http://cs.lth.se/EDA040

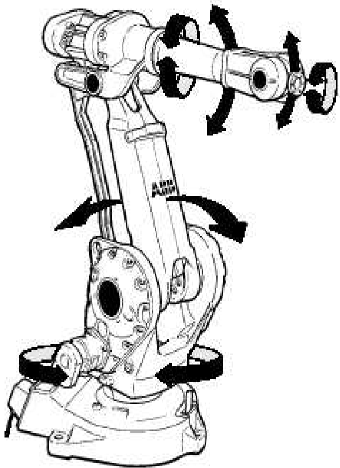
Lectures during HT 2010: EDA040

Concurrent and Real-time programming

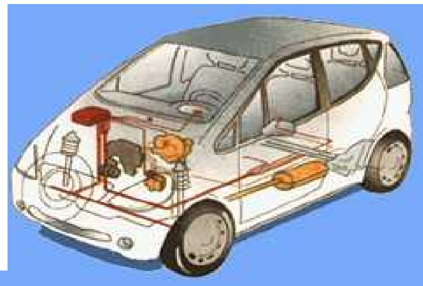
Department of Computer Science, Lund

klas@cs.lth.se

Lecture 9 [F9]: **Summary and hints for the Exam**



Example Products



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.

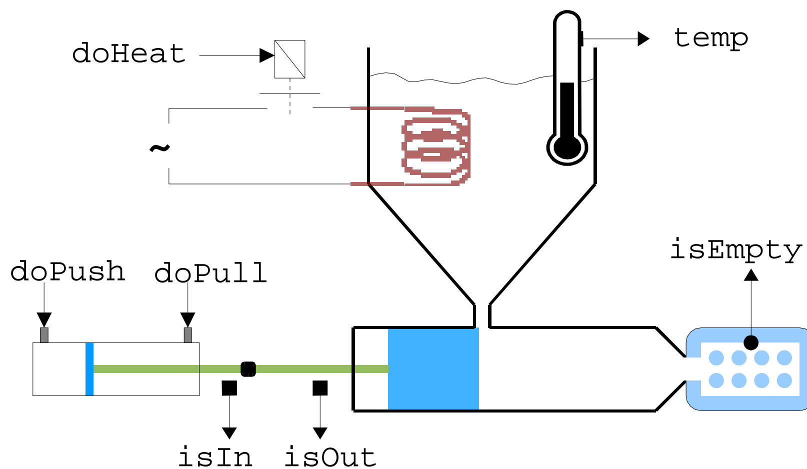
Note 2: The "Complete locking" requires that thread to be scheduled; with strict priorities a thread C could pass and lock.

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);
}
```



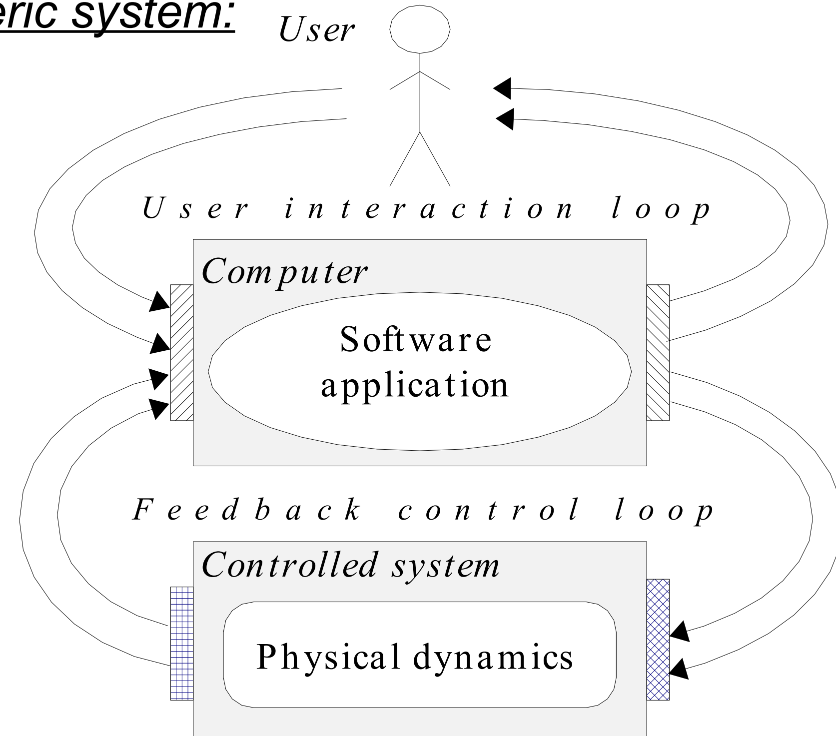
Concurrent computing

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.

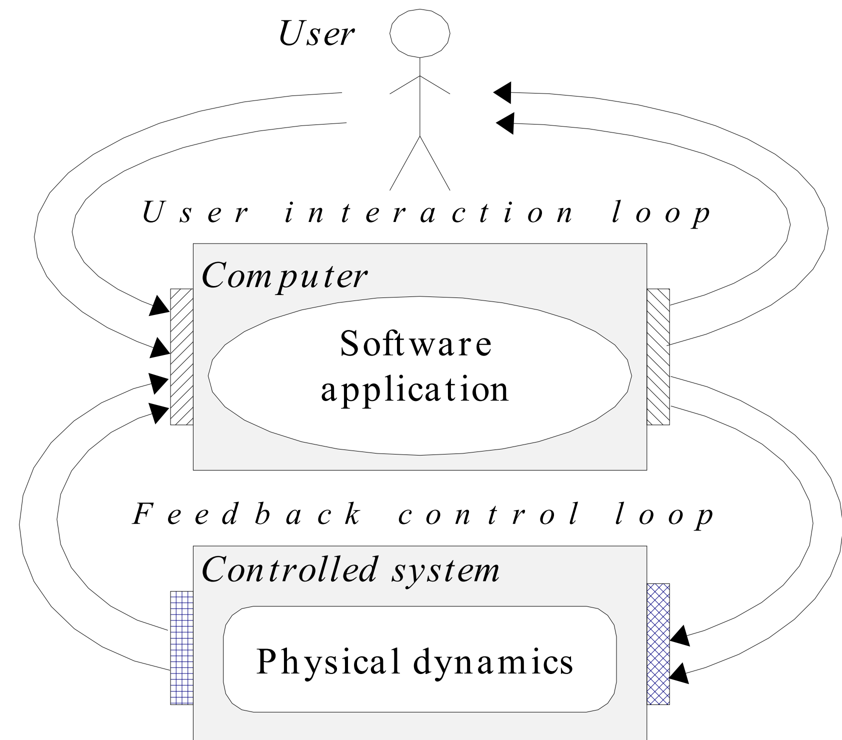
The *software application* should be a *reactive* system, responding to time increments and external events, but not consuming computing when nothing happens.

Concurrent real-time computing

The software must

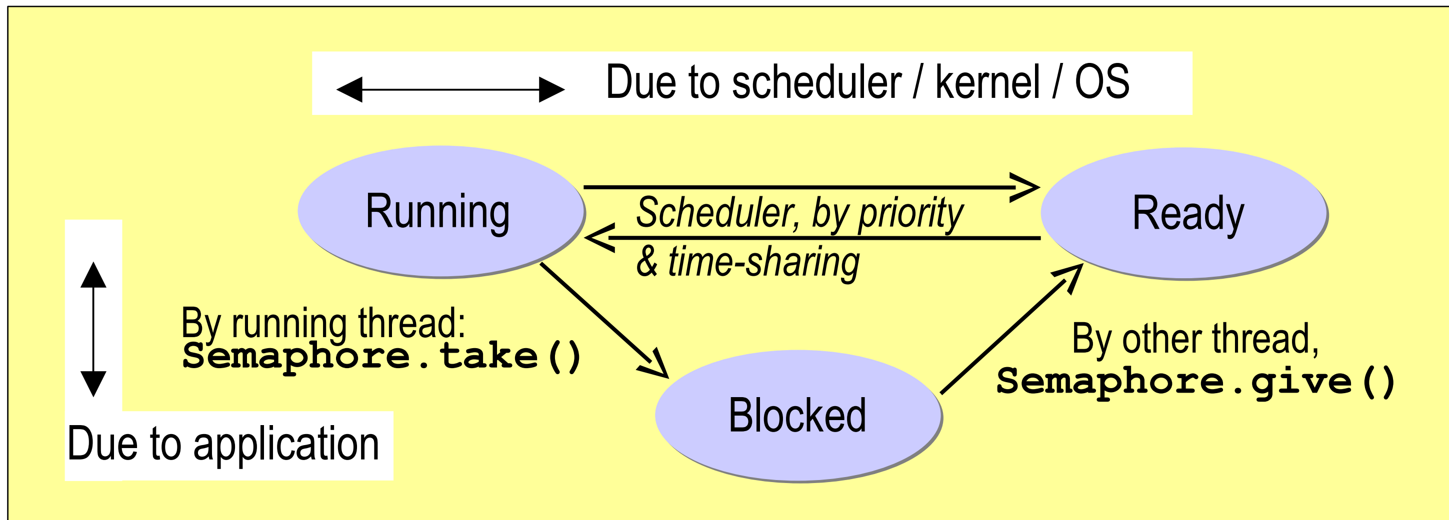
- perform computations *logically* correct
- act on input events *concurrently*
- respond **timely**

otherwise the systems may fail (is not correct).



For ***real-time correctness***, the software (when run on an appropriate platform/OS) must ensure that the concurrency-correct ***result/output is produced on time***.

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!

Threads and processes

Threads in OS-process

- Semaphore object
- Monitor (shared data)
- Messages (event buff)

- sub-milliseconds

- Hosted in os-process
- Language support
- class Thread

EDA040 practice

OS processes

- OS-Sem resource
- File, file-lock
- Pipe, socket

- milliseconds

- Hosted in computer
- OS/API support
- classes System, Runtime, Process

See EDA055

Embedded threads

- Sem object and/or HW
- Monitor (on-board memory)
- Messages (network, fieldbus, event buff)
- microseconds

- Hosted on board
- Language support
- class Thread

EDA040 content

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.

Context switch

A *context switch* takes place when the system changes running process/thread.

In a typical preemptive kernel, switching from one thread to another may look like:

Step	Phase
• Turn off interrupts. • Push PC, CPU registers (Ax, Dx, SR) on stack.	Save
• Save stack pointer in process record. • Get new process record and restore stack pointer from it.	Switch
• Pop CPU registers (SR, Ax, Dx) from stack, and pop PC. • Turn on interrupts.	Restore

Hence, each thread has its own stack, allocated at thread creation on the heap.

Preemption

The preemption strategy of the system determines when a context switch can occur.

- *Nonpreemptive scheduling*: The running thread continues until it voluntarily releases the CPU;
  - explicitly by calling `yield()` or
  - implicitly via `(synchronized)` operations that may block.
- *Preemptive scheduling*: The (HW interrupt driven) scheduler can interrupt the running process at any time.
- Scheduling based on *preemption points*: A context switch can only occur at certain points (def. by lang. or run-time syst.)

For proper timing, our programs assume **preemption**!

Java as such does not prescribe the preemption model.

Monitors (objects & concept)

- In OOP we use classes as a (more powerful) mean to accomplish abstract data-types.
- Objects with such mutually exclusive methods are then monitor objects.

- With methods like

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

the monitor concept is implemented by using semaphores.

Object categories

- Plain passive object:
  - Thread safe by reentrant methods (`java.lang.Math`)
  - Explicitly thread unsafe; to be used by a single thread (`java.util.HashSet`)
  - Implicitly thread unsafe; has to be assumed if not documented.
- Monitor object:
  - Mutually exclusive methods, e.g., by using synchronized.
  - Should be passive; do not mix monitors and threads!
- Thread object:
  - Active object (if started but not terminated); drives execution.
  - Don't call me, I'll call you!

synchronized - wait - notify

Condition queue

In addition to locking the object for exclusive access (*mutex*): Temporarily unlock until someone *signals* that the state has changed:

Real-time Monitor

We assume these monitor properties:

- High priority threads should be given precedence, even to threads which have been waiting longer (desired starvation risk).
- Immediate resumption not guaranteed (depends on OS/scheduler)
- The condition being waited for might not be true anymore when a blocked thread resumes execution.
- Waiting for a condition must be coded: `while (!ok) wait();`
- Use '`notifyAll`' to avoid problems (practical - wakes all).
- The notify not necessarily called last in the method.

When previously blocked threads precedes those with same priority + notify last + one level of priority: equivalent with Hoare Monitor.

A special type of monitor is the buffer, forming a *mailbox for messages*

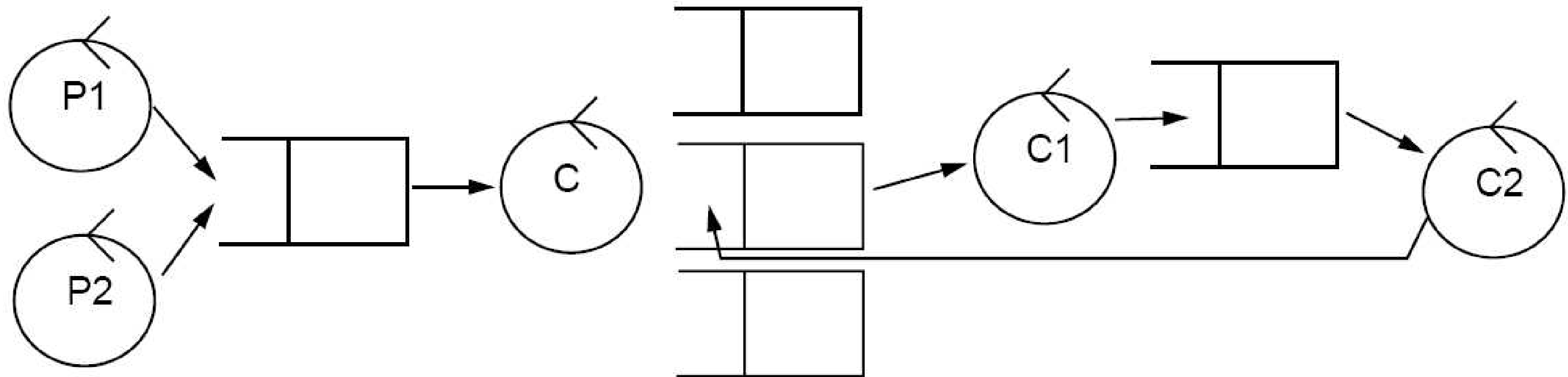
- *While monitors in general are for operations on shared data, a monitor with operations post (called by a producer thread) and fetch (called by consumer thread) comprises a data flow.*
- *Data can provide information and/or synchronization.*
- *Originally and traditionally data is then referred to as* ***messages****, and the buffer is a* ***mailbox****.*
- *Between threads (the same program and memory space) a message can be an* **`Object`** *ref.*

```
class Buffer // Providing mailbox
{
  synchronized void post(Object obj)
  {
    while (buff.size()>=maxSize) {
      wait();
    }
    if (buff.isEmpty()) notifyAll();
    buff.add(obj);
  }

  synchronized Object fetch()
  {
    while (buff.isEmpty()) {
      wait();
    }
    if (buff.size()>=maxSize) notifyAll();
    buff.remove(buff.size());
  }
}
```

Systems of mailboxes

Communication between threads often forms a network of mailboxes.



- The same principle for (operating system) processes and threads.
- A thread can put a letter in several mailboxes.
- A mailbox can, in Java, handle various types of messages - subclassing of message (**`RTEvent`**).
- A thread has in most cases only one mailbox it reads from (otherwise problems - fetching a messag is a blocking operation).
- Message objects need to be ***serialized*** (transformed into a stream of bytes) in order to be sent to another OS process.
- Within a OS process (between threads), we can send pointers/references or a copy of the object.
- How does the receiver know that another thread does not modify the contents of a message???

Mailbox == Monitor == Semaphore

A Mailbox can easily be implemented using a Monitor

Also a Semaphore is sort of a monitor.

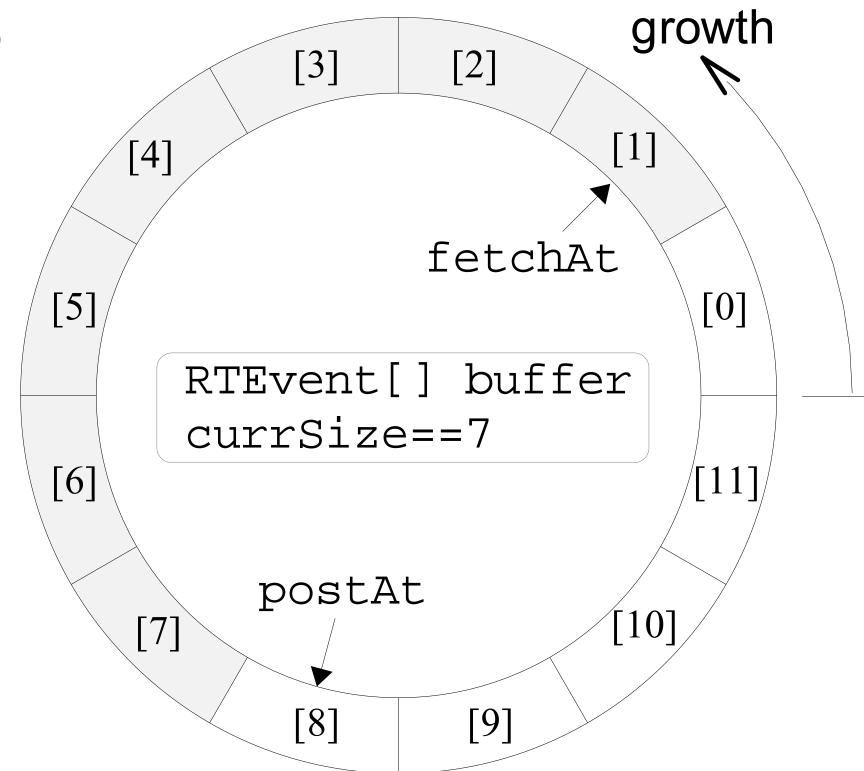
Suppose we only send empty messages, then a Mailbox is equivalent to a Semaphore:

- The value of the counter of the Semaphore corresponds to the number of messages in the mailbox.
- Send message - give()
- Receive message - take()

All three constructions are thus equally powerful, but practical in different situations.

The RTEventBuffer class

- As `RTEvent`, part of `se.lth.cs.realtime.`**`event`**
- Constructors:
```
public RTEventBuffer()
RTEventBuffer(int maxSize)
RTEventBuffer(int maxSize,
              Object lock)
```
- Example with `maxSize==12` and `currSize==7`, internal attributes:
- Obtain message/event by `RTEvent` **`fetch`**`()` or specific final methods.
- Send massage/event by `RTEvent` **`post`**`(RTEvent ev)` or specific final methods.



The JThread utility class

- Part of the `se.lth.cs.realtime` package
- Subclass of `java.lang.Thread`; thus it is a **Java Thead**, hence **`JThread`**.
- Encapsulates an `RTEventBuffer`, exposed via a public **`putEvent`** method.
- Default `run` method is a cyclic call of `perform`
- Internally the `perform` (or `run`) method refers to the `mailbox` attribute like

```
event = mailbox.doFetch();
```

- Methods `sleepUntil` and `terminate` are also provided (compare lab).

Exercises and Labs

Exercise 1:

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

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.

Exercises and Labs

Exercise 3:

- Monitors
- Wait and notify
- Use while for checking conditions.

Lab 2 (&exercise 4):

- Deadlock
- Monitors: Sharing data the OOP way.
- Elevator simulation.

Lab 3 (&exercise 5):

- Mailboxes and message passing.
- Washing machine.

Exercise 6:

- Sheduling.
- Blocking
- Schedulability analysis.

The exam

Separately:

- Comments on earlier exam (transparencies)
- Exam hints and date on home page
- Course developments (todos):
  - Booklet (revised and next two chapters)
  - Java2C (https://launchpad.net/ljrt)
  - Many more code examples (in proj too)
  - More on deployment and networking.
  - RT/Java-reference