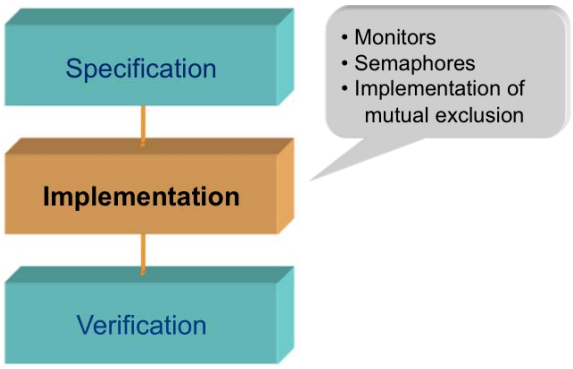
CHALMERS

Real-Time Systems



CHALMERS

Monitors

Monitors:

- A monitor is a construct offered by some programming languages, e.g., Modula-1, Concurrent Pascal, Mesa.
- 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 variables.

CHALMERS

Monitors

Monitors vs. protected objects:

- Monitors are similar to protected objects in Ada 95. Both are passive 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 monitor-like construct:
  - Java's synchronized methods correspond to monitor procedures
  - However, Java has no mechanism that corresponds to condition variables; a thread that gets woken up must check manually whether the resource is available.

CHALMERS

Monitors

Operations on condition variables:

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

`send(cond_var)`: wake up first task in the queue corresponding to `cond_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 `send` must be the last statement in a monitor procedure.
3. Queuing tasks that are awoken by a call to `send` has priority over tasks waiting to enter the monitor.

CHALMERS

Example: simple resource manager

```
monitor body Simple_Resource is     -- NOT Ada 95
  Resource_Max : constant := 8;
  R : Integer range 0..Resource_Max := Resource_Max;
  CR : condition_variable;

  procedure Acquire is
  begin
    if R = 0 then Wait(CR); end if;
    R := R - 1;
  end Acquire;

  procedure Release is
  begin
    R := R + 1;
    Send(CR);
  end Release;
end Simple_Resource;
```

CHALMERS

Example: circular buffer

Problem: Write a monitor **`Circular_Buffer`** that handles a circular buffer with room for 8 data records of type **`Data`**.

- The monitor should have two entries, **`Put`** and **`Get`**.
- Producer tasks should be able to insert data records in the buffer via entry **`Put`**. If the buffer is full, a task that calls **`Put`** should be blocked.
- Consumer tasks should be able to remove data records from the buffer via entry **`Get`**. If the buffer is empty, a task that calls **`Get`** should be blocked.

We solve this on the blackboard!

CHALMERS

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.
- Semaphores are used in real-time kernels and operating systems to implement e.g. rendezvous, protected objects or monitors.
- Semaphores were proposed by (Dutchman) E. W. Dijkstra. It is therefore common to see the notation **P** and **V** for the operations wait and signal, respectively.

CHALMERS

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;
```

CHALMERS

Example: semaphores in Ada 95

Problem: Write a package **`Semaphores`** that implements semaphores in Ada 95.

- The package should define a protected object **`Semaphore`**.
- The object should receive an initial value when it is created.
- The object should have two entries, **`Wait`** and **`Signal`**, that work in accordance with the definition of semaphores.

We solve this on the blackboard!

CHALMERS

Using semaphores

Simple resource manager with critical regions

```
with Semaphores; use Semaphores;
Resource_Control : Semaphore(1);
task A;
task B;
task body A is
begin
  loop
    Resource_Control.Wait;
    ...         -- Critical region with statements using the resource
    Resource_Control.Signal;
  end loop;
end A;
task body B is
begin
  loop
    Resource_Control.Wait;
    ...         -- Critical region with statements using the resource
    Resource_Control.Signal;
  end loop;
end B;
```

CHALMERS

Mutual exclusion

Methods for implementing mutual exclusion:

- By disabling the processor's interrupt service mechanism
  - Only works for single-processor systems
- With atomic processor instructions
  For example: the test-and-set instruction
  - Variables can be tested and updated in one operation
  - Necessary for systems with two or more processors
- With software
  - Dekker's algorithm, Peterson's algorithm
  - Requires no dedicated hardware support

CHALMERS

Disabling interrupts

In single-processor systems, the mutual exclusion is guaranteed by disabling the processor's interrupt service mechanism ("interrupt masking") while the critical region is executed.

This way, unwanted task switches in the critical region (caused by e.g. timer interrupts) are avoided. However, all other tasks are unable to execute during this time.

Therefore, critical regions should only contain such instructions that really require mutual exclusion (e.g., code that handles the operations **wait** and **signal** for semaphores).

This method does not work for multi-processor systems!

CHALMERS

Disabling interrupts

```
procedure Main is
  task A;
  task B;

  task body A is
  begin
    loop
      Disable_Interrupts;          -- turn off interrupt handling
      ...                          -- critical region
      Enable_Interrupts;           -- A leaves critical region
      ...                          -- remaining program code
    end loop;
  end A;

  task body B is
  begin
    loop
      Disable_Interrupts;          -- turn off interrupt handling
      ...                          -- critical region
      Enable_Interrupts;           -- B leaves critical region
      ...                          -- remaining program code
    end loop;
  end B;
begin
  null;
end Main;
```

CHALMERS

Test-and-set instruction

In multi-processor systems with shared memory, a test-and-set instruction is used for handling critical regions.

A test-and-set instruction is a processor instruction that reads from and writes to a variable in one atomic operation.

The functionality of the test-and-set instruction can be illustrated by the following Ada procedure:

```
procedure testandset(lock, previous : in out Boolean) is
begin
  previous := lock;          -- lock is read and its value saved
  lock := true;              -- lock is set to "true"
end testandset;
```

The combined read and write of **lock** must be atomic. In a multi-processor system, this is guaranteed by locking (disabling access to) the memory bus during the entire operation.

CHALMERS

Test-and-set instruction

```
procedure Main is
  lock : Boolean := false;          -- shared flag
  task A;
  task B;

  task body A is
    previous : Boolean;
  begin
    loop
      loop
        testandset(lock, previous); -- A waits if critical region is busy
       exit when not previous;
      end loop;
      ...                           -- critical region
      lock := false;                -- A leaves critical region
      ...                           -- remaining program code
    end loop;
  end A;

   .
   .
   .
```

CHALMERS

Test-and-set instruction

```
   .
   .
   .
  task body B is
    previous : Boolean;
  begin
    loop
      loop
        testandset(lock, previous); -- B waits if critical region is busy
       exit when not previous;
      end loop;
      ...                           -- critical region
      lock := false;                -- B leaves critical region
      ...                           -- remaining program code
    end loop;
  end B;

begin
  null;
end Main;
```

CHALMERS

Dekker's algorithm

Accomplishes mutual exclusion with the aid of:

1. Shared memory
   a) shared flags
   b) shared counters
2. Busy-wait loops

Requires no dedicated hardware!

Fundamental assumption:
A task will not terminate in a critical region.

Peterson's algorithm is a simplification of Dekker's algorithm.

CHALMERS

Derivation of Dekker's algorithm
(for two tasks)

Attempt 1: A counter variable indicates which task is next in line to get access to the critical region.
- This guarantees mutual exclusion
- The execution order is fixed (P1 P2 P1 P2 …) which is an inefficient solution if the tasks request the critical region at different intervals
- If a task terminates outside its critical region, deadlock occurs

Attempt 2a: Two flags are used to indicate which task is currently within the critical region.
- Since testing and updating of the flags are non-atomic operations, mutual exclusion cannot be guaranteed

CHALMERS

Derivation of Dekker's algorithm

(for two tasks)

Attempt 2b: Each task first sets its own flag to "true" and then examines the other flag.

- If the tasks arrive at the critical region at the same time, deadlock can occur

Attempt 2c: If a task does not get access to the critical region, it clears its own flag and a new attempt is made later.

- If the tasks request the critical region at exactly the same interval, starvation can occur

Solution: Let the tasks take turns to try to get access to the critical region. The combination of "turns" indicator and "current-use" flags guarantees that the algorithm reaches mutual exclusion and avoids both deadlock and starvation.

CHALMERS

Dekker's algorithm

(for two tasks)

```
procedure Dekker is
  in1   : Boolean := false;          -- shared flag
  in2   : Boolean := false;          -- shared flag
  turn  : Integer range 1..2 := 1;   -- indicator for turns

  task A;
  task B;

  task body A is
  begin
    loop
      in1 := true;                   -- attempt to enter
      while in2 loop                 -- examine if B makes an attempt
        if turn = 2 then             -- examine whose turn
          in1 := false;              -- allow B to enter
          while turn = 2 loop        -- wait until B has finished
            null;
          end loop;
          in1 := true;               -- make a new attempt to enter
        end if;
      end loop;
      ...                            -- critical region
      turn := 2;
      in1 := false;                  -- A leaves critical region
      ...                            -- remaining program code
   end loop;
  end A;
```

CHALMERS

Dekker's algorithm

(for two tasks)

```
   .
   .
   .
task body B is
begin
  loop
    in2 := true;                    -- attempt to enter
    while in1 loop                  -- examine if A makes an attempt
      if turn = 1 then              -- examine whose turn
        in2 := false;               -- allow A to enter
        while turn = 1 loop         -- wait until A has finished
          null;
        end loop;
        in2 := true;                -- make a new attempt to enter
      end if;
    end loop;
    ...                             -- critical region
    turn := 1;
    in2 := false;                   -- B leaves critical region
    ...                             -- remaining program code
  end loop;
end B;

begin
  null;
end Dekker;
```