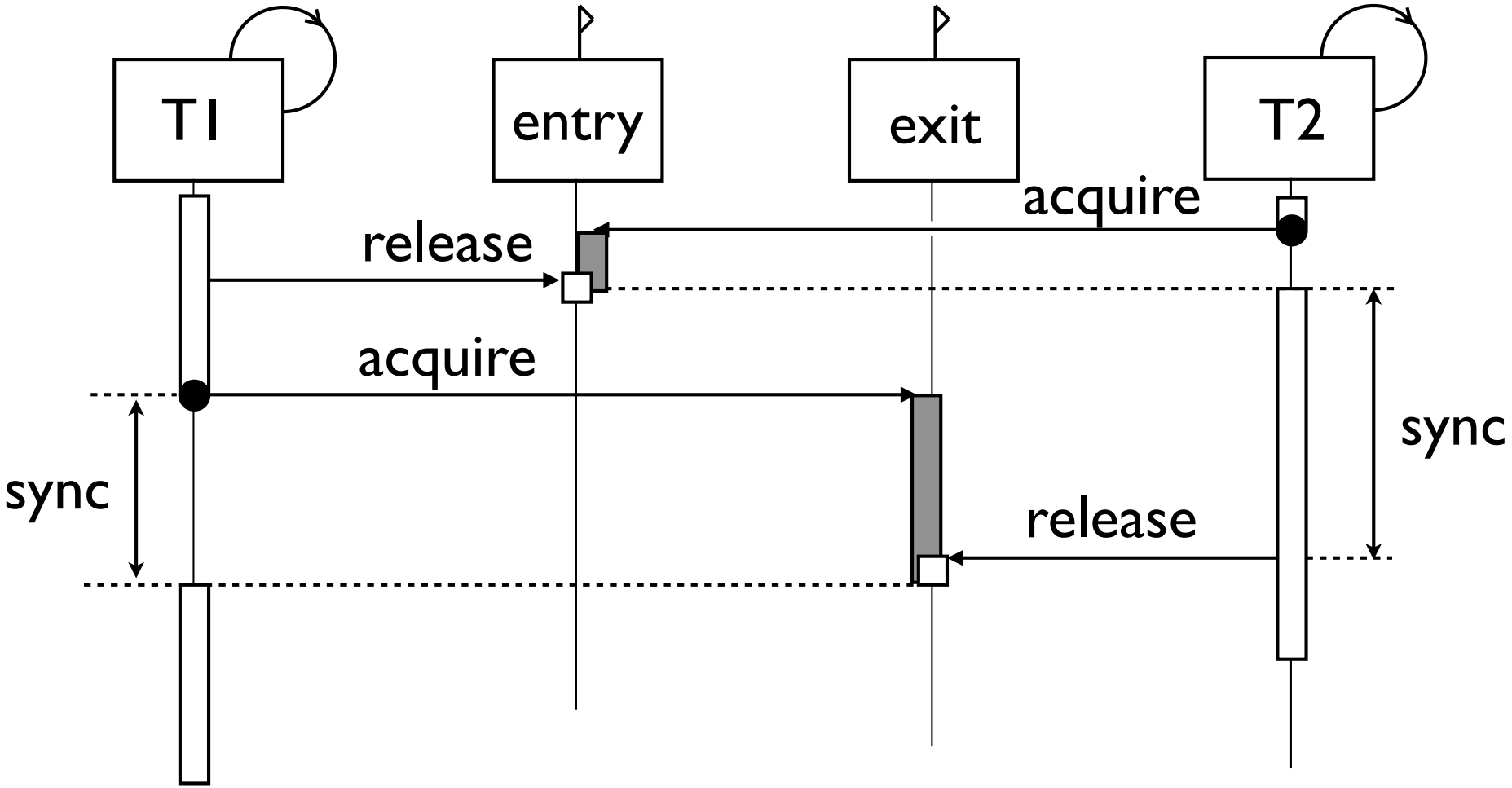
Anrop till **acquire** - **release** i följd måste komma från samma tråd. Används **ReentrantLock** (**lock()/unlock()**) i stället kollas detta.

Signalering



En tråd anropar **acquire** (för att vänta på en händelse) och en annan **release** (för att signalera att händelsen inträffat) Stöds av **Semaphore**-klassen.

Rendezvous



Säker dataöverföring / manipulation med T1 från T2 under *sync*-tiden. T1 har exklusiv tillgång till datan genom att hålla i semaforen förutom mellan sitt *release* och T2s *release*

Övning 1

ÖVNINGAR ÄR INGA FÖRELÄSNINGAR - Arbeta med materialet!

Förbered er - studera teorin som övningen handlar om före övningen och titta gärna igenom uppgifterna i förväg.

Övning 1 handlar om semaforer och deras tillämpning - ställ frågor om sådant som är oklart om semaforer.

Laboration 1 presenteras

Sammanfattning

- Trådar, kapplöpningsproblem, hantering av gemensamma resurser
- Semaforer
- Typer av semaforer
- Användning av semaforer

- Man ska kunna lösa uppgifterna till övning 1 och påbörja arbetet med lab1

- Lästips:
  - e-bok: delar av Kap 5 (s 103-129)
  - kompendium: Kap 2-1 (Threads) samt 2-2 (Semaphores)