Distributed System:

- several computational entities ("nodes") with their own local memory,
- these entities communicate with each other (e.g., message passing, or shared memory),
- and the system presents itself as a homogenous system to the user (or attempts to do so).

A “node” can be a computer of its own, a core on a given processor, or even one of several processes on a single-core machine.

In any case will refer to the program running on a node as a “process”.

This section deals with the core ideas and obstacles for implementing concurrent and distributed systems, and gives some very basic hints how these are implemented in Ada.
Reasons to use Distributed Systems:

Compared to sequential systems, a Distributed System may provide

- an *improved performance*,
- an *improved failure tolerance*,
- or both.

Furthermore, implementing a system with heterogenous tasks as a Distributed System may be a *more natural model* than a strictly sequential implementation.

Consider, e.g., an automotive system to control the engine and to check the road for obstacles and to display the relevant information to the user. Instead of a single process handling all these tasks, this system is more naturally implemented by three different processes with some amount of communication between them.
Concurrency is difficult

... and often ends in the roadside ditch!

... or wherever
Questions

- **Reliability / Availability:**
  How can the system deal with individual nodes failing, or with the communication between certain nodes failing?

- **Concurrency and Synchronisation:**
  - Can several nodes work in parallel on the same task?
  - When do they need to synchronise?
  - How can they synchronise?

- **Secure Communication:**
  How to protect the system if one is using insecure communication channels (e.g., the internet, wireless communication, . . .)?
  (No topic for this lecture!)
Sequential $\leftrightarrow$ Concurrent

Two actions are sequential, if one can only be performed after the other. Examples:

- Make Coffee, drink it. (*Sequential:* drinking is only possible if the coffee has been made before.)
- Drink Coffee and tea using the same cup. (*Sequential, but undefined which comes first.*)
- Drink Coffee and tea using two different cups. (*Parallel:* Alice may be drinking her coffee while Bob is nipping from his tea.)
Alas: The Human Factor

- Humans are bad at thinking in parallel lines of action.
- Intuitively, people assume that concurrent actions are performed in the “correct” order.
- This causes “Race Conditions”.
- Most programming languages (including Ada) have a sequential semantic:
  - If statement $B$ follows statement $A$, then $A$ has to be performed first and $B$ has to be performed then.
  - The optimizer may change this order, but the optimized program must functionally behave as if $A$ has been performed before $B$.
- Programming languages with a parallel semantic have not been successful, so far. (Some exception: purely functional languages.)
- However, programming language with a sequential semantic can still provide mechanisms for concurrent programs . . .
Support for Concurrent Programming

- Some modern programming languages provide such support (Java’s “Threads”, Ada’s “Tasks”, ...).
- Otherwise, there are libraries providing such support (e.g., pthreads). The main concept is the MutEx “Mutual Exclusion”.
A “Race Condition” Example

A setuid-program

```c
if (access(file, R_OK) != 0) {
    exit(1);
}
```

```c
fd = open(file, O_RDONLY);
// read content of fd ...
```

The if-Statement sorts out users without read permission for the file. Readers with read permission can actually read the file.

Where is the problem? A Time-of-check-to-time-of-use Attack:

1. Generate a file $x$ which you are allowed to read!
2. Start the program
3. Change $x$ into a symlink, which links to a file $y$ for which you have no read permission . . . but which you would like to read.
   With good timing (and, maybe, luck) you will actually read $y$ . . .
Therac-25
A computer-controlled radiation therapy machine

- Two modes for radiation therapy:
  - **direct mode**  radiation = dosage at the patient
  - **X-ray mode** (for inner body parts; beam passes through a wolfram-“target”) radiation = dosage \* 100

- predecessors have not been computer-controlled, but used mechanical locks

- In use from 1983 on. Until 1987 several patients suffered from radiation poisoning, three killed.
**direct mode** radiation = dosage at the patient

**X-ray mode** radiation = dosage * 100

Implementation as two independent processes:
- R to control the radiation, and
- M to control the mode and move the target.

**A race condition** (found in 1987 by accident):

1. Operator entered correct dosage, but wrong mode (“X-Ray”).
2. M selected X-ray; R calculated the radiation (= 100 * dosage)
3. Operator corrected the wrong mode *within less than 8 seconds*.
4. M switched into direct mode; R started beaming – without re-calculating the radiation!
5. Patient received 100 times more radiation than intended (not always – a safety mechanism could trigger an early abort).

**The Short-term solution:** Disable the key to correct wrong data!
6.1: Mutual Exclusion

Extract from the *pthreads*-Tutorial:

Mutex

- ... abbreviation for “mutual exclusion”
  ... one of the primary means of implementing thread synchronisation and for protecting shared data when multiple writes occur.

- A mutex variable acts like a “lock” protecting access to a shared data resource.
  ... only one thread can lock (or own) a mutex variable at any given time.
  ... if several threads try to lock a mutex only one thread will be successful. No other thread can own that mutex until the owning thread unlocks that mutex.
Mutexes can be used to prevent “race” conditions. An example of a race condition involving a bank transaction . . .

<table>
<thead>
<tr>
<th>Action 1</th>
<th>Action 2</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>withdraw 200.00</td>
<td>deposit 300.00</td>
<td>1000.00</td>
</tr>
<tr>
<td>X := Read Account</td>
<td>Y := Read Account</td>
<td>1000.00</td>
</tr>
<tr>
<td>X := X - 200.00</td>
<td>Y := Y + 300.00</td>
<td>1000.00</td>
</tr>
<tr>
<td>Write Account(X)</td>
<td>Write Account(Y)</td>
<td>800.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1300.00</td>
</tr>
</tbody>
</table>

In the above example, a mutex should be used to lock the “Balance” while a thread is using this shared data resource.
Very often the action performed by a thread owning a mutex is the updating of global variables. . . . the final value is the same as what it would be if only one thread performed the update. The variables being updated belong to a “Critical Section”.

A typical sequence in the use of a mutex is as follows:
1. Create and initialize a mutex variable
2. Several threads attempt to lock the mutex
3. Only one succeeds and that thread owns the mutex
4. The owner thread performs some set of actions
5. The owner unlocks the mutex
6. Another thread acquires the mutex and repeats the process
7. Finally the mutex is destroyed

When several threads compete for a mutex, the losers **block** at that call – an unblocking call is available with “trylock” instead of the “lock” call.

When protecting shared data, it is the programmer’s responsibility to make sure every thread that needs to use a mutex does so.
Simplifying the example

Thread 1
\[ a = \text{Data}; \]
\[ a++; \]
\[ \text{Data} = a; \]

Thread 2
\[ b = \text{Data}; \]
\[ b--; \]
\[ \text{Data} = b; \]

Data may have been changed by either +1, 0, or −1.
Using pthreads-Mutex-Variables

Thread 1

```c
pthread_mutex_lock (&mut);

/* critical */ a = Data;
/* critical */ a++;
/* critical */ Data = a;

pthread_mutex_unlock (&mut);
```

Thread 2

```c
pthread_mutex_lock (&mut);

/* Thread 2 blocks */

/* critical */ a = Data;
/* critical */ a++;
/* critical */ Data = a;

pthread_mutex_unlock (&mut);
```

/* critical */ b = Data;
/* critical */ b--;
/* critical */ Data = b;

pthread_mutex_unlock (&mut);

Data is the same as before – as it should be!
Mutual Exclusion: Summary

- Important concepts you should understand: “critical regions”, “blocked process”, “mutual exclusion”.

- Libraries like pthreads enable the programming of Concurrent and Distributed Systems – even if the underlying programming language doesn’t provide any such support.

- Note that critical sections must be found “by hand”, mutex-lock-/unlock-operations must be inserted by the programmer, . . .

- The shared variable may be a complex data structure instead of a single number.

- Different critical regions may overlap. This is complicated (in which order shall I call mutex_unlock) and may cause “deadlocks”.

- All in all, the usage of mutex-variables is the “assembler language” approach towards concurrency.

Secure and reliable software should follow a higher-level approach!
6.2: Example: Control a central heating

- User can set a target temperature.
- The water pump
  - is switched on if it is too cool in any room and
  - is switched off if it is sufficiently warm in all rooms.
- The burner
  - is switched on if the water in the boiler is too cool and
  - is switched off if the water is sufficiently warm.
with Boiler; with House;

procedure Control is
  procedure Boiler_Control(...) is ...  
  procedure House_Control(...) is ...
begin
  loop -- the infinite *control* *loop*

    -- first deal with user input
    if House.User_Has_Pressed_Button then
      House.Read(Target_Temperature);
    end if;

    -- then deal with the boiler
    Boiler_Control(72, 2); -- "hardwired" flow temperature

    -- then deal with the pump
    House_Control(Target_Temperature);

  end loop;
end Control;
procedure Boiler_Control

(Target_Temperature, Accuracy: Boiler.Temperature) is

T: Boiler.Temperature := Boiler.Sensor;

begin

if T < Target_Temperature − Accuracy then
    Boiler.Burner.Switch_On
elsif T > Target_Temperature + Accuracy then
    Boiler.Burner.Switch_Off;
end if;

end Boiler_Control;
procedure House_Control

procedure House_Control

(Target_Temperature: House.Tenth_Part_Of_Celsius) is

use House;
T: Tenth_Part_Of_Celsius := Tenth_Part_Of_Celsius’Last;
Temperature: Tenth_Part_Of_Celsius;

begin
for I in Sensor_ID loop
  Temperature := Sensor(I);
  if Temperature < T then
    T := Temperature;
  end if;
end loop; — T = lowest Temperature in any room

if Target_Temperature < (T–5) then
  Pump.Switch_On;
elsif Target_Temperature > (T+5) then
  Pump.Switch_Off;
end if;
end House_Control;
Busy Waiting

From Wikipedia:

In software engineering, busy-waiting or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input is available, or if a lock is available. . . .

Spinning can be a valid strategy in certain circumstances, . . .

In general, however, spinning is considered an anti-pattern and should be avoided, as processor time that could be used to execute a different task is instead wasted on useless activity.

(my emphasis)
Busy Waiting (2)

Beyond wasting processor time, there are further issues with this approach:

- One process (namely, the user input) can block the others.
- Another process is critical: The burner must be switched off, after some time.
- If the user can’t make up her mind, and the burner just happens to be switched on, then BOOOOM!
An improved solution

- We will represent our logical processes as three **tasks** in Ada.
- Our tasks don’t terminate.
- But when they have done their job, they will sleep (without consuming CPU time), until the job needs to be done again. We use Ada’s `delay`-statement for this purpose.
- Our solution has still some weaknesses – ideally, our tasks should have different priorities (e.g., controlling the burner should have a higher priority than reading the user input).
- Ada provides the tools to assign priorities to tasks. But for the sake of simplicity, we will skip that issue here.
Source code for the improved solution

```plaintext
with Boiler; with House;

procedure Improved_Control is

pragma atomic (Target_Temperature);
pragma volatile (Target_Temperature);

-- specify three dependent tasks
task Boiler_Control;
task House_Control;
task User_Input;

-- the task implementations
task body Boiler_Control is ... 
task body House_Control is ... 
task body User_Input is ... 

begin -- main program can only terminate
    null; -- after termination of all dependent tasks
end Improved_Control;
```
task body User_Input is
begin
  loop
    if House.User_Has_Pressed_Button then
      House.Read(Target_Temperature);
    end if;
    delay 0.5; -- check about twice a second if
    -- the user has pressed a button
  end loop;
end User_Input;
task body Boiler_Control is
  -- hardwired flow temperature: 72 Celsius (+−2)

  T: Boiler.Temperature := Boiler.Sensor;

begin
  loop
    if T < 70 then
      Boiler.Burner.Switch_On;
      delay 0.05;  -- critical!
    elsif T > 74 then
      Boiler.Burner.Switch_Off;
      delay 5.0;  -- not critical
    end if;
  end loop;
end Boiler_Control;
task body House_Control is
  use House;
  T: Tenth_Part_Of_Celsius
       := Tenth_Part_Of_Celsius 'Last;
  Temperature: Tenth_Part_Of_Celsius;

begin
  loop
    for I in Sensor_ID loop
      Temperature := Sensor(I);
      if Temperature < T then
        T := Temperature;
      end if;
    end loop;  —— T is the smallest
      —— Temperature anywhere

    if Target_Temperature < (T−5) then
      Pump.Switch_On; delay 1.0;
    elsif Target_Temperature > (T+5) then
      Pump.Switch_Off; delay 5.0;
    end if;
  end loop;
end House_Control;
Communication / Synchronisation

- Our processes communicate via a single global variable Target_Temperature.
- If more than one process would write that variable, we would have to worry about synchronisation.
- In our case, synchronisation is not an issue, since namely User_Input is the only process writing to Target_Temperature.
- The remaining synchronisation issues are solved by
  - "pragma atomic (Target_Temperature);" and
  - "pragma volatile(Target_Temperature);".

(What would be the problem if these pragmas were missing?)
6.3: Tasks in Ada

- A **task** in Ada is a *limited object* (no assignment and no comparison defined).
- We can specify **task types**:

```ada
task type T is
    ... — specification of operations
end T;
```

- Then we have to provide the implementation:

```ada
task body T is
    ... — implementation of operations
end T;
```

- And finally, we can generate objects of such a type:

```ada
Alice, Berta: T; — generates 2 task objects
Workers: array (1 .. 5) of T; — another 5 task objects
type T_Acc is access T;
Acc_Worker: T_Acc; — does not generate any task object!
begin
    Acc_Worker := new T; — yet another task object (number 8)
```
Properties of Ada-Tasks

▶ Once a task object has been generated, it runs. (It does not need some `start` command.)
▶ The calling task can only terminate after the termination of all tasks depending on it.
▶ There is always an “environment task”. All other tasks depend either directly or indirectly on the environment task. The environment task (or “main program”, as one might also call it) can be quite trivial, as in

```ada
begin
    null;
end Improved_Control;
```
Ada-Tasks and Exceptions

- Within a task, raising and handling of exceptions works as usual.
- **But pay heed to the following:** If an exception is not handled within a task, the task will just terminate without any further action.
- For beginners with programming Ada tasks – as most of you are – this leads to a sometimes very surprising behaviour.
- On the other hand, the obvious alternative to “no further action” would be to propagate the exception to the calling task. But that might be the worse of two evils. *The calling task would have to be prepared to handle any exception from any of the tasks it had called at any time – driving the programmer crazy!*
- Since Ada 2005, the calling task can define callback procedures, which will be called when a dependent task terminates.
Periodic Activities (1)

```ada
task Periodic;

task body Periodic is
begin
  loop
    Action( "Periodic" );
    delay 1.0;
  end loop;
end Periodic;
```

- Repeats Action every $t + 1$ seconds (at most), where $t$ is the time to perform the Action.
- No Busy-Wait.
- The `delay` argument is from a predefined numerical type `Duration`.
task More_Periodic;

task body More_Periodic is
  Interval: constant Duration := 1.0;
  use type Ada.Calendar.Time;
  Next_Time: Ada.Calendar.Time;
begin
  Next_Time := Ada.Calendar.Clock + Interval;
  loop
    Action("More_Periodic");
    delay until Next_Time;
    Next_Time := Next_Time + Interval;
  end loop;
end More_Periodic;
The famous “Dining Philosophers” problem.
Some times, the philosophers starve at the dining table.
The Setting in which Deadlocks can Occur

- Some resources are limited, i.e., if a process has to reserve the resource before using it, and other processes who need to access the resource block until the resource is given free.
- Processes reserve resources one after the other, in any order. (The alternative would be to require that a process reserves all resources it might ever need at its initialisation.)
- There is no “preemption”, i.e., once a resource has been reserved by a process, it will not be given free except by the process itself. (The alternative would be to take away the resource from the process, without its cooperation.)
Directed Graphs

- View a distributed system as a directed graph.
- Vertices are processes and resources.
- If a process $P$ has reserved a resource $R$: edge $P \rightarrow R$.
- If a process $P'$ waits for resource $R$ to become available: $R \rightarrow P'$.
- Any cycle of the form

$$ P \rightarrow R \rightarrow P' \rightarrow R' \rightarrow \cdots \rightarrow P $$

is a **deadlock**.
- None of the processes $P, P', \ldots$ involved in a deadlock will ever run again.
- None of the resources $R, R', \ldots$ involved will ever become free again.
Avoiding Deadlocks – or Trying to Survive with them

- Finding potential deadlocks by *testing* is essentially hopeless.
- Ideally, one tries to *statically prove* that a given Distributed System cannot deadlock.
- Often, one lives with the existence of potential deadlocks. One still tries to
  1. discover the occurrence of a deadlock and
  2. handle the case by aborting (at least) one of the involved processes, thus freeing the locked ressource(s).
- Deadlock avoidance, recognition and handling is a research topic in computer science, with plenty of unsolved problems.
6.5: Communication between Tasks

- Data-Based communication:
  - “Mutual Exclusion” (see above)
  - Shared variables, such as Target_Temperature above
  - Famous textbook examples: A semaphor, a monitor, . . .
  - Ada’s high-level concept: protected types (see below)

- Communication by Message Passing:
  - Process S sends a message to another process R:
    send Message to S —— no Ada syntax.
  - On the other hand, R waits for a message from S, or from any process
    wait for Message from E —— not possible in Ada!
    wait for Message —— from anybody —— no Ada Syntax!
Guards

In general, the $R$ will not wait for a single message, but for one of several messages. Also, there are situations, where $R$ cannot handle some messages:

```
select —— Rendezvous — a bit like Ada
    not(Buffer_Full) => wait for Write_Into_Buffer(Item);
or
    not(Buffer_Full) => wait for Read_From_Buffer(Item);
end select;
```

The abstract concept of a guard:

- If $n \geq 1$ guards are true then any of the $n$ alternatives is chosen (it is not specified, which one).
- It would be an error if, at any point of time, no guard is true!
Ada Rendezvous

- Ada-rendezvous are synchronous:
  - Regardless of sender $S$ or receiver $R$ – whoever is first at the rendez-vous point, has to wait for the other one.

- One could also imagine asynchronous communication:
  - $S$ sends a message to $R$ and then goes on with its own work – without waiting for $R$ to read the message.

- Theoretically, synchronous and asynchronous communication are equally powerful – one can be realized by the other one. Thus, most programming languages with support for concurrency support either synchronous or asynchronous communication, but not both.

- Initially, Ada (83) supported only synchronous communication (message passing via rendezvous). This lead to too many work-around solutions for problems which actually required asynchronous communication.

Ada 95 introduced **protected types** for asynchronous communication. These are as powerful as rendezvous.
protected type Cnt is

  entry Decr;
  entry Incr;
  function Val
    return Integer;

private
  N: Natural := 0;
end Signal_Object;

end Cnt;

protected body Cnt is

  entry Decr when Val > 0 is
    begin
      N := N − 1;
    end Decr;

  entry Incr when Val < ... is
    ...
  ...

  function Val
    return Integer is
      begin
        return N;
      end Val;

end Cnt;

If C is of type Cnt then

- tasks can call C.Incr;
  C.Decr; or M := C.Val;

- without having to worry about mutual exclusion.