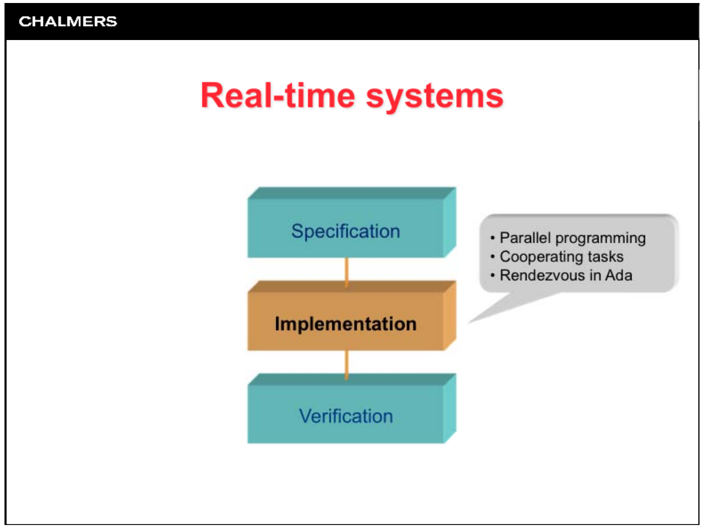


CHALMERS

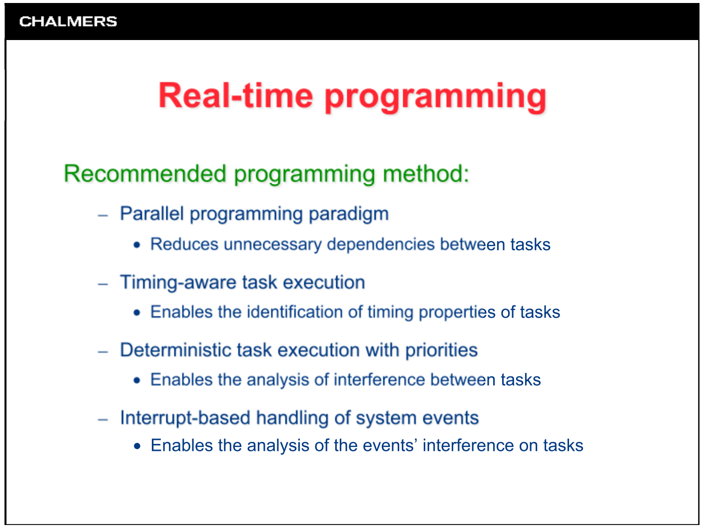
Real-time systems

Specification

Implementation

Verification

- Parallel programming
- Cooperating tasks
- Rendezvous in Ada

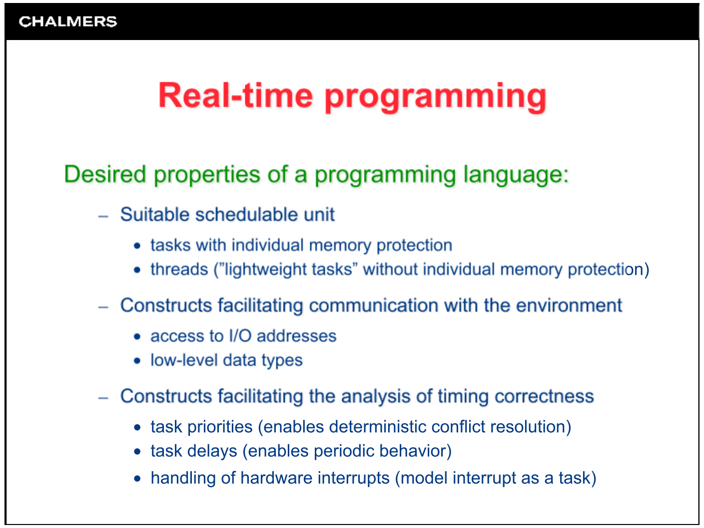


CHALMERS

Real-time programming

Recommended programming method:

- Parallel programming paradigm
  - Reduces unnecessary dependencies between tasks
- Timing-aware task execution
  - Enables the identification of timing properties of tasks
- Deterministic task execution with priorities
  - Enables the analysis of interference between tasks
- Interrupt-based handling of system events
  - Enables the analysis of the events' interference on tasks

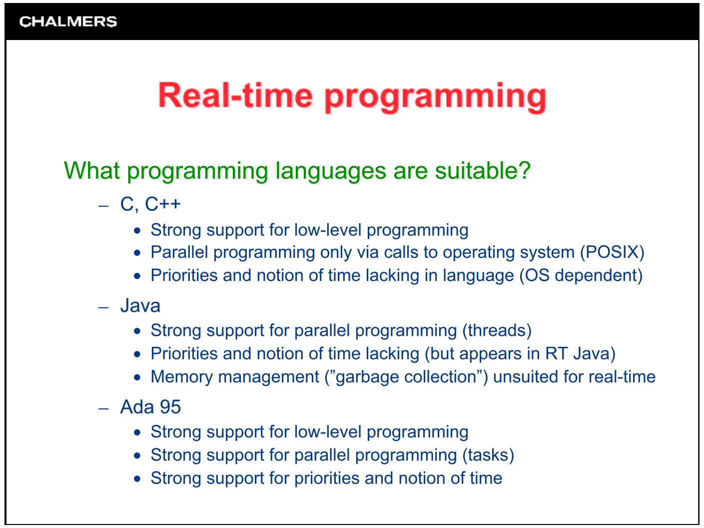


CHALMERS

Real-time programming

Desired properties of a programming language:

- Suitable schedulable unit
  - tasks with individual memory protection
  - threads ("lightweight tasks" without individual memory protection)
- Constructs facilitating communication with the environment
  - access to I/O addresses
  - low-level data types
- Constructs facilitating the analysis of timing correctness
  - task priorities (enables deterministic conflict resolution)
  - task delays (enables periodic behavior)
  - handling of hardware interrupts (model interrupt as a task)

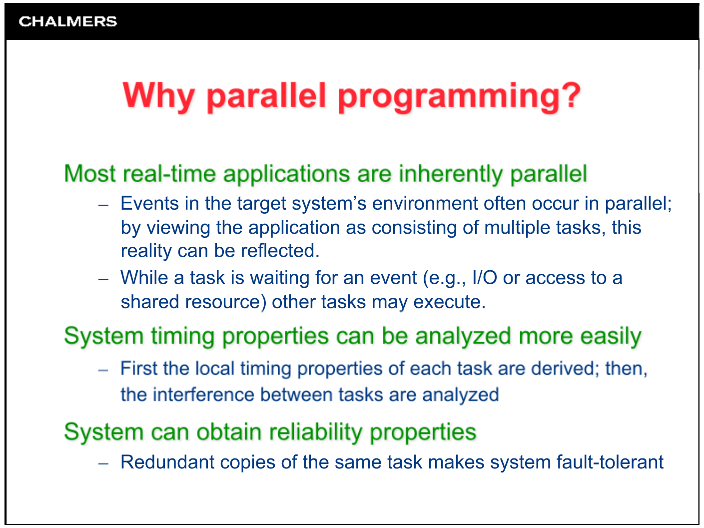


CHALMERS

Real-time programming

What programming languages are suitable?

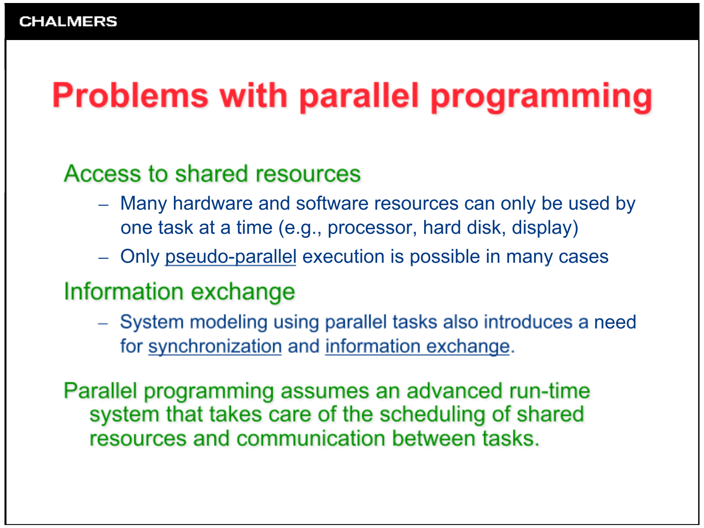
- C, C++
  - Strong support for low-level programming
  - Parallel programming only via calls to operating system (POSIX)
  - Priorities and notion of time lacking in language (OS dependent)
- Java
  - Strong support for parallel programming (threads)
  - Priorities and notion of time lacking (but appears in RT Java)
  - Memory management ("garbage collection") unsuited for real-time
- Ada 95
  - Strong support for low-level programming
  - Strong support for parallel programming (tasks)
  - Strong support for priorities and notion of time



CHALMERS

Why parallel 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 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.
- System timing properties can be analyzed more easily
  - 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

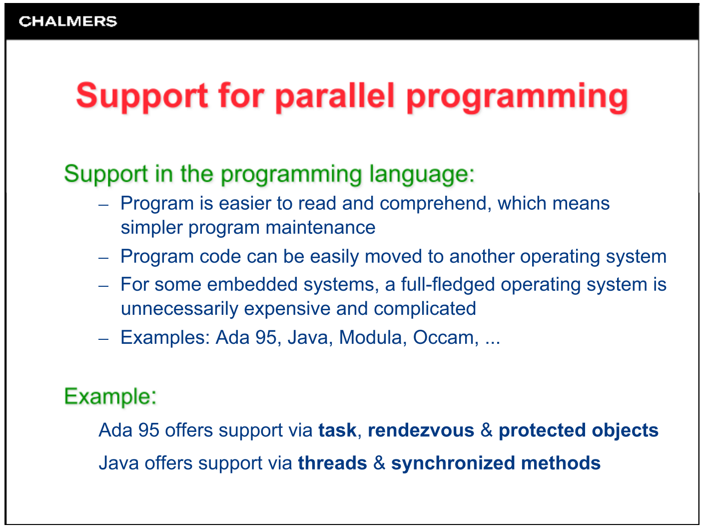


CHALMERS

Problems with parallel programming

- Access to shared resources
  - Many hardware and software resources can only be used by one task at a time (e.g., processor, hard disk, display)
  - Only pseudo-parallel execution is possible in many cases
- Information exchange
  - System modeling using parallel tasks also introduces a need for synchronization and information exchange.

Parallel programming assumes an advanced run-time system that takes care of the scheduling of shared resources and communication between tasks.



CHALMERS

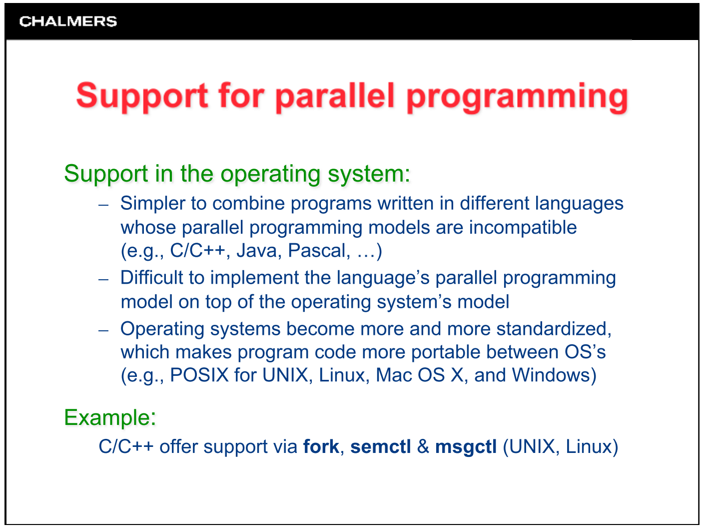
Support for parallel programming

- 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**



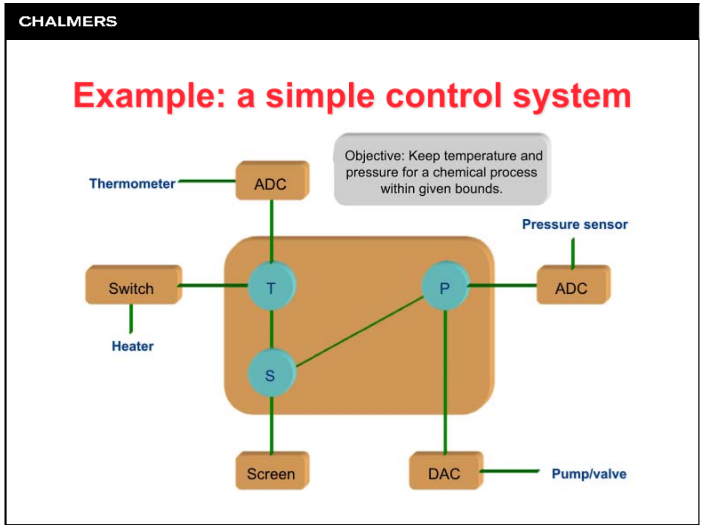
CHALMERS

Support for parallel programming

- Support in the operating system:
  - Simpler to combine programs written in different languages whose parallel programming models are incompatible (e.g., C/C++, Java, Pascal, …)
  - Difficult to implement the language's parallel programming model on top of the operating 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:

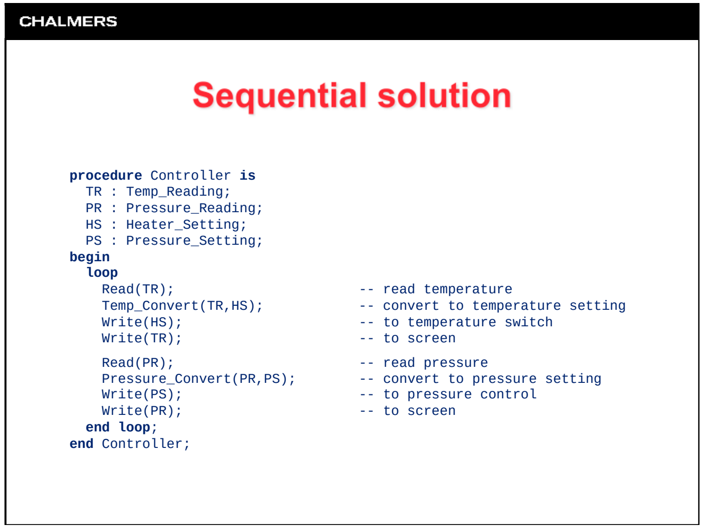
C/C++ offer support via **fork**, **semctl** & **msgctl** (UNIX, Linux)



CHALMERS

Example: a simple control system

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

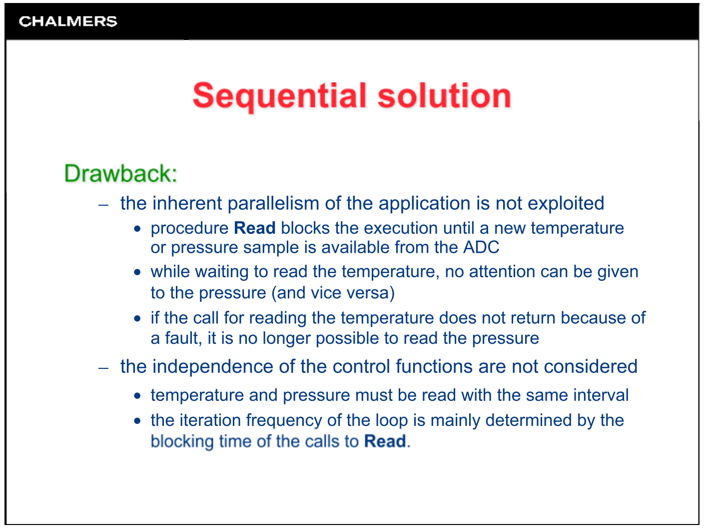


CHALMERS

Sequential solution

```
procedure Controller is
  TR : Temp_Reading;
  PR : Pressure_Reading;
  HS : Heater_Setting;
  PS : Pressure_Setting;
begin
  loop
    Read(TR);                        -- read temperature
    Temp_Convert(TR,HS);             -- convert to temperature setting
    Write(HS);                       -- to temperature switch
    Write(TR);                       -- to screen

    Read(PR);                        -- read pressure
    Pressure_Convert(PR,PS);         -- convert to pressure setting
    Write(PS);                       -- to pressure control
    Write(PR);                       -- to screen
  end loop;
end Controller;
```

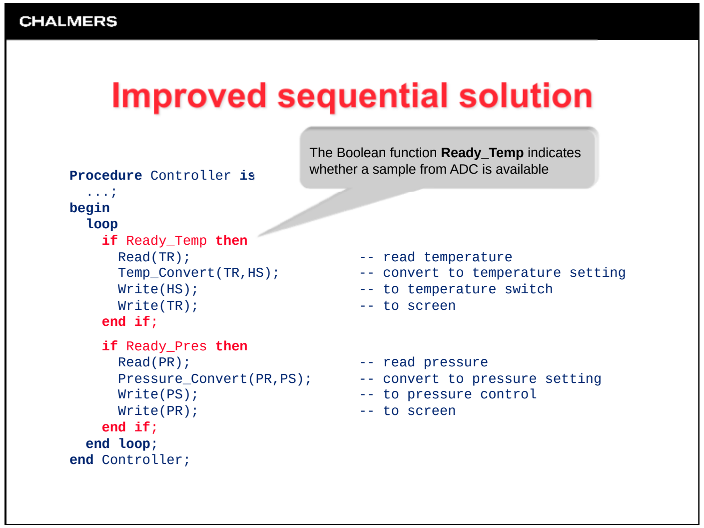


CHALMERS

Sequential solution

Drawback:

- the inherent parallelism of the application is not exploited
  - procedure **Read** blocks the execution until a new temperature or pressure sample is available from the ADC
  - while waiting to read the temperature, no attention can be given to the pressure (and vice versa)
  - if the call for reading the temperature does not return because of a fault, it is no longer possible to read the pressure
- 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 **Read**.



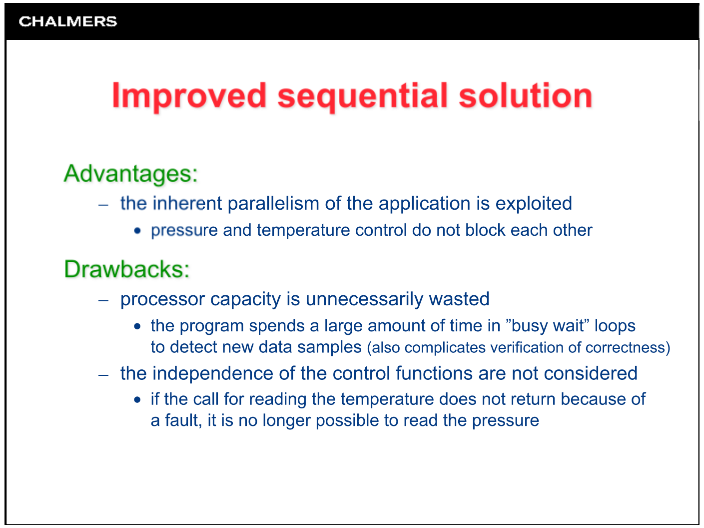
CHALMERS

Improved sequential solution

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

```
Procedure Controller is
  ...;
begin
  loop
    if Ready_Temp then
      Read(TR);                      -- read temperature
      Temp_Convert(TR,HS);           -- convert to temperature setting
      Write(HS);                     -- to temperature switch
      Write(TR);                     -- to screen
    end if;

    if Ready_Pres then
      Read(PR);                      -- read pressure
      Pressure_Convert(PR,PS);       -- convert to pressure setting
      Write(PS);                     -- to pressure control
      Write(PR);                     -- to screen
    end if;
  end loop;
end Controller;
```



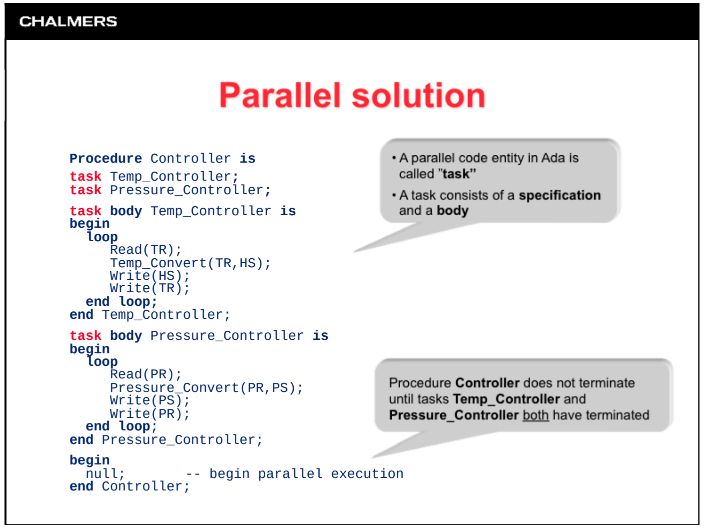
Improved sequential solution

Advantages:

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

Drawbacks:

- processor capacity is unnecessarily wasted
  - the program spends a large amount of time in ”busy wait” loops to detect new data samples (also complicates verification of correctness)
- the independence of the control functions are not considered
  - if the call for reading the temperature does not return because of a fault, it is no longer possible to read the pressure

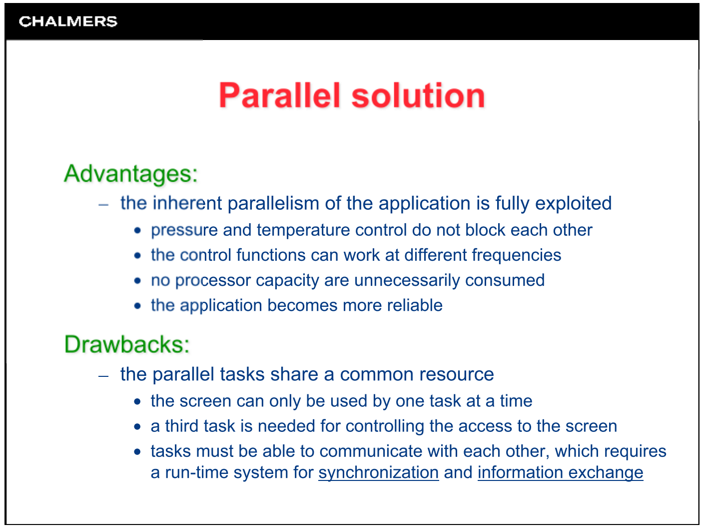


Parallel solution

```
Procedure Controller is
task Temp_Controller;
task Pressure_Controller;
task body Temp_Controller is
begin
  loop
     Read(TR);
     Temp_Convert(TR,HS);
     Write(HS);
     Write(TR);
  end loop;
end Temp_Controller;
task body Pressure_Controller is
begin
  loop
     Read(PR);
     Pressure_Convert(PR,PS);
     Write(PS);
     Write(PR);
  end loop;
end Pressure_Controller;
begin
  null;        -- begin parallel execution
end Controller;
```

- A parallel code entity in Ada is called ”**task**”
- A task consists of a **specification** and a **body**

Procedure **Controller** does not terminate until tasks **Temp_Controller** and **Pressure_Controller** both have terminated



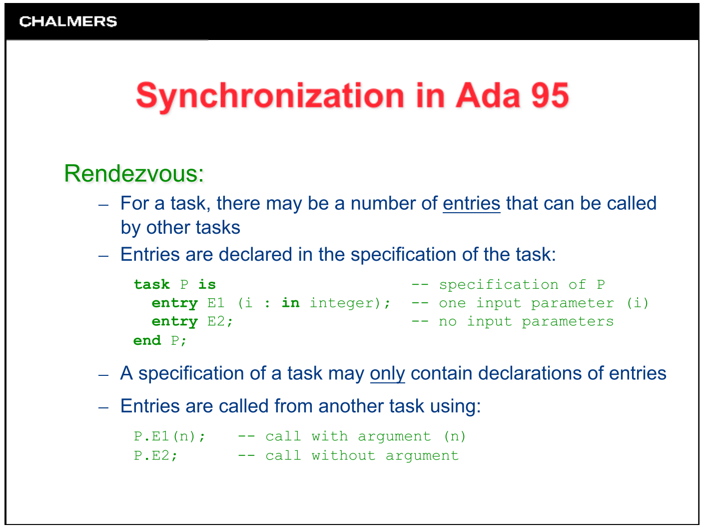
Parallel solution

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 third task is needed for controlling the access to the screen
  - tasks must be able to communicate with each other, which requires a run-time system for synchronization and information exchange



Synchronization in Ada 95

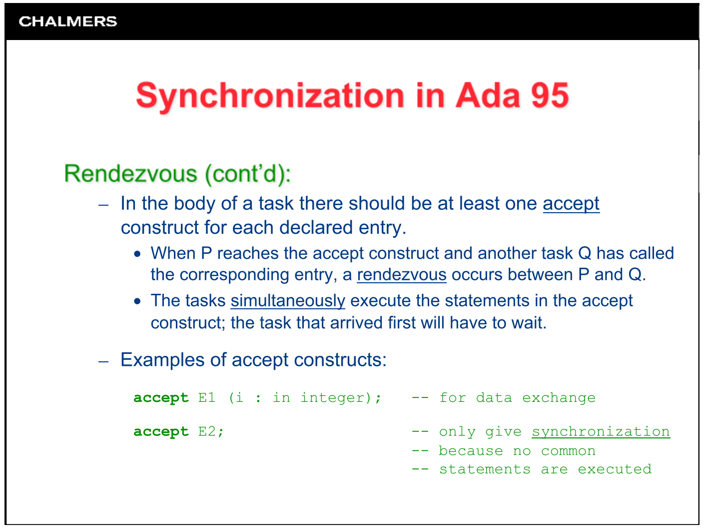
Rendezvous:

- For a task, there may be a number of entries that can be called by other tasks
- Entries are declared in the specification of the task:

```
task P is                          -- specification of P
  entry E1 (i : in integer);       -- one input parameter (i)
  entry E2;                        -- no input parameters
end P;
```

- A specification of a task may only contain declarations of entries
- Entries are called from another task using:

```
P.E1(n);    -- call with argument (n)
P.E2;       -- call without argument
```



CHALMERS

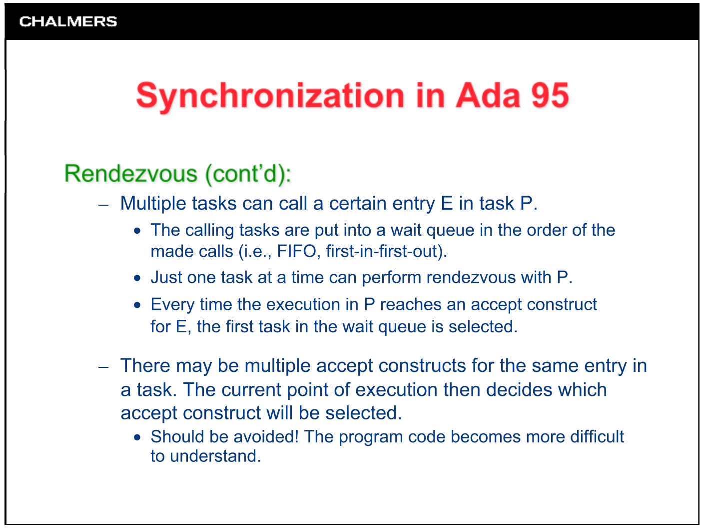
Synchronization in Ada 95

Rendezvous (cont'd):

- In the body of a task there should be at least one accept construct for each declared entry.
  - When P reaches the accept construct and another task Q has called the corresponding entry, a rendezvous occurs between P and Q.
  - The tasks simultaneously execute the statements in the accept construct; the task that arrived first will have to wait.
- Examples of accept constructs:

```
accept E1 (i : in integer);   -- for data exchange

accept E2;                    -- only give synchronization
                              -- because no common
                              -- statements are executed
```

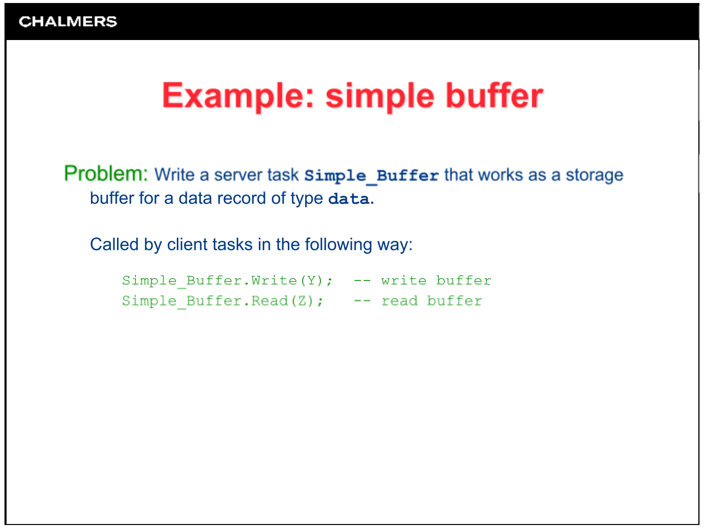


CHALMERS

Synchronization in Ada 95

Rendezvous (cont'd):

- Multiple tasks can call a certain entry E in task P.
  - The calling tasks are put into a wait queue in the order of the made calls (i.e., FIFO, first-in-first-out).
  - Just one task at a time can perform rendezvous with P.
  - Every time the execution in P reaches an accept construct for E, the first task in the wait queue is selected.
- There may be multiple accept constructs for the same entry in a task. The current point of execution then decides which accept construct will be selected.
  - Should be avoided! The program code becomes more difficult to understand.



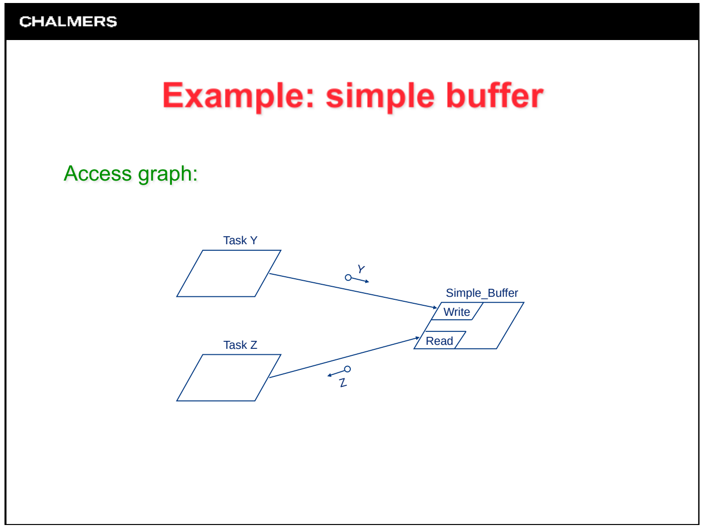
CHALMERS

Example: simple buffer

Problem: Write a server task **Simple_Buffer** that works as a storage buffer for a data record of type **data**.

Called by client tasks in the following way:

```
Simple_Buffer.Write(Y);  -- write buffer
Simple_Buffer.Read(Z);   -- read buffer
```



CHALMERS

Example: simple buffer

Access graph:

CHALMERS

Example: simple buffer

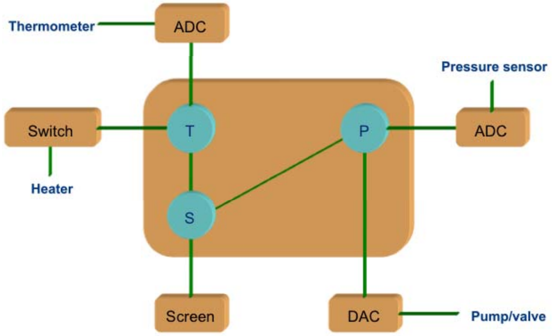
```
task Simple_Buffer is
  entry Write(d : in data);
  entry Read(d : out data);
end Simple_Buffer;

task body Simple_Buffer is
  buffer : data;
begin
  loop
    accept Write(d : in data) do
      buffer := d;    -- save client data in buffer
    end Write;

    accept Read(d : out data) do
      d := buffer;    -- return buffer data to client
    end Read;
  end loop;
end Simple_Buffer;
```

CHALMERS

Example: control system



CHALMERS

Synchronized solution

```
task Screen_Controller is
  entry Write_p(PR : in Pressure_Reading);
  entry Write_t(TR : in Temp_Reading);
end Screen_Controller;

task body Screen_Controller is
begin
  loop

    accept Write_p(PR : in Pressure_Reading) do
      put_p(PR);      -- write pressure value to screen
    end Write_p;

    accept Write_t(TR : in Temp_Reading) do
      put_t(TR);      -- write temperature value to screen
    end Write_t;

  end loop;
end Screen_Controller;
```

CHALMERS

Synchronized solution

```
Procedure Controller is
task Temp_Controller;
task Pressure_Controller;

task body Temp_Controller is
begin
  loop
     Read(TR);
     Temp_Convert(TR,HS);
     Write(HS);
     Screen_Controller.Write_t(TR); -- entry call
  end loop;
end Temp_Controller;

task body Pressure_Controller is
begin
  loop
     Read(PR);
     Pressure_Convert(PR,PS);
     Write(PS);
     Screen_Controller.Write_p(PR); -- entry call
  end loop;
end Pressure_Controller;

begin
  null;       -- begin parallel execution
end Controller;
```

CHALMERS

Synchronized solution

Drawbacks:

- the independence of the control functions are not considered
  - the screen task writes pressure and temperature every other call (predetermined sequences)
  - the sequential coding of the accept constructs in the screen task introduces a (unnecessary) dependence between the tasks **Temp_Controller** and **Pressure_Controller**
  - this solution works poorly if one of the control functions needs to write its value more often than the other (i.e., using different iteration frequencies)

⇒ the screen task needs a mechanism for considering available accept constructs simultaneously

CHALMERS

Synchronization in Ada 95

Alternative rendezvous:

- Multiple accept alternatives can be ”open” at the same time in the called task by enclosing them with **select**:

```
select
  accept E1 ( ... ) do
     ...
or
  accept E2 ( ... ) do
     ...
else
  ...                  -- do something else
end select;
```

- If rendezvous cannot occur instantly, a task can refrain from waiting and instead choose the **else** alternative in the select construct.

CHALMERS

Synchronization in Ada 95

Alternative rendezvous (cont’d):

- A corresponding action can be made for a calling task:

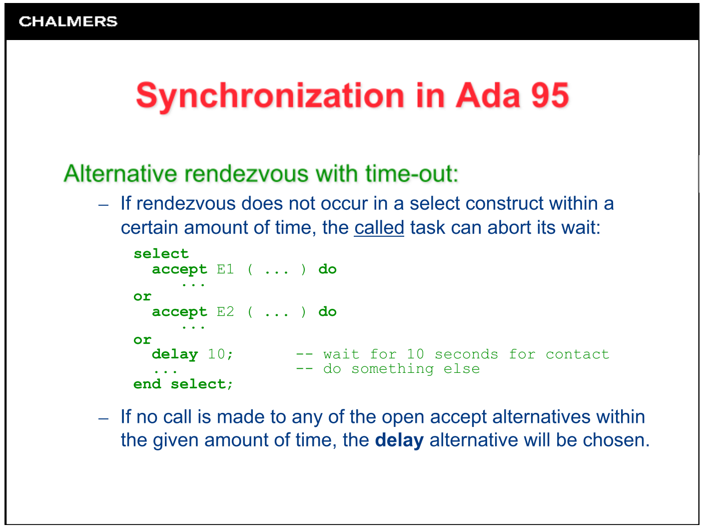
```
select
  P.E1 ( ... )     -- try to establish contact
else
  ...              -- perform error handling
end select;
```

CHALMERS

Improved screen task

```
task Screen_Controller is
  entry Write_p(PR : in Pressure_Reading);
  entry Write_t(TR : in Temp_Reading);
end Screen_Controller;

task body Screen_Controller is
begin
  loop
    select
      accept Write_p(PR : in Pressure_Reading) do
        put_p(PR);    -- write pressure value to screen
      end Write_p;
    or
      accept Write_t(TR : in Temp_Reading) do
        put_t(TR);    -- write temperature value to screen
      end Write_t;
    end select;
  end loop;
end Screen_Controller;
```



CHALMERS

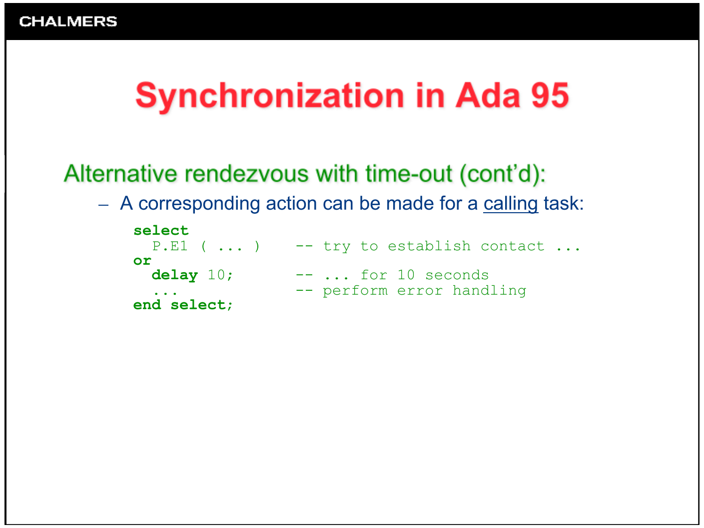
Synchronization in Ada 95

Alternative rendezvous with time-out:

- If rendezvous does not occur in a select construct within a certain amount of time, the called task can abort its wait:

```
select
  accept E1 ( ... ) do
     ...
or
  accept E2 ( ... ) do
     ...
or
  delay 10;        -- wait for 10 seconds for contact
 ...               -- do something else
end select;
```

- If no call is made to any of the open accept alternatives within the given amount of time, the **delay** alternative will be chosen.



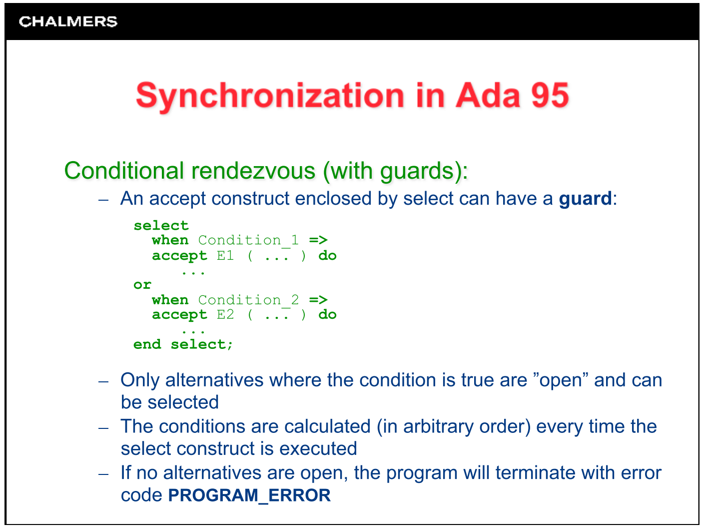
CHALMERS

Synchronization in Ada 95

Alternative rendezvous with time-out (cont'd):

- A corresponding action can be made for a calling task:

```
select
  P.E1 ( ... )    -- try to establish contact ...
or
  delay 10;       -- ... for 10 seconds
  ...             -- perform error handling
end select;
```



CHALMERS

Synchronization in Ada 95

Conditional rendezvous (with guards):

- An accept construct enclosed by select can have a **guard**:

```
select
  when Condition_1 =>
  accept E1 ( ... ) do
     ...
or
  when Condition_2 =>
  accept E2 ( ... ) do
     ...
end select;
```

- Only alternatives where the condition is true are ”open” and can be selected
- The conditions are calculated (in arbitrary order) every time the select construct is executed
- If no alternatives are open, the program will terminate with error code **PROGRAM_ERROR**