http://cs.LTH.se/EDA040

Real-Time and Concurrent Programming

Lecture 2 (F2):

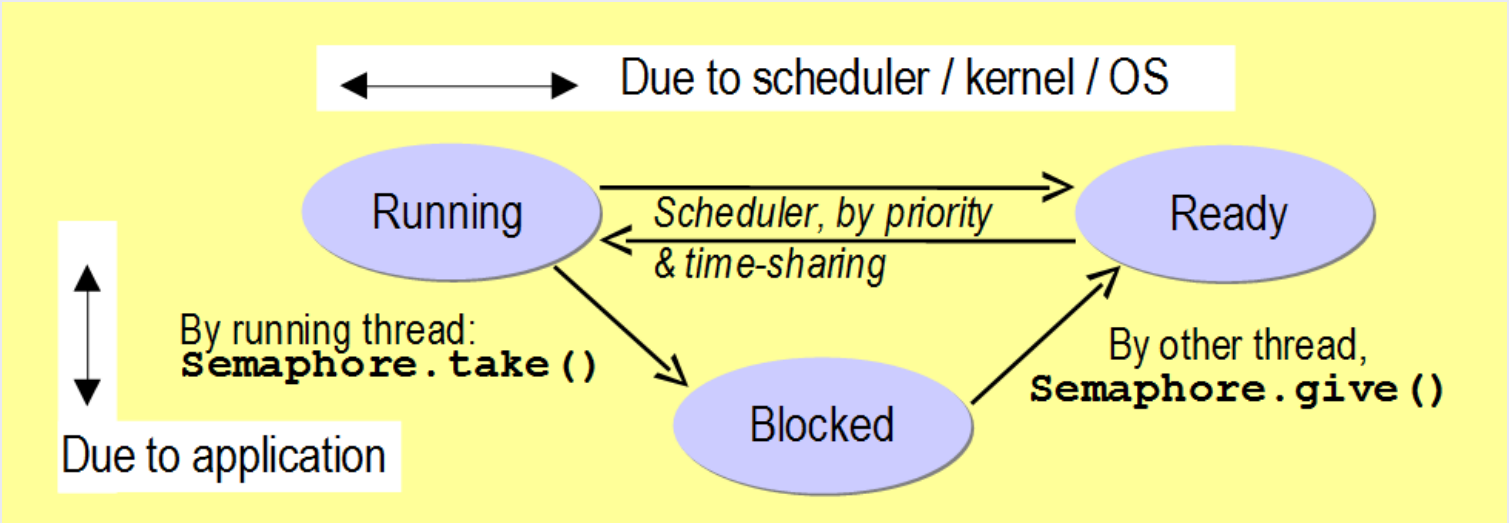
Threads

Klas Nilsson

2016-08-30

Execution states and semaphores

A Thread is concurrently executing within a program, being blocked most of the time



Scheduling states

- Running
- Ready
- Blocked

Required for first lab:

- You have to use Semaphores appropriately in your code such that is gets concurrently correct!
- Proper blocking (only Ready/Running when needed).

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

```
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 – cont.

```
static void sleep(long t);   // Suspend for at least 't' ms.

static void yield();         // Reschedule to let others run.

void interrupt();            // Set interrupt request flag.
static boolean interrupted(); // Check&clear intr. flag.
boolean isInterrupted();     // Check flag without clearing.

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; if you call `run`, there is no concurrency; *you code `run` and call `start`*!
- 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();
}
```

Usage: Implement `Runnable` and pass the object (typically this) to the `Thread` constructor, 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.

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();
    t1 = new MyThread("Thread 1",s);
    t1.start();
    t2 = new MyThread("Thread 2",s);
    t2.start();
  }
}
```

```
class MyThread extends Thread {
  String theName;
  Semaphore theSem;
  public MyThread(
       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();
  }
}
```

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

```
class MyThread extends Thread {
  String theName;
  CountingSem mySem,hisSem;
  public MyThread(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++) {
    }
  }
}
```

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

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

Preemption alternatives

The preemption strategy determines when acontext switch can occur.

Nonpreemptive scheduling:

The running thread continues until it voluntarily releases the CPU (hands over to the scheduler);

- 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.

Preemption points:

A context switch can only occur at certain points (according to language, compiler and/or run-time system)

For proper timing, our programs assume preemptive scheduling!
Java as such does not prescribe the preemption model.

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).
- The `start` method calls `run` that defines the work to perform. If you (not the system) call `run`, the context of the calling thread is used and there are no concurrency added!
- After returning from `run`, the execution thread is dead (`isAlive` returning `false`), but the `Thread` object remains (but cannot be restarted).

Concurrent real-time computing and correctness

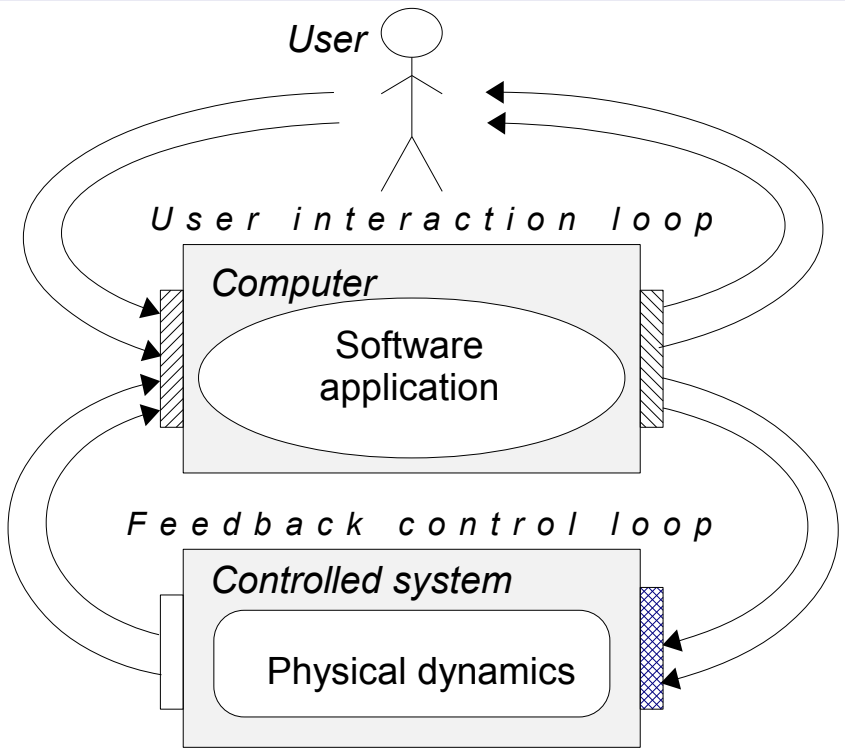
Correctness

The software must

- perform computations logically correct in each activity, and
- react on input events concurrently while giving the correct output for any permitted execution order,
- respond timely, meeting all deadlines!

otherwise the systems may fail (is not correct).

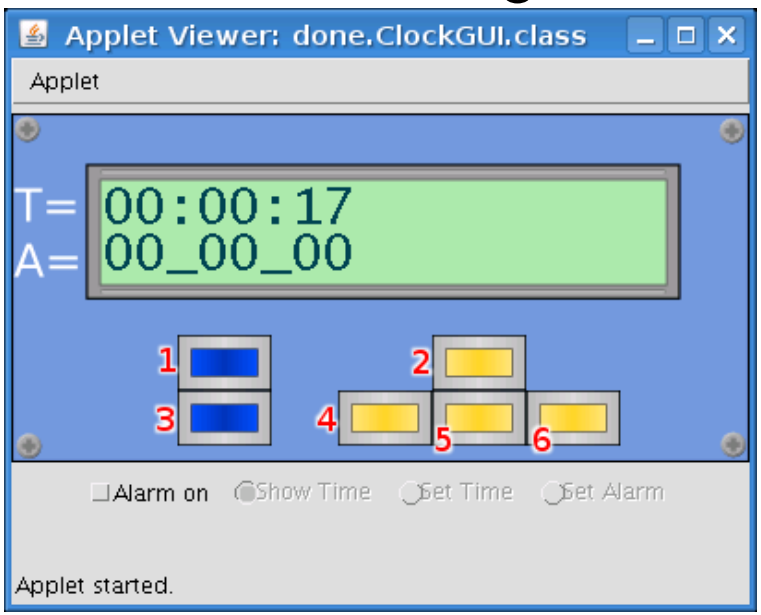
Generic system

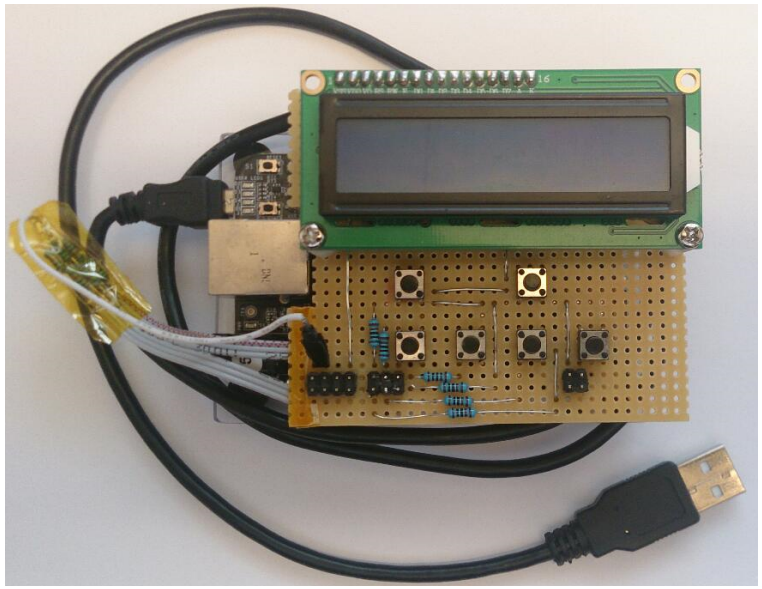


For real-time correctness, the software (when run on an appropriate platform/OS) must ensure that the concurrency-correct **result/output is produced on time**.

Requirements on a solution for Lab 1

Consider: IO (and how is the HW accessed), what is triggering (the need for) execution, what are the activities (threads; active objects) that are motivated by the asynchronous interaction with the environment, what shared data is there (to be put in passive objects with mutual exclusion), data flow, who is calling whom, and where to put the application logic.





Requirements

Specification:

1. Call `showTime` (with the current clock time as an argument) once a second, which will update the display unless the (clock or alarm) time is being edited.
2. Alarm time and clock time should be possible (concurrently correct) to set, but the actual editing of the displayed time value is done outside the application program (in device driver or low-level handler).
3. Check and issue alarm, beeping every second for up to 20 seconds or until any button is pressed.
4. Optional: Implement program termination.
5. Optional: Make ticking available for external calling.

Coding practices 1(2)

The code is required to follow practices for concurrent programming:

- Threads have their data protected (or private).
- No added public (or package visible) methods in thread classes.
- Constructors construct, passive objects (not calling start)
- Thread objects are started by calling start.
- Thread.run is public but should never be called (except via start/OS).
- Signaling is done via a `Semaphore`, or specifically a `CountingSem`.
- Mutual exclusion is done by means of a `MutexSem`, which should be protected within the class(es) for shared data.
- A class for shared data should have no static attributes, and all public methods should have mutex protection. We may refer to those methods as operations on shared data.

Coding practices 2(2)

- All operations on shared data should have an application meaning, and they may not call other such operations on the same object. That implies that small getters and setters based on attributed only should not exist (publicly).
- The software should need practically no CPU time, that is, all threads being blocked almost all the time. Specifically, busy-wait is forbidden.
- Polling is forbidden, so `Thread.sleep` is only useful for awaiting an update of time.
- The number of threads should reflect the required concurrency of the application.
- A minimum of external classes should be used, to promote execution in small embedded systems. Specifically, Date and Time classes should not be used.