http://cs.lth.se/EDA040

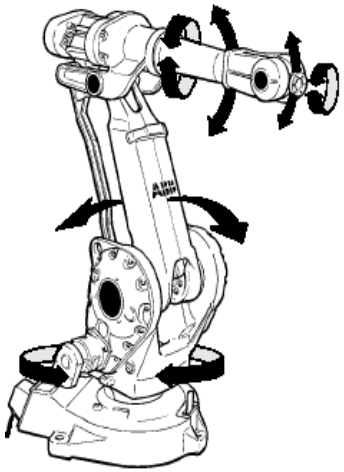
Lectures during HT 2013: EDA040

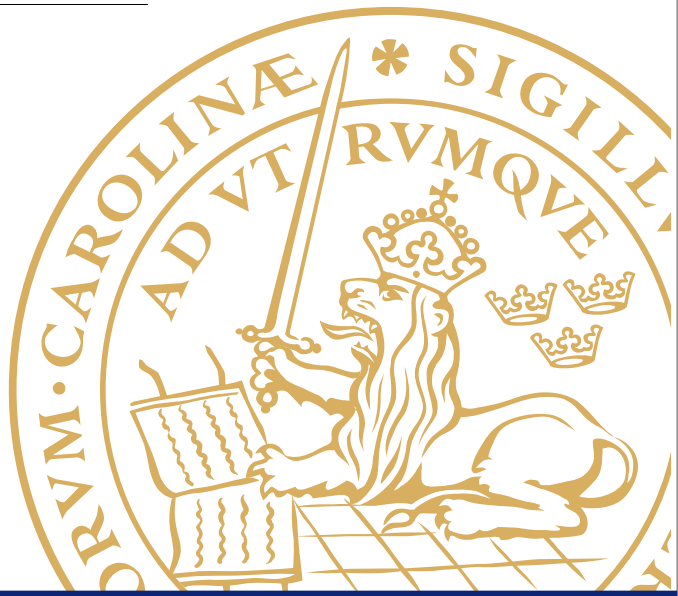
Concurrent and Real-time programming

Department of Computer Science, Lund

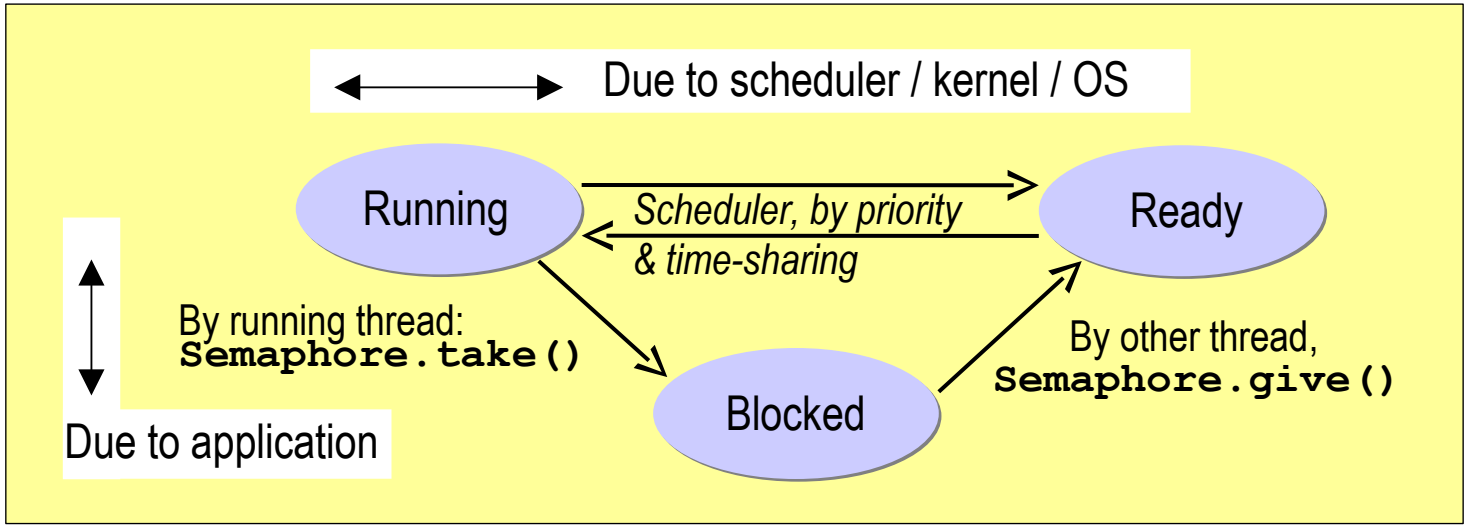
klas@cs.lth.se

Lecture 2 [F2]: More on Semaphores and threads of execution





Execution states and semaphores [F1]



Scheduling state:

- Running
- Ready
- Blocked

Required [Lab 1]:

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

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:

• 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

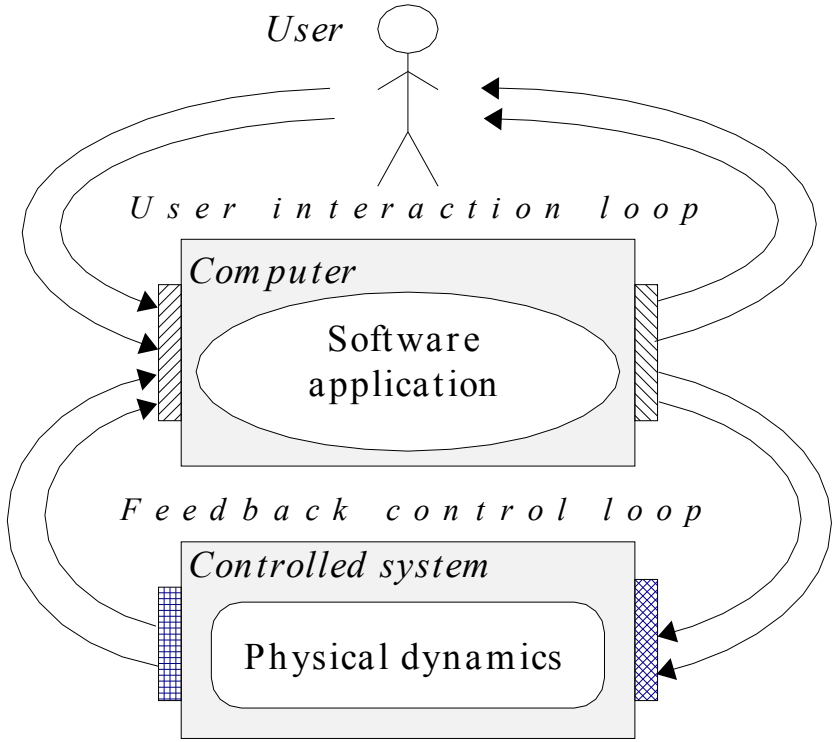
Hence, each thread has its own stack, allocated at thread creation on the heap.

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***.

Preemption

http://en.wikipedia.org/wiki/Preemption_%28computing%29

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.

Mutual exclusion - can it be implemented without system calls?

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

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

```
ad {
{

tion();

n();
;
  }
}
```

Critical Section (CS)

- Like the three lines of code from the bank account example.
- We will concentrate on the construction of pre/postProtocol.
- Assumption: A thread will not block inside its critical region.
- Requirements:
  Mutual exclusion, No deadlock, No starvation, and Efficiency.

Mutual exclusion – the thread

```
class T extends Thread {

  public void run() {
    while (true) {
      nonCriticalSection();
      prePotocol();
      criticalSection();
      postProtocol();
    }
  }
}
```

Critical Section (CS)

- Like the thread-class example from the bank account example.
- We will concentrate on the construction of pre/postProtocol.
- Assumption: A thread will not block inside its critical region.
- Requirements:
  Mutual exclusion, No deadlock, No starvation, and Efficiency.

Required Mutex Properties

Requirements:

R1. Mutual exclusion: Execution of code in critical sections must not be interleaved.

R2. No deadlock: If one or more threads tries to enter a CS, one must do so eventually.

R3. No starvation: A thread must be allowed to enter its CS eventually.

R4. Efficiency: Small overhead when only one active thread.

Can that be accomplished by ordinary (Java) code?

Mutual exclusion – version 1

```
int turn=1;
```

```
class V1 extends Thread {
  public void run() {
    while (true) {
      nonCS1();
      while (turn!=1);
      CS1();
      turn = 2;
    }
  }
}
```

```
class V1 extends Thread {
  public void run() {
    while (true) {
      nonCS2();
      while (turn!=2);
      CS2();
      turn = 1;
    }
  }
}
```

R1. **Mutual exclusion:** OK

R2. **No deadlock**: OK since one of the threads can always proceed.

R3. **No starvation**: Alternating protocol; OK.

R4. **Efficiency**: Does not work for one thread only. Busy-wait; inefficient! No good for many threads.

#: Not acceptable!

Mutual exclusion – version 2

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

```
class V2 extends Thread {
  public void run() {
    while (true) {
      nonCS1();
      while (c2!=1);
      c1 = 0;
      CS1();
      c1 = 1;
    }
  }
}
```

```
class V2 extends Thread {
  public void run() {
    while (true) {
      nonCS2();
      while (c1!=1);
      c2 = 0;
      CS2();
      c2 = 1;
    }
  }
}
```

R1. **Mutual exclusion:** NO!
E.g. with the interleaving:

```
c1 = 1;
       c2 = 1;
while (c2!=1);
       while (c1!=1);
c1 = 0;
       c2= 0;
CS1();
       CS2();
```

#: Not a solution, but could work for a long time (until interrupt in pre1 or pre2)!

Mutual exclusion – version 3

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

```
class V3 extends Thread {
  public void run() {
    while (true) {
      nonCS1();
      c1 = 0;
      while (c2!=1);
      CS1();
      c1 = 1;
    }
  }
}
```

```
class V3 extends Thread {
  public void run() {
    while (true) {
      nonCS2();
      c2 = 0;
      while (c1!=1);
      CS2();
      c2 = 1;
    }
  }
}
```

R1. **Mutual exclusion:** OK.

R2. **Deadlock (but still using the CPU)**:

```
c1 = 0;
      c2 = 0;
while (c2!=1);        // Forever ..
      while (c1!=1); // ..and ever.
....
      ....
```

#: **Not a solution, but could work for a long time!**

Mutual exclusion – version 4

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

```
class V4 extends Thread {
  //..
      nonCS1();
      c1 = 0;
      while (c2!=1){
        c1 = 1;  //**
        c1 = 0;
      }
      CS1();
      c1 = 1; //..
}
```

```
class V4 extends Thread {
  //..
      nonCS2();
      c2 = 0;
      while (c1!=1){
        c2 = 1;  //**
        c2 = 0;
      }
      CS2();
      c2 = 1; //..
}
```

R1. **Mutual exclusion:** OK (as for V3).

R2. **No Deadlock:** OK (yield at *//***).

R3. **No Starvation**: Failure, one thread may execute but never get the resource (called *Livelock;* threads neither block progress).

#: Not acceptable!

```
c1 = 0;
     c2 = 0;
     while (c1!=1);
     c2 = 1;
while (c2!=1);
CS1();
c1 = 1;
nonCS1();
```

```
c1 = 0;
     c2 = 0;
     while ..
     c2 = 1;
while (c2!=1);
CS1();
c1 = 1;
nonCS1();
```

Dekkers Algorithm

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

```
class DA extends Thread {
  //..
    nonCS1();
    c1 = 0;
    while (c2!=1){
      if (turn==2){
        c1 = 1;
        while (turn==2);
        c1 = 0;
      }
    }
    CS1();
    c1 = 1;
    turn = 2;
  //..
```

```
class DA extends Thread {
  //..
    nonCS2();
    c2 = 0;
    while (c1!=1){
      if (turn==1){
        c2 = 1;
        while (turn==1);
        c2 = 0;
      }
    }
    CS2();
    c2 = 1;
    turn = 1;
  //..
```

R1. **Mutual exclusion:** OK.

R2. **No Deadlock:** OK.

R3. **No Starvation:** OK.

R4. **Efficiency**: Not good!

#: Dekkers Algorithm (can be extended to many threads, but gets very complex) solves the mutex problem, but with busy-wait (CPU used also when nothing to do).
Useful in some multi-processor systems.

Mutual exclusion – semaphore

```
MutexSem mutex = new MutexSem();
```

```
class V1 extends Thread {
  public void run() {
    while (true) {
      nonCS1();
      mutex.take();
      CS1();
      mutex.give();
    }
  }
}
```

```
class V1 extends Thread {
  public void run() {
    while (true) {
      nonCS2();
      mutex.take();
      CS2();
      mutex.give();
    }
  }
}
```

R1. **Mutual exclusion:** OK

R2. **No deadlock**: OK.

R3. **No starvation**: OK (give starts blocked thread directly).

R4. **Efficiency**: Works well also for a signle thread, waiting threads are put to sleep (not using any CPU time).

#: Acceptable!

Test-and-Set

The problem in version 2 arose since the following is not atomic:

```
while (c2!=1)  // Load
c1 = 0;        // Store
```

All computers have an instruction that corresponds to `TestAndSet` which performs both these instruction atomically. It stores a new value and returns the old value:

```
while (TestAndSet(c,0)==0) ;
CS();
c = 1;
```

- A simple solution is thus possible assuming hardware support.
- Still Busy-wait -- inefficient, the waiting thread should be blocked.
- Useful for synchronization in machines with several CPUs and shared memory.
- In the announced JDK1.5 there will be **`compareAndSet`**-methods that implement *Test-and-Set for built-in datatypes*, as part of the new package `java.util.`**`concurrent.atomic`**
- Since we focus on single (embedded) CPUs, you should know about Test-and-Set but it should not be used in your programs within this course!

Variants of Semaphores

Blocked-Set Semaphore

Give - wakes arbitrary waiting thread.

- Starvationwhen N >= 3 - two threads happen to alternate

Blocked-Queue Semaphore

Give wakes threads in FIFO order (i.e. the thread that have been waiting longest)

- Starvation impossible

Blocked-Priority Semaphore

Give wakes the thread that has the highest priority (when equal FIFO order)

- Starvation possible if N >= 3 and two high priority threads, but that is desirable!

Binary Semaphore

- Efficient mutex-implementation in some RTOS, see the `BinarySem` class.

Multistep Semaphore

- To reserve several resources at once/atomically see the `MultistepSem` class.

Semaphore – basics [F1]

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.

Execution thread vs. thread object

- An executing thread is a entity in the run-time system, accessed implicitly via a thread object.
- The thread object, before `start()` has been called, is like any other object (but start is native; not implemented in Java).
- When `myThreadObject.start();` is called, the (native) `start` method calls some OS (Win32, Linux, OS-X, etc) routine that creates the thread of execution (represented by the thread object).
- Start calls `run` that defines the work to perform. *If you call run, the context of the calling thread is used and there are* ***no concurrency added!***

Abstractions [not on exam]

Thread: Performs execution using a processor.

Execution state: Thread status stored in context.

Mutual exclusion: Restriction on context switching

In Java we have

- threads represented by objects of type **`Thread`**,
- state of execution as in sequential programming,
- `synchronized` methods for mutual exclusion.

Objects and concurrency [not on exam]

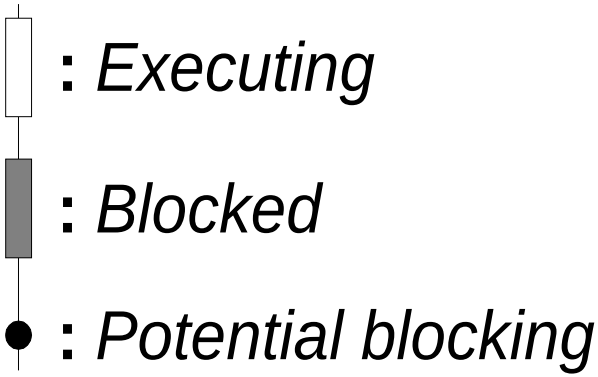
Object properties		Implicit mutual exclusion of methods		Comment
Thread	Exec. state	No	Yes	
No	No	Object 1	Monitor 2	Passive objects
No	Yes	Coroutine 3	’ 4	Not in Java
Yes	No	— 5	— 6	Not useful
Yes	Yes	Thread-object 7	Task 8	Active objects

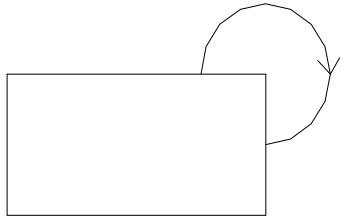
3) Fibers (by Microsoft) when managed by OS.

- We use the abstractions to cope with complexity.
- In Java we will use Objects (as in OOP), plus Monitors and Thread-objects for concurrency.

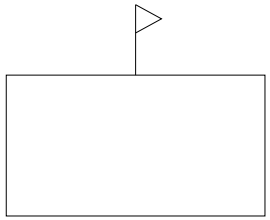
Objects and blocking

- Threads objects are referred to as active objects.
- Other objects are passive objects, called/driven by (the threads represented by) active objects.
- Semaphores provide blocking operations; how to present (potential) blocking in sequence diagrams?



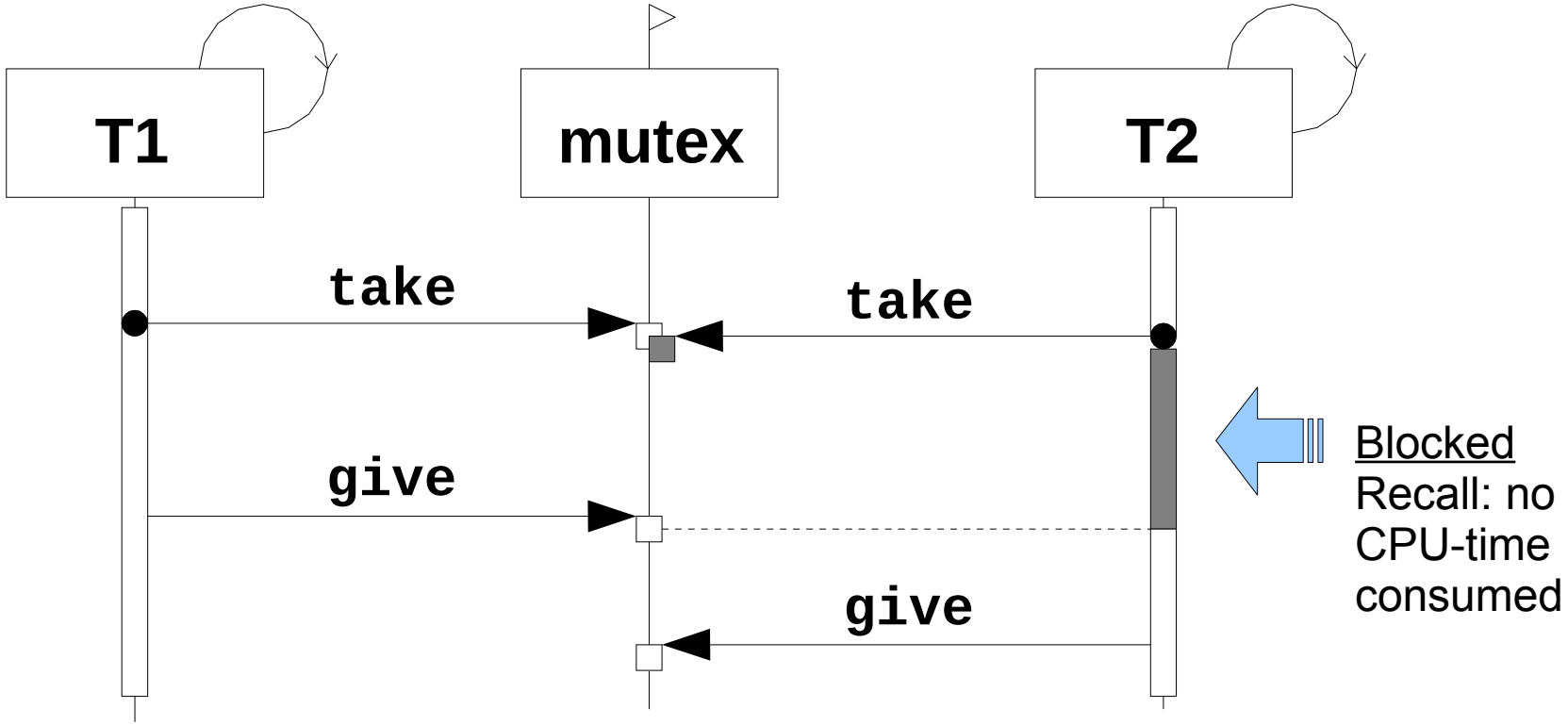


Thread
(active obj.)



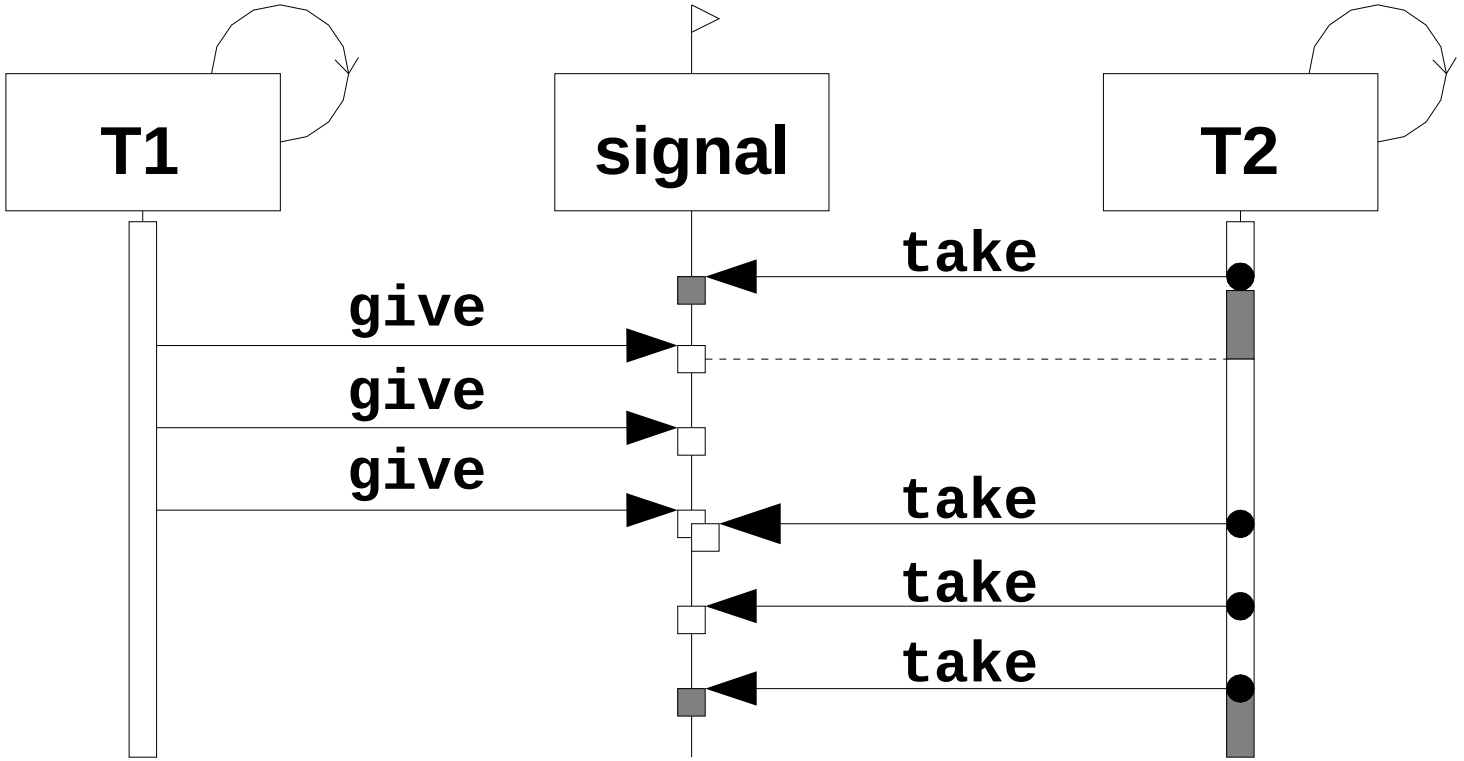
Semaphore
(passive obj.)

Mutex sequence/blocking example



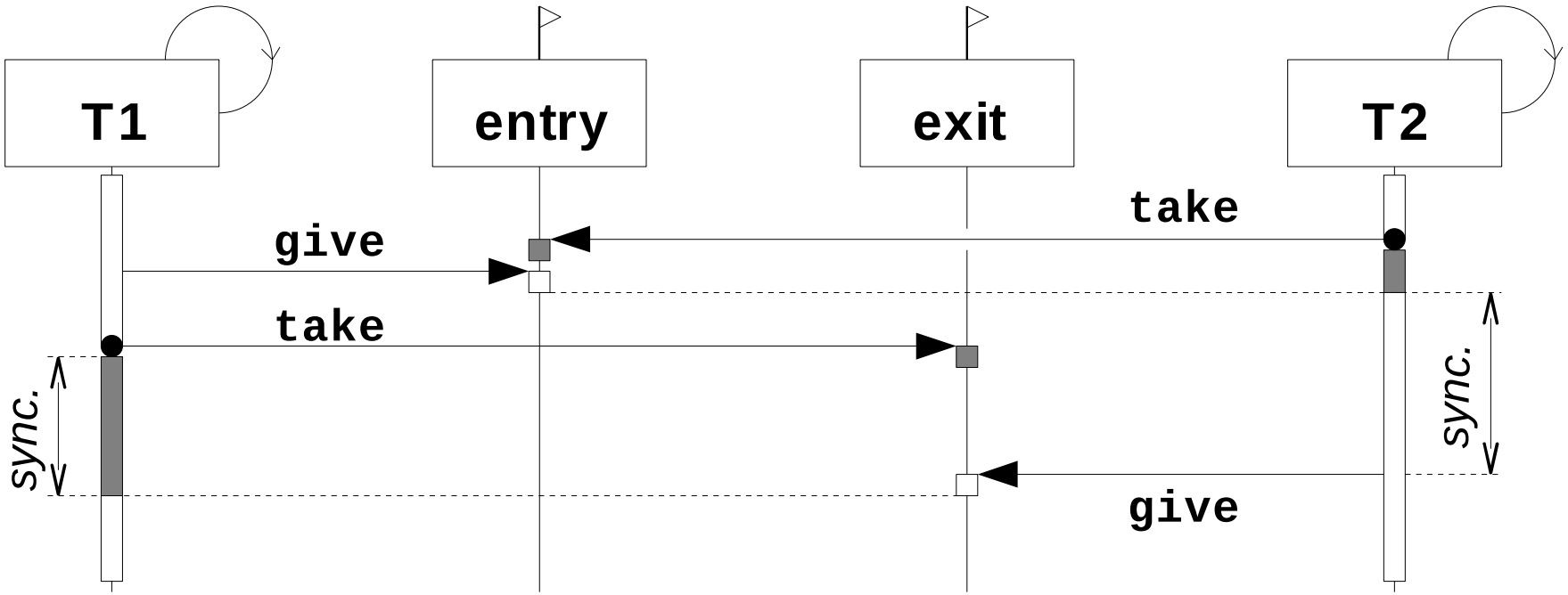
Successive **`take - give`** calls always by the same thread; checked internally in **`MutexSem`** class.

Signaling



- One thread calling **`take`** and another calling **`give`**
- Supported by the **`CountingSem`** class.

Rendez-vous [ref Ada language]



- Safe data transfer/manipulation with T1 by T2 during *sync.*
- T1 has exclusive access except between give and end sync.

Semaphore usage

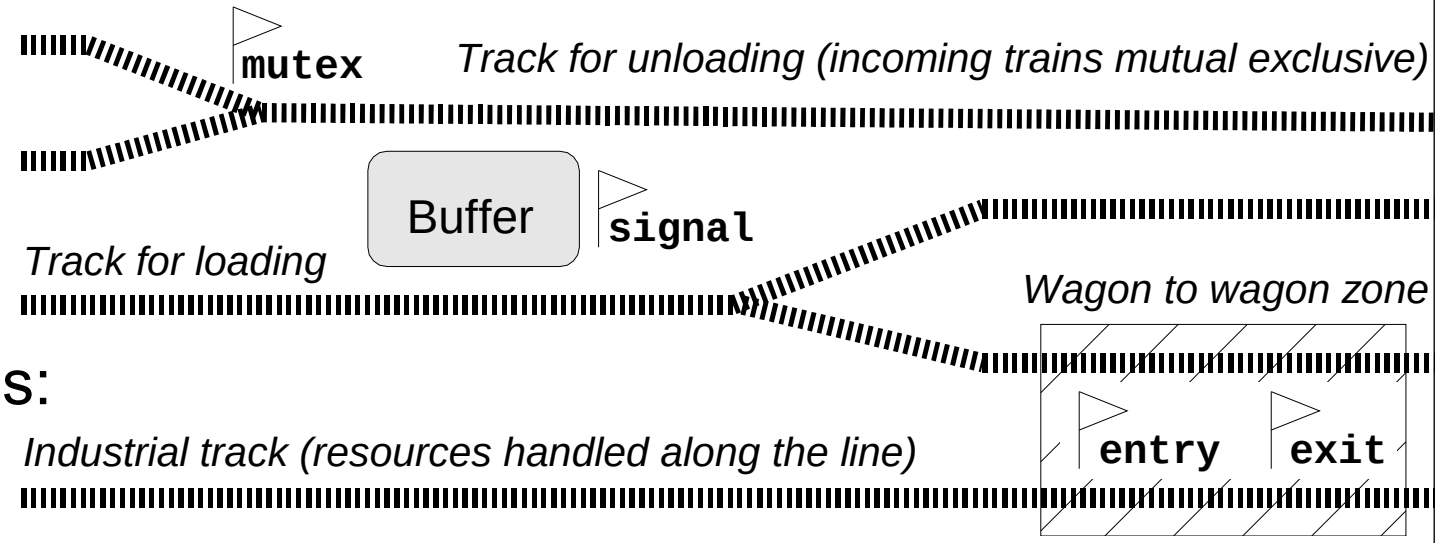
	Thread A:	Thread B:

• Mutual Exclusion	•
	mutex.take();
	***
	mutex.give();
	•
		•
		mutex.take();
		***
		mutex.give()
		•

• Signaling	• buffer.give() •	• buffer.take() •

• Rendez-vous	***
	entry.give();
	exit.take();
	***
		•
		entry.take();
		***
		exit.give();

Semaphore – analogy with railroad tracks (seen from above)



- Mutual Exclusion:
- Signaling:
- Rendez-vous:

- Rendez-vous requires no buffer, but both trains need to be in the tranfer area at the very same time.

Event Processing

- Single-threaded execution in graphics toolkits.
- We will use asynchronous events to promote concurrency (or synchronous depending on object properties).
- The incrementation of time (system clock) is a special (time-based) event.
- Method calls can be done as event processing.
- **When no event (or call) reaches your software, there should be no execution carried out!!!!!!**

Software – practical comments

On blackboard:

- Cross compilation (compiler on host, for target)
- Embedded execution (CPU runs in machine)
- Device drivers (cannot be written in Java)
- Lab practices (how are embedded threads run)

To Do (cross compilation during lab)

- Read booklet 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 signalling (from buttons).
  - Shared data: The current time, the alarm time, and the display mode. I.e., different, since activites are connected in both hardware and software. Mutual exclusion as for the bank account needed. Semaphores for mutual exclusion.

Design, implement, debug, and demo your C-translated software!