Real-Time Systems

Last modified: 02.06.2021

by L.J. Opalski

Overview of Real-Time Systems

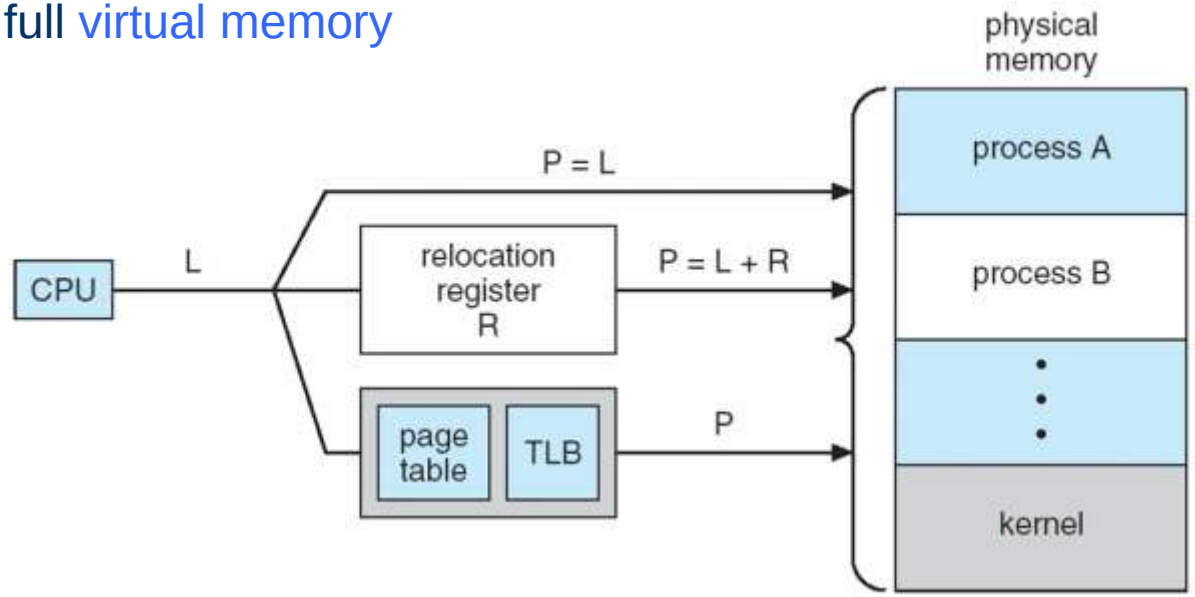
- A real-time (RT) system requires that results be produced within a specified deadline period.
  - An embedded system is a computing device that is part of a larger system (i.e. automobile, airliner)
  - A safety-critical system is a real-time system with catastrophic results in case of failure
- A hard real-time system guarantees that real-time tasks be completed within their required deadlines
- A soft real-time system provides priority of real-time tasks over non real-time tasks

Features of Real-Time Kernels

- Most real-time systems do not provide the features found in a standard desktop system. Reasons include:
  - Real-time systems are typically single-purpose
  - Real-time systems often do not require interfacing with a user
  - Features found in a desktop PC require more substantial hardware that what is typically available in a real-time system
  - Cost-effective features may introduce non-deterministic latency
  - Real-time applications must be able to control the execution sequence of their concurrent process in order to meet externally defined response requirements.

Memory in Real-Time Systems

- Address translation may occur via:
  - (1) Real-addressing mode where programs generate actual addresses
  - (2) Relocation register mode
  - (3) Implementing full virtual memory



- The concept of memory locking is introduced in RT systems to eliminate the indeterminacy introduced by paging and swapping, and to support an upper bound on the time required to access the memory mapped into the address space of a process.

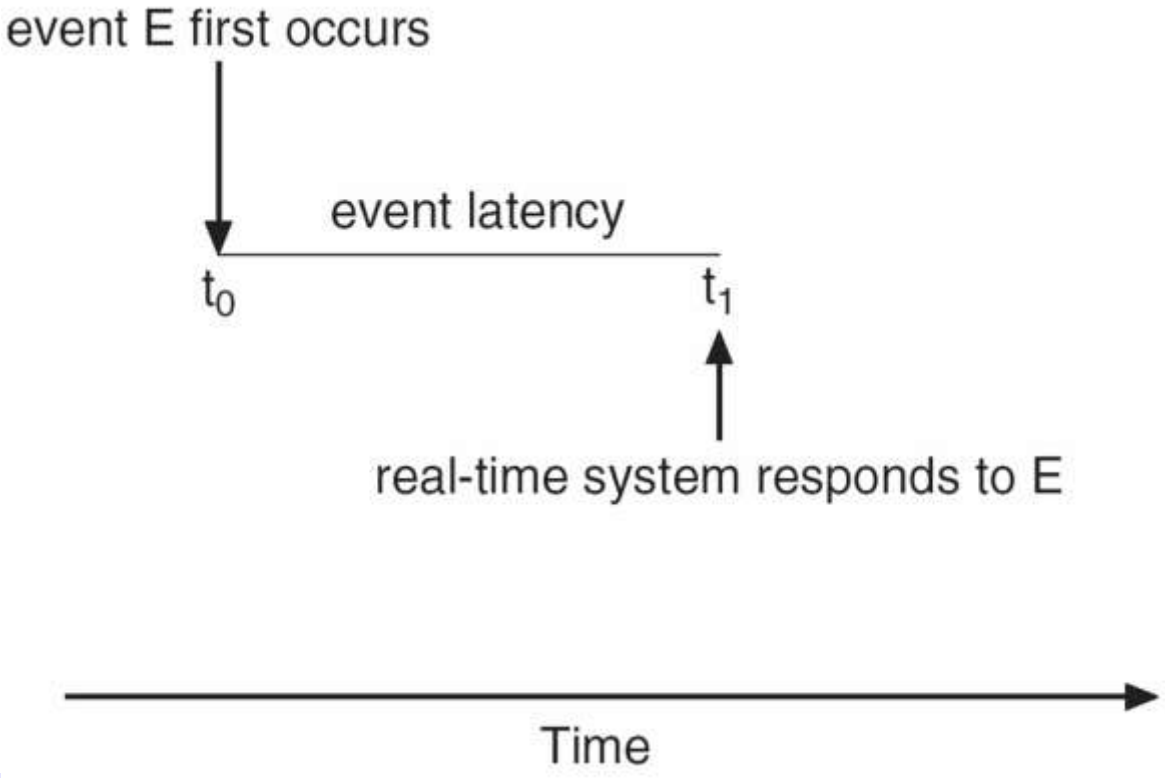
“Must have” for Real-Time Systems

- In general, real-time operating systems must provide:

  (1) **Preemptive**, priority-based, **scheduling**

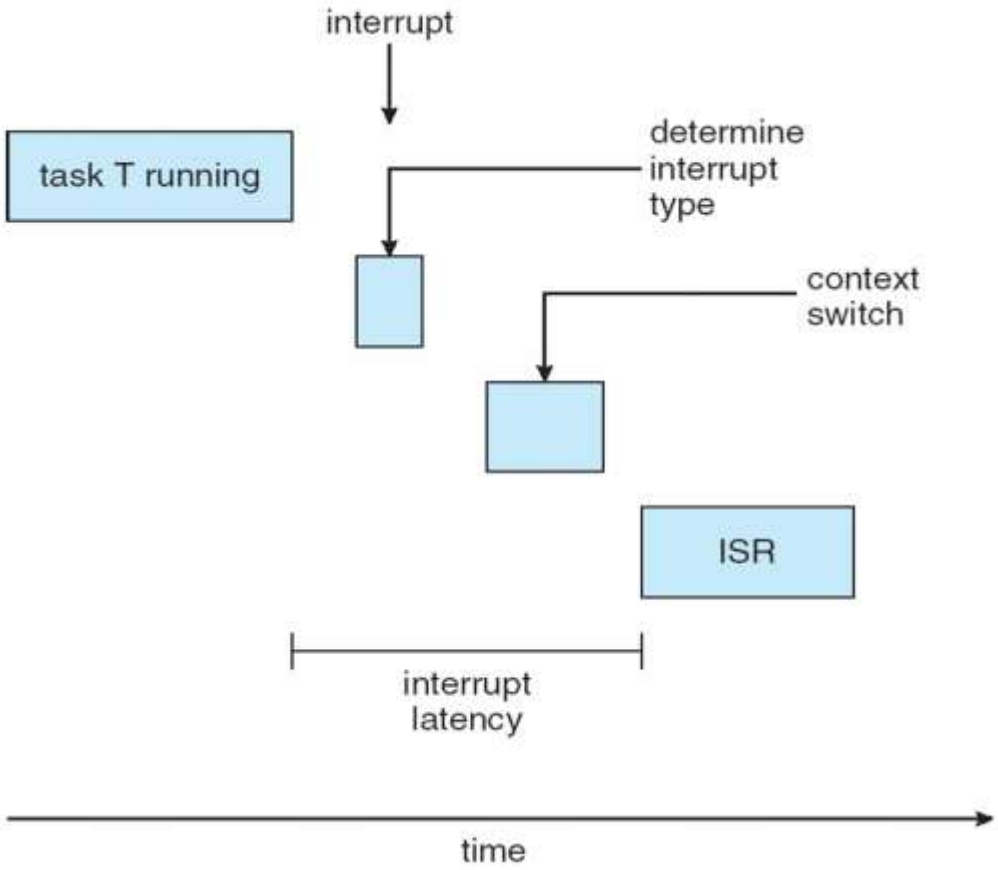
  (2) **Preemptive kernels**

  (3) Minimized **event latency**



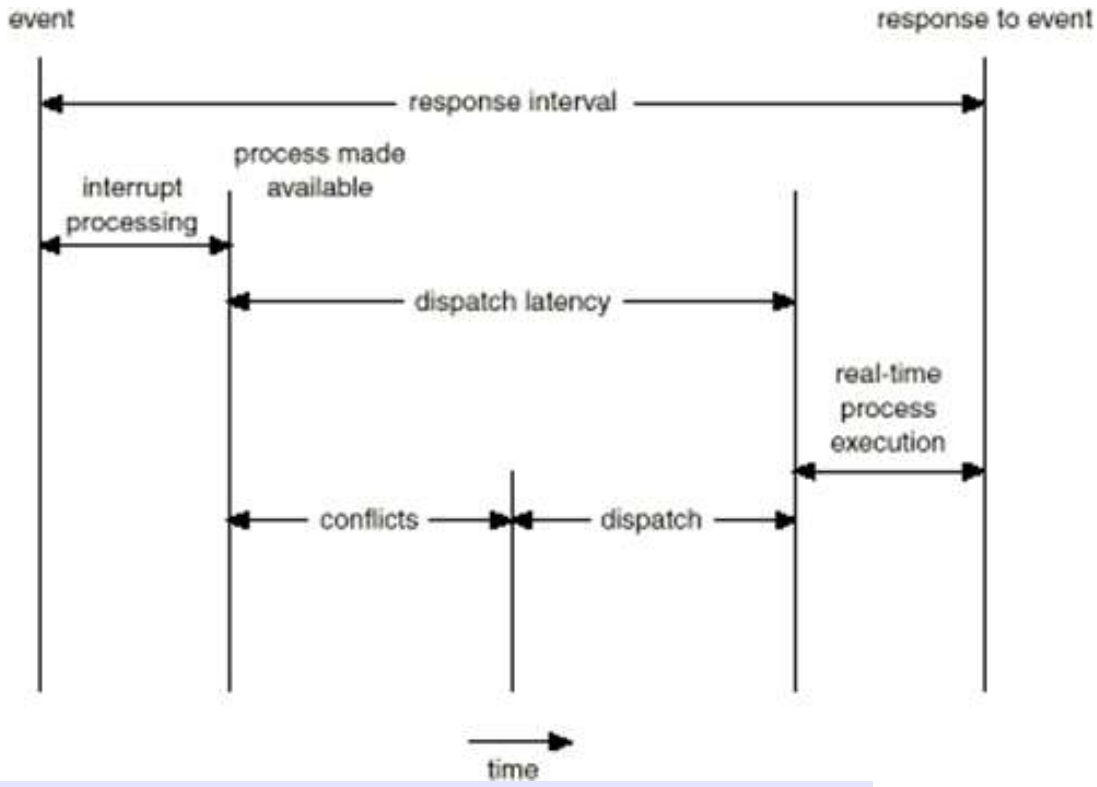
Interrupt Latency

- **Interrupt latency** is the period of time from when an interrupt arrives at the CPU to when it is serviced
- In RT systems interrupt latency must be **bounded** to guarantee the **deterministic behavior** of the system.



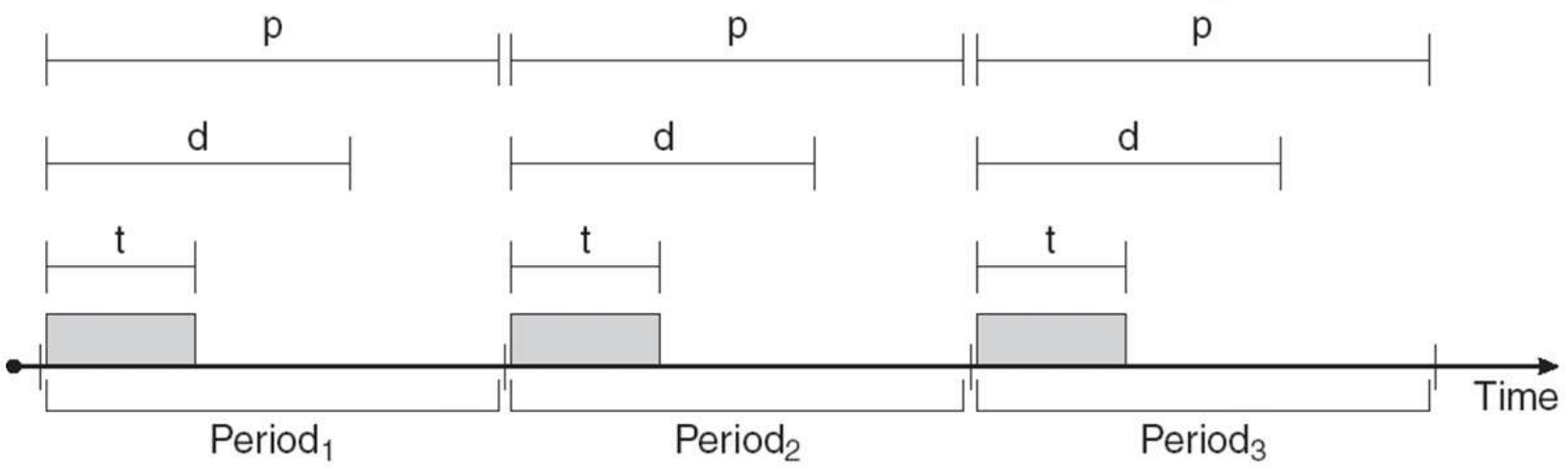
Dispatch Latency

- Dispatch latency is the amount of time required for the scheduler to stop one process and start another
- Dispatch latency can be reduced by:
  - preemption points in long-duration system calls
  - preemptive kernel



Real-Time CPU Scheduling

- Important characteristics of tasks, which RT processes realize:
  - Aperiodic task – has to be started or finished by specified time
  - Periodic task – require the CPU at specified (periodic) intervals
    - ***p*** is the duration of the period
  - ***d*** is the deadline by when the process must be serviced
  - ***t*** is the processing time



- Necessary condition of scheduling feasibility: $\sum_{i=1}^{N} \frac{t_i}{p_i} \leq 1$

Scheduling of tasks when P_2 has a higher priority than P_1

p_1=50, p_2=100 – periods

t_1=20, t_2=35 – processing times

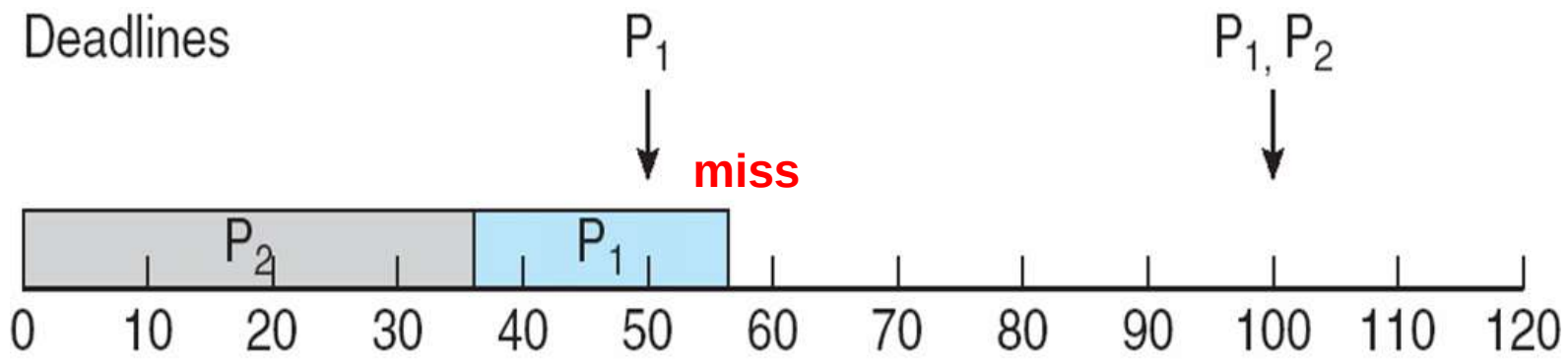
CPU utilization:

P_1: t_1/p_1=20/50=0.4

P_2: t_2/p_2=35/100=0.35

Both: 0.4+0.35=0.75<1 -> it seems that we might meet deadlines

Surprise (?)

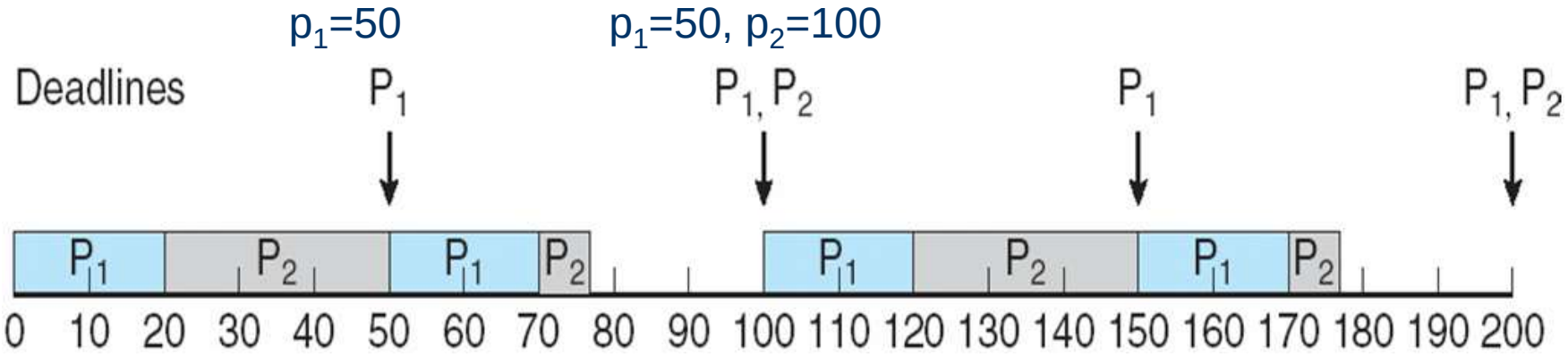


Priority-based CPU allocation **failed** in this example for assumed priorities

Rate Monotonic Scheduling (RMS)

- **Rate Monotonic Scheduling** (RMS) algorithm assumes that a process priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority

- P_1 is assigned a higher priority than P_2.(t_1=20; t_2=35)



- This time the priorities resulted in meeting deadlines. Is the RM scheduling always successful?

Missed Deadlines with Rate Monotonic Scheduling

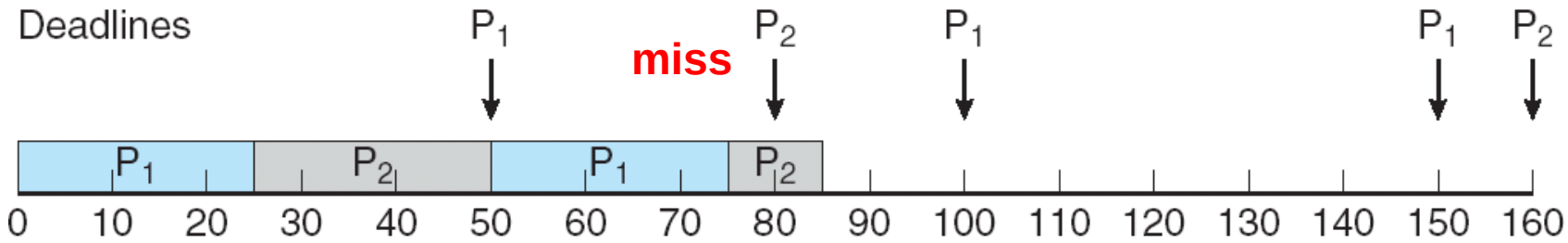
- **Rate-monotonic scheduling** is considered **optimal** in that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities.
- Example.

p_1=50, p_2=80 – periods; t_1=25, t_2=35 – processing times

CPU utilization:

P_1: t_1/p_1=25/50=0.5; P_2: t_2/p_2=35/80=0.44

Necessary condition: 0.5+0.44=0.94<1 is satisfied



Note:

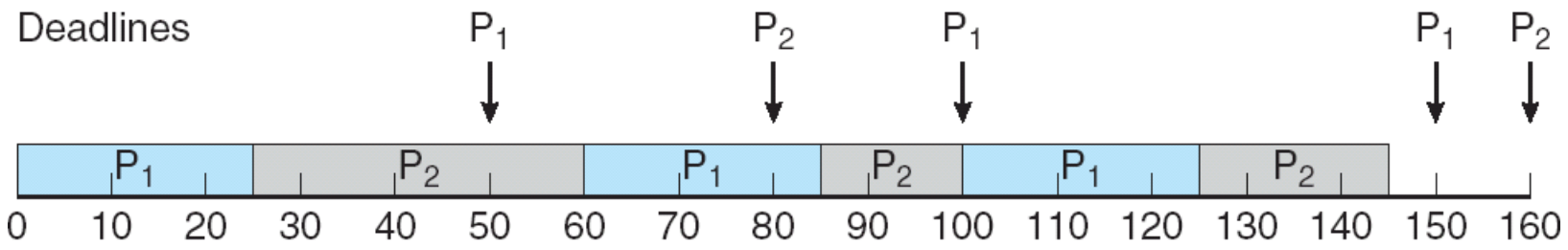
- The worst-case CPU utilization for scheduling N processes is bounded by $N(2^{1/N}-1)$. For N=2 it makes 0.83<0.94 - hence the above failure. $\lim_{N\to\infty} N(2^{1/N}-1) = \ln(2) \approx 0.6931$
- RMS (a priority scheduler) can be affected by **priority inversion**

Earliest Deadline First (EDF) Scheduling

- **Earliest Deadline First** (EDF) Scheduling assigns priorities (dynamically) according to current **deadlines**:

  the earlier the deadline, the higher the priority;

  the later the deadline, the lower the priority



- Unlike the rate-monotonic algorithm EDF scheduling does not require that processes be periodic, nor must a process require a constant CPU time per burst.
- EDF only requires that a process announces its deadline to the scheduler when it becomes runnable.

POSIX Real-Time Scheduling

The POSIX.1b standard

- Each thread is controlled by an associated **scheduling policy** and **priority**.
- The scheduling **contention scope** of a thread defines the set of threads with which the thread competes for use of the processing resources. System or process scope.
- There is, conceptually, one thread list of **runnable threads** for each **priority**. A **scheduling policy** defines allowable operations on the set of lists (including moving threads between and within lists). Associated with each policy is a **priority range**.
- The thread, which is at the head of **the highest priority** non-empty thread list becomes a running thread, regardless of its associated policy.

POSIX Real-Time Thread Scheduling

Four thread scheduling policies are required by POSIX 1003.1 Standard:

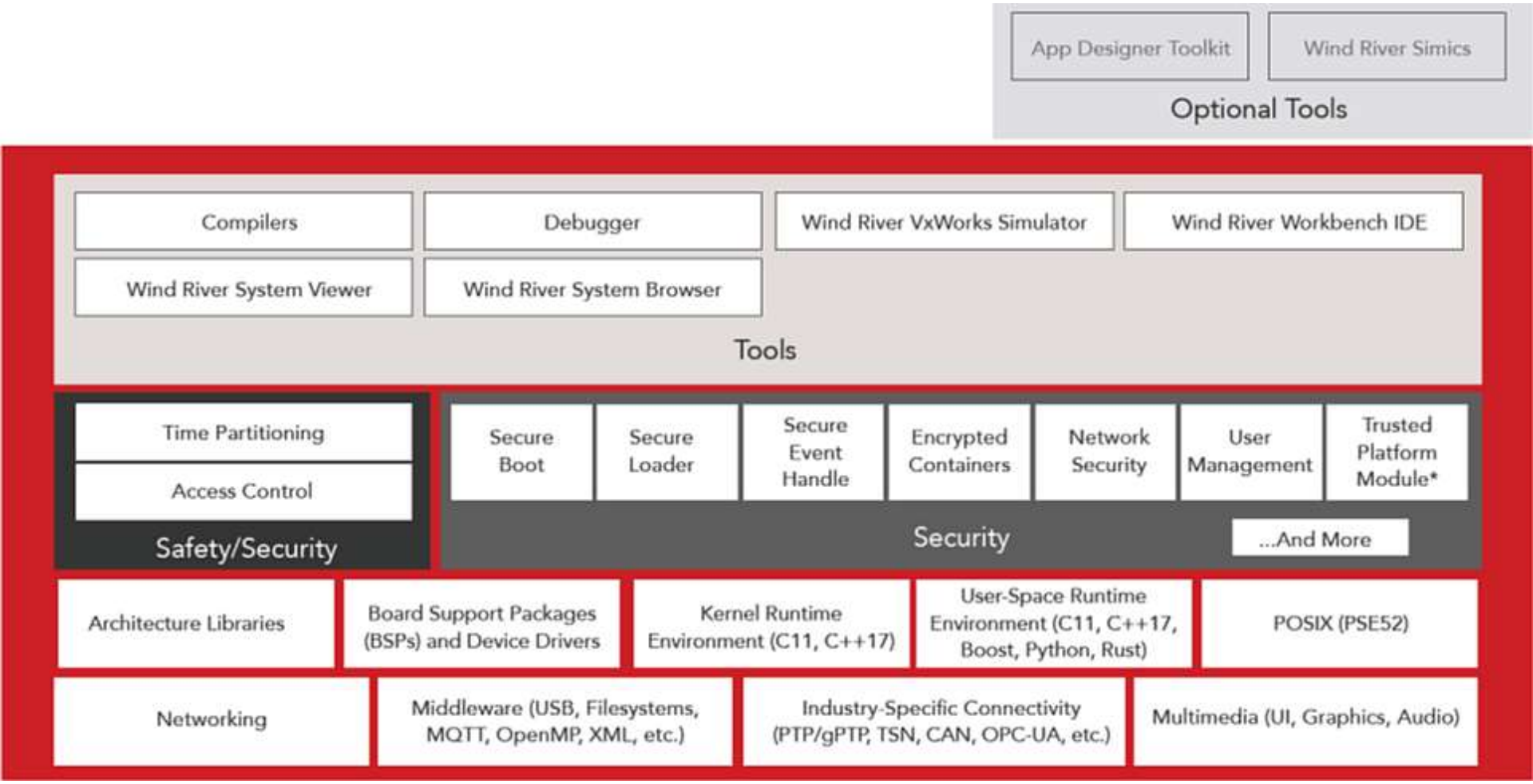
- Scheduling classes for real-time threads:
  1. **SCHED_FIFO** - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
  2. **SCHED_RR** - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
  3. **SCHED_SPORADIC** – high priority for at most specified time, if the time is exceeded – the priority drops; the execution capacity is replenished after some specified time.
- Another scheduling policy
  4. **SCHED_OTHER** – can behave like FIFO or RR policy, but it is not a RT policy, so all RT threads preempt threads with SCHED_OTHER policy. Threads subject to this policy can have dynamic priorities (e.g. to implement ageing).

Linux: additional classes of no-RT processes: SCHED_BATCH, SCHED_IDLE and of RT processes: **SCHED_DEADLINE** (Global Earliest Deadline First scheduling) – instead of SCHED_SPORADIC. See man sched(7).

POSIX RT profiles

- POSIX 1003.1 Standard
  - Allows writing portable real-time applications
  - Very large: inappropriate for embedded real-time systems
- POSIX 1003.13 – defines four real-time system subsets (profiles)
  - Minimal (PSE51): for small embedded systems. Platform: Small embedded system, with no MMU, no disk, no terminal. Model: controller of a “Toaster”
  - Controller (PSE52): for industrial controllers. Platform: Special purpose controller, with no MMU, but with a disk containing a simplified file system. Model: industrial robot controller
  - Dedicated (PSE53): for large embedded systems. Platform: Large embedded system with file system on disk, with an MMU; software is complex and requires memory protection and network communications. Models: avionics controller, cellular phone cell node.
  - Multi-Purpose (PSE54): for large general-purpose systems with RT requirements. Platform: Large real-time system with all the features, including a development environment, network communications, file system on disk, terminal and GUI, etc. Model: workstation with RT requirements: e.g. for air traffic control systems, or for telemetry systems for Formula One racing cars.

Example RT system: VxWorks



https://www.windriver.com/products/vxworks

Architectures: ARM (64b, v7), Intel (IA), Power PC, RISC-V,….
Multiprocessor systems: AMP i SMP; multi-OS (Type 1 hypervisor)
Open source Raspberry Pi Board Support Package

Wind Microkernel

- The Wind microkernel features:
  - small size (20kB)
  - multitasking with processes and threads
  - preemptive and non-preemptive round-robin scheduling
  - manages interrupts (with bounded interrupt and dispatch latency times)
  - shared memory and message passing interprocess communication support
- There is a support to two levels of virtual memory
  - Control of the page cache on a per-page basis, eg. allowing for non-cacheable pages
  - Optional virtual memory component VxVMI along with hardware MMU allow selected data areas to be marked as private (accessible from a specific task)
- POSIX PSE52 interface support.
- Many certificates available, handy for development of aerospace, automobile, medical and other safety-critical systems

RTLinux

- RTLinux is a hard real-time microkernel
- In RTLinux the standard Linux kernel runs as a fully preemptive task in a small operating system
- The real-time (RT) kernel handles all interrupts – directing each interrupt to a handler in the standard kernel or to an interrupt handler in the RT kernel.
- RTLinux prevents the standard Linux kernel from ever disabling interrupts, thus ensuring that it cannot add latency to the RT system
- RTLinux provides additional scheduling policies, including rate-monotonic scheduling (RMS) and earliest-deadline-first scheduling (EDF).