http://cs.lth.se/EDA040

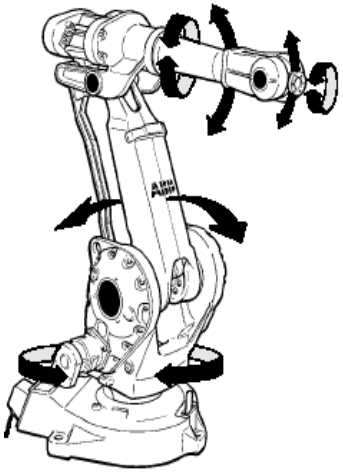
Lectures during HT 2011 (draft): EDA040

Concurrent and Real-time programming

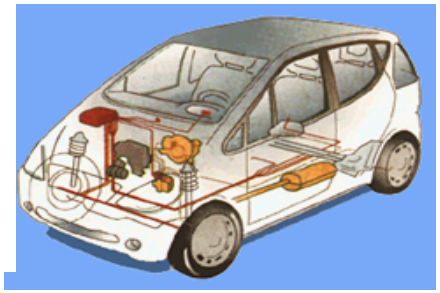
Department of Computer Science, Lund

klas@cs.lth.se

Lecture 1 [F1]: Embedded concurrent software



Example Products



Upcoming Lab

Exercise 1:

- Semaphores
- Signaling and mutex
- Place operations inside methods and objects.

Basics for Lab 1.

Lab 1 (&exercise 2):

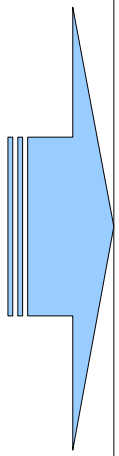
- Use semaphores to implement software for an alarm clock.
- Event-driven and time-driven threads sharing time data.

Essence of threads and semaphores.

How to design and implement embedded real-time software

Concurrency

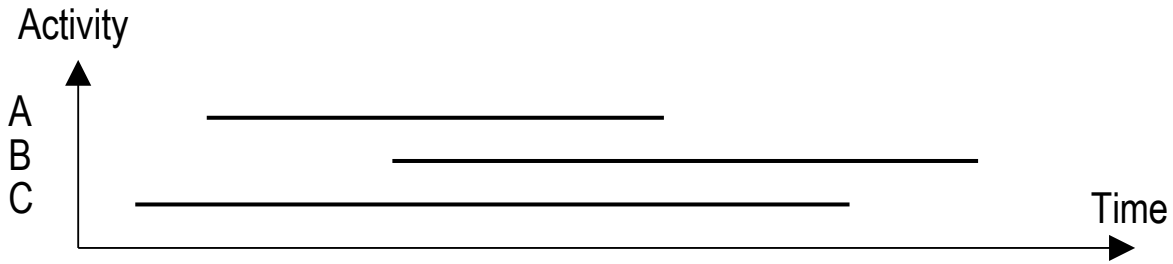
- Sequential computations
- Parallel/physical environment
- Shared data
- Mutual exclusion
- Signaling
- Timing and correctness



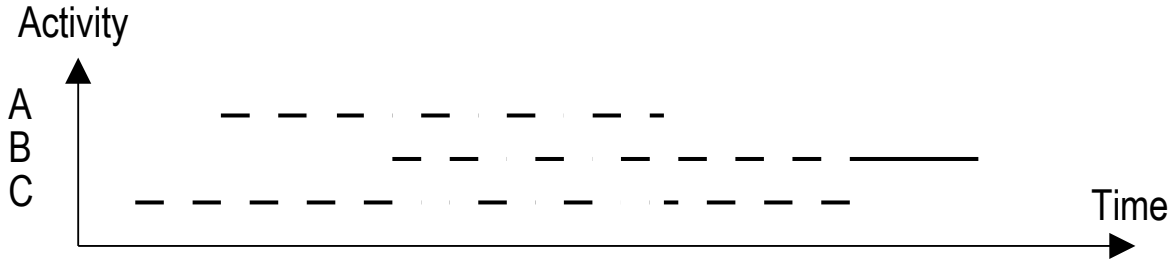
Programming

- Java (or C/C#/..) execution
- Concurrent threads (or processes or tasks)
- OOP
- Mutex-semaphore
- Counting-semaphore
- Real-time scheduling

Concurrency



Logical concurrency

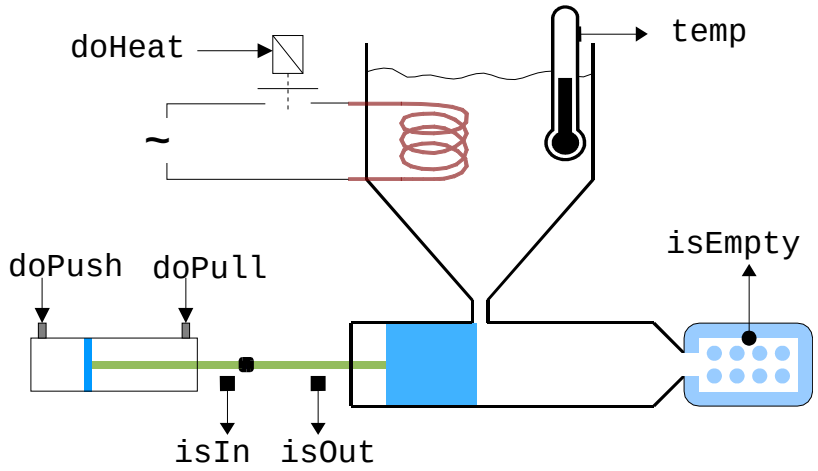


Time-sharing concurrency

Each concurrent activity needs to be a logically correct sequential program.
All concurrent activities together must be concurrently correct.

The LEGO-brick machine – 1

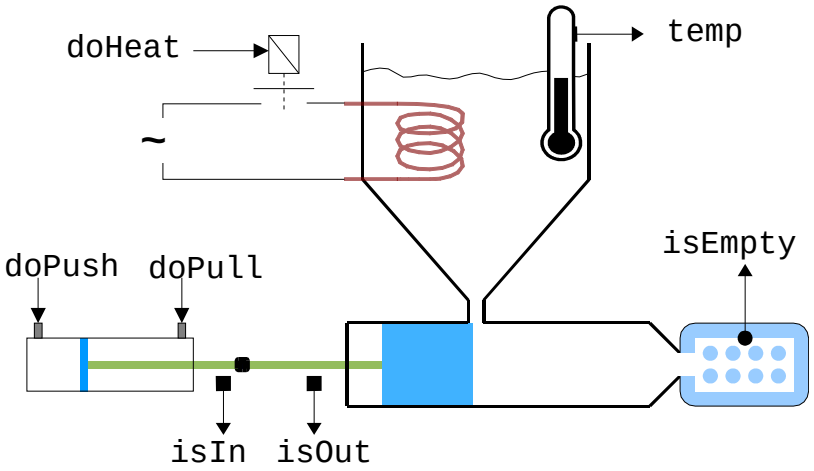
Control temperature (periodic) and piston (events). With one program assuming fast piston:




```
// Starts with piston back
while (true) {
   while (! isEmpty) {
      tempControl();
      sleep(tsamp);
   }
   On(doPush);
   while (! isOut) {
      tempControl();
      sleep(tsamp);
   }
   off(doPush);
on(doPull);
   while (! isIn) {
      tempControl();
      sleep(Ts);
   }
   off(doPull);
}
```

The LEGO-brick machine – 2

More natural with two concurrent programs:




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

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

[Sequential] Software execution

```
int x = 2;
while (x>1) {
     if (x==10) x = 0;
     x++;
};
/* What is x now? */
```

Answer: 1 (of course).

Why do many beginners give the answer 0?

They do not think sequentially yet.

The bank account – 1

Case 1: Withdraw first

A: Read 5000
A: Amount = 5000 - 1000
A: Write 4000
B: Read 4000
B: Amount = 4000 + 10000
B: Write 14000

Case 2: Salary first

B: Read 5000
B: Amount = 5000 + 10000
B: Write 15000
A: Read 15000
A: Amount = 15000 - 1000
A: Write 14000

Two activities (threads or processes) independently make their transactions, but not at the same time. Correct result in both cases.

The bank account - 2

Timing 1:

A: Read 5000
B: Read 5000
A: Amount = 5000 - 1000
B: Amount = 5000 + 10000
A: Write 4000
B: Write 15000

Timing 2:

A: Read 5000
B: Read 5000
B: Amount = 5000 + 10000
B: Write 15000
A: Amount = 5000 - 1000
A: Write 4000

Concurrency fault: Wrong result for some interleavings.

Needed: Mutual exclusion (critical sections, atomic actions)

Critical sections

- Parts of a program that access a shared resource
- May not be interrupted by each other, or by another invocation of itself.
- Accomplished by Semaphores, Monitors, or Mailboxes.
- In low-level/native code, interrupts can be disabled.

Real-world objects and actions

Sequential	*Parallel*
• Single actor actions.	• Physical dynamics.
• Software execution.	• Electronic hardware.

Embedded computers; control software:

Sequential programs behaving concurrently and in real time to control a parallel environment.

- The program is part of a control loop.
- We must think both sequentially and concurrently.

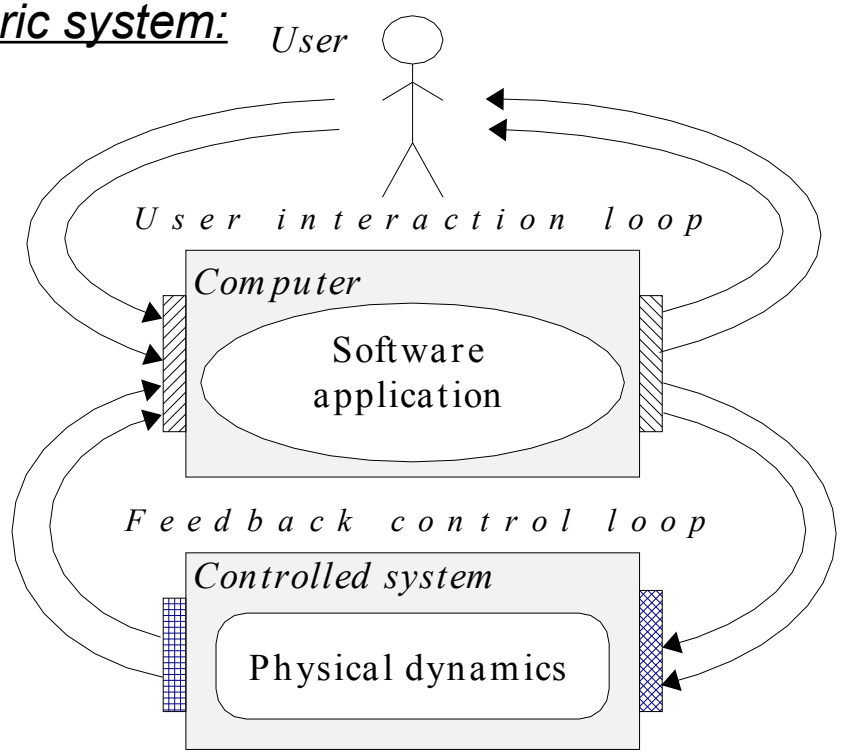
Concurrent computing

The software must

- perform computations **logically** correct
- react on input events **concurrently**

otherwise the systems may fail (is not correct).

Generic system:



The dynamics can be informational (see bank ex.), with no strict real-time requirements.

The *software application* should be a *reactive* system, responding to time increments and external events, but not consuming computing when nothing happens.

A Thread

```
class MyActivity extends Thread {
    public MyActivity(Object argument) {
        // Init here, done before start.
   }
    public void run() {
        while (true) {
            // The work done after start.
        }
   }
}
```

Class Thread – part 1

```
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 the run of 'target'.

    void start();                     // Create thread that calls run.
    void run() {};                    // Work defined in subclass.

    static Thread currentThread();    // Get executing thread.

    void setPriority(int pri);        // Change the priority to 'pri'.
    int getPriority();                // Return priority of thread.
```

Class Thread – part 2

```
    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.
    static boolean interrupted(); // Check+clear flag for current Thread.
    boolean isInterrupted();       // Check flag for this thread.
    boolean isAlive();             // True if started but not dead.
    void join();                   // Waits for this thread to die.
}
```

A ***`Thread`*** *object* represents a concurrent thread of execution, but it *is just an object*! Hence, if you call run, there is no concurrency, etc. etc.

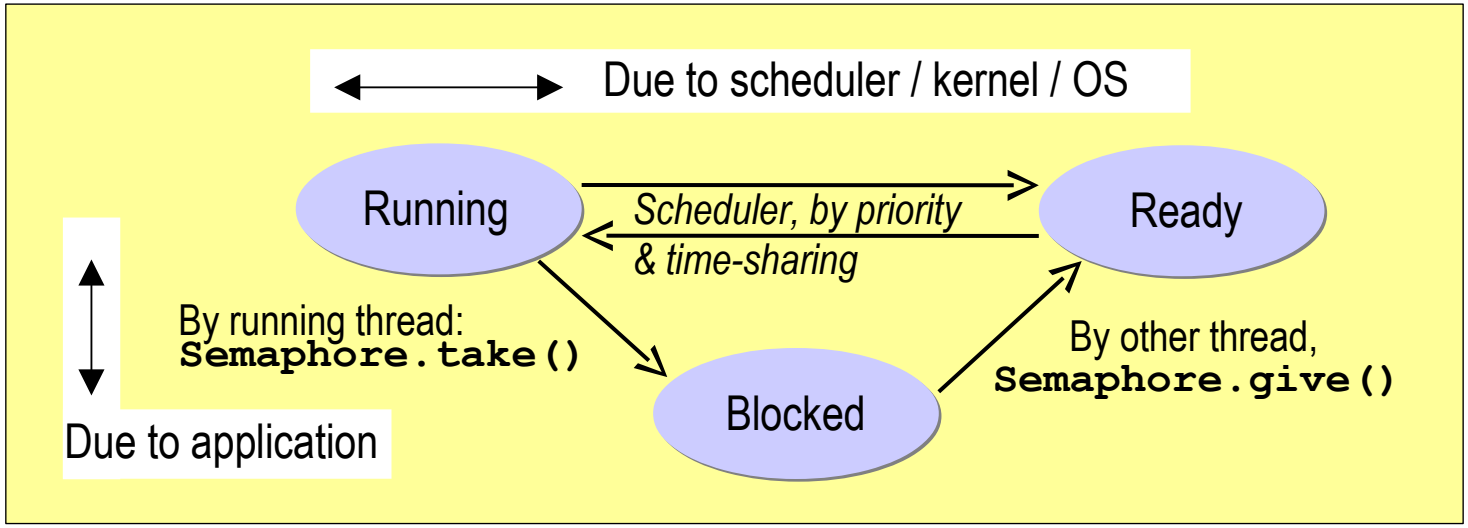
The thread of execution runs the called methods of any object, and in such a method (e.g., in passive object; not being a thread) the calling thread can be obtained via call of static method: **`Thread caller = Thread.currentThread();`**

Runnable

```
public interface Runnable {
    public void run();
}
```

Implement `Runnable` and and pass object (typically `this`) to `Thread` constructor (see next slide), and the thread object will use the provided `run` method. The class can then inherit from something else than a `Thread`, but still become a `Thread` object.

Execution states and semaphores



Scheduling state:

- Running
- Ready
- Blocked

Required [Lab 1]:

You have to use Semaphores appropriately in your code such that is gets concurrently correct!

Semaphore - basics

Minimal mechanism for synchronization
Positive integer variable with two operations, take and give:

```
class SemaphorePrinciple {
    int count;
    public void take() {
              while (count < 1)
             “suspend executing process”;
         --count; // Got the semaphore, continuing ....
    }
    public void give() {
                 if (”any thread suspended”)
                          ”resume the first one in queue”;
                 count++;
    }
}
```

- NOTE: `take` and `give` are ’**atomic**’ (odelbara) operations which require system support to implement, for example by disabling hardware interrupts or a Test-and-Set instruction.
- NOTE: In `take`, as long as `count==0` the caller is **blocked**, i.e. the execution is stopped which differs from methods that can be implemented by ordinary Java code.

Java – Mutex Semaphore

- Declaration

```
import se.lth.cs.realtime.semaphore.*;
Semaphore mutex1;          // Not that good/clear.
MutexSem mutex2;           // Better, more expressive.
```

- Create, initialize.

```
/* Possible but not good: */
mutex1 = new CountingSem(); // Assigns the value zero.
mutex1 = new CountingSem(1);// Assigns the value 1.

/* Preferred: */
mutex2 = new MutexSem(); // Assigns 1 (or true) to internal state.
```

- Use

```
mutex2.take();
amount += change; // Bank account transaction in critical section
mutex2.give();
```

- Apart form clarity, using a `MutexSem` for mutual exclusion enables better error detection and timing.

Semaphore mutex example

```
import se.lth.cs.realtime.semaphore.*;

class ThreadTest {
  public static void main(String[] args) {
    Thread t1,t2;
    Semaphore s;
    s = new MutexSem(1);
    t1 = new RogersThread("Thread one",s);
    t1.start();
    t2 = new RogersThread("Thread two",s);
    t2.start();
  }
}
```

```
class RogersThread extends Thread {
  String theName;
  Semaphore theSem;
  public RogersThread(
      String n, Semaphore sem) {
    theName = n;
    theSem = sem;
  }
  public void run() {
    theSem.take();
    for(int t=1;t<=100;t++) {
      System.out.println(theName + ":" + t);
      for(int y=1;y<=1000000;y++) { }
    }
    theSem.give();
  }
}
```

Bank account mutual exclusion

Case 1: Withdraw first

A: Lock account (take sem)
A: Read 5000
B: Lock account (take; blocks)
A: Amount = 5000 - 1000
A: Write 4000
A: Unlock (give sem; unblock B)
B: Complete locking
B: Read 4000
B: Amount = 4000 + 10000
B: Write 14000
B: Unlock (give sem)

Case 2: Salary first

B: Lock account (take sem)
B: Read 5000
A: Lock account (take; blocks)
B: Amount = 5000 + 10000
B: Write 15000
B: Unlock (give sem; unblock A)
A: Complete locking
A: Read 15000
A: Amount = 15000 - 1000
A: Write 14000
A: Unlock (give sem)

Note 1:Except the four lines with *italic* font, threads A&B are ready or running, managed by OS scheduler.

Note 2: The “Complete locking” requires that thread to be scheduled; with strict priorities a thread C could pass and lock.

Semaphores - usage

- **Mutual exclusion**

```
// Thread A
mutex.take();
***
mutex.give();
•
```

```
// Thread B
•
mutex.take();
***
mutex.give()
•
```

- **Signaling**

```
// Thread A:
•
elements.give();
•
```

```
// Thread B:
•
elements.take();
•
```

Semaphore signalling example

```
import se.lth.cs.realtime.semaphore.*;

class ThreadTest {
  public static void main(String[] args) {
    Thread t1,t2;
    CountingSem s1,s2;
    s1 = new CountingSem(1);
    s2 = new CountingSem(0);
    t1 = new RogersThread("One",s1,s2);
    t1.start();
    t2 = new RogersThread("Two",s2,s1);
    t2.start();
  }
}
```

```
class RogersThread extends Thread {
  String theName;
  CountingSem mySem,hisSem;
  public RogersThread(String n,
       CountingSem s1,CountingSem s2) {
    theName = n;
    mySem = s1;
    hisSem = s2;
  }
  public void run() {
    for(int t=1;t<=100;t++) {
      mySem.take();
      System.out.println(theName + ":" + t);
      hisSem.give();
      for(int y=1;y<=1000000;y++) { }
    }
  }
}
```

Threads and processes

Threads in OS-process	OS processes	Embedded threads
• Semaphore object	• OS-Sem resource	• Sem object and/or HW
• Monitor (shared data)	• File, file-lock	• Monitor (on-board memory)
• Messages (event buff)	• Pipe, socket	• Messages (network, fieldbus, event buff)
• sub-milliseconds	• milliseconds	• microseconds
• Hosted in os-process	• Hosted in computer	• Hosted on board
• Language support	• OS/API support	• Language support
• class Thread	• classes System, Runtime, Process	• class Thread
EDA040 practice	*See EDA050*	*EDA040 content*

To Do

- Read provided pages according to home page.
- Study Exercise 1 and the answers; understand.
- The Lab 1 (Exercise 2) is your assignment, and you should be able to run your final program on the hardware.
- Compare with the LEGO-brick machine:
  - Two activities: One time-based (incrementing the time each second), and one event-based (reacting on buttons). I.e., similar. Semaphores needed for signaling (from buttons).
  - Shared data: The current time, the alarm time, and the display mode. I.e., different, since activities are connected in both hardware and software. Mutual exclusion as for the bank account needed. Semaphores for mutual exclusion.

Design, implement, debug, and show your programmed hardware!