DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Real-Time Linux

- Pre-emptible Linux Kernel - from 2.4 to 2.6
- Synchronization in the Kernel: single and multiple processor
- Spinlocks
- Kernel Threads
- Realtime extensions (Linux MRG)

Pre-emptible Kernel

- Up to Linux 2.4, the kernel was not pre-emptible
  - This means that a kernel operation had to finish before giving the chance for another task to resume, i.e. before calling schedule()
  - Interrupts of course occur, including those the may make a high priority task newly ready, but the call to schedule() is deferred until the termination of the pending kernel operation

- As a consequence, a high priority task that becomes again eligible for execution may be deferred for an undefined amount of time
  - Some kernel operation may require hundreds of ms

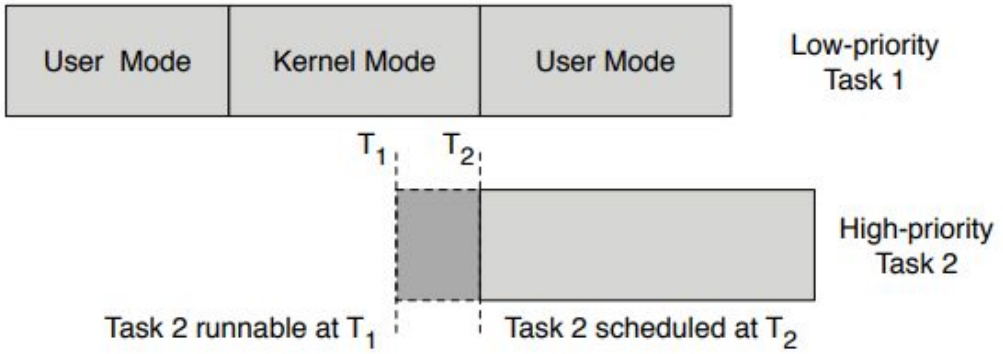
- Linux 2.4 could therefore not be considered a real-time OS

- From Linux 2.6 the kernel becomes interruptible, i.e. when an event occurs that makes a high priority task ready again, control is returned to the latter even before terminating the kernel operation that shall be resumed later.

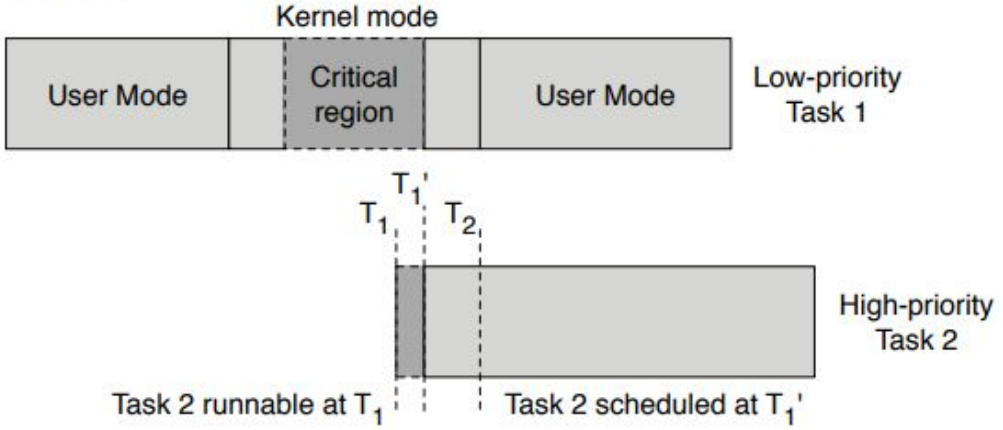
- In this case it is possible to avoid in most cases unbounded wait for high priority tasks, making Linux 2.6 a soft real-time system

- This comes not for free, and it is necessary to protect critical regions inside the kernel to avoid data structure corruption

Pre-emptible Kernel - cont



Preemptive Kernel



kernel threads

- When invoked via soft interrupt (the only way to move programmatically to kernel mode) kernel code is running in ***interrupt context***
- As such a kernel section cannot be preempted, only suspended by a higher priority interrupt (i.e. an HW interrupt)
- Therefore it is necessary to let most kernel operations be carried out by Kernel Threads
- Kernel Threads do not live within the context of any process, but they share the same kernel virtual address
- In this way, using the same scheduler mecanism it is possible to suspend a kernel thread when another thread/process with higher priority becomes ready, and eventually to resume it.
- Kernel threads are also important in reducing the amount of code executed at interrupt level in drivers, moving actions that are not essential (such as writing an immediate answer into the device registers) to a kernel thread and therefore reducing again the dead time in system responsiveness
    - An interrupt routine can be interrupted only by a higher priority HW interrupt

Protecting critical sections in Kernel threads

- It ***cannot*** rely on the scheduler mechanism because critical sections may be required also by interrupt routines and therefore a lower level mechanism is required
- On a single processor system an immediate way to ensure mutual exclusion is to disable HW interrupts when the critical section is accessed
- For multiprocessor (multicore) systems this is no more enough, as the critical section can be accessed by code running on a different processor
- In this case ***spinlocks*** are used, based on shared memory, looping until the critical section is accessible
  - As interrupts are disabled, only a separate processor can be competing for the critical section
- Spinlocks require some sort of atomic Test&Set HW mechanism, that is provided by most, if not all, platforms

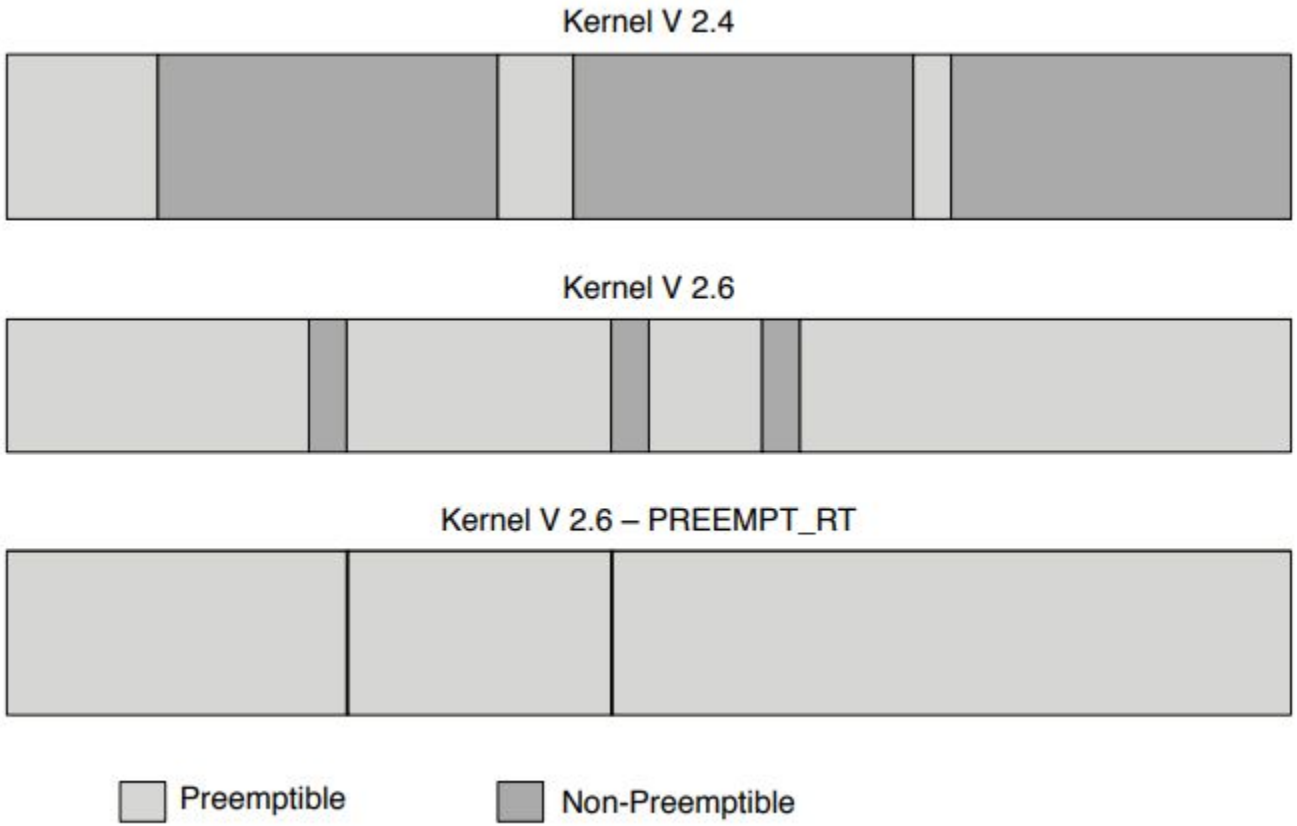
PREEMPT_RT patch (Linux MRG)l

- In pre-emptible kernel, system latency is mainly due to
  - critical sections in the kernel code that are protected by spinlocks; preemption is in fact disabled as long as a single spinlock is active;
  - Interrupt Service Routines (ISRs) running outside any task context, and therefore potentially introducing delays in the system reaction to events because the scheduler cannot be invoked until no pending interrupts are present.

- The PREEMPT RT Linux patch represents one step further toward hard realtime Linux performance. PREEMPT RT provides the following features:
  - Preemptible critical sections
  - Priority inheritance for in-kernel spinlocks and semaphores;
  - Preemptible interrupt handlers.

- Real-time performance of the (free) Linux MRG are now very close to those of specialized (expensive) real-time systems such as vxWorks (that has been used in the NASA Mars missions)

PREEMPT_RT patch (Linux MRG)l



Pre-emptible critical sections

- In PREEMPT RT, a new locking mechanism is made available. It is no more implemented as the cyclic atomic test and set but is internally implemented by a semaphore called rt-semaphore.
  - This semaphore is implemented in a very efficient way if the underlying architecture supports atomic compare and exchange;
  - otherwise, an internal spinlock mechanism is used. Therefore, the task entering a critical section can now be interrupted

- The integrity of the critical section section is still ensured since, if a new task tries to acquire the same lock, it will be put on wait.
  - spinlocks are still required for ISR because they are outside the kernel thread context

- The impact of this different, semaphore-based mechanism is not trivial since it may lead to deadlocks using spinlocks
  - In fact, in the case the task owning a spinlock is put on wait, there would be no chance for other tasks to gain processor usage

Priority inheritance for in kernel spinlocks and semaphores

- Priority inversion may lead to potentially unbounded delays in task response even when task priorities (fixed) are accurately planned

- Observe that for spinlocks priority inversion cannot occur because the task taking ownership disables interrupts and therefore implicitly becomes the highest priority task in the system (interrupts disabled => the task cannot be preempted by higher priority tasks).

- For rt-semaphores PREEMPT_RT implements priority inheritance
  - The kernel knows the priority of the threads owning the semaphore and of the requesting thread and in case can therefore increment the priority of the current owner, lowering it at its original value when the semaphore is released

Pre-emptible Interrupt handlers

- Well-written device driver avoids defining lengthy ISR code.
  - Rather, the ISR code should be made as short as possible,delegating the rest of the work to kernel threads.
  -
- PREEMPT RT takes one step further and forces almost all interrupt handlers to run in task context unless marked SA NODELAY to cause it to run in interrupt context.
  - By default, only a very limited set of interrupts is marked as SA NODELAY
  - among them, only the timer interrupt is normally used.

- In this way, it is possible to define the priority associated with every interrupt source in order to guarantee a faster response to important events
  - This would not have been possible if ISR were handled at interrupt level, even when using Interrupt Priority levels (normally limited to 8)