Embedded Operating System
RTOS chacateristics

Graziano Pravadelli

Real-time system

A computer system in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the *physical instant at which these results are produced*

A system that is required to *react to stimuli* from the environment (including the passage of physical time) *within time intervals dictated by the environment*

“Real” and “Time”

Time

- Main difference to other classes of computation

Real

- Reaction to external events must occur during their evolution

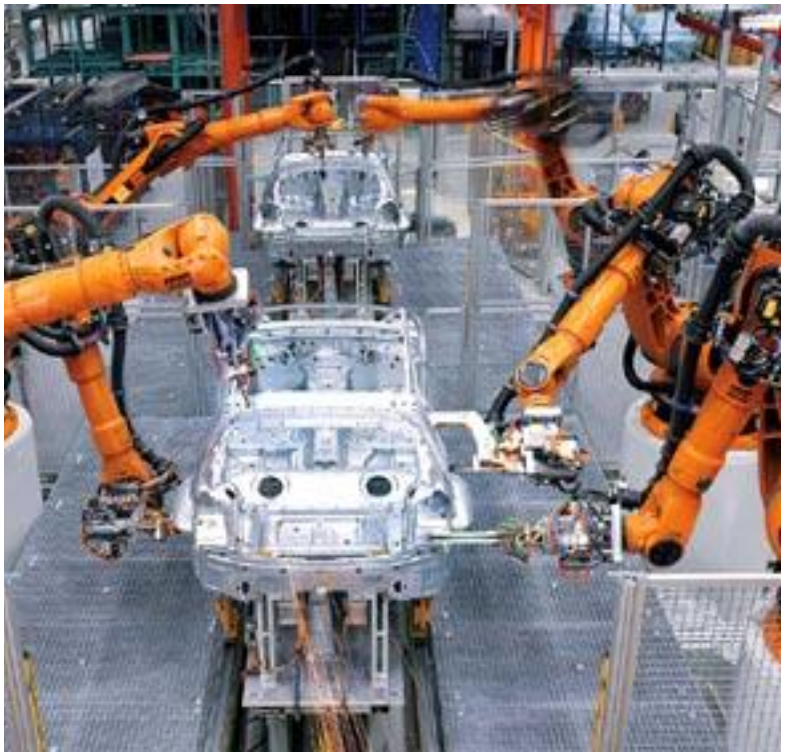
Concept of deadline

- Maximum time within which the task must be completed
- System time (internal time) must be measured with the same time scale used to measure the controlled environment (external time)
- **Real time does not mean fast but predictable**
- **After deadline, a computation is not just late, it is wrong!**

Examples of real time systems

- Plant control
- Control of production processes
- Industrial automation
- Environmental acquisition and monitoring
- Railway switching systems
- Automotive applications
- Flight control systems
- Telecommunication systems
- Robotics
- Military systems
- Space missions
- Household appliances
- Virtual/augmented reality

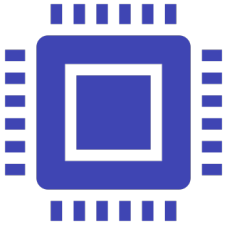
From portable devices to larger systems



Hard vs. soft vs. firm time

Hard RT task	if missing its deadline may cause catastrophic consequences on the environment under control
Soft RT task	if meeting its deadline is desirable (e.g., for performance reasons) but missing does not cause serious damage
Firm RT task	If missing its deadline does not cause damage but the results is useless after the deadline

Hard real time systems



Typical hard real time activities

Sensory data acquisition

Detection of critical conditions

Low-level control of critical system components

Application areas

Automotive

- power-train control, air-bag control, steer by wire, brake by wire

Aircraft

- engine control, aerodynamic control

Soft real time systems

- **Typical soft real time applications**
  - Command interpreter of user interface
  - Keyboard handling
  - Displaying messages on screen
  - Transmitting streaming data
- **Application areas**
  - User interaction
  - Comfort electronics

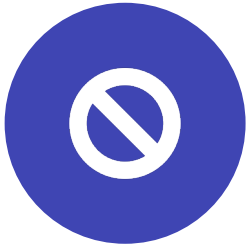
Firm real time systems

- **Typical firm real time activities**
  - Video encoding/decoding
  - Image processing
  - Data transmission in distributed systems
- **Applications areas**
  - Networked applications
  - Multimedia systems

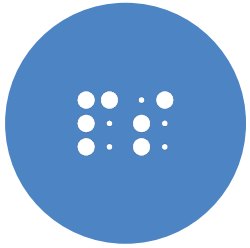
Misconception about RT computing

- RT system designed by using
  - Assembly-level ad hoc techniques
  - Programming timers
  - Low-level drivers for device handling and interrupt priorities
- Pros
  - Very optimized and efficient code
- Cons
  - So much!

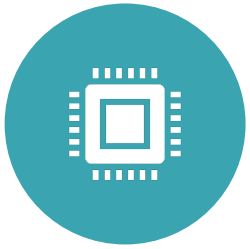
Disadvantages



Tedious programming



Difficult code understanding... for others



Difficult software maintainability... but also for original developers

Difficult verification of time constraints... unpredictable system!

n Postulates

Any software bug will tend to maximize the damage.

2. *The worst software bug will be discovered six months after the field test.*

Green's Law

If a system is designed to be tolerant to a set of faults, there will always exist an idiot so skilled to cause a nontolerated fault.

Corollary

Dummies are always more skilled than measures taken to keep them from harm.

ohnson's First Law

If a system stops working, it will do it at the worst possible time.

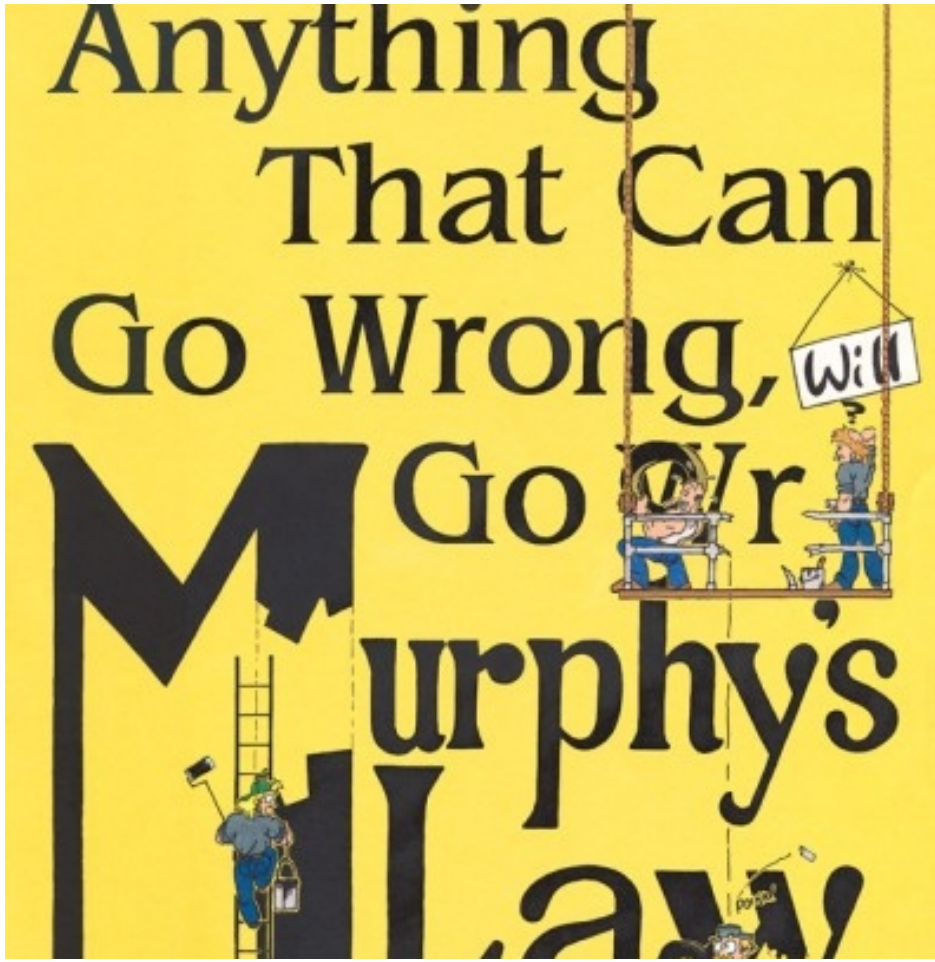
Second Law

A tragic example (Gulf War 25/02/1991)

- What
  - A radar sighted a Scud missile directed to Saudi Arabia, classified the event as a false alarm
  - A few minutes later, the Scud fell on the city of Dhahran
- Why
  - Long interrupt handling routine running with disable interrupts caused a delay of about 57 microseconds per minute
  - The computer had been working for about 100 hours thus accumulating a total delay of 343 milliseconds
  - This caused a prediction error of 687 meters!
- Solution
  - The bug was corrected by a few preemption points inside the interrupt handler

And then...

- Necessary
  - robust guarantee of the performance of a real-time system under all possible operating conditions
  - capability of anticipate pessimistic scenarios
- Achievable only by:
  - Using more sophisticated design methodologies
  - combined with static analysis of source code and specific operating systems mechanisms taking care of timing constraints

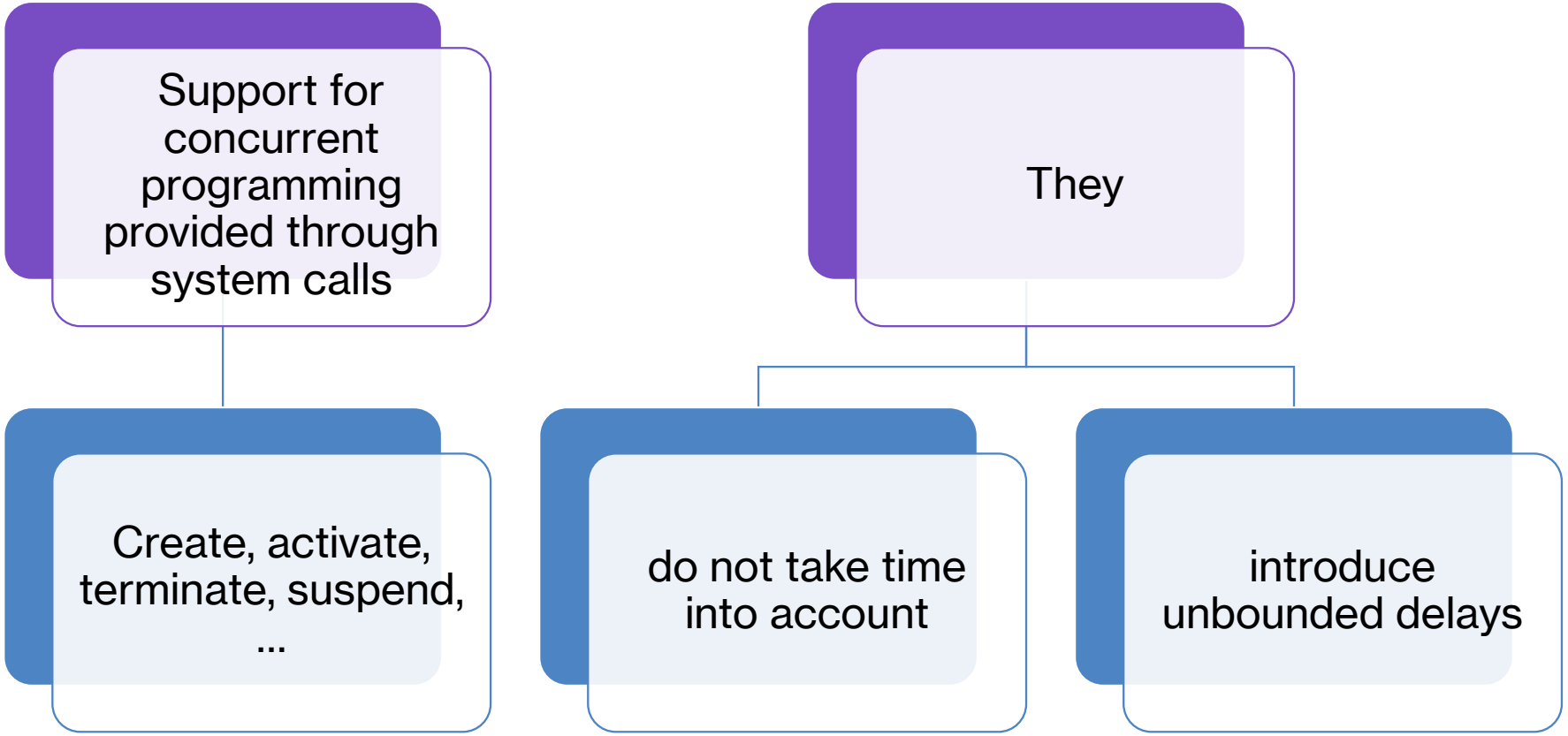


Conventional OS: inadequate for RT

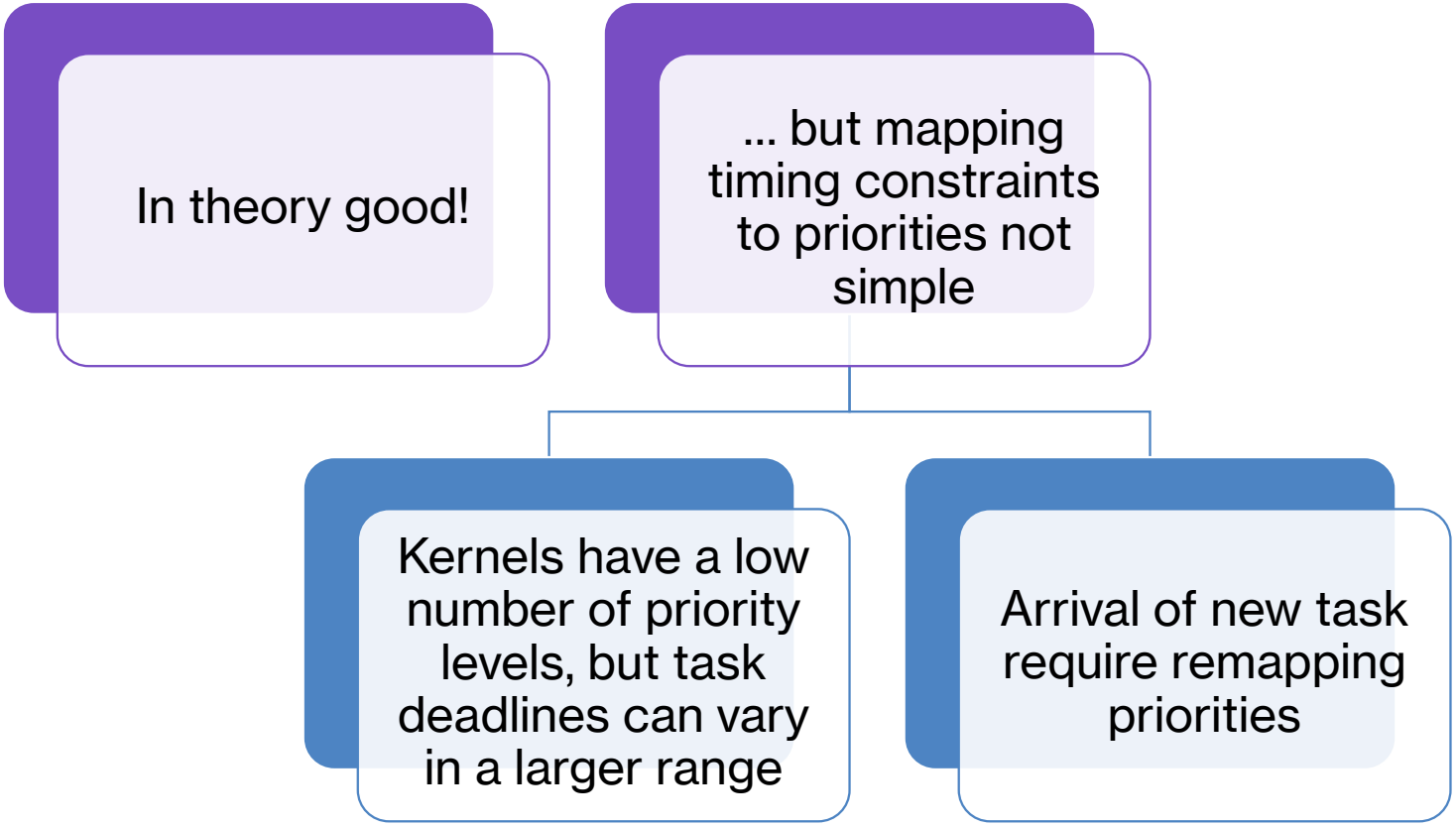
Aim: minimal response time

- Multitasking
- Priority-based scheduling
- Quick reaction to external interrupts
- Process communication and synchronization
- Small kernels
- Internal time reference

(Un)desirable: Multitasking



(Un)desirable: Priority-based scheduling



(Un)desirable: Quick reaction to external interrupts

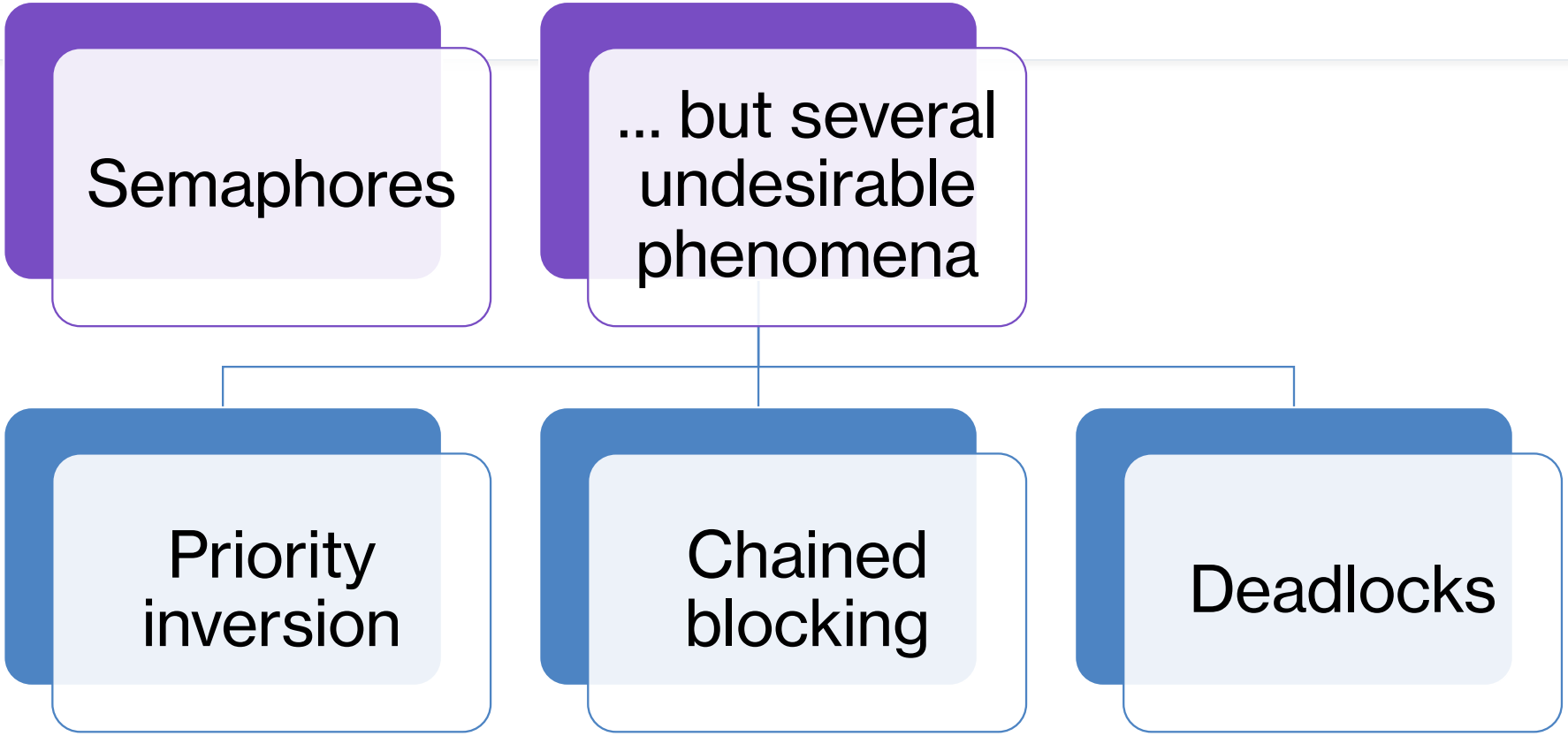
By setting interrupt priority higher than process priority

By reducing portions of code with interrupt disabled

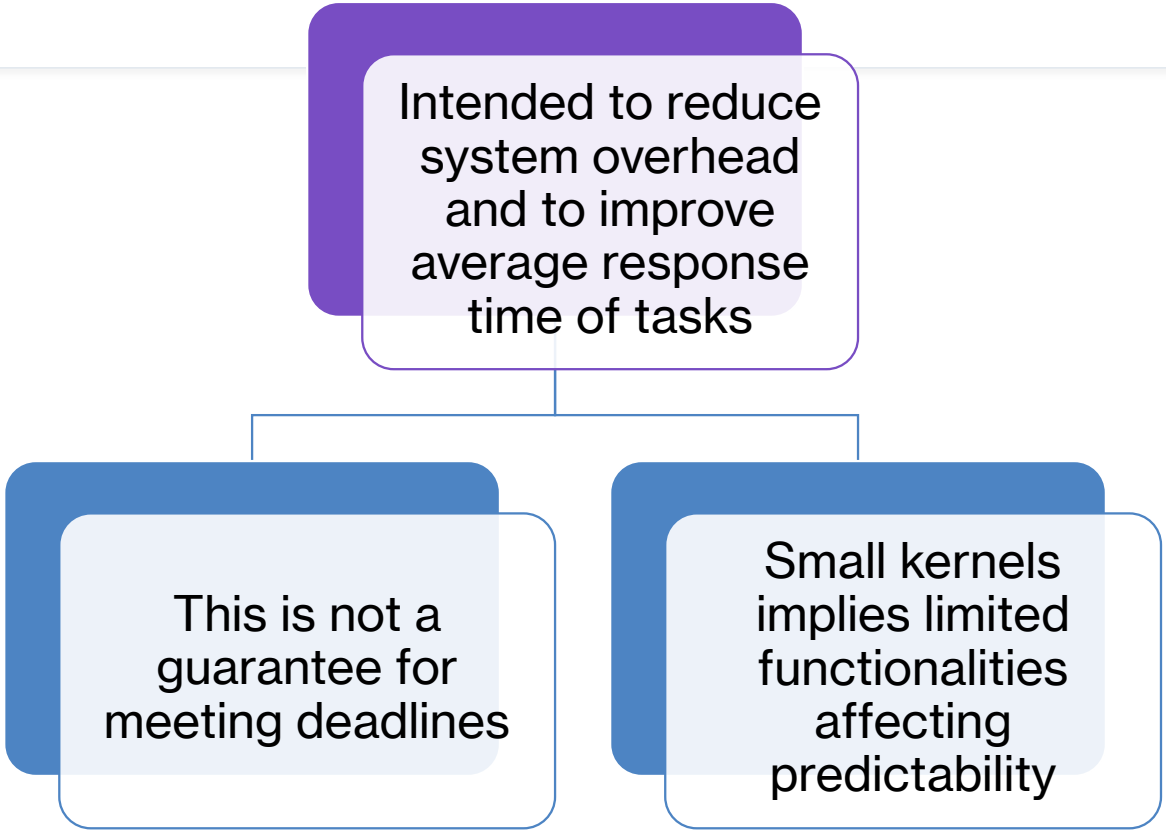
... but this introduces unbounded delays for process execution

- User process interrupted by drivers even if more important
- Number of interfering interrupts per process not bounded in advance

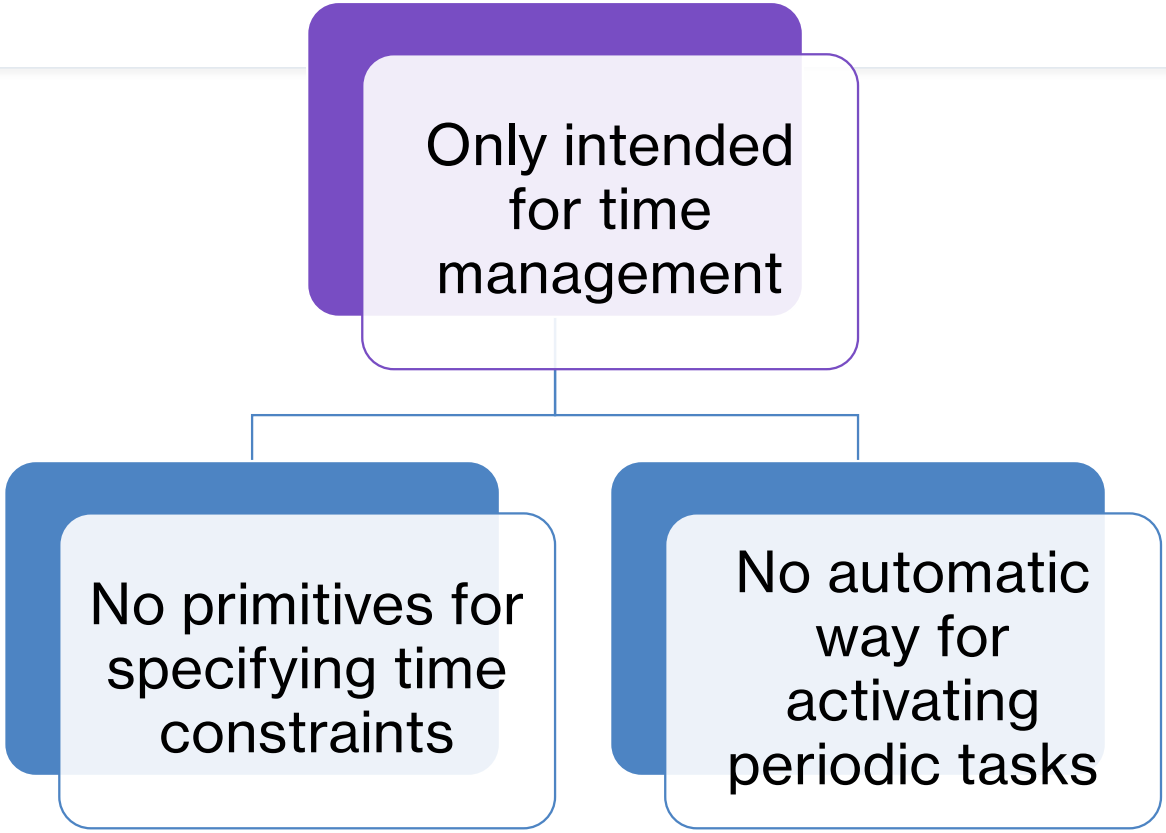
(Un)desirable: Process communication and synchronization



(Un)desirable: Small kernels



(Un)desirable: Internal time reference



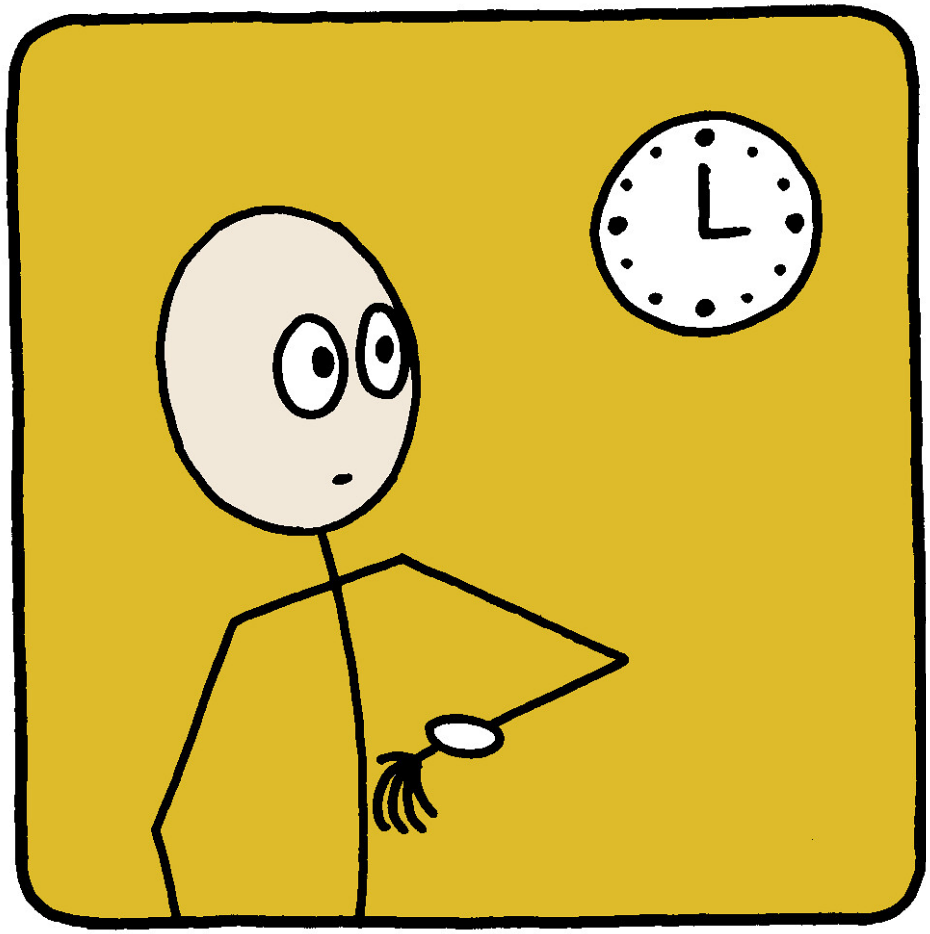
Real time OS: Desirable features

Aim: predictability for peak load

Timeliness	Predictability	Efficiency
Robustness	Fault tolerance	Maintainability

RT desirable: Timeliness

- Results correct in value and time
- Specific kernel mechanism required



RT desirable: Predictability

- Prediction of scheduling decision
- Timing requirements guaranteed in advance before system activation
- Missing deadlines notified in advance

RT desirable: Efficiency

- RT systems embedded in small devices need efficient management of
  - Computational power
  - Memory
  - Energy
  - Weight
  - Space

RT desirable: Robustness

- Reasoning for peak-load conditions
- The system must no collapse due to overloaded scenarios

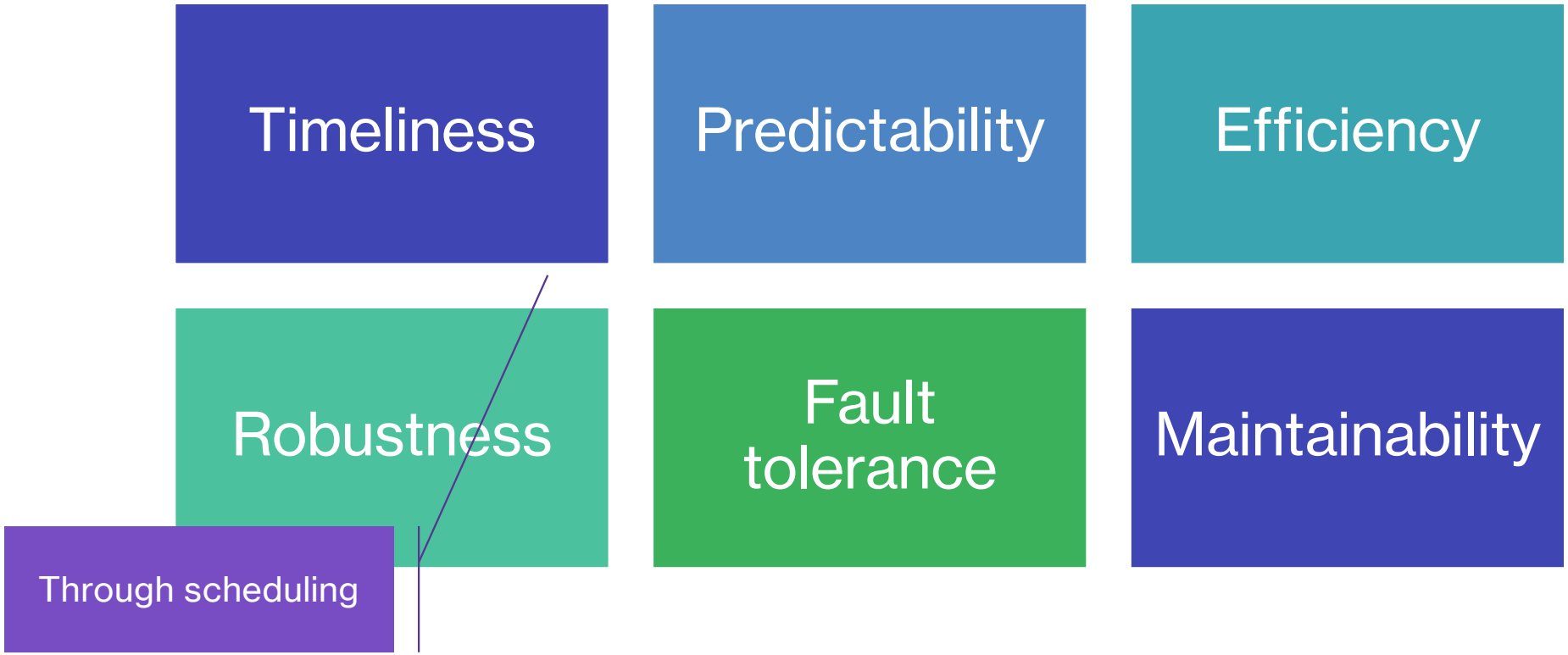
RT desirable: Fault tolerance

- HW/SW failure should not cause the system to crash
- Critical components must be fault tolerant

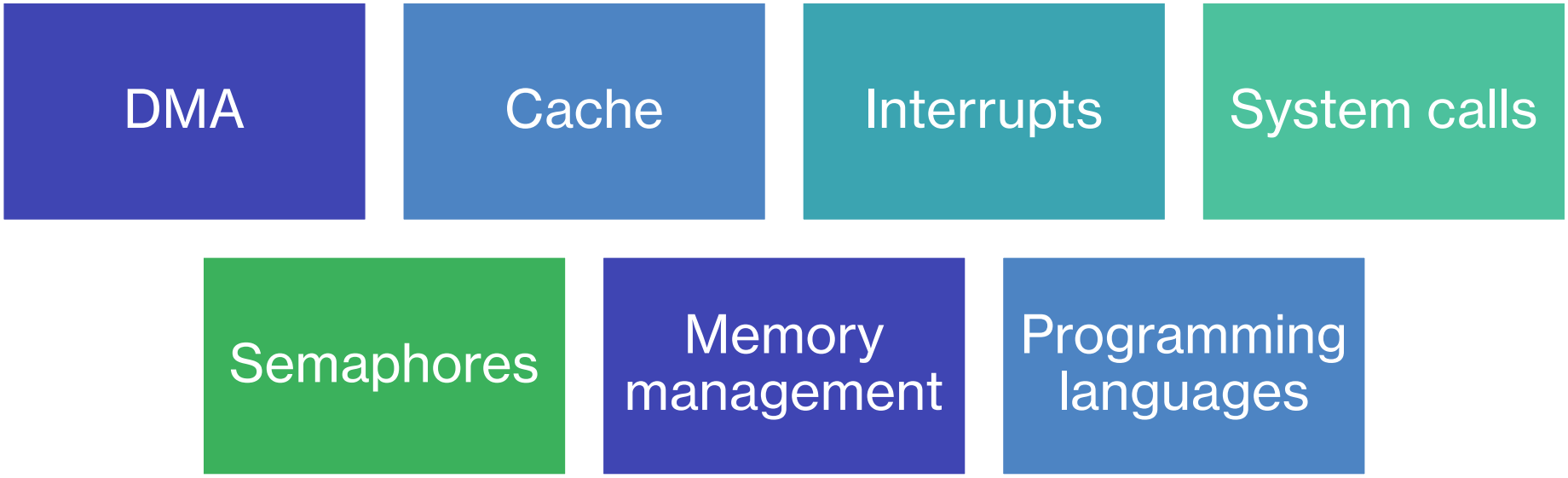
RT desirable: Maintainability

- Modular structure
- Simple to modify

Real time OS: Desirable features



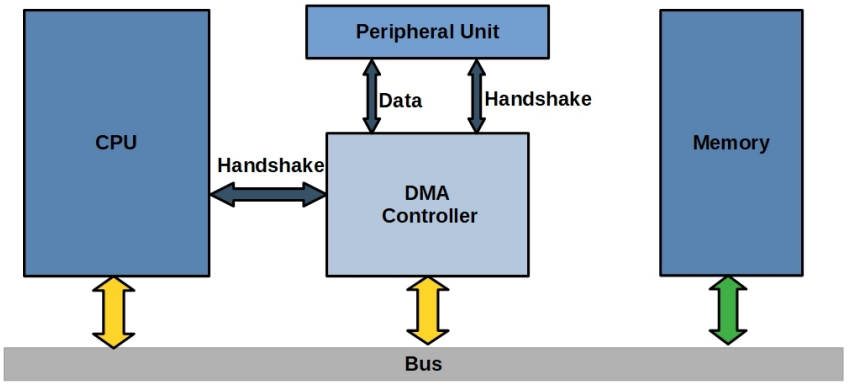
Achieving predictability



These may negatively influence predictability

Achieving predictability: DMA

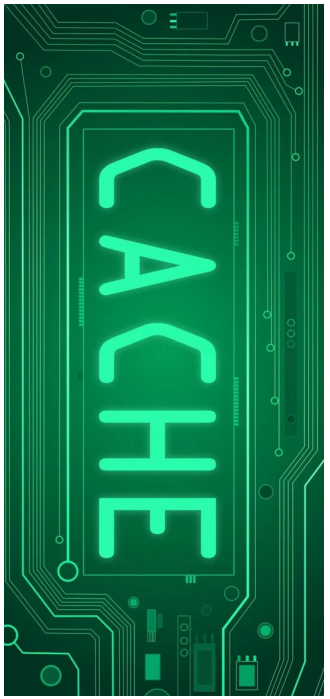
- **Purpose**
  - to transfer data between a device and the main memory relieving CPU
- **Problem**
  - I/O device and CPU share the same bus
  - CPU blocked when DMA is transferring



- **Solutions**
  - Cycle stealing
    - The DMA steals a CPU memory cycle to execute a data transfer
    - The CPU waits until the transfer is completed
    - Source of non-determinism!
  - Time-slice method
    - Each memory cycle is split in two adjacent time slots
      - One for the CPU
      - One for the DMA
    - More costly, but more predictable!

Achieving predictability: Cache

- **Purpose**
  - Motivated by spatial and temporal locality of memory references
    - Originally only for speeding up memory access
    - Now mainly for avoiding conflicts with other devices
- **Problem**
  - Source of non determinism (difference in accessing time)
    - 80% hit vs. 20% miss
    - 90% reading vs. 10% writing
    - Preemption destroys cache
- **Solution**
  - To obtain a high predictability it is better to have processors without cache



Achieving predictability: Interrupts

- **Purpose**
  - I/O interrupts increases reactivity with the environment
  - Hardware details for peripheral management are encapsulated inside drivers
- **Problem**
  - Interrupts have higher priority than (more urgent?) user tasks
  - Execution of interrupt service routine has a cost
    - How may interrupts?
    - How long is the ISR?

Interrupts: Solution 1

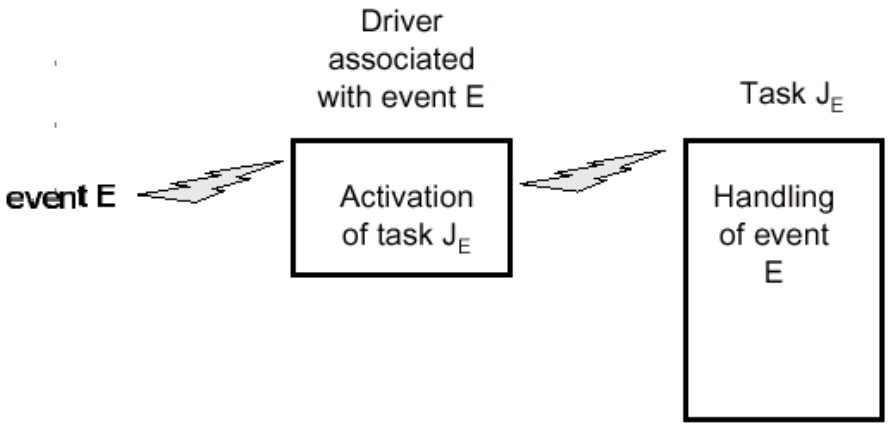
- **Disable all interrupts, but timer interrupts**
- Characteristics
  - All peripheral devices have to be handled by tasks
  - Data transfer by polling
  - Great flexibility, time for data transfers can be estimated precisely
  - No change of kernel needed when adding devices
- Problems
  - Degradation of processor performance (busy waiting)
  - Task must know low level details of the drive

Interrupts: Solution 2

- **Disable all interrupts but timer interrupts,**
- **Handle devices by special, timer-activated kernel routines**
- Advantages
  - Unbounded delays due to interrupt driver eliminated
  - Periodic device routines can be estimated in advance
  - Hardware details encapsulated in dedicated routines
- Problems
  - Degradation of processor performance (still busy waiting - within I/0 routines)
  - More inter-process communication than first solution
  - Kernel has to be modified when adding devices

Interrupts: Solution 3

- **Enable external interrupts**
- **Reduce the drivers to the least possible size**
  - Driver only activates proper task to take care of device
  - The task executes under direct control of OS, just like any other task
  - Control tasks then have higher priority than device task



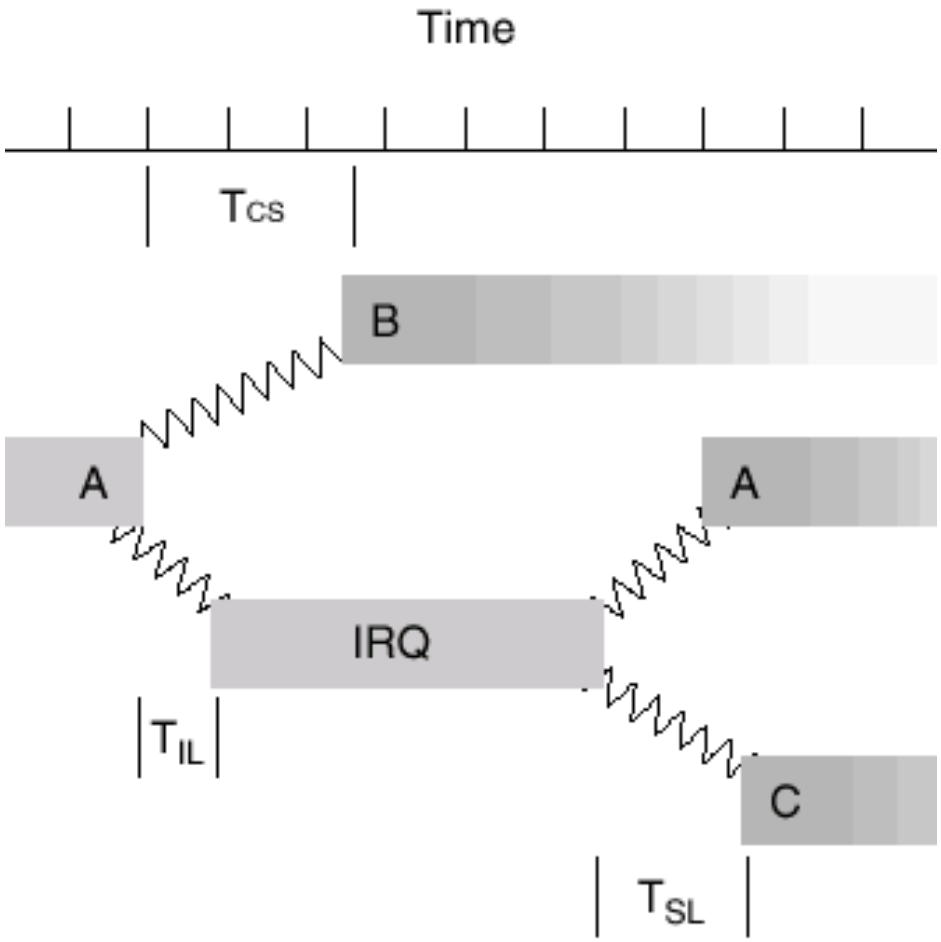
Interrupts: Solution 3

- Advantages
  - Busy waiting eliminated
  - Unbounded delays due to unexpected device handling dramatically reduced (not eliminated!)
  - remaining unbounded overhead may be estimated relatively precisely

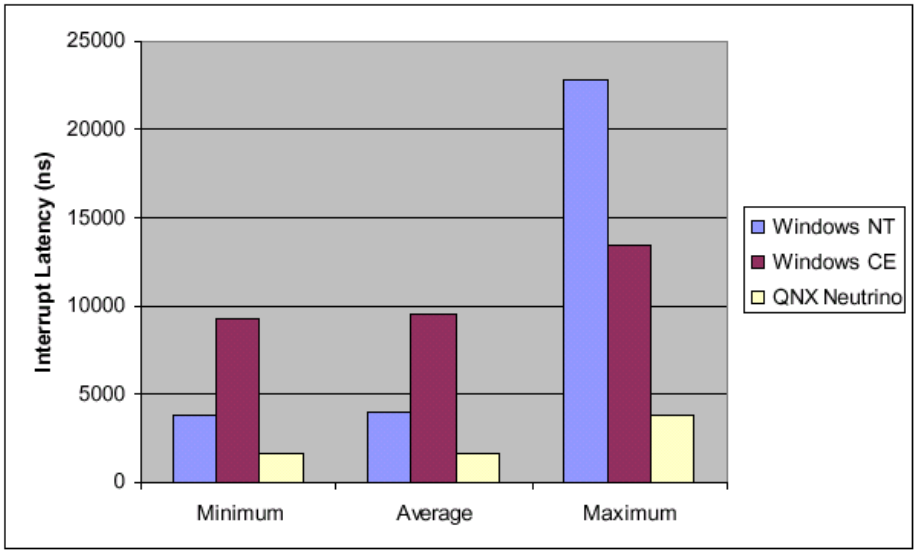
ARTS, HARTIK, SPRING

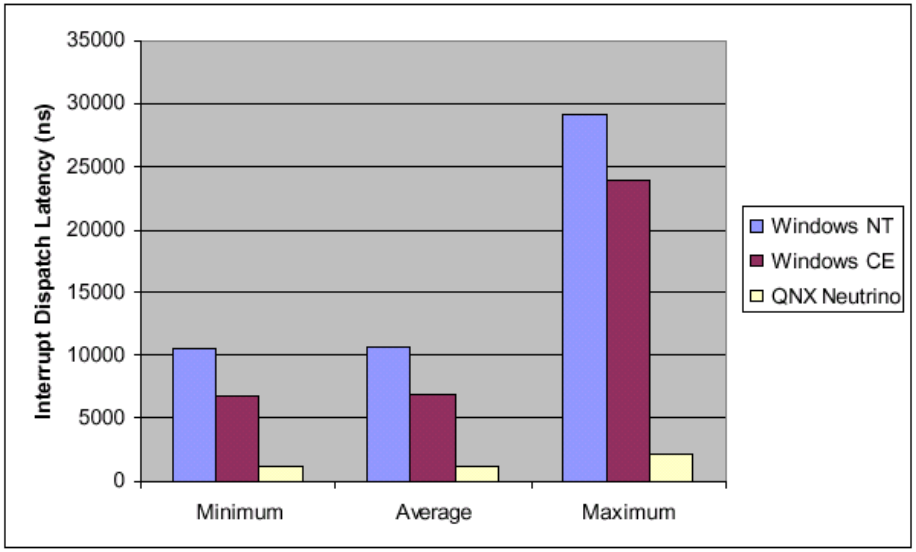
RTOS timing figures

- Interrupt latency (TIL)
  - the time from the start of the physical interrupt to the execution of the first instruction of the interrupt service routine
- Scheduling latency (interrupt dispatch latency) (TSL)
  - the time from the execution of the last instruction of the interrupt handler to the first instruction of the task made ready by that interrupt
- Context-switch time (TCS)
  - the time from the execution of the last instruction of one user-level process to the first instruction of the next user-level process
- Maximum system call time
  - should be predictable & independent of the # of objects in the system



RTOS and interrupts - example





Achieving predictability: System calls

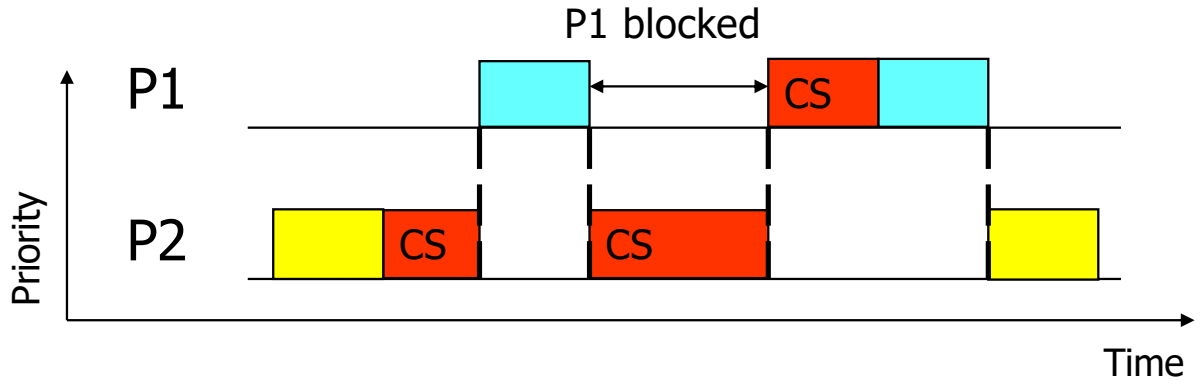
- All system calls have to be characterized by bounded execution time
- Each kernel primitive should be preemptable
- Non-preemtable calls could delay the execution of critical activities

Achieving predictability: Semaphore

- **Purpose**
  - Managing process synchronization
- **Problem**
  - Usual semaphore mechanism not suited for real time applications
  - Priority inversion problem
    - High priority task is blocked by low priority task for unbounded time
- **Solution**
  - Priority Inheritance
  - Priority ceiling

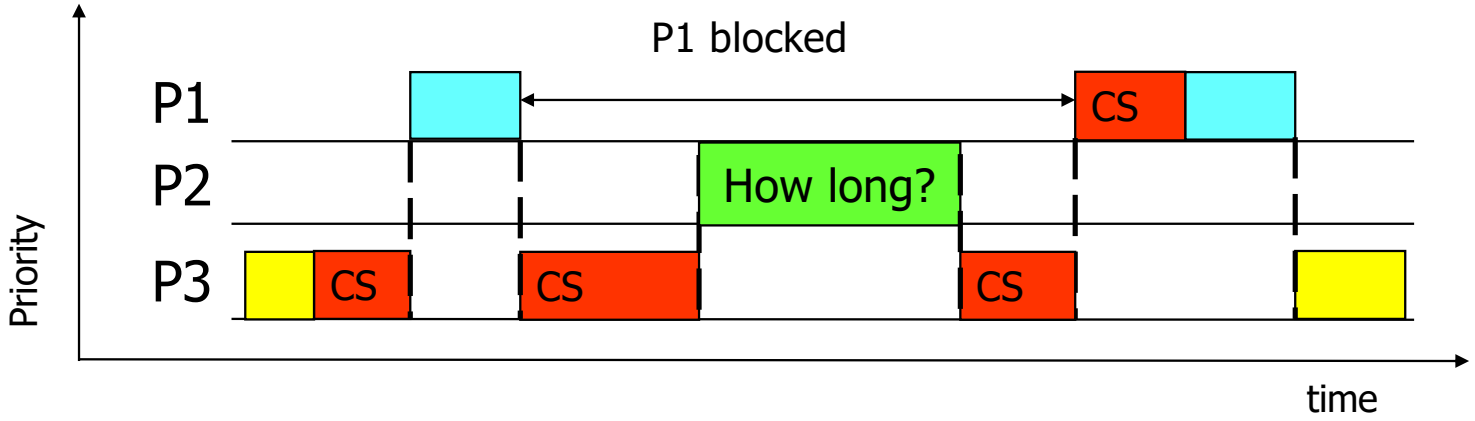
Priority inversion: Acceptable

- P1, P2 share a Critical Section, Priority(P1) > Priority (P2)
- P1 must wait until P2 exits CS even if P(P1) > P(P2)
- Maximum blocking = time needed by P2 to execute its CS
  - Direct consequence of mutual exclusion



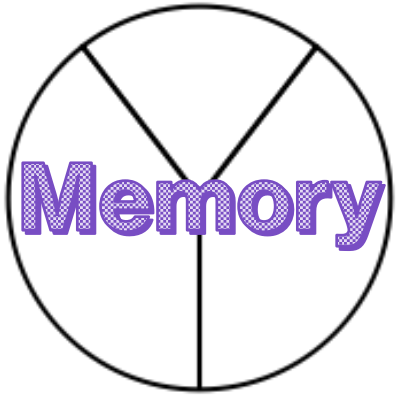
Priority inversion: Unacceptable

- Medium-priority task preempts a lower-priority task that is using a shared resource on which a higher-priority task is pending
- If the higher-priority task is ready to run, but a medium-priority task is currently running instead, a priority inversion is said to occur



Achieving predictability: Memory management

- **Problem**
  - Conventional demand paging causes non-deterministic delays
    - Page fault & page replacement may cause unpredictable delays
    - May use selective page locking to increase determinism
- **Solutions**
  - Memory segmentation
  - Static partitioning if applications require similar amounts of memory
- **Cost to pay**
  - Reduced flexibility in dynamic environment
  - Designer must balance between predictabiliy and flexibility



Achieving predictability: Programming languages

- Current programming languages not expressive enough to prescribe precise timing
  - Need of specific RT languages
- **Desirable features**
  - No dynamic data structures
    - prevent the possibility of correctly predict time needed to create and destroy dynamic structures
  - No recursion
    - impossible estimation of execution time for recursive programs
  - Only time-bound loops
    - to estimate the duration of cycles
- Example of RT programming language (OS support required)
  - Real-Time Concurrent C
  - Real-Time Euclid

within deadline *(d) statement-1*
[else *statement-2*]