Presented material is based on

- Other sources are marked in the text accordingly
* Real-time systems are processing systems which have to react according to some (hard) time constraint.

* Example: Sensor writes to a finite (ring-)buffer. The reader task has to guarantee that no entry is lost.

* Correctness means not only functional correctness; results must be delivered just-in-time. There are many techniques and formalisms to reason about the correctness of so-called real-time systems. In the lecture you have seen only one of them, namely Lustre and Luke
A real-time task is a piece of software subject to stringent timing constraints.

Its invocation has to be done in-time such that a dedicated deadline is (always) meet.

Example: The reader task has to guarantee that each data item is fetched from the buffer at most after \((n-1)p\) time units, where \(p\) is the period of the sensor and \(n\) the size of the buffer.
Tasks are mapped to resources, each with its own architecture/OS dependent best and worst-case execution time. Determining this interval is far from trivial.

Task processes in-coming events

If input has been processed, signal is send to down-streamed component

task is inactive, if no input is in the buffer and output has been released.
* DMA (Direct Memory Access)
* Caches
* Interrupt-handling
* System calls
* Semaphores
* Memory management
* Programming languages

* Predictability in RT-OS: a never-ending story
* DMA (Direct Memory Access): CPU and DMA-device are racing for bus access. With the DMA device at a higher priority, worst-case waiting times of the task currently executing on the CPU are unknown.

* Caches: program and temporal locality can make caching a useful technique for increasing the performance (lowering) the execution time of a real-time task. However, in case of misses the cache introduces additional overheads, as well as maintaining RAM and cache consistency. Assuming a miss each time might be too pessimistic. For ruling this out some architectures, e.g., in avionics simply turn the caches off

* Interrupts: commonly interrupt handlers, e.g., of I/O-devices, are assigned a static priority over-ruling application’s task. However, in control systems this might be not acceptable, as the number of interfering interrupts can not be bounded, worst-case execution times of real-time tasks is difficult to estimate.

* Predictability, an open issue
Interrupt disabling

* straight-forward solution with non-interruptible task execution, (task is executing a busy wait when accessing I/O-devices). Disable interrupts when executing a task

* Kernel does not need to be modified when replacing an I/O-device

* Real-time task needs knowledge about the I/O-device for implementing the accesses. Usage of a library as an interface can solve this issue.

* Suffers from low performance, over-dimensioning of platforms due to waste of CPU cycles.

* On multi-core platforms this might be difficult to implement

* Interrupt-handling in RT-OS
Periodic interrupt handling by kernel routines

* non-interruptible task execution; reduces the non-determinism imposed by the non-predictable occurrence of interrupts

* Worst-case waiting times for accesses to I/O-devices deterministic, the interrupt handling is wrapped in a system call.

* More wasteful as real-time tasks have to wait for the interrupt handling kernel routines to finish (busy waiting).

* Overhead due to kernel-thread invocation and the kernel-layered communication with the I/O device.

* Kernel needs to be updated upon replacement of peripheral devices.

* Interrupt-handling in RT-OS
Interrupt handling by real-time tasks

* Interrupt handlers are of minimal size

* The interrupt handling itself is left to some dedicated task which is the device manager

* Device managing task may have a low priority, hence, execution of other control-relevant real-time tasks can be preferred.

* Real-time task does not need to stall while interrupt is served, its execution can be suspended (reduced busy waiting).

* Non-determinism imposed by interrupts is eliminated. This scheme does not disable interrupts.

* Interrupt-handling in RT-OS
* DMA (Direct Memory Access)
* Caches
* Interrupt-handling
* System calls
* Semaphores
* Memory management
* Programming languages

* Predictability in RT-OS
* **System calls:** bound on their execution time is necessary, calls need to be preemptive for scheduling real-time tasks executing control functions

* **Memory management:** memory segmentation with fixed-memory scheme and static partitioning. If provided, garbage collection needs to be done in bounded time.

* **Semaphores:** priority-inversion needs to be ruled out

* **Predictability, an open issue**
Trouble of Mars Pathfinder is a classic example of an error induced by priority inversion in RT systems.
* Disabling preemption when executing critical sections (difficult to do, interrupt handling on multi-core architectures)

* Priority ceiling: synchronization protocol for shared resources to avoid unbounded priority inversion and deadlocks due to wrong nesting of critical sections.
Priority ceiling: synchronization protocol for shared resources to avoid unbounded priority inversion and **deadlocks** due to wrong nesting of critical sections.

* Each resource is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may lock the resource.

* If the resource is already taken the requesting task is blocked on that resource.

* If the resource is available, the current priority barrier of the resource is tested:
  * If the task has a higher priority than the current priority of the resource, it allocates the resource, with the resource inheriting the task’s priority.
  * If the task has no higher priority, it gets the resource only if he already holds the resource with the currently highest priority (barrier).

* A task will not get scheduled if any resource it may lock actually has been locked by another task, and therefore the priority ceiling protocol prevents deadlocks.