CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

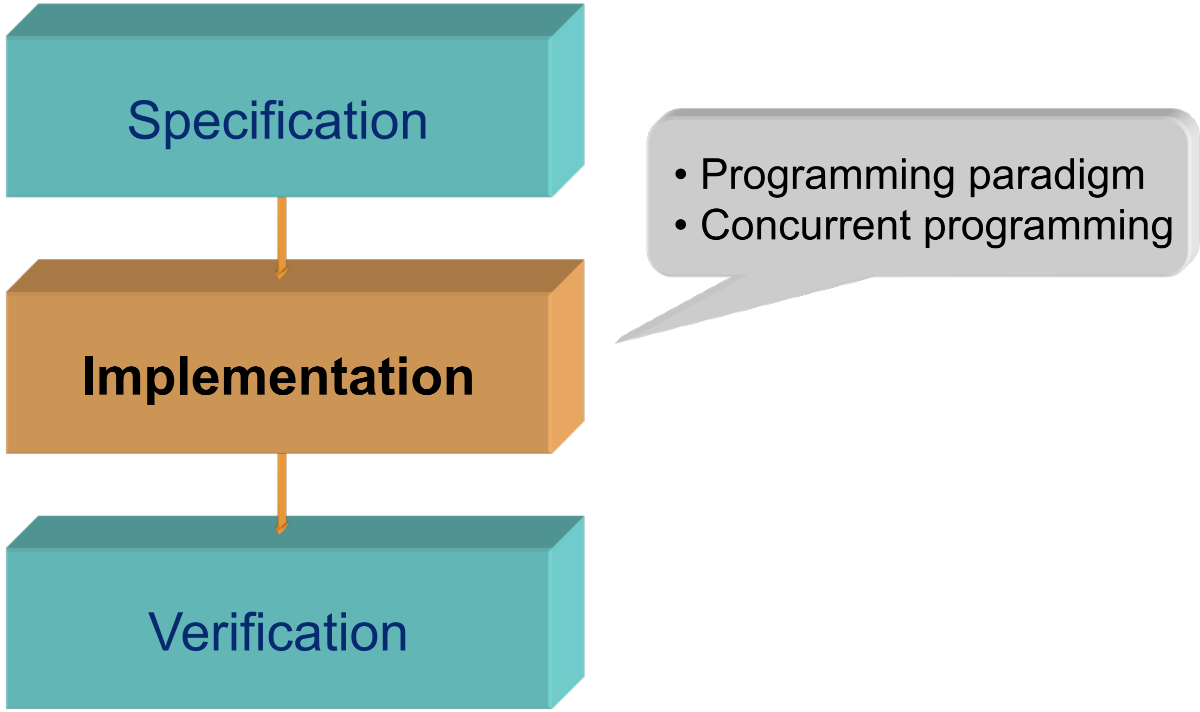
Real-Time Systems

Lecture #2

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-time systems



Real-time programming

Recommended programming paradigm:

- Concurrent programming
  - Reduces unnecessary dependencies between tasks
  - Enables a composable schedulability analysis
- Reactive programming
  - Certifies that tasks are activated only when work should be done; tasks are kept idle otherwise
  - Maps directly to the task model used in schedulability analysis
- Timing-aware programming
  - Certifies that timing constraints are visible at the task level
  - Enables priority-based scheduling of tasks, which in turn facilitates schedulability analysis

Real-time programming

Desired properties of a real-time programming language:

- Support for partitioning software into units of concurrency
  - tasks or threads (Ada 95, Java or POSIX C)
  - object methods (C/C++ using the TinyTimber kernel)
- Support for communication with the environment
  - access to I/O hardware (e.g. view I/O registers as variables)
  - machine-level data types (e.g. bit-field type, address pointers)
- Support for the schedulability analysis
  - notion of (high-resolution) time ($\Rightarrow$ timing-aware programming)
  - task priorities (reflects constraints $\Rightarrow$ timing-aware programming)
  - task delays (idle while not doing useful work $\Rightarrow$ reactive model)
  - hardware interrupt handlers (event generators $\Rightarrow$ reactive model)

Real-time programming

What programming languages are suitable?

- C, C++
  - Support for machine-level programming
  - Concurrent programming via run-time system (POSIX, TinyTimber)
  - Priorities and notion of time via run-time system (POSIX, TinyTimber)
- Java
  - Support for machine-level programming
  - Support for concurrent programming (threads)
  - Support for priorities and notion of time (Real-Time Java)
- Ada 95
  - Support for machine-level programming
  - Support for concurrent programming (tasks)
  - Support for priorities and notion of time

Why concurrent programming?

Most real-time applications are inherently parallel

- Events in the target system's environment often occur in parallel
- By viewing the application as consisting of multiple tasks, this parallel reality can be reflected
- While a task is waiting for an event (e.g., I/O or access to a shared resource) other tasks may execute

Enables a composable schedulability analysis

- First, the local timing properties of each task are derived
- Then, the interference between tasks are analyzed

System can obtain reliability properties

- Redundant copies of the same task makes system fault-tolerant

Issues with concurrent programming

Access to shared resources

- Many hardware and software resources can only be used by one task at a time (e.g., processor, data structures)
- Only pseudo-parallel access is possible in many cases

Synchronization and information exchange

- System modeling using concurrent tasks also introduces a need for synchronization and information exchange.

Concurrent programming must hence be supported by an advanced run-time system that handles the scheduling of shared resources and communication between tasks.

Support for concurrent programming

Support in the programming language:

- Program is easier to read and comprehend, which means simpler program maintenance
- Program code can be easily moved to another operating system
- For some embedded systems, a full-fledged operating system is unnecessarily expensive and complicated
- Examples: Ada 95, Java, Modula, Occam, ...

Example:

Ada 95 offers support via task, rendezvous & protected objects

Java offers support via threads & synchronized methods

Support for concurrent programming

Support in the run-time system:

- Simpler to combine programs written in different languages whose concurrent programming models are incompatible
- There may not exist a simple one-to-one mapping between the language's model and the run-time system's model
- Operating systems become more and more standardized, which makes program code more portable between OS's (e.g., POSIX for UNIX, Linux, Mac OS X, and Windows)

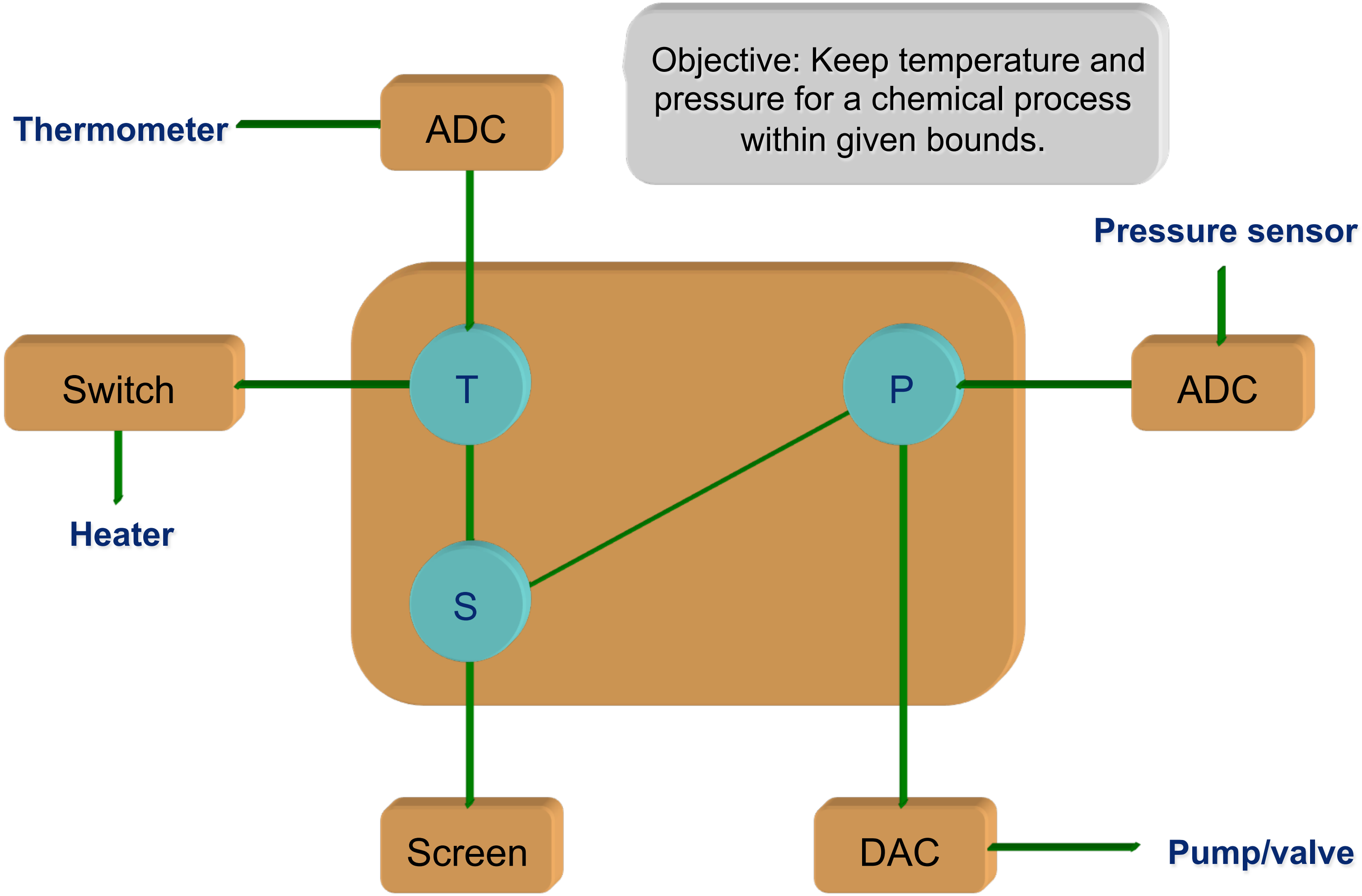
Example:

UNIX, Linux, etc offer support via fork, semctl & msgctl

POSIX offers support via threads & mutex methods

TinyTimber offers support via reactive objects & mutex methods

Example: a simple control system



Sequential solution (Ada95)

```
procedure Controller is
   TR : Integer;
   PR : Integer;
   HS : Integer;
   PS : Integer;
begin
   loop
      TR := T_Read;                      -- read temperature
      HS := Temp_Convert(TR);            -- convert to heater setting
      T_Write(HS);                       -- set heater switch
      PrintLine("Temperature: ", TR);    -- write message on operator screen

      PR := P_Read;                      -- read pressure
      PS := Pressure_Convert(PR);        -- convert to pump setting
      P_Write(PS);                       -- set pump control
      PrintLine("Pressure: ", PR);       -- write message on operator screen
   end loop;
end Controller;
```

Sequential solution (Java)

```
public class Controller {
  public static void main(String[] args) {
    int TR;
    int PR;
    int HS;
    int PS;

    while (true) {
      TR = T_Read();                    // read temperature
      HS = Temp_Convert(TR);            // convert to heater setting
      T_Write(HS);                      // set heater switch
      PrintLine("Temperature: ", TR); // write message on operator screen

      PR = P_Read();                    // read pressure
      PS = Pressure_Convert(PR);        // convert to pump setting
      P_Write(PS);                      // set pump control
      PrintLine("Pressure: ", PR);    // write message on operator screen
    }
  }
}
```

Sequential solution

Drawback:

- the inherent parallelism of the application is not exploited
  - Procedures `T_Read` and `P_Read` block the execution until a new temperature or pressure sample is available from the sensor
  - while waiting to read the temperature, no attention can be given to the pressure (and vice versa)
- the independence of the control functions are not considered
  - temperature and pressure must be read with the same interval
  - the iteration frequency of the loop is mainly determined by the blocking time of the calls to `T_Read` and `P_Read`.
  - if the call for reading the temperature does not return because of a fault, it is no longer possible to read the pressure

Improved sequential solution (Ada95)

```
procedure Controller is
  ...
begin
  loop
    if Ready_Temp then
      TR := T_Read;
      HS := Temp_Convert(TR);
      T_Write(HS);
      PrintLine("Temperature: ", TR);
    end if;
    if Ready_Pres then
      PR := P_Read;
      PS := Pressure_Convert(PR);
      P_Write(PS);
      PrintLine("Pressure: ", PR);
    end if;
  end loop;
end Controller;
```

The Boolean function **Ready_Temp** indicates whether a sample from the sensor is available

The Boolean function **Ready_Pres** indicates whether a sample from the sensor is available

Improved sequential solution (Java)

```
public class Controller {
  public static void main(String[] args) {
    ...

    while (true) {
      if (Ready_Temp()) {
        TR = T_Read();
        HS = Temp_Convert(TR);
        T_Write(HS);
        PrintLine("Temperature: ", TR);
      }

      if (Ready_Pres()) {
        PR = P_Read();
        PS = Pressure_Convert(PR);
        P_Write(PS);
        PrintLine("Pressure: ", PR);
      }
    }
  }
}
```

The Boolean method **Ready_Temp** indicates whether a sample from the sensor is available

The Boolean method **Ready_Pres** indicates whether a sample from the sensor is available

Improved sequential solution

Advantages:

- the inherent parallelism of the application is exploited
  - pressure and temperature control do not block each other

Drawbacks:

- the program spends a large amount of time in “busy wait” loops
  - processor capacity is unnecessarily wasted
  - schedulability analysis is made complicated/impossible
- the independence of the control functions is not considered
  - temperature and pressure must be read with the same interval
  - if the call for reading the temperature does not return because of a fault, it is no longer possible to read the pressure

Concurrent solution

Step 1: Make concurrent:

– Partition the software into units of concurrency

Ada95:

Create two units of type **task**, `T_Controller` and `P_Controller`, each containing the code for handling the data from respective sensor. The concurrent execution of the code will be automatically initiated.

Java:

First declare two classes, `T_Controller` and `P_Controller`, each class being a subclass of the predefined `Thread` class. Each class should provide a `run` method containing the code for handling the data from respective sensor.

Then, create one thread object from each declared class. Finally, initiate the concurrent execution of the code by calling the predefined `start` method associated with each thread object.

Concurrent solution

Step 2: Make reactive:

- Tasks should be idle if there is no work to be done

  Ada95:
  The task calls the blocking procedure `T_Read` or `P_Read` to idle.

  Java:
  The thread calls the blocking method `T_Read` or `P_Read` to idle.

- Activate task as a reaction to an incoming event

  Ada95:
  The call to procedure `T_Read` or `P_Read` unblocks when data becomes available at a sensor, thus activating the calling task.

  Java: The call to method `T_Read` or `P_Read` unblocks when data becomes available at a sensor, thus activating the calling thread.

Concurrent solution (Ada95)

```
procedure Controller is
task T_Controller;
task P_Controller;

task body T_Controller is
begin
  loop
     TR := T_Read;
     HS := Temp_Convert(TR);
     T_Write(HS);
     PrintLine("Temperature: ", TR);
  end loop;
end T_Controller;

task body P_Controller is
begin
  loop
     PR := P_Read;
     PS := Pressure_Convert(PR);
     P_Write(PS);
     PrintLine("Pressure: ", PR);
  end loop;
end P_Controller;

begin
  null;        -- begin concurrent execution of the two tasks
end Controller;
```

Concurrent solution (Java)

```
public class T_Controller extends Thread {
  public void run() {
    while (true) {
      TR = T_Read();
      HS = Temp_Convert(TR);
      T_Write(HS);
      PrintLine("Temperature: ", TR);
    }
  }
}

public class P_Controller extends Thread {
  public void run() {
    while (true) {
      PR = P_Read();
      PS = Pressure_Convert(PR);
      P_Write(PS);
      PrintLine("Pressure: ", PR);
    }
  }
}
```

Concurrent solution (Java)

```
public class Controller {
  public static void main(String[] args) {
    T_Controller TC = new T_Controller; // create temperature thread
    P_Controller PC = new P_Controller; // create pressure thread

    TC.start();       // begin concurrent execution of first thread
    PC.start();       // begin concurrent execution of second thread
  }
}
```

Concurrent solution

Advantages:

- the inherent parallelism of the application is fully exploited
  - pressure and temperature control do not block each other
  - the control functions can work at different frequencies
  - no processor capacity are unnecessarily consumed
  - the application becomes more reliable

Drawbacks:

- the parallel tasks share a common resource
  - the screen can only be used by one task at a time
  - a resource handler must be implemented, for controlling the access to the screen (to avoid garbled text)
  - the resource handler must guarantee *mutual exclusion* (*mutex*)