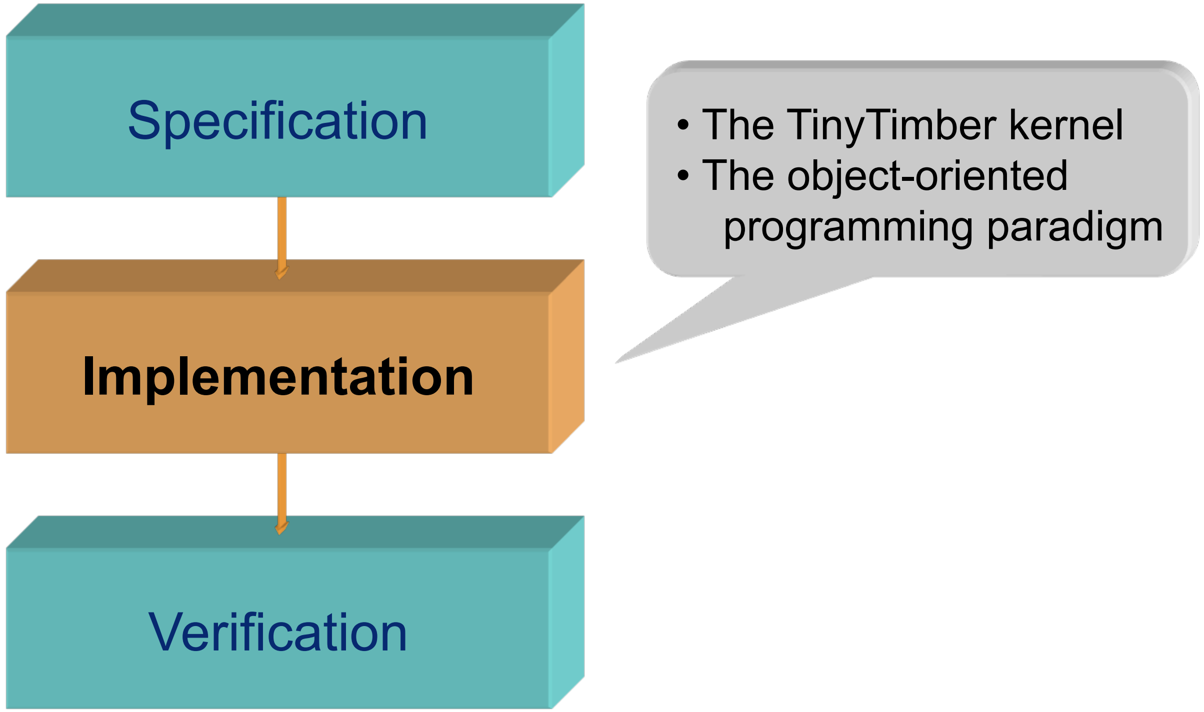
Real-Time Systems

Lecture #3

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-time systems



TinyTimber – the context

Timber – the programming language:

- Full-featured language:
  - Higher-order and strongly typed language
  - Semantics in the purely functional tradition (non-lazy)
  - Features time-constrained concurrent reactive objects
- International research project:
  - Participants: Chalmers, Luleå University of Technology, Oregon Graduate Institute, Portland State University and Kansas State University (project period: 2000–2003).
- Direct descendant of O'Haskell:
  - An object-oriented extension of the Haskell language
  - Result of PhD work at Chalmers (J. Nordlander, 1999)

TinyTimber – the context

Timber – the programming language:

- Salient feature #1: "CORT" properties
  - Concurrent: Code execution concurrency is implicit, by means of the Timber object, and thus does not require the use of any dedicated concurrency constructs (e.g. threads or tasks).
  - Object oriented: Timber uses objects to encapsulate a local state and methods to manipulate this state.
  - Reactive: a Timber object is a passive entity, and the relative execution order of its methods is solely determined by events (e.g. hardware interrupts or invocations from other methods).
  - Timing aware: each reaction (method invocation) is associated with a programmable timing window, that can optionally be used to constrain start time and/or completion time of the reaction.

TinyTimber – the context

Timber – the programming language:

- Salient feature #2: “Deterministic” properties
  - Through its language design Timber code is free from indefinitely blocking language constructs. Thus, an object is always fully responsive when not actively executing code.

    This is in contrast to the common infinite event-loop pattern in other languages, where blocking calls are used to partition a linear thread of execution into event-handling fragments.
  - Through its language design Timber methods that belong to the same object are mutually exclusive. Consequently, object state is guaranteed to always be consistent.

TinyTimber – the context

Timber – the programming language:

- Recall the desired properties of a RT language:
  - Concurrent
  - Reactive
  - Timing aware
- What does the object-oriented (OO) approach offer?
  - Encapsulation (entity with data and code)
  - Reliability and maintainability (of object data and code)
  - Natural unit of concurrency (object with run-time context)
  - Natural place to implement mutual exclusion ("mutex").

OO programming

Prominent features of OO programming:

- Encapsulation:
  - Encapsulation implies the existence of an entity that binds together data and a set of operations that manipulate the data.
  - The operations are referred to as <u>methods</u>.
  - The conventions for calling a method (e.g. parameter types, return value type) is referred to as the <u>interface</u> of the method.
  - The strongest form of encapsulation does not allow external code to directly access the data in the entity, but stipulates <u>access through methods only</u>. Thus, the data is for all practical purposes considered to be <u>hidden</u> from the external code.

OO programming

Prominent features of OO programming:

- Classes:
  - A detailed description of the internal format of encapsulated data (members) and set of methods (code) is called a class.
  - A class may be composed of members that refer to a class.
  - A class may be defined by extending the functionality of an existing class by means of class inheritance.
- Objects:
  - An instantiation of a class (i.e. memory storage is allocated to its members) is called an object.
  - The contents (values) of the object members is referred to as the state of the object.

OO programming

Advantages with OO programming:

- Reliability and maintainability:
  - Encapsulation means that the object state is safe from outside (deliberate or unintentional) interference and misuse.
  - Encapsulation means that calling code does not need to be edited even if the internal format (data or code) of the class should change, as long as method interfaces stay the same.
- Natural unit of concurrency:
  - The object state can include run-time context of its methods.
  - Methods belonging to different objects thereby get independent run-time contexts, and may therefore execute concurrently.
  - Offered by Java (thread objects) and Timber (reactive objects).

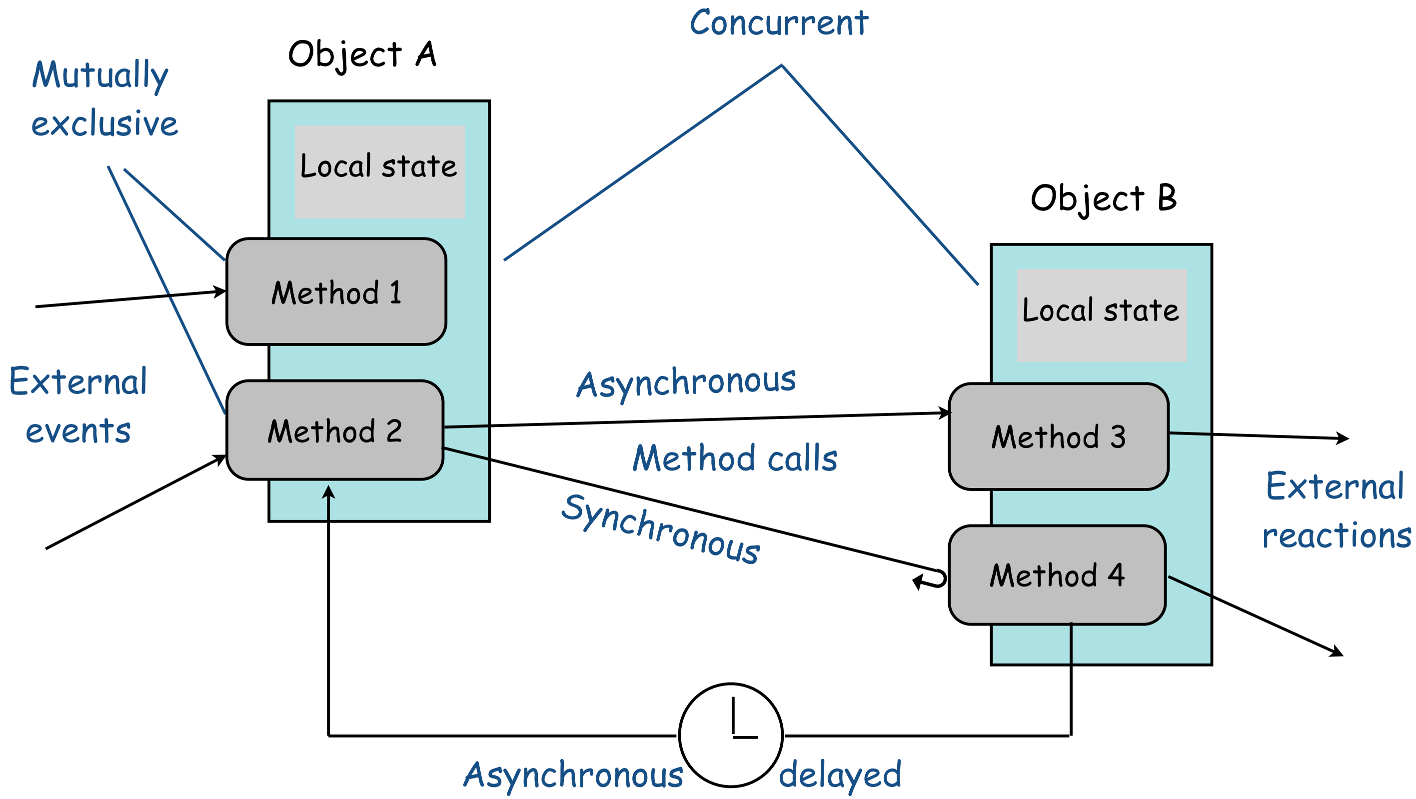
OO programming

Advantages with OO programming:

- Natural place to implement mutual exclusion:
  - Encapsulation facilitates “locking” the object while manipulating it via method calls, thereby guaranteeing state consistency.
  - Among an object’s methods only one may execute at a time, and must also complete its code before the object is unlocked (such code is commonly referred to as a “critical region”).
  - Methods belonging to the same object can therefore not execute concurrently (i.e., they are mutually exclusive)
  - Offered by Ada 95 (protected objects), Java (synchronized methods) and Timber (mutex methods).

TinyTimber – the context

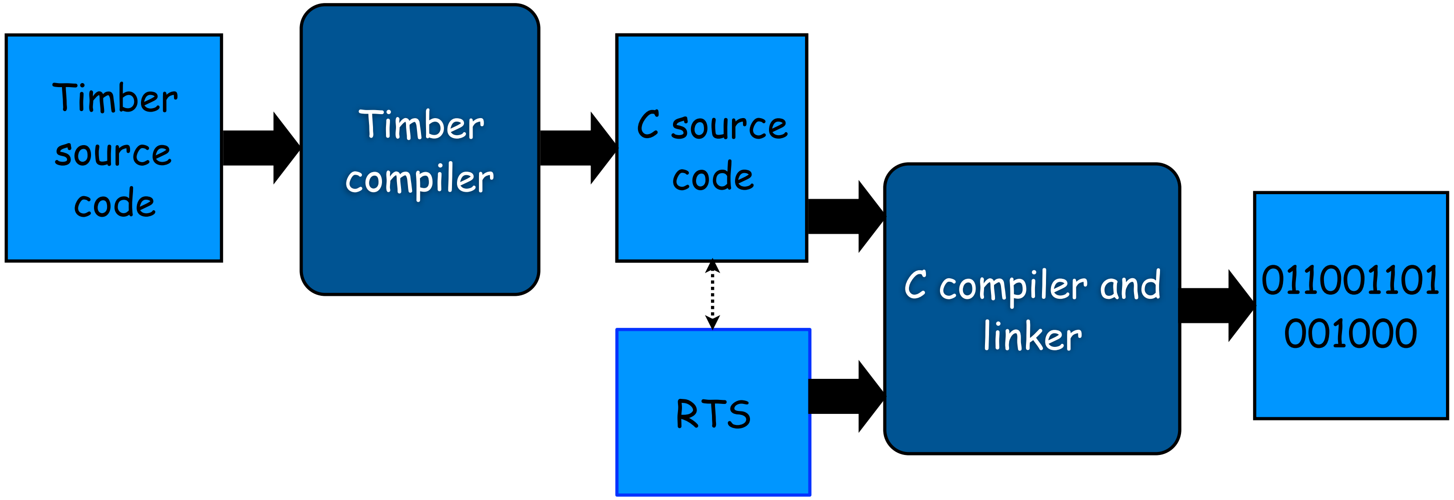
OO programming – visualized as an access graph:



An OO program is a collection of objects that act on each other via method calls

TinyTimber – the context

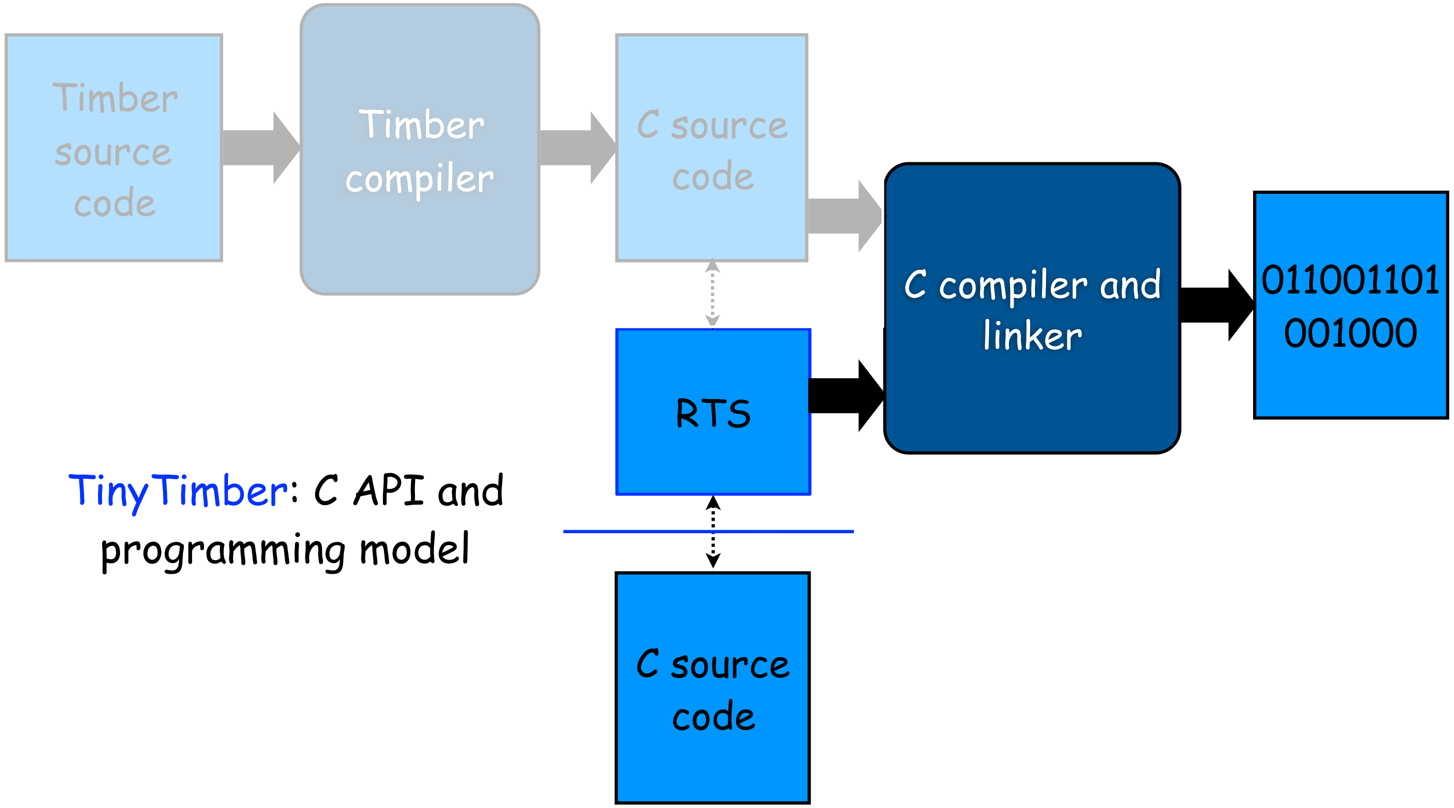
Timber – the compiler workflow:



Note: C language back-end

TinyTimber – the context

TinyTimber – the real-time kernel:



TinyTimber: C API and programming model

TinyTimber – the context

TinyTimber – the real-time kernel:

- Originates from the Timber run-time system:
  - TinyTimber was redesigned to make it a standalone run-time system, thereby not requiring the use of the Timber language.
  - TinyTimber is completely written in the C language.
- Retains the API of the Timber run-time system:
  - Caveat: requires special C coding conventions for the method interface.
- Fully supports the salient features #1 and #2 of Timber:
  - Caveat: requires special C coding conventions to retain these features of Timber.

TinyTimber – the context

TinyTimber – C coding conventions (OO programming):

- Use a C struct to define the members of a class:
  - The first member in the C struct must be of type `Object`, a predefined parent class containing run-time information.
- Use a C function to define a method:
  - The function must have two parameters, and a return type of either `int` or `void`.
  - The first parameter must be a pointer to the class to which the method belongs. The second parameter must be of type `int`.

  (Work-arounds for these restrictions will be given in Exercise #2)
- Use a variable of the C struct type to create an object:
  - The predefined `initObject()` macro should be used as a constructor for the first member in the class.

TinyTimber – the context

Example − C coding conventions (OO programming):

```
// TinyTimber class
typedef struct {
  Object super;    // NOTE: 'Object' type makes struct a TinyTimber class
  int theData;     // NOTE: 'theData' cannot be encapsulated (hidden)
} SharedInteger;

// TinyTimber methods
int Read(SharedInteger *self, int unused) {      // NOTE: methods are not
    return self->theData;                        //  part of the class ...
}
void Write(SharedInteger *self, int newValue) { // ... and therefore need
    self->theData = newValue;                    //  a pointer to the class
}

// TinyTimber object constructor (NOTE: not part of the class)
#define initSharedInteger(initialValue) { initObject(), initialValue }

SharedInteger myData = initSharedInteger(42);  // Create TinyTimber object
```

OO programming

Compare with Java implementation (OO programming):

```
// Java class
class SharedInteger
{
  private int theData;    // NOTE: 'theData' can be encapsulated (hidden)

  public SharedInteger(int initialValue) {   // Java object constructor
    theData = initialValue;                  // NOTE: part of the class
  }
  public synchronized int Read() {           // Java methods (mutex)
    return theData;                          // NOTE: part of the class
  }
  public synchronized void Write(int newValue) {
    theData = newValue;
  }
}

SharedInteger myData = new SharedInteger(42);     // Create Java object
```

TinyTimber – the context

TinyTimber – C coding conventions (OO programming):

- Ensure encapsulation:
  - Access to class members should be done via methods calls only, even if the C language does not provide any mechanism for hiding the members in the corresponding C struct.
- Ensure determinism (mutex methods):
  - Mutex method calls are done synchronously or asynchronously.
  - A synchronous call is done via the predefined **`SYNC()`** macro; the calling code waits until the method call returns.
  - An asynchronous call is done via the predefined **`ASYNC()`** macro; this spawns a concurrent execution of the method code, and the calling code continues to execute (without waiting).

TinyTimber – the context

TinyTimber – C coding conventions (OO programming):

- Ensure determinism (non-blocking property):
  - The object methods cannot contain indefinitely blocking code
  - No-no #1: infinite loops (e.g. 'while (1)') must not be used
  - No-no #2: synchronous calls to a method within the same object as the calling code must not be used (as it will lead to deadlock)
- Ensure concurrency:
  - Program code that should execute concurrently must reside in methods that belong to separate objects.
  - Recall that code in methods that belong to the same object can never execute concurrently (due to mutex methods).

TinyTimber – the context

TinyTimber – C coding conventions (OO programming):

- Ensure timing awareness:
  - Programmable timing windows may be used for method calls by means of the following set of predefined macros.

`AFTER()`: corresponds to an `ASYNC()` call that takes place after an initial delay (offset).

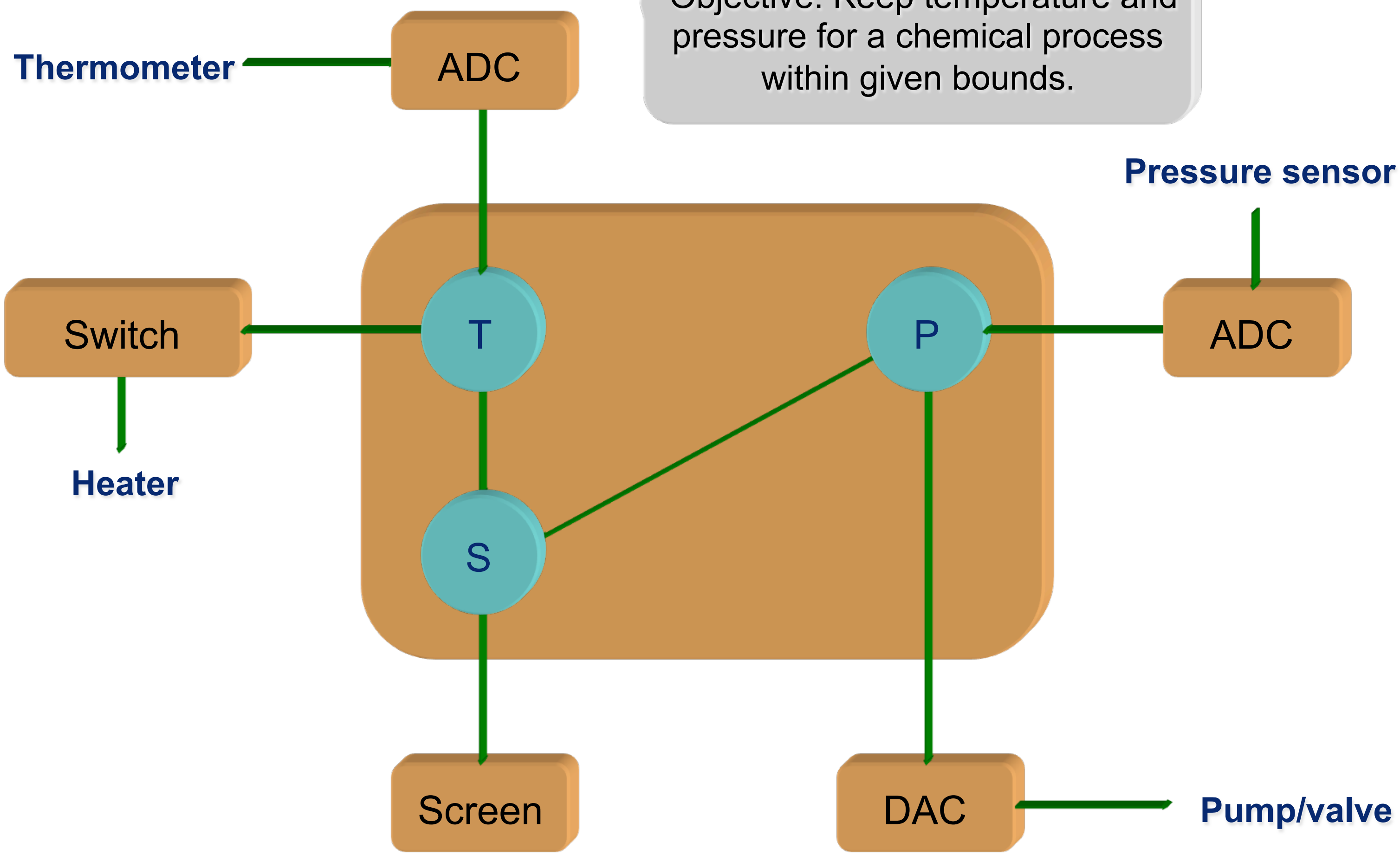
`BEFORE()`: corresponds to an `ASYNC()` call with a deadline on the method code completion.

`SEND()`: equal to a combined `AFTER()` and `BEFORE()` call.

(More details regarding timing windows will be given in Lecture #6)

A simple control system (revisited)

Objective: Keep temperature and pressure for a chemical process within given bounds.



Concurrent solution

Step 1: Make concurrent:

– Partition the software into units of concurrency

TinyTimber:

First declare a class `Task` with its first member being of predefined type `Object`, and define two methods associated with the declared class, `T_Controller` and `P_Controller`, containing the code for handling the data from respective sensor.

Then, create two objects from the declared class, one for each of the defined methods. This will allow for concurrent execution of the code.

Finally, create two interrupt handlers, one for each sensor, that will call the respective method when data becomes available on the sensor.

Concurrent solution

Step 2: Make reactive:

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

  TinyTimber:
  Since methods `T_Controller` and `P_Controller` must be called to be activated they are by default idle.

- Activate task as a reaction to an incoming event

  TinyTimber:
  An interrupt handler calls (activates) its respective method when data becomes available at a sensor.

  (More details on interrupt handlers and how to associate them with reactive objects will be given in Lecture #5)

Concurrent solution (TinyTimber)

```
typedef struct {
  Object super;      // NOTE: 'Object' type makes struct a TinyTimber class
} Task;

#define initTask() { initObject() }

Task T_Task = initTask();    // Create two new concurrent objects
Task P_Task = initTask();

// Declare the methods for each new object

void T_Controller(Task *, int);
void P_Controller(Task *, int);

// Define two new objects of class Sensor (definition not shown here),
// representing the sensors

Sensor sensor_t = initSensor(SENSOR_PORT0, &T_Task, T_Controller);
Sensor sensor_p = initSensor(SENSOR_PORT1, &P_Task, P_Controller);
```

Concurrent solution (TinyTimber)

```
// Define the methods for handling the input data. Each method is
// called with the data from the sensor as parameter.

void T_Controller(Task *self, int data) {
  int HS;

  HS = Temp_Convert(data);           // convert to heater setting
  T_Write(HS);                       // set heater switch
  PrintLine("Temperature: ", data); // write message on operator screen
}

void P_Controller(Task *self, int data) {
  int PS;

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

...
```

Concurrent solution (TinyTimber)

```
...

// Initialize the two sensor objects

void kickoff(Task *self, int unused) {
  SENSOR_INIT(&sensor_t);
  SENSOR_INIT(&sensor_p);
}

// Install interrupt handlers for the sensors, and then kick off
// the TinyTimber run-time system

void main() {
  INSTALL(&sensor_t, sensor_interrupt, SENSOR_INT0);
  INSTALL(&sensor_p, sensor_interrupt, SENSOR_INT1);
  TINYTIMBER(&P_Task, kickoff, 0);
}
```

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*)

Solid concurrent solution (TinyTimber)

```
/*
 * TinyTimber objects guarantee mutual exclusion for their declared
 * methods, if the caller uses a synchronous or asynchronous call:
 * the call to the method will then be blocked if any of the methods
 * in the object are currently being used.
 */

typedef struct {
  Object super;     // NOTE: 'Object' type makes struct a TinyTimber class
} ScreenController;

#define initScreenController() { initObject() }

ScreenController myScreen = initScreenController();   // Create new object

void T_Printline(ScreenController *self, int data) {  // NOTE: methods are
    PrintLine("Temperature: ", data);                 // not part of class
}

void P_Printline(ScreenController *self, int data) {
    PrintLine("Pressure: ", data);
}
```

Solid concurrent solution (TinyTimber)

```
/*
 * In a TinyTimber synchronous call the caller will be blocked
 * if any of the methods in the object are currently being used.
 */

void T_Controller(Task *self, int data) {
  Heater_Setting HS;

  HS = Temp_Convert(data);
  T_Write(HS);
  SYNC(&myScreen, T_PrintLine, data);      // NOTE: calling code waits
}                                          // for T_PrintLine to complete

void P_Controller(Task *self, int data) {
  Pressure_Setting PS;

  PS = Pressure_Convert(data);
  P_Write(PS);
  SYNC(&myScreen, P_PrintLine, data);      // NOTE: calling code waits
}                                          // for P_PrintLine to complete
```

Solid concurrent solution (TinyTimber)

```
/*
 * In a TinyTimber asynchronous call the caller can continue executing
 * immediately after posting the method call, regardless of whether
 * any of the methods in the object are currently being used or not.
 */

void T_Controller(Task *self, int data) {
  Heater_Setting HS;

  HS = Temp_Convert(data);
  T_Write(HS);
  ASYNC(&myScreen, T_PrintLine, data);     // NOTE: spawns a concurrent
}                                          // call to T_PrintLine

void P_Controller(Task *self, int data) {
  Pressure_Setting PS;

  PS = Pressure_Convert(data);
  P_Write(PS);
  ASYNC(&myScreen, P_PrintLine, data);     // NOTE: spawns a concurrent
}                                          // call to P_PrintLine
```

Solid concurrent solution (Ada95)

```
-- In Ada95 protected objects can guarantee mutual exclusion for their
-- declared procedures: a calling task will be blocked if any of the
-- procedures in the object are currently being used.

protected type Screen_Controller is
  procedure T_Printline(data : Integer);
  procedure P_Printline(data : Integer);
end Screen_Controller;

protected body Screen_Controller is
begin
  procedure T_Printline(data : Integer) is
  begin
    Printline("Temperature: ", data);
  end T_Printline;

  procedure P_Printline(data : Integer) is
  begin
    Printline("Pressure: ", data);
  end P_Printline;
end Screen_Controller;

myScreen : Screen_Controller;                -- Create new object
```

Solid concurrent solution (Ada95)

```
...

task body T_Controller is
begin
  loop
     TR := T_Read;
     HS := Temp_Convert(TR);
     T_Write(HS);
     myScreen.T_PrintLine(TR);
  end loop;
end T_Controller;

task body P_Controller is
begin
  loop
     PR := P_Read;
     PS := Pressure_Convert(PR);
     P_Write(PS);
     myScreen.P_PrintLine(PR);
  end loop;
end P_Controller;
```

Solid concurrent solution (Java)

```
// Objects in Java can guarantee mutual exclusion if their methods are
// declared as synchronized: a call to the method will then be blocked
// if any of the methods in the object are currently being used.

class ScreenController
{
  public synchronized void T_Println(int data) {
     Println("Temperature: ", data);
  }
  public synchronized void P_Println(int data) {
     Println("Pressure: ", data);
}

ScreenController myScreen = new ScreenController();   // Create new object
```

Solid concurrent solution (Java)

```
...

public class T_Controller extends Thread {
  public void run() {
    while (true) {
      TR = T_Read();
      HS = Temp_Convert(TR);
      T_Write(HS);
      myScreen.T_PrintLine(TR);
    }
  }
}

public class P_Controller extends Thread {
  public void run() {
    while (true) {
      PR = P_Read();
      PS = Pressure_Convert(PR);
      P_Write(PS);
      myScreen.P_PrintLine(PR);
    }
  }
}
```