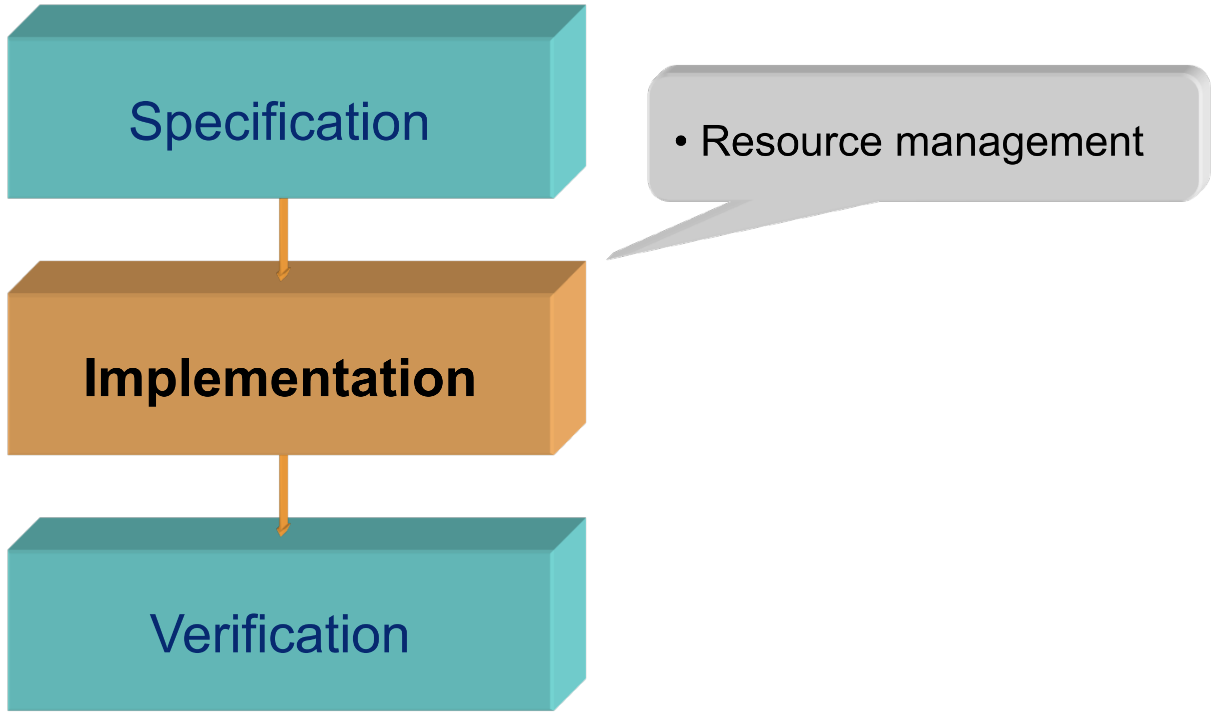
Real-Time Systems

Lecture #4

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

Real-Time Systems



Resource management

Resource management is a general problem that exists at several levels in a real-time system.

- Shared resources internal to the the run-time system:
  - CPU time
  - Memory pool (for dynamic allocation of memory)
  - Data structures (queues, tables, buffers, …)
  - I/O device access (ports, status registers, …)
- Shared resources specific to the application program:
  - Data structures (buffers, state variables, databases…)
  - Displays (to avoid garbled text if multiple tasks use it)
  - Entities in the application environment (seats in a cinema or an aircraft, a car parking facility, etc)

Resource management

Classification of resources:

- Exclusive access: there must be only one user at a time.
  - Exclusiveness is guaranteed through mutual exclusion
  - Program code that is executed while mutual exclusion applies is called a critical region
  - Examples: manipulation of data structures or I/O device registers
- Shared access: there can be multiple users at a time.
  - Resource manager makes sure that the number of users are within acceptable limits
  - The program code for the resource manager is a critical region
  - Classical computer science example: Dining Philosophers Problem

Resource management

Operations for resource management:

- `acquire`: to request access to a resource
- `release`: to release a previously acquired resource

The `acquire` operation can be either blocking or non-blocking:

- Blocking: the task that calls `acquire` is blocked if the resource is not available. Blocked tasks are stored in a queue, in FIFO or priority order. When the requested resource becomes available one of the blocked tasks is unblocked and is activated via a *call-back functionality*.
- Non-blocking: `acquire` returns a status code to the calling task indicating whether access to the resource was granted or not.

Resource management

Problems with resource management:

- Deadlock: tasks blocks each other and none of them can use the resource.
  - Deadlock can only occur if the tasks require access to more than one resource at the same time
  - Deadlock can be avoided by following certain guidelines
- Starvation: Some task is blocked because resources are always assigned to other (higher priority) tasks.
  - Starvation can occur in most resource management scenarios
  - Starvation can be avoided by granting access to resources in FIFO order

In general, deadlock and starvation are problems that must be solved by the program designer!

Resource management

Example #1: Assume that two tasks, A and B, want to use two different resources at the same time …

```
R1, R2 : Shared_Resource;

task A;
task body A is
begin
   R1.Acquire;
   R2.Acquire;
   ...           -- program code using both resources
   R2.Release;
   R1.Release;
end A;

task B;
task body B is
begin
   R2.Acquire;
   R1.Acquire;
   ...           -- program code using both resources
   R1.Release;
   R2.Release;
end B;
```

A task switch from A to B after this code line causes deadlock.

Resource management

Example #1: Assume that two tasks, A and B, want to use two different resources at the same time …

```
R1, R2 : Shared_Resource;

task A;
task body A is
begin
   R1.Acquire;
   R2.Acquire;
   ...          -- program code using both resources
   R2.Release;
   R1.Release;
end A;

task B;
task body B is
begin
   R1.Acquire;
   R2.Acquire;
   ...          -- program code using both resources
   R2.Release;
   R1.Release;
end B;
```

Deadlock can be avoided if the tasks acquire the resources in the same order.

Resource management

Example #2: The dining philosophers problem …

- Five philosophers live together in a house.
- The house has one round dinner table with five plates of rice.
- There are five sticks available: one stick between every pair of plates.
- The philosophers alternate between eating and thinking. To be able to eat the rice, a philosopher needs two sticks.
- Sticks are a scarce resource: only two philosophers can eat at the same time.

How is deadlock and starvation avoided?

Resource management

Example #2: The dining philosophers problem …

- The following solution will cause deadlock if all philosophers should happen to take the left stick at exactly the same time:

```
loop
  Think;
  Take_left_stick;
  Take_right_stick;
  Eat;
  Drop_left_stick;
  Drop_right_stick;
end loop;
```

- One way to avoid deadlock and starvation is to only allow four philosophers at the table at the same time.

Resource management

Example #3: A potential issue in our daily life …

Deadlock

Conditions for deadlock to occur:

1. Mutual exclusion
   – only one task at a time can use a resource
2. Hold and wait
   – there must be tasks that hold one resource at the same time as they request access to another resource
3. No preemption
   – a resource can only be released by the task holding it
4. Circular wait
   – there must exist a cyclic chain of tasks such that each task holds a resource that is requested by another task in the chain

Deadlock

Guidelines for avoiding deadlock:

1. Tasks should, if possible, only use one resource at a time.
2. If (1) is not possible, all tasks should request resources in the same order.
3. If (1) and (2) are not possible, special precautions should be taken to avoid deadlock. For example, resources could be requested using non-blocking calls.

   Example: the TinyTimber kernel can detect deadlock situations when a synchronous call is made. In such situations `SYNC()` will not make the intended method call and instead return a value of (-1) to notify the caller.

Resource management

Program constructs for resource management:

- Ada 95 uses protected objects.
- Older languages (e.g. Concurrent Pascal, Modula) use monitors.
- Java uses reentrant locks (can be used to build e.g. monitors) or synchronized methods.

When programming in languages (e.g. C and C++) that do not provide the constructs mentioned above, mechanisms provided by the real-time kernels or operating system must be used.

- POSIX offers semaphores and methods with mutual exclusion.
- The TinyTimber kernel offers methods with mutual exclusion.

  To allow TinyTimber to support general `acquire` and `release` operations a suitable object type (e.g. monitor or semaphore) must be added to the kernel.

Protected objects

Protected objects:

- A protected object is a synchronization mechanism offered by Ada 95.
- A protected object offers operations with mutual exclusion for data being shared by multiple tasks.
- A protected operation can be an entry, a procedure or a function. The latter is a read-only operation.
- Protected entries are guarded by a Boolean expression called a barrier.

  The barrier must evaluate to ”true” to allow the entry body code to be executed. If the barrier evaluates to “false”, the calling task will block until the barrier condition changes.

Protected objects

Implementing an exclusive resource in Ada 95:

```
protected type Exclusive_Resource is
  entry Acquire;
  procedure Release;
private
  Busy : Boolean := false;
end Exclusive_Resource;

protected body Exclusive_Resource is
  entry Acquire when not Busy is
  begin
    Busy := true;
  end Acquire;

  procedure Release is
  begin
    Busy := false;
  end Release;
end Exclusive_Resource;
...
```

Protected objects

```
R : Exclusive_Resource;        -- resource with one user

task A, B;       -- tasks using the resource

task body A is
begin
  ...
  R.Acquire;
  ...            -- critical region with code using the resource
  R.Release;
  ...
end A;

task body B is
begin
  ...
  R.Acquire;
  ...            -- critical region with code using the resource
  R.Release;
  ...
end B;
```

Monitors

Monitors:

- A monitor is a synchronization mechanism originally offered by some older languages, e.g., Concurrent Pascal, Modula.
- A monitor encapsulates data structures that are shared among multiple tasks and provides procedures to be called when a task needs to access the data structures.
- Execution of monitor procedures are done under mutual exclusion.
- Synchronization of tasks is done with a mechanism called condition variable. Each such variable represents a given Boolean condition for which the tasks should synchronize.

Monitors

Monitors vs. protected objects:

- Monitors are similar to protected objects in the sense that both are objects that can guarantee mutual exclusion during calls to procedures manipulating shared data.
- The difference between monitors and protected objects are in the way they handle synchronization:
  - Protected objects use entries with barriers (auto wake-up)
  - Monitors use condition variables (manual wake-up)

Java offers a class that facilitates creation of monitors:

The `ReentrantLock` class includes support for mutual exclusion and the possibility to create `Condition` objects, that directly correspond to the monitor's condition variables.

Monitors

Operations on condition variables:

`wait(c_var)`: the calling task is blocked and is inserted into a queue corresponding to condition `c_var`.

`signal(c_var)`: wake up first task in the queue corresponding to condition `c_var`. No effect if the queue is empty.

Properties:

1. After a call to `wait` the monitor is released (e.g., other tasks may execute the monitor procedures).
2. A call to `signal` must be located after the procedure code that manipulates the internal state of the monitor.

Monitors

Implementing an exclusive resource with Java monitor:

```
class Exclusive_Resource {

  private boolean Busy;

  private final Lock lock = new ReentrantLock();

  private final Condition notBusy = lock.newCondition();

  public Exclusive_Resource() {
    Busy = false;
  }

  ...
```

Monitors

```
    public void Acquire() {
      lock.lock();
      try {
        while (busy)
          notBusy.await();      // block the task if resource busy

        Busy = true;
      } finally {
        lock.unlock();
      }
    }

    public void Release() {
      lock.lock();

      Busy = false;             // manipulate internal state of monitor
      notBusy.signal();         // then wake up a blocked task (if one exists)

      lock.unlock();
    }
  } // class Exclusive_Resource

  ...
```

Monitors

```
Exclusive_Resource R = new Exclusive_Resource(); // resource with one user

public class A extends Thread {  // concurrent thread using the resource
  public void run() {
    ...
    R.Acquire();
    ...                 // critical region with code using the resource
    R.Release();
    ...
  }
}

public class B extends Thread {  // concurrent thread using the resource
  public void run() {
    ...
    R.Acquire();
    ...                 // critical region with code using the resource
    R.Release();
    ...
  }
}
```

Semaphores

Semaphores:

- A semaphore is a passive synchronization primitive that is used for protecting shared and exclusive resources.
- Synchronization is done using two operations, `wait` and `signal`. These operations are atomic (indivisible) and are themselves critical regions with mutual exclusion.

A semaphore is less powerful than a protected object or a monitor, but is still quite useful as it can implement resource handlers for both exclusive (single-user) and shared (multi-user) resources.

Semaphores

A semaphore `s` is an integer variable with value domain ≥ 0

Atomic operations on semaphores:

`Init(s,n)`: assign `s` an initial value `n`

```
Wait(s):     if s > 0 then
                s := s - 1;
             else
                "block calling task";

Signal(s):   if "any task that has called Wait(s) is blocked"
             then
                "allow one such task to execute";
             else
                s := s + 1;
```

Semaphores

Implementing semaphores in Ada 95:

```
protected type Semaphore (InitialValue : Natural := 0) is
    entry Wait;
    procedure Signal;
private
    Value : Natural := InitialValue;
end Semaphore;

protected body Semaphore is
    entry Wait when Value > 0 is
    begin
      Value := Value - 1;
    end Wait;
    procedure Signal is
    begin
      Value := Value + 1;
    end Signal;
end Semaphore;

...
```

Semaphores

```
R : Semaphore(1);      -- exclusive resource (only one user)

task A, B;             -- tasks using the resource

task body A is
begin
   ...
   R.Wait;
   ...                 -- critical region with code using the resource
   R.Signal;
   ...
end A;

task body B is
begin
   ...
   R.Wait;
   ...                 -- critical region with code using the resource
   R.Signal;
   ...
end B;
```

Semaphores

Implementing semaphores in Java:

```
class Semaphore {

  private int Value;

  private final Lock lock = new ReentrantLock();

  private final Condition notBusy = lock.newCondition();

  public Semaphore(int InitialValue) {
    Value = InitialValue;
  }

  ...
```

Semaphores

```
  ...

  public void Wait() {
    lock.lock();
    try {
      while (Value == 0)
        notBusy.await();      // block the task if resource busy

      Value = Value - 1;
    } finally {
      lock.unlock();
    }
  }

  public void Signal() {
    lock.lock();

    Value = Value + 1;        // manipulate internal state of monitor
    notBusy.signal();         // then wake up a blocked task (if one exists)

    lock.unlock();
  }
} // class Semaphore
```

Semaphores

```
Semaphore R = new Semaphore(1);      // exclusive resource (only one user)
public class A extends Thread {   // concurrent thread using the resource
  public void run() {
    ...
    R.Wait();
    ...                   // critical region with code using the resource
    R.Signal();
    ...
  }
}

public class B extends Thread {   // concurrent thread using the resource
  public void run() {
    ...
    R.Wait();
    ...                   // critical region with code using the resource
    R.Signal();
    ...
  }
}
```