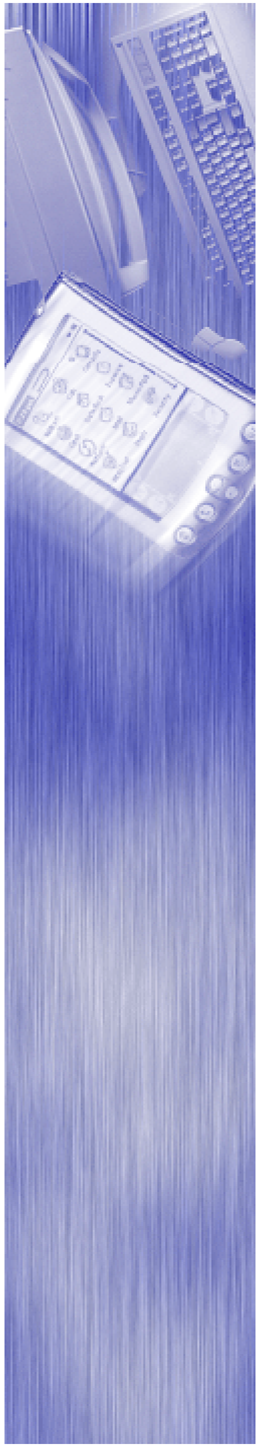


CS632/SEP564: Embedded Operating Systems (Fall 2008)

Real-Time Support

KAIST

Introduction (1)

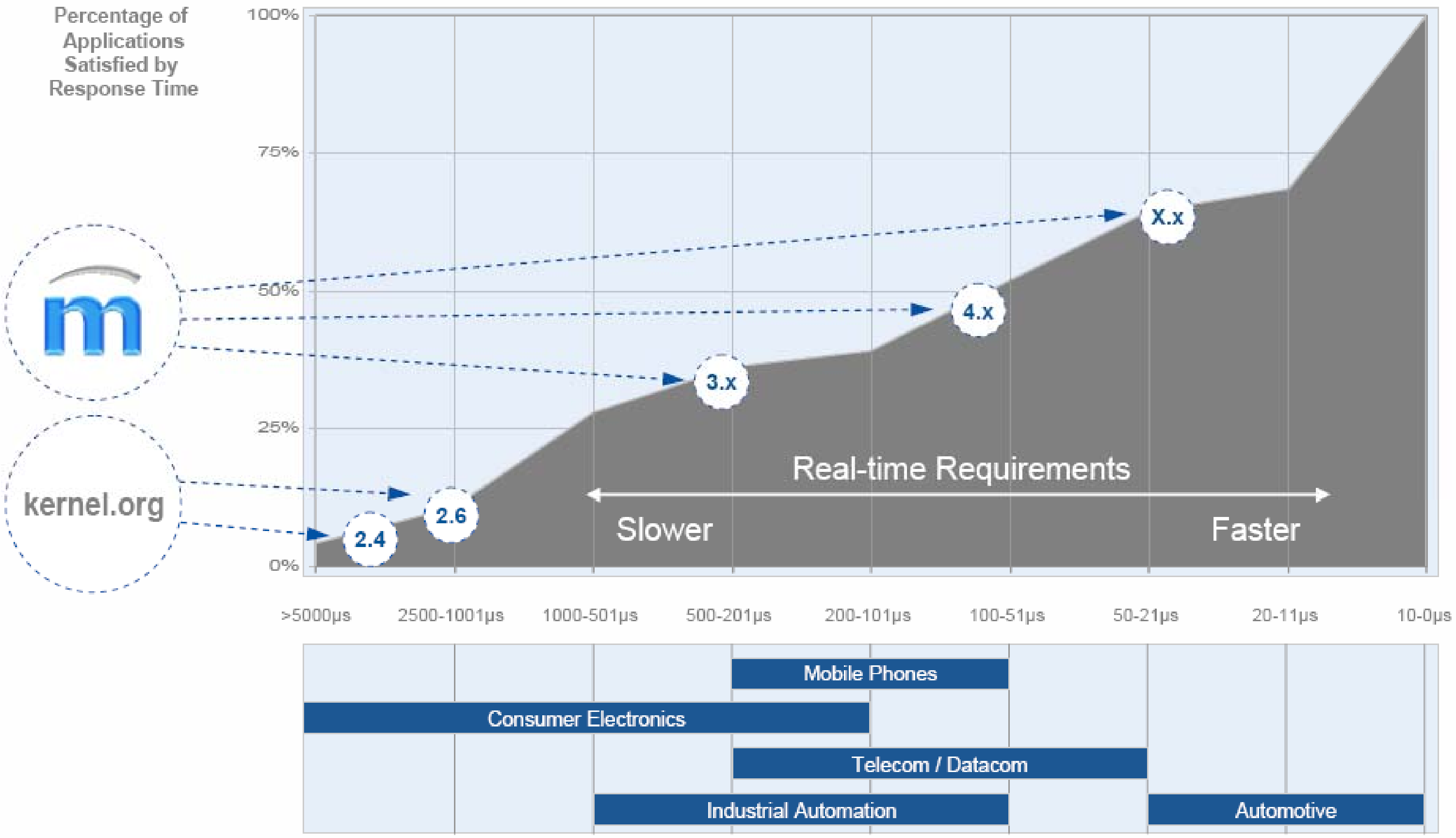
- **Real time applications**
  - A typical real time application spends most of its time waiting for external events, but:
    - As soon as the event fires, the system must be ready to resume the real time application
      - » Response time from 100us to milliseconds
    - The real time application must have all the resources required to complete its task.
      - » Spin locks?
  - Other non-critical processes may be running at the same time.
    - A time-sharing system must reach a compromise between real time and non-real time applications.

Introduction (2)

- **Some examples**
  - Space shuttle avionics control software
    - – Early versions were ~1 MIPS IBM “space qualified” systems
    - – Iterated execution loop
      - » 24x/second, approximately 41ms/cycle
  - Cell phone radio protocol stack for GSM support
    - – Requires ~300 microsecond worst-case response
  - MD-11 flight control computers
    - – Two Motorola 68020’s, one Intel 80386, and one Honeywell SDP-185 processors (all ~1-3 MIPS)
    - – Iteration rates similar to space shuttle avionics

Introduction (3)

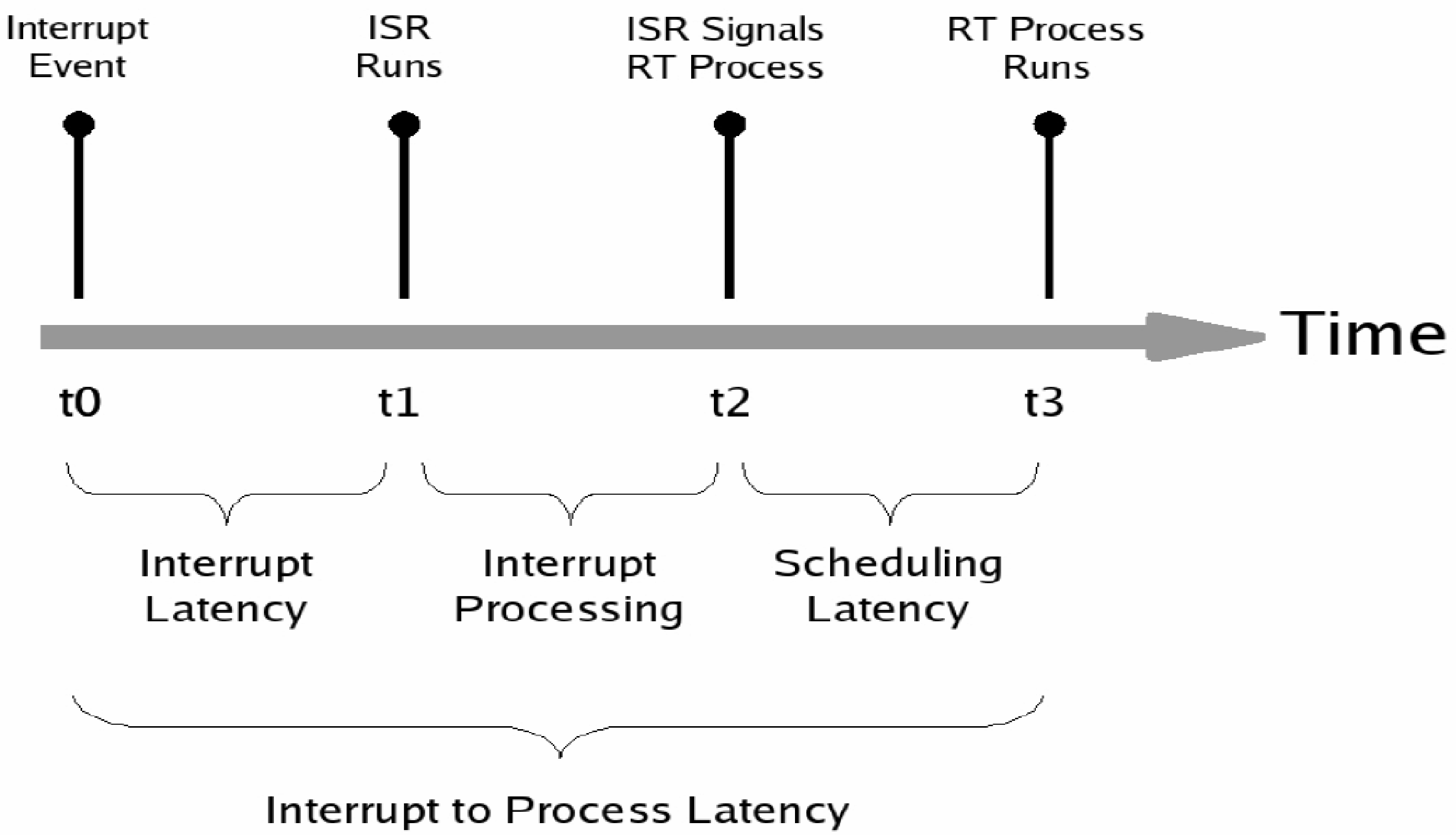
- **The real-time difference**



Sources: *The Embedded Software Strategic Market Intelligence Program*, VDC, July 2005 & *RTOSes Balance Performance with Ease of Use*, COTS Journal, November 2004

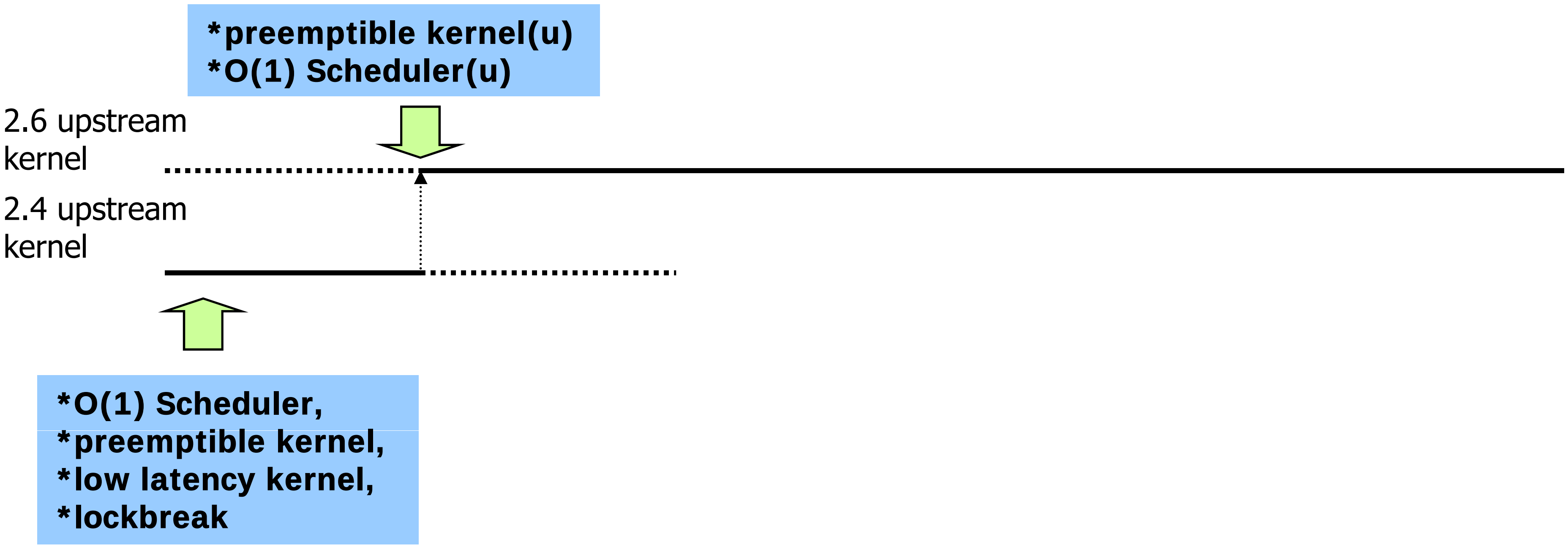
Introduction (4)

- **Real-time control flow**



Source: Montavista

Real-Time Support in 2.4



O(1) Scheduler

- **Linux 2.4 scheduler**
  - A single runqueue
  - A runqueue has both RT tasks and normal tasks.
  - Non-deterministic
    - At the end of each epoch, scheduling is delayed unpredictably depending on the number of tasks.
  - Not suitable for real-time systems

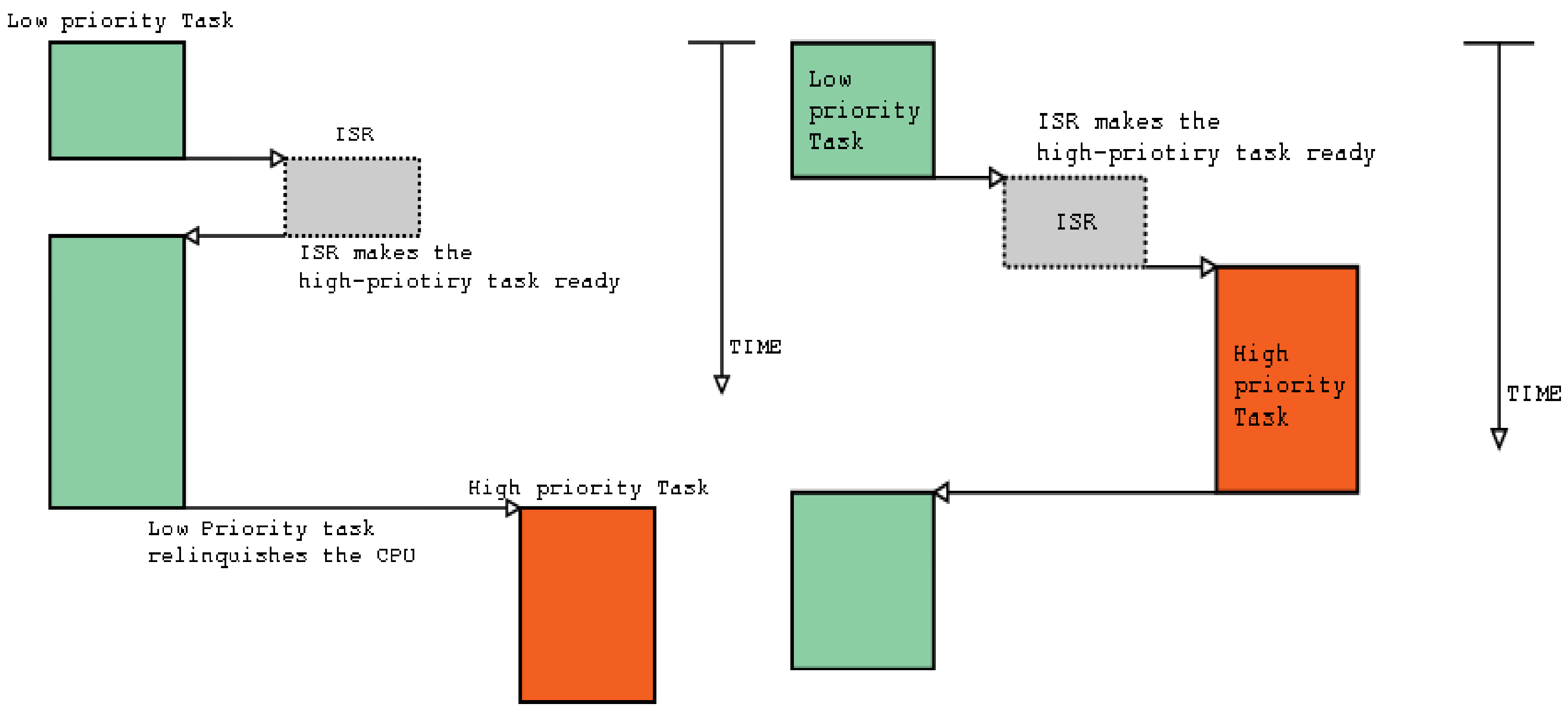
  - O(1) scheduler
    - Proposed in Linux kernel 2.5
    - Adopted officially in Linux kernel 2.6

Preemptible Kernel (1)

- **Overview**
  - Proposed in Linux kernel 2.4.x by Robert Love
  - Officially adopted in Linux kernel 2.5.4
  - Kernel compile option is provided in Linux kernel 2.6.x
    - –DCONFIG_PREEMPT
  - Implemented using SMP locking mechanism
    - Preemptible kernel in a single CPU
      ≈ Non-preemptible kernel in SMPs
    - Minimal kernel modifications
  - Results
    - Improve system responsiveness
    - Decrease system throughput

Preemptible Kernel (2)

- **Non-preemptible vs. preemptible kernel**



Non-preemptible kernel **Preemptible kernel**

Preemptible Kernel (3)

- **Implementation**
  - Kernel is preemptible unless it is in the preemption locked region.
  - If a spin lock is held, the kernel is not preemptible.
    - Hence, the critical section protected by a spin lock is the preemption locked region.

```
spin_lock();
/* preemption locked region */
spin_unlock();
```

  - In reality, some situations do not require a spin lock, but do need kernel preemption disabled.
    - e.g., when per-CPU data or CPU states are accessed

Preemptible Kernel (4)

▪ Preemption count

- Stores the number of held locks and preempt_disable() calls.
- If the number is zero, the kernel is preemptive.
- Stored in the current thread's thread_info structure
  - current_thread_info()->preempt_count

```
                                                                  <linux/preempt.h>
#define preempt_disable()                do { inc_preempt_count(); barrier(); } while (0)
#define preempt_enable_no_resched()  do { barrier(); dec_preempt_count(); } while (0)
#define preempt_check_resched()         \
                    do { if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \
                              preempt_schedule(); } while (0)
#define preempt_enable()                 do { preempt_enable_no_resched(); barrier(); \
                                              preempt_check_resched(); } while (0)
```

Preemptible Kernel (5)

- **Spin lock implementations**

```
                                                    <linux/spinlock.h>
#define spin_lock(lock)            _spin_lock(lock)
#define spin_unlock(lock)          _spin_unlock(lock)
```

```
                                                    <kernel/spinlock.c>
void __lockfunc _spin_lock(spinlock_t *lock)
{
        preempt_disable();
        _raw_spin_lock(lock);
}
void __lockfunc _spin_unlock(spinlock_t *lock)
{
        _raw_spin_unlock(lock);
        preempt_enable();
}
```

Preemptible Kernel (6)

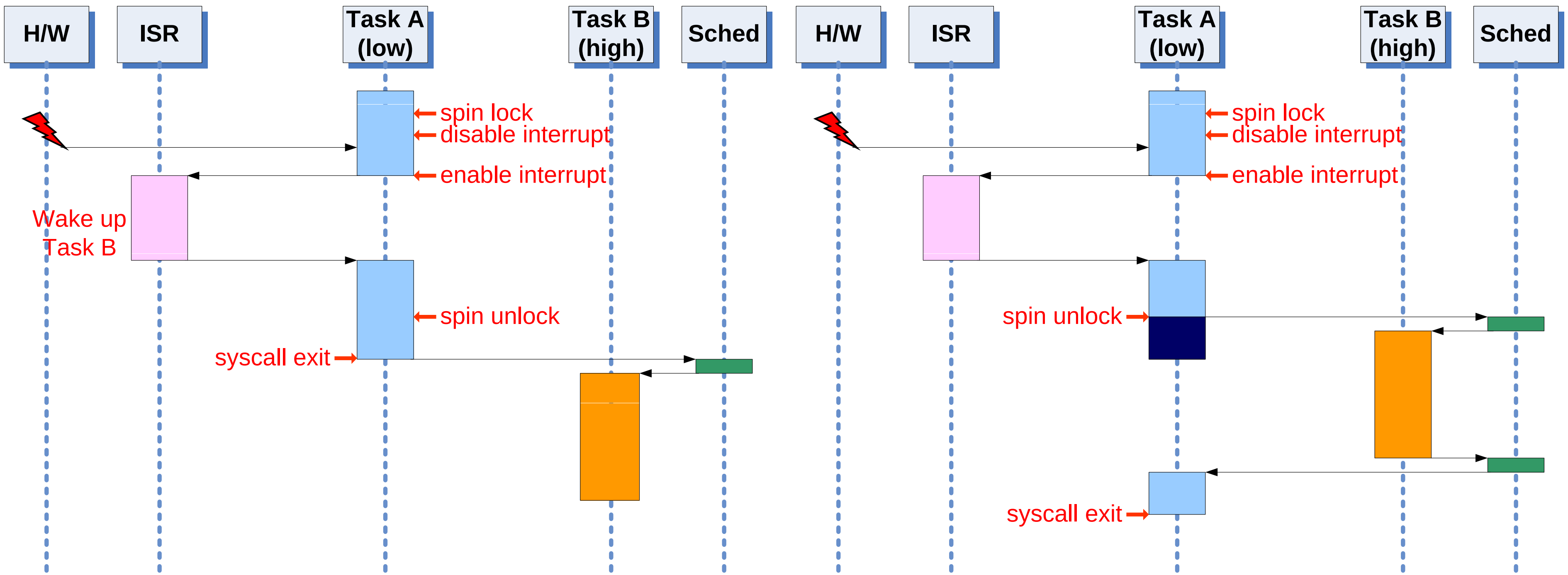
- **Accessing per-processor data**

```
                                                        <linux/smp.h>
#define get_cpu()  ({preempt_disable(); smp_processor_id(); })
#define put_cpu()  preempt_enable()
```

```
int cpu;
{
…
        /* disable kernel preemption &  set cpu to the current CPU */
        cpu = get_cpu();
        /* manipulate per-processor data */
        …
        /* reenable kernel preemption & cpu is no longer valid */
        put_cpu();
…
}
```

Preemptible Kernel (7)

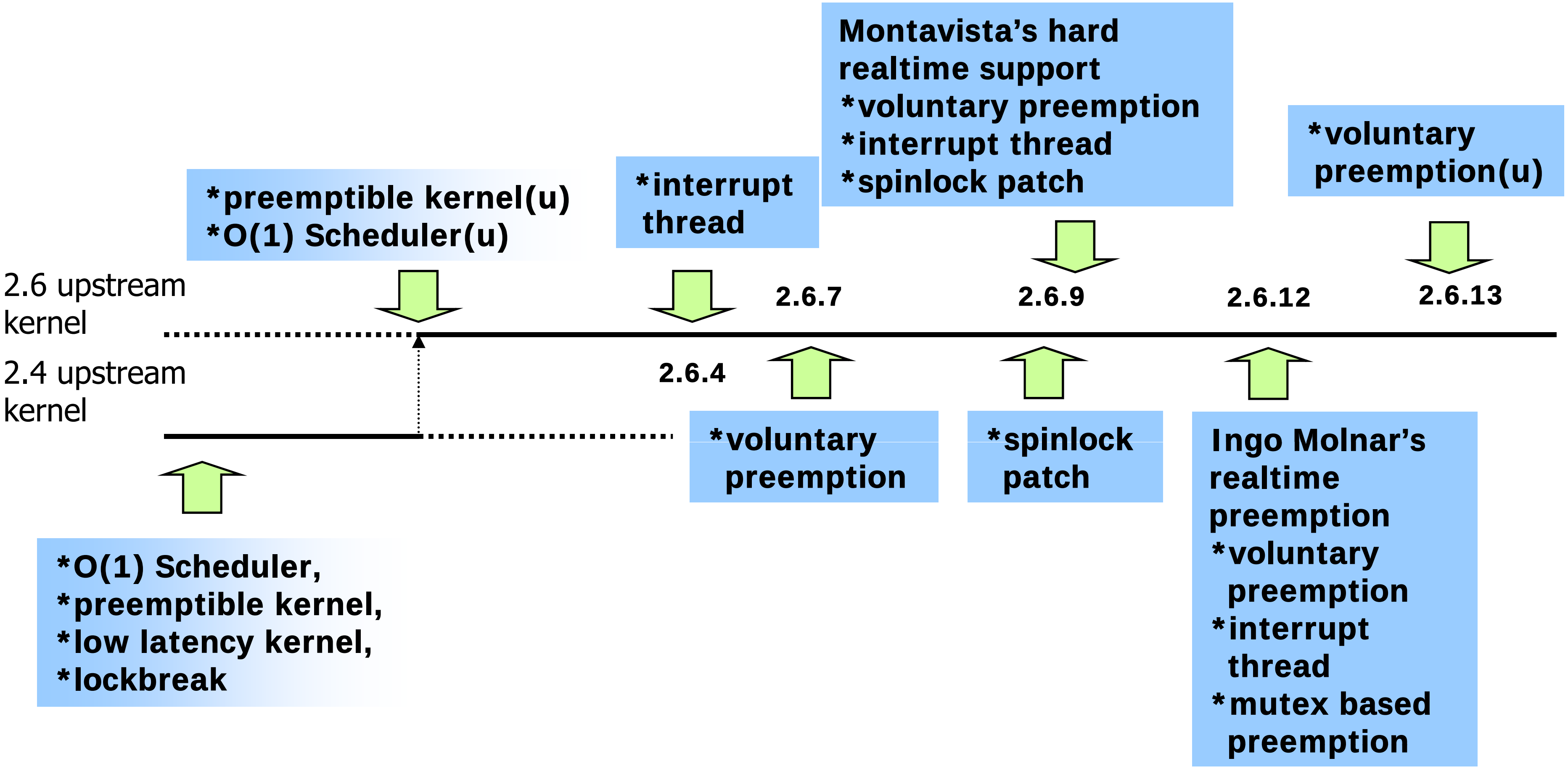
- **Vanilla kernel vs. preemptible kernel**



Vanilla kernel (Non-preemptible kernel)

Preemptible kernel

Real-Time Support in 2.6



Voluntary Preemption (1)

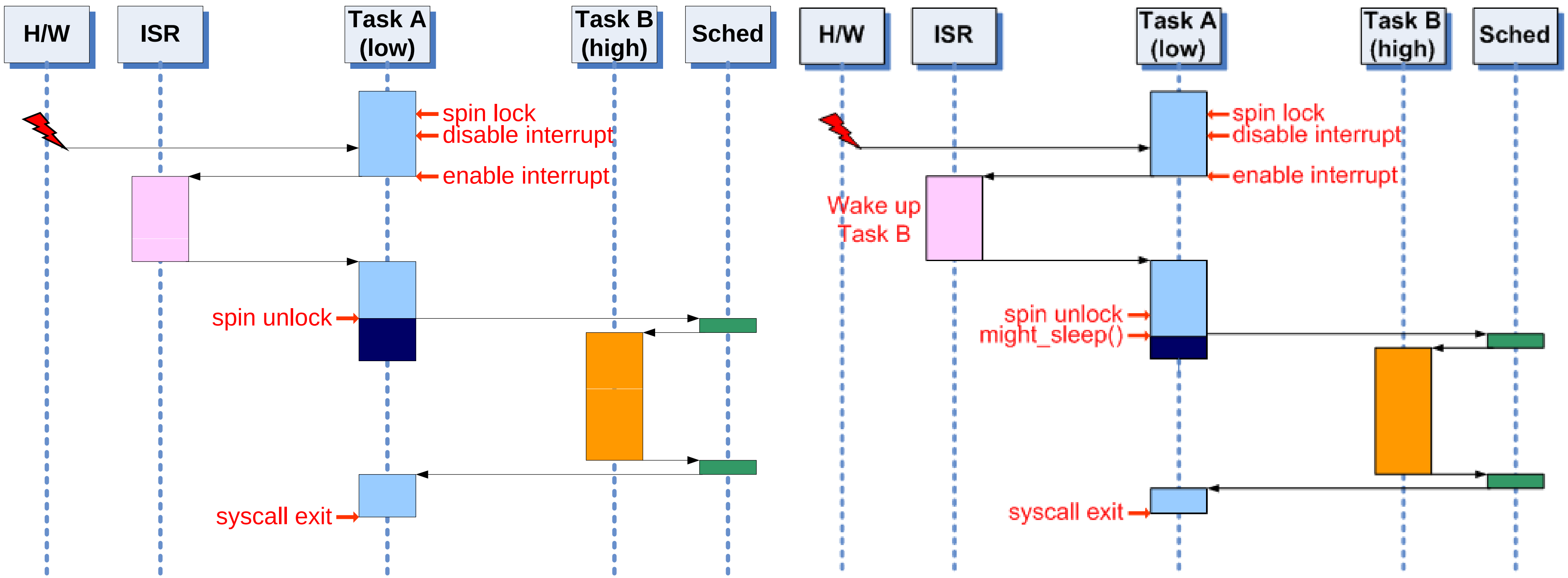
- **Background**
  - Complaints on the Linux kernel mailing list
    - By Jackit (Java Audio Connection Kit) people
    - The 2.6 kernel is not suitable for serious audio work due to high scheduling latencies.
    - Up to 50ms with 2.6.7 preemptible kernel on 2GHz+ x86
  - Proposed by Ingo Molnar (with Arjan van de Ven) in 2004 for Linux kernel 2.6.7
  - Add several scheduling points to the source code
    - Systematically via might_sleep() macro
  - Includes lock break feature
  - Officially adopted in 2.6.13, but not with kernel preemption.

Voluntary Preemption (2)

- **Mechanism**
  - Reuse a rich but currently inactive set of scheduling points that already exist in the 2.6 kernel
    - might_sleep() debugging checks → cond_resched()
    - Any code point that does might_sleep() is in fact ready to sleep at that point.
    - Reduce complexity and impact quite significantly.
  - There were still a number of latency sources.
    → Identify and fix them by hand, either via
    - Additional might_sleep() checks
    - Explicit rescheduling points
    - Lock-break

Voluntary Preemption (3)

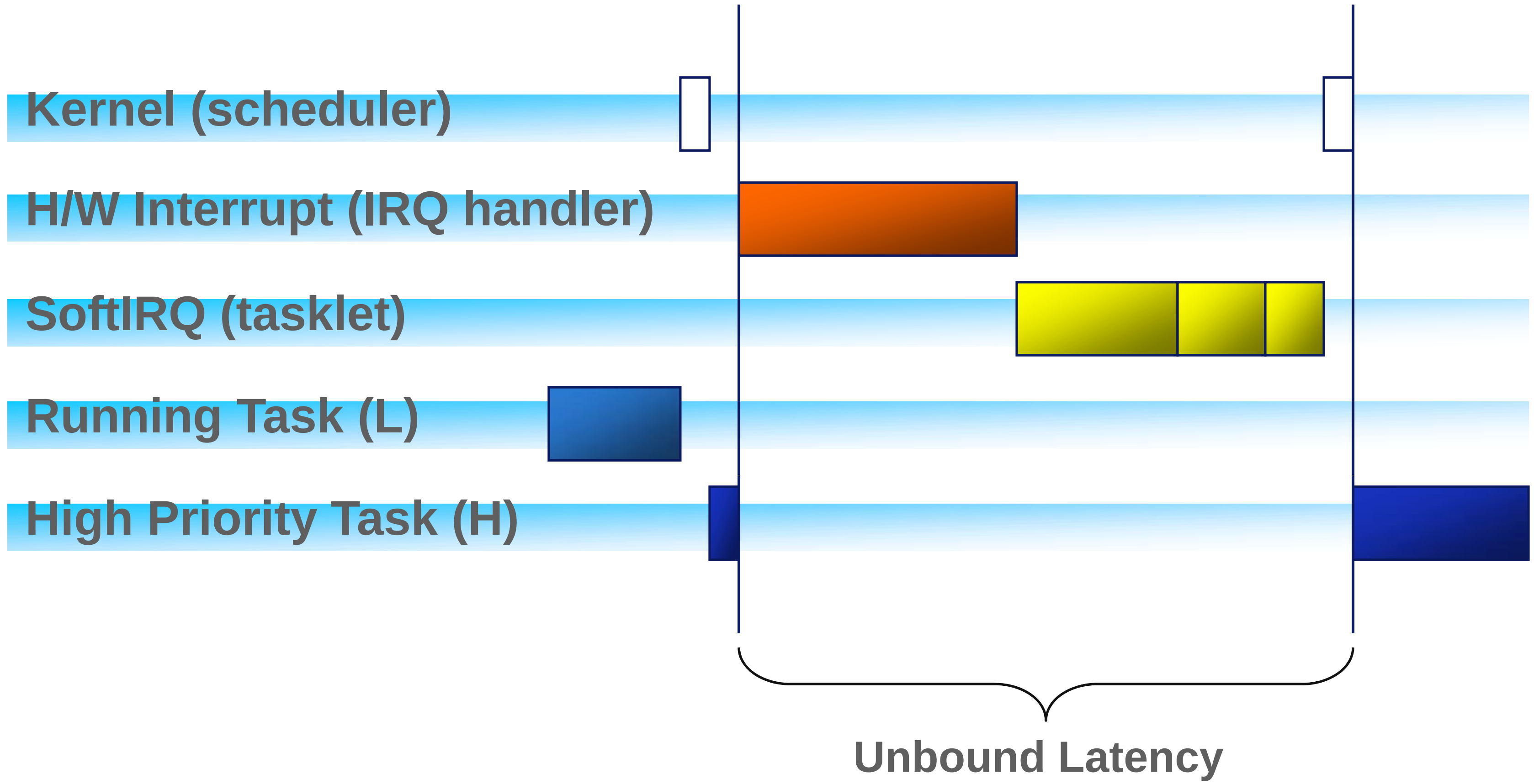
- **Using voluntary preemption**



Preemptible kernel **Voluntary preemption**

IRQ Threads (1)

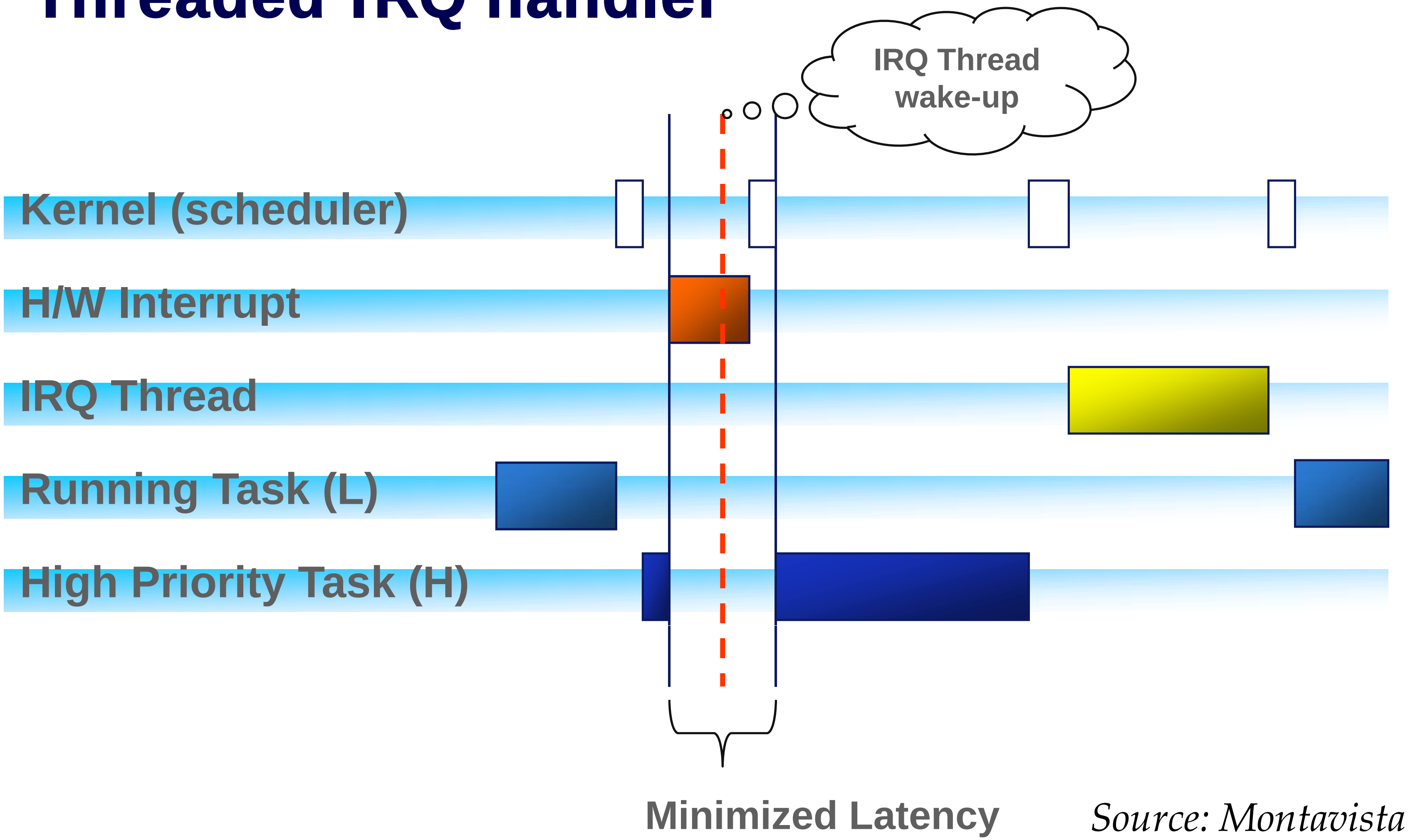
- **Standard IRQ mechanism**



Source: Montavista

IRQ Threads (2)

- **Threaded IRQ handler**



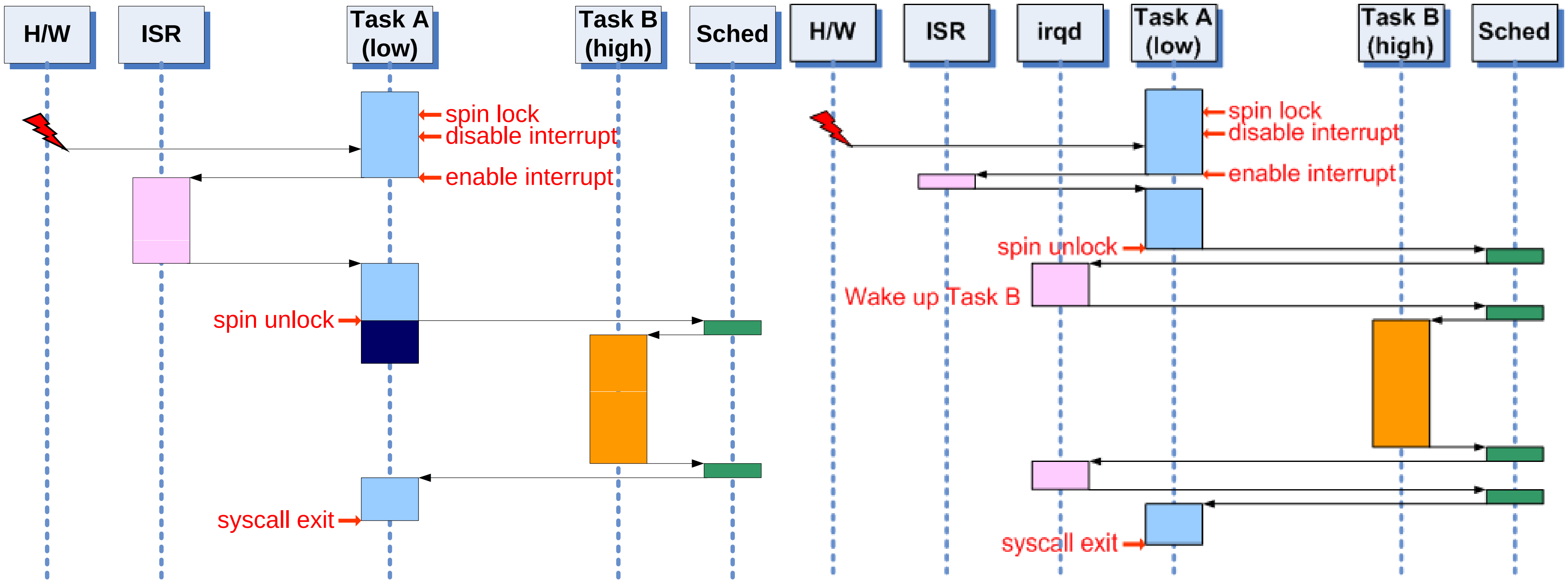
Source: Montavista

IRQ Threads (3)

- **Mechanism**
  - Proposed by Scott Wood in 2004 for Linux kernel 2.6.4.
  - Motivation
    - Interrupt handlers are not preempted → increases latency
  - Run IRQ handlers in (kernel) threads
    - Softirqs are also run in threads
    - Timer interrupt handler is not threaded (with SA_NODELAY)
  - IRQ threads
    - One thread for each hardware IRQ
    - Real-time priorities are assigned (25~50)
    - Scheduled under SCHED_FIFO policy
    - Interrupt handlers are now scheduled and preempted by normal system scheduler

IRQ Threads (4)

- **Using IRQ threads**



Preemptible kernel

Preemptible kernel + IRQ threads

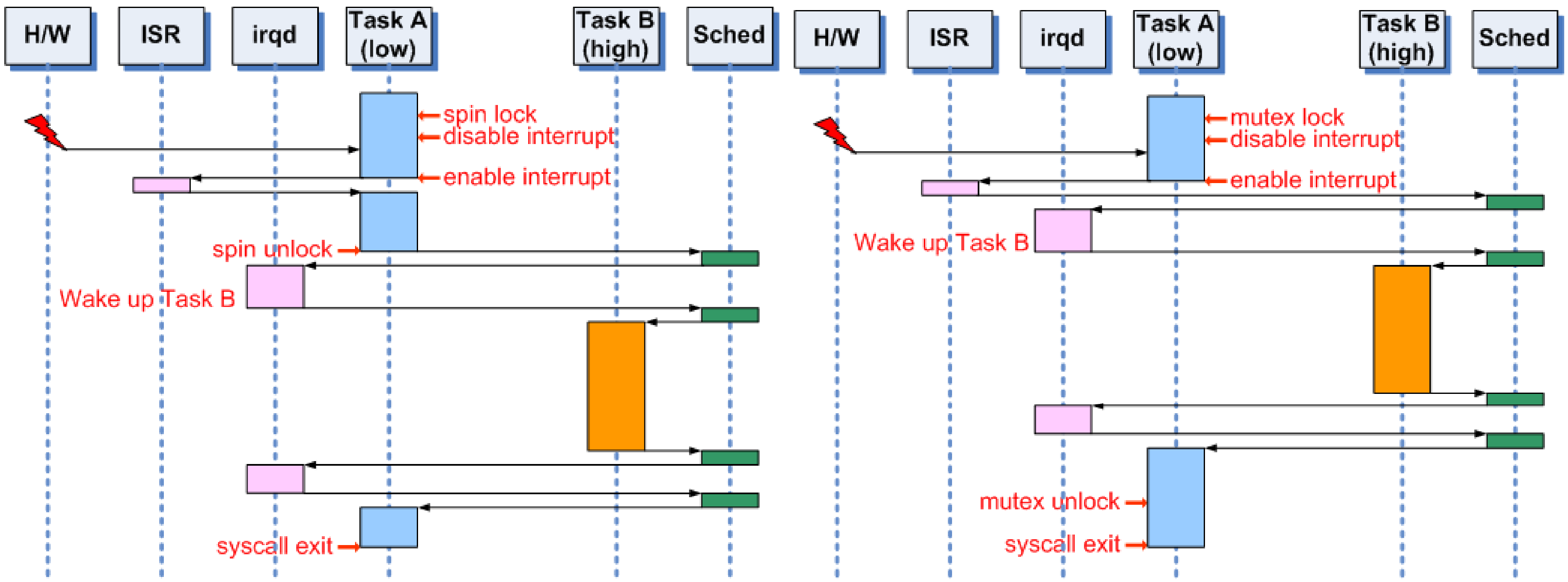
Mutex-based Preemption (1)

- **Background**
  - Spin lock regions are not preemptible.
  - OK if the newly scheduled task does not require the same spin lock.

- **Mechanism**
  - Minimize the use of spin lock
  - Replace spin locks with mutex locks
    - Spinlock: 491 (6%)
    - Mutex lock: 7678 (93.9%)
  - Preemption is enabled inside the mutex region

Mutex-based Preemption (2)

- **Using mutex lock**

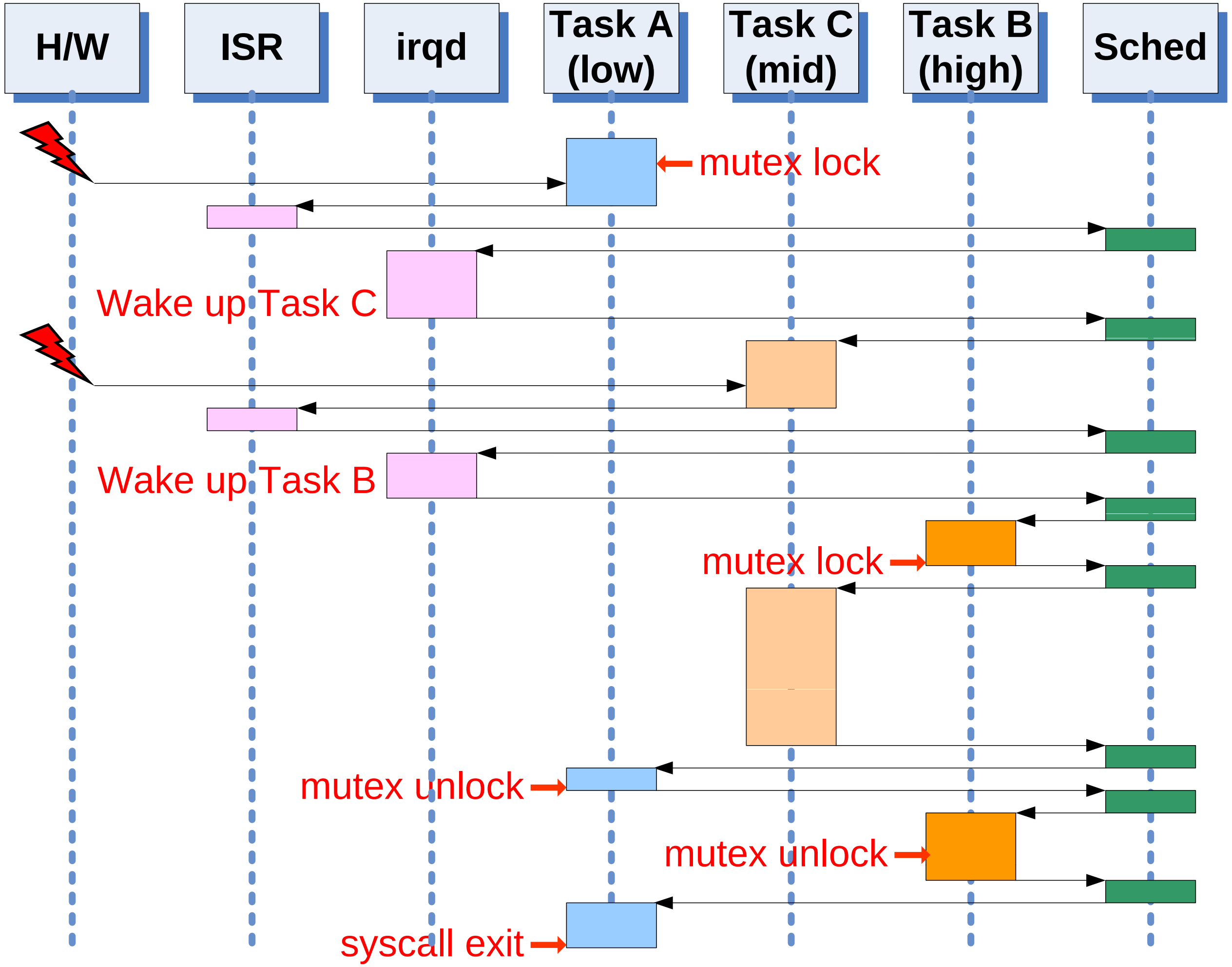


Preemptible kernel + IRQ threads

Preemptible kernel + IRQ threads + mutex lock

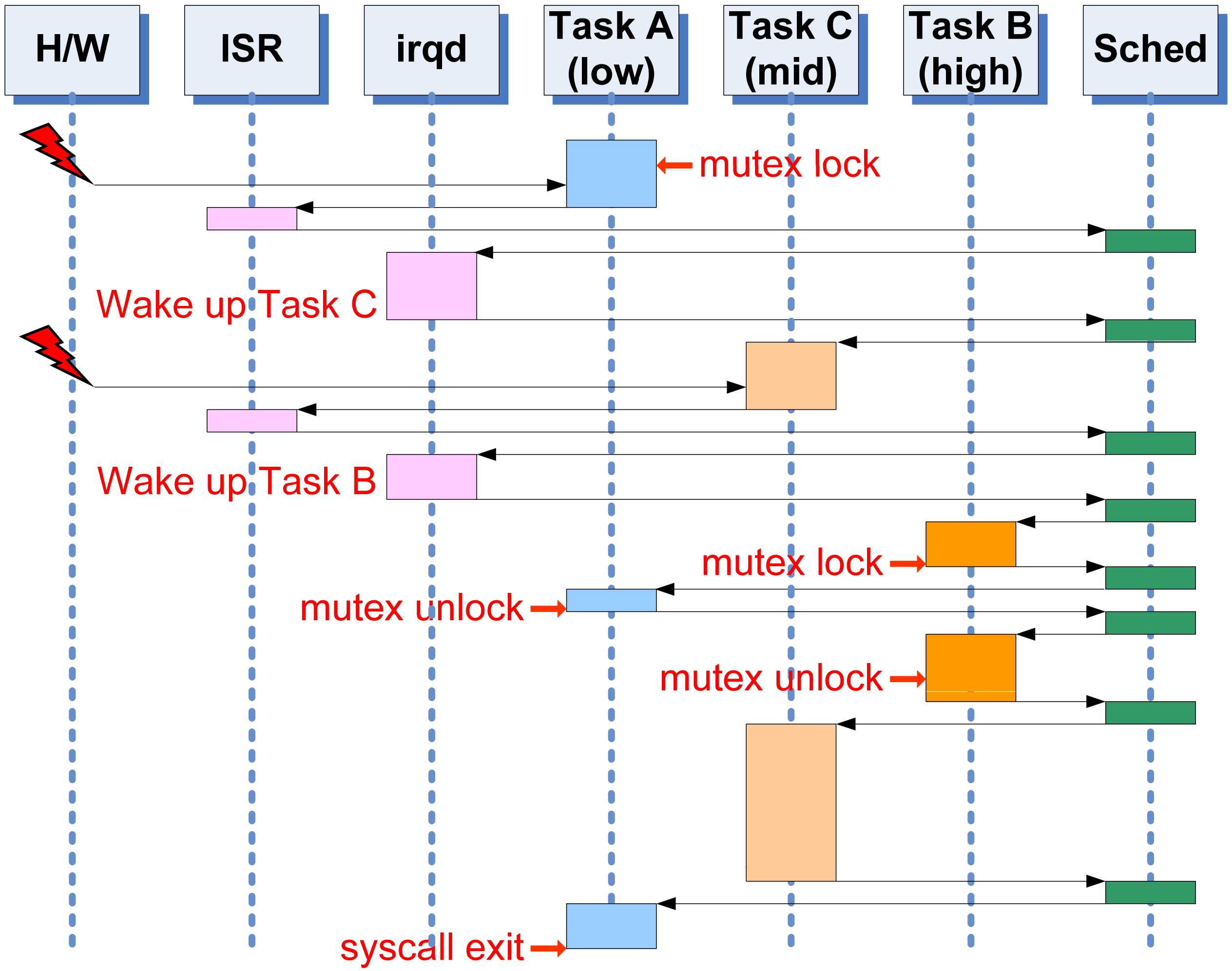
Priority Inheritance (1)

- **Priority inversion problem**



Priority Inheritance (2)

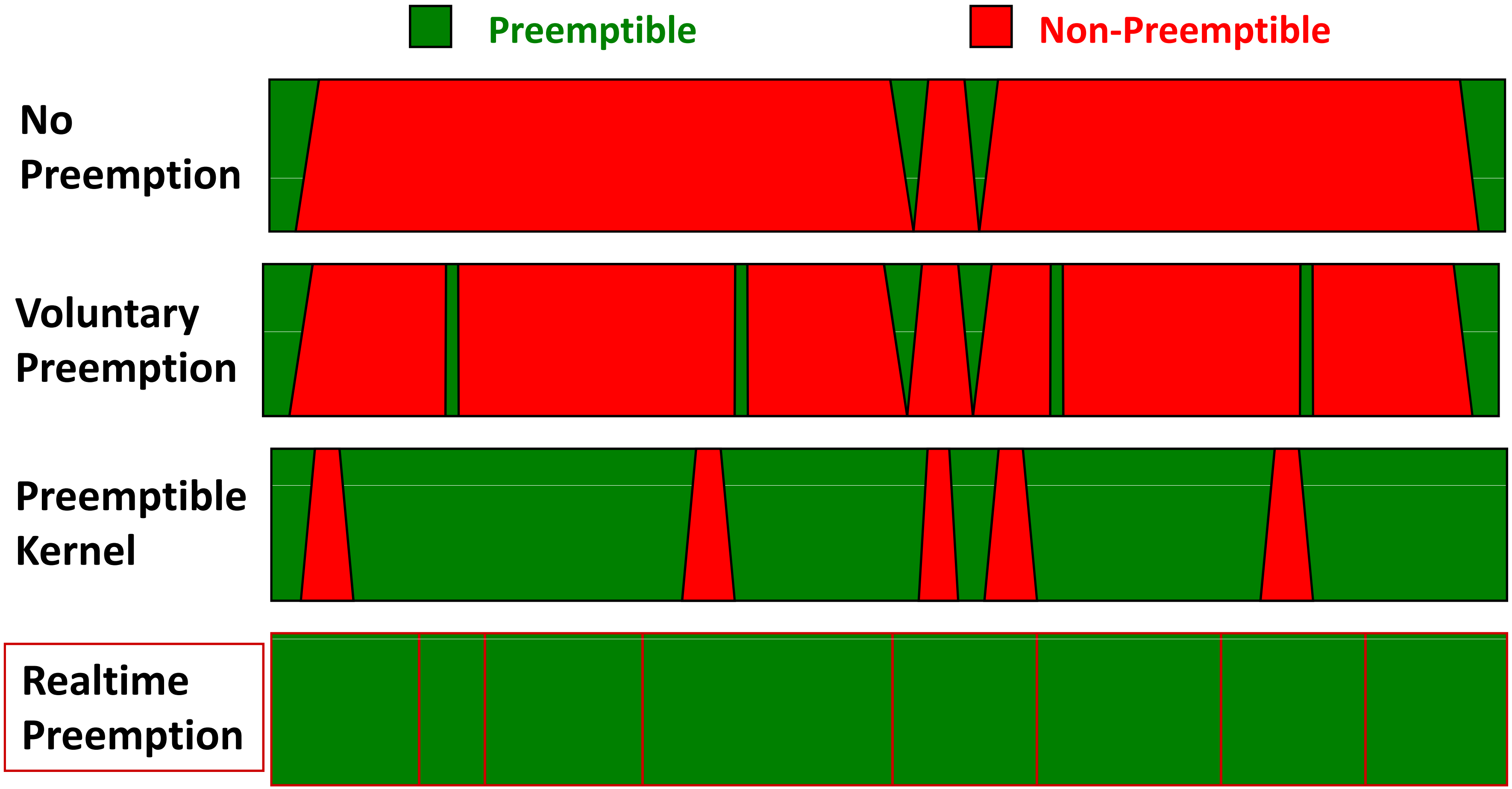
- **With priority inheritance**



Real-Time Preemption

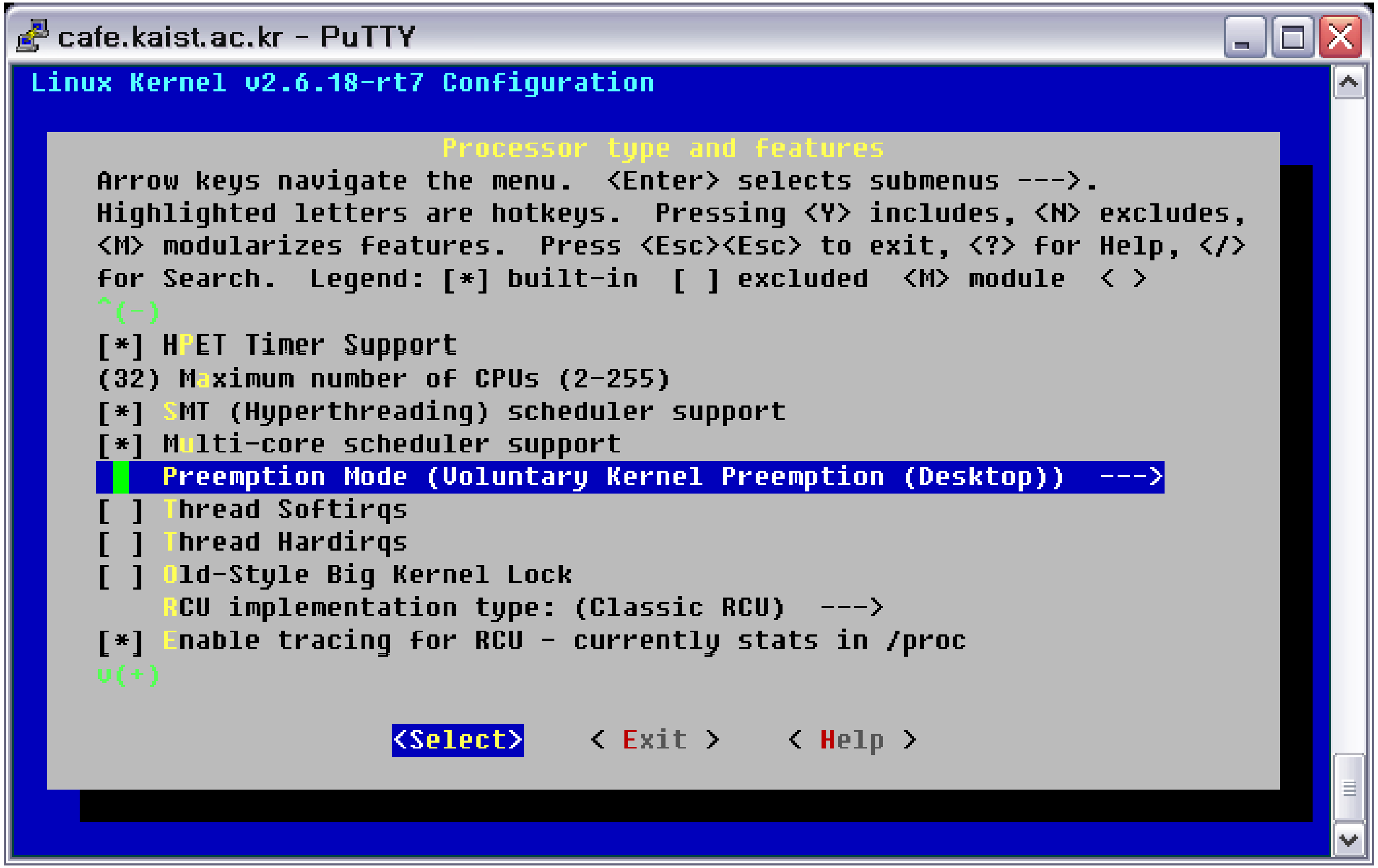
- **Real-time complete preemption patch**
  - Proposed by Ingo Molnar in 2004 for Linux kernel 2.6.12-rc2
    - http://people.redhat.com/mingo/realtime-preempt/
  - Recommended for < 100usec response time
  - Incorporate most of existing real-time features
    - Includes preemptible kernel
    - Prioritized interrupt thread
    - Mutex-based preemption
    - Priority inheritance mechanism

Comparison



Source: Montavista

Kernel Configuration (1)



Kernel Configuration (2)

Kernel Configuration (3)

- **No forced preemption**
  - CONFIG_PREEMPT_NONE=y
  - Traditional Linux preemption model
  - Best for throughput (batch jobs)
  - Provide good latencies most of the time
  - No guarantees for latency and occasional long delays
  - Recommended for server environment
  - Preemption points
    - At the end of an interrupt handler and before returning to user mode
    - At the end of a system call and before returning to user mode

Kernel Configuration (4)

Voluntary kernel preemption

- CONFIG_PREEMPT_VOLUNTARY=y
- Reduce the latency by adding more "explicit preemption points" to the kernel (might_sleep())
- Recommended for general desktop environment

```
#ifdef  CONFIG_PREEMPT_VOLUNTARY
#define might_resched() cond_resched()
#else
#define might_resched() do { } while (0)
#endif

#define might_sleep() do { might_resched(); } while (0)
```

Kernel Configuration (5)

- **Preemptible kernel**
  - CONFIG_PREEMPT_DESKTOP=y
  - CONFIG_PREEMPT=y
  - Make all kernel code that is not executing in a critical section preemptible.
  - Voluntary preemption is not enabled.
    - Long-held spin lock is possible.
  - Recommended for low-latency desktop environment

Kernel Configuration (6)

- **Complete preemption**
  - CONFIG_PREEMPT_RT=y
  - CONFIG_PREEMPT=y
  - CONFIG_PREEMPT_SOFTIRQS=y
  - CONFIG_PREEMPT_HARDIQRS=y
  - Preemptible in the critical section
  - Reduces the latency of the kernel by replacing almost every spinlock with preemptible mutexes
  - Priority inheritance mechanism
  - Recommended for real-time environment

Kernel Configuration (7)

- **Thread softirqs**
  - CONFIG_PREEMPT_SOFTIRQS=y
  - All softirqs will execute in ksoftirqd's context.

- **Thread hardirqs**
  - CONFIG_PREEMPT_HARDIRQS=y
  - All (or selected) interrupt handlers will run in their own kernel thread context.

Evaluation (1)

Benchmarks

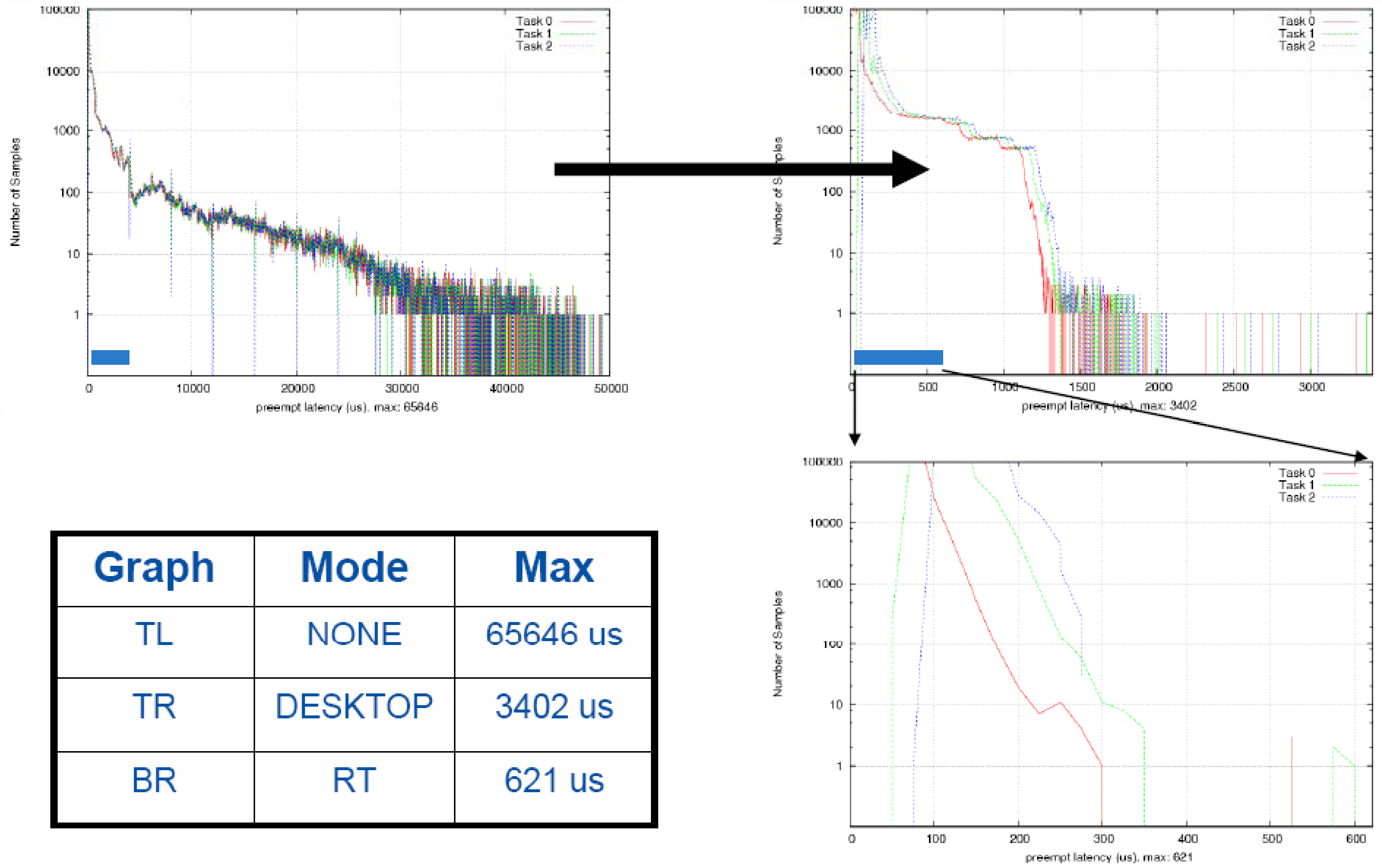
montavista

- **Workload applied to the target system:**
  - Lmbench
  - Netperf
  - Hackbench
  - Dbench
  - Video Playback via MPlayer
- **CPU utilization during test:**
  - 100% most of the time
- **Test Duration:**
  - 20 hours

Evaluation (2)

Interrupt to Userspace, ARM

montavista



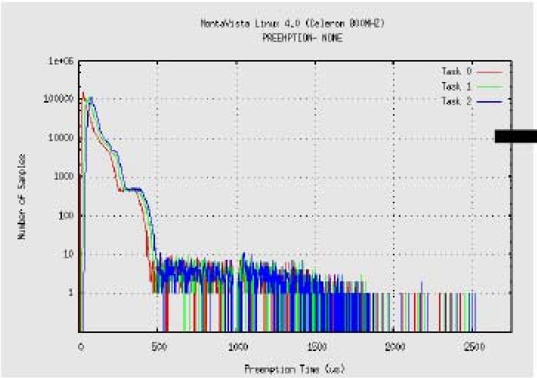
Graph	Mode	Max
TL	NONE	65646 us
TR	DESKTOP	3402 us
BR	RT	621 us

Evaluation (3)

Interrupt to Userspace, 800Mhz Celeron

montavista

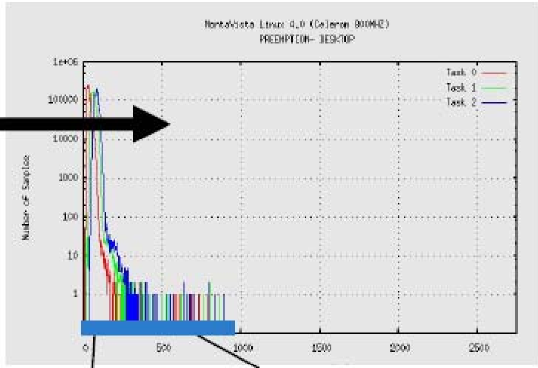
Linux 2.6 Kernel – No Preemption



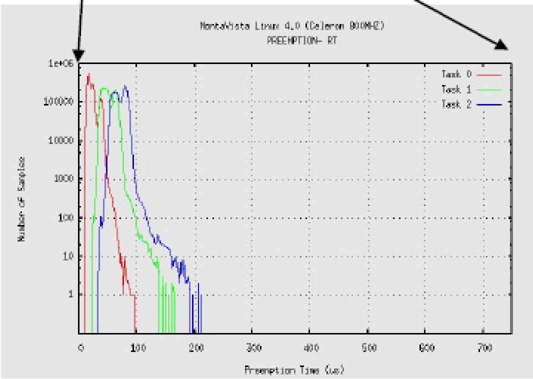
Source:

Brief History of Real-Time Linux

Linux 2.6 Kernel – Preemptible Kernel



Linux 2.6 Kernel – Real-Time Preemption



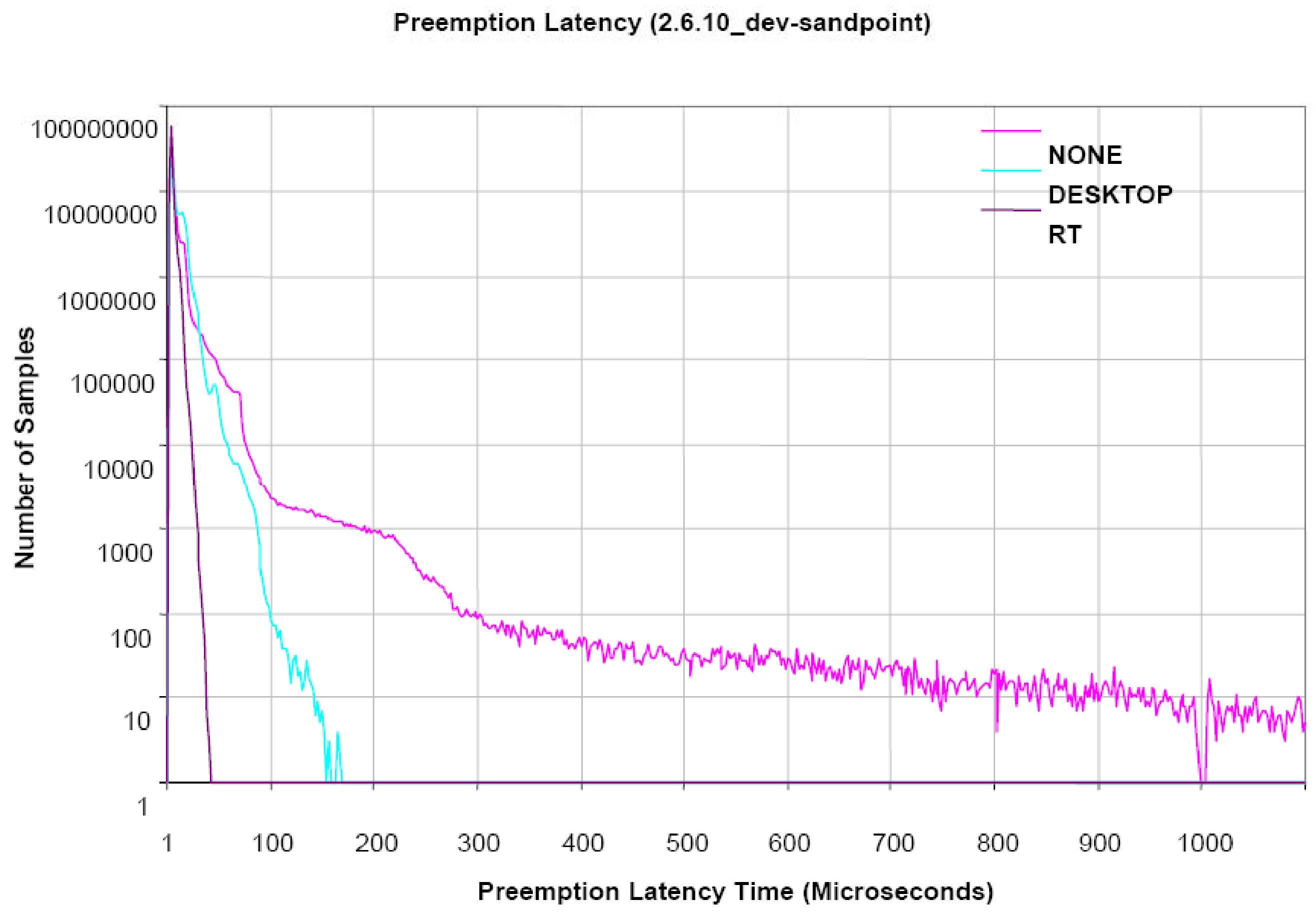
Source:

Graph	Mode	Max
TL	NONE	~2500 us
TR	DESKTOP	~900 us
BR	RT	~100 us

Evaluation (4)

Linux 2.6 Kernel – PPC 7457 Sandpoint

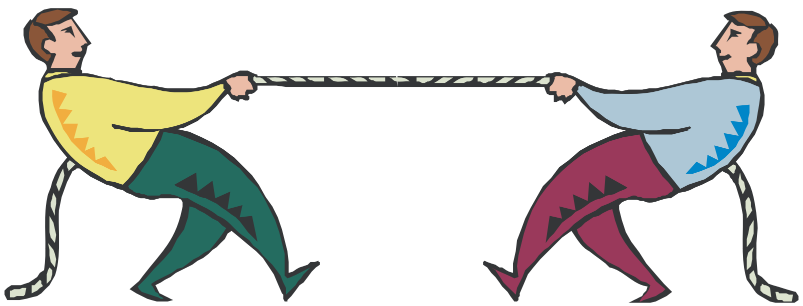
montavista



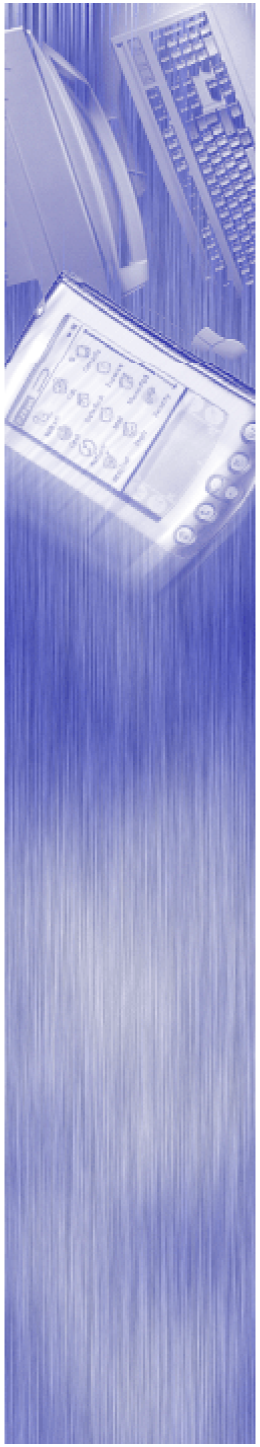
Responsiveness vs. Throughput

- **Overhead for real-time preemption**
  - More frequent task switching!
  - Mutex operations more complex than spinlock operations
  - Priority inheritance on mutex increases task switching
  - Priority inheritance increases worst-case execution time
- **Efficiency and responsiveness are inversely related.**

Throughput



High responsiveness



CS632/SEP564: Embedded Operating Systems (Fall 2008)

Time Management

KAIST

Introduction

- **Why is timing measurement important?**
  - Many kernel functions are time driven
    - CPU time sharing
    - Updating resource usage statistics
  - Keeping the current time and date
    - System timer
    - time(), gettimeofday(), timestamps for files and network packets
  - Maintaining software timers
    - Dynamic timers
    - setitimer(), alarm()

System Timer (1)

- **System timer**
  - Issue a timer interrupt at a preprogrammed frequency.
  - tick: the time between any two successive timer interrupts.
  - Work executed periodically by the timer interrupt:
    - Updating the system uptime
    - Updating the time of day
    - Rebalancing the scheduler runqueues (on SMP)
    - Checking whether the current process has exhausted its timeslice and, if so, causing a reschedule
    - Running any dynamic timers that have expired
    - Updating resource usage and processor time statistics

System Timer (2)

- **HZ: tick rate**
  - Frequency of the timer interrupt
    - #define HZ 250 <asm-i386/param.h>
    - Very architecture dependent

Architecture	HZ
alpha	1024
arm	100
cris	100
h8300	100
i386	100 / 250 / 300 / 1000
ia64	1024
m68k	100
m68knommu	50 / 100 / 1000
mips	100
mips64	100 / 1000

Architecture	HZ
parisc	100 / 1000
ppc	1000
ppc64	1000
s390	100
sh	100 / 1000
sparc	100
sparc64	1000
um	100
v850	24 / 100 / 122
x86-64	1000

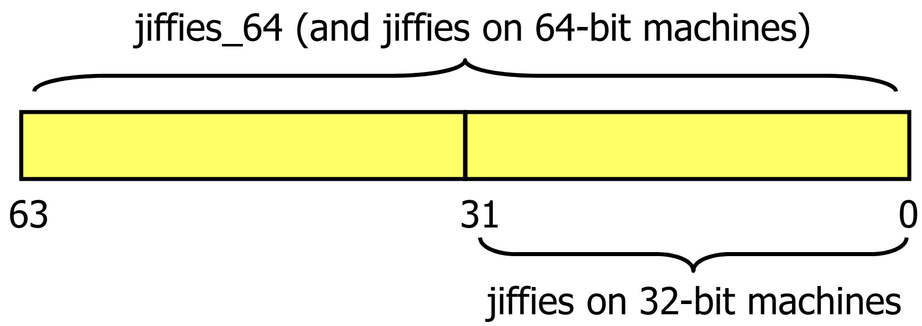
System Timer (3)

- **Larger HZ values**
  - Kernel timers execute with finer resolution and increased accuracy.
  - System calls such as poll() and select() execute with improved precision.
  - Measurements, such as resource usage or the system uptime, are recorded with a finer resolution.
  - Process preemption occurs more accurately.

  - Higher overheads due to more frequent timer interrupts.

Jiffies (1)

- **Jiffies**
  - The global variable that holds the number of ticks that have occurred since the system booted.
    - Incremented by one during each timer interrupt
    - The system uptime = jiffies/HZ (seconds)
  - Internal representation
    - u64 jiffies_64;
      unsigned long volatile jiffies; <linux/jiffies.h>
    - Linker script overlays the jiffies variable over the jiffies_64



Jiffies (2)

▪ Jiffies wraparound

- 32-bit jiffies: 497days (HZ=100), 49.7days (HZ=1000)
- Timer-wraparound-safe macros
  - time_after(a,b): returns true if the time a is after time b
  - #define time_after(a,b)   ((long)(b) – (long)(a) < 0);
  - #define time_before(a,b)   time_after(b,a)

```
unsigned long timeout = jiffies + HZ/2;
…
if (timeout > jiffies) {
   /* not timed out, yet */
}
else {
   /* timed out */
}
```

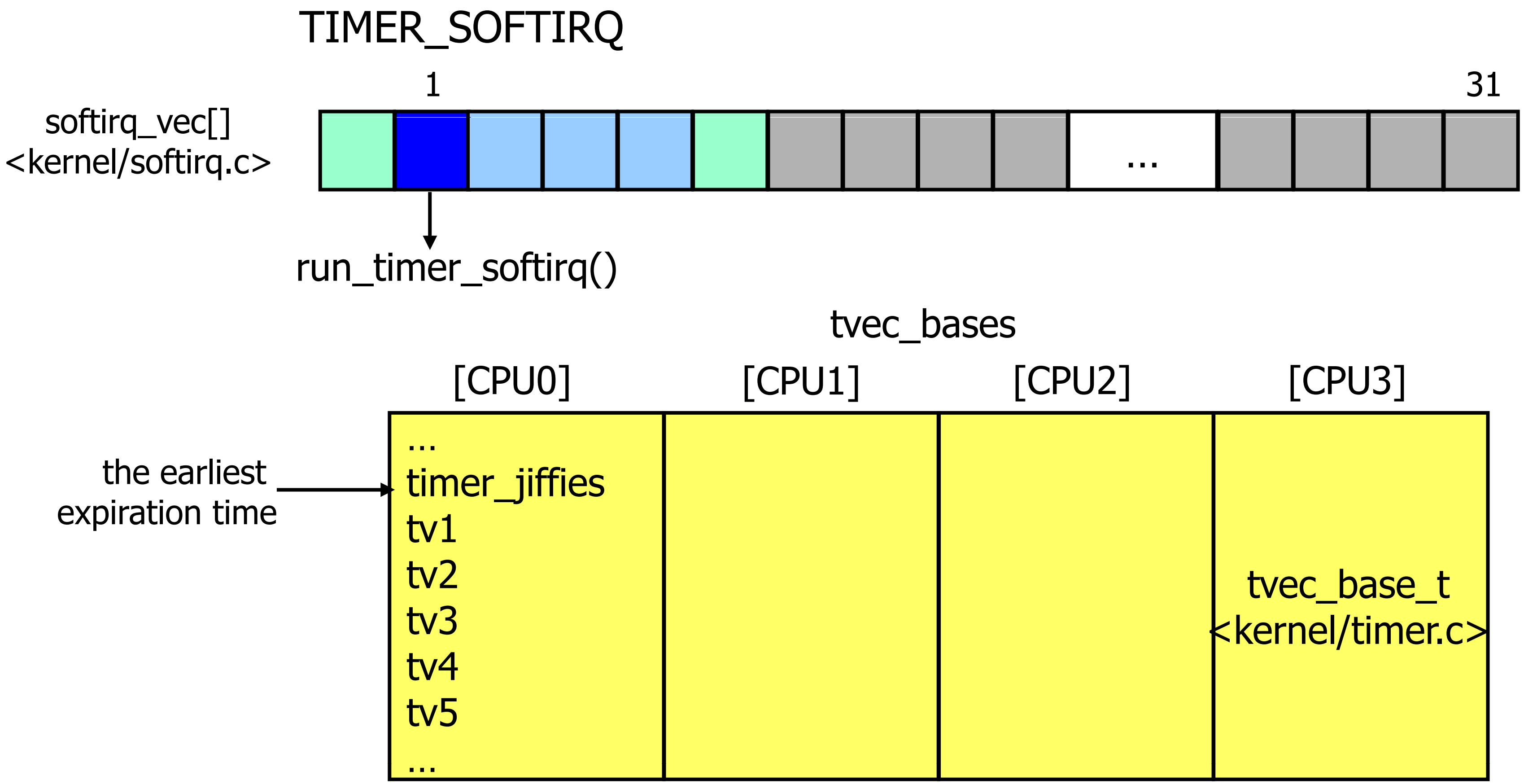
```
unsigned long timeout = jiffies + HZ/2;
…
if (time_before(jiffies, timeout)) {
   /* not timed out, yet */
}
else {
   /* timed out */
}
```

Timers (1)

- **Dynamic timers or kernel timers**
  - The given function will run after the timer expires.
  - The timer is destroyed after it expires.
  - The kernel executes timers in bottom-half context as softirqs.
  - The kernel cannot ensure that timer functions will start right at their expiration times.
    - Not appropriate for real-time applications in which expiration times must be strictly enforced.
  - Insertion, deletion, and lookup operation should be fast.
  - Partition timers according to their expiration time.
  - The timers are also split among the CPUs.

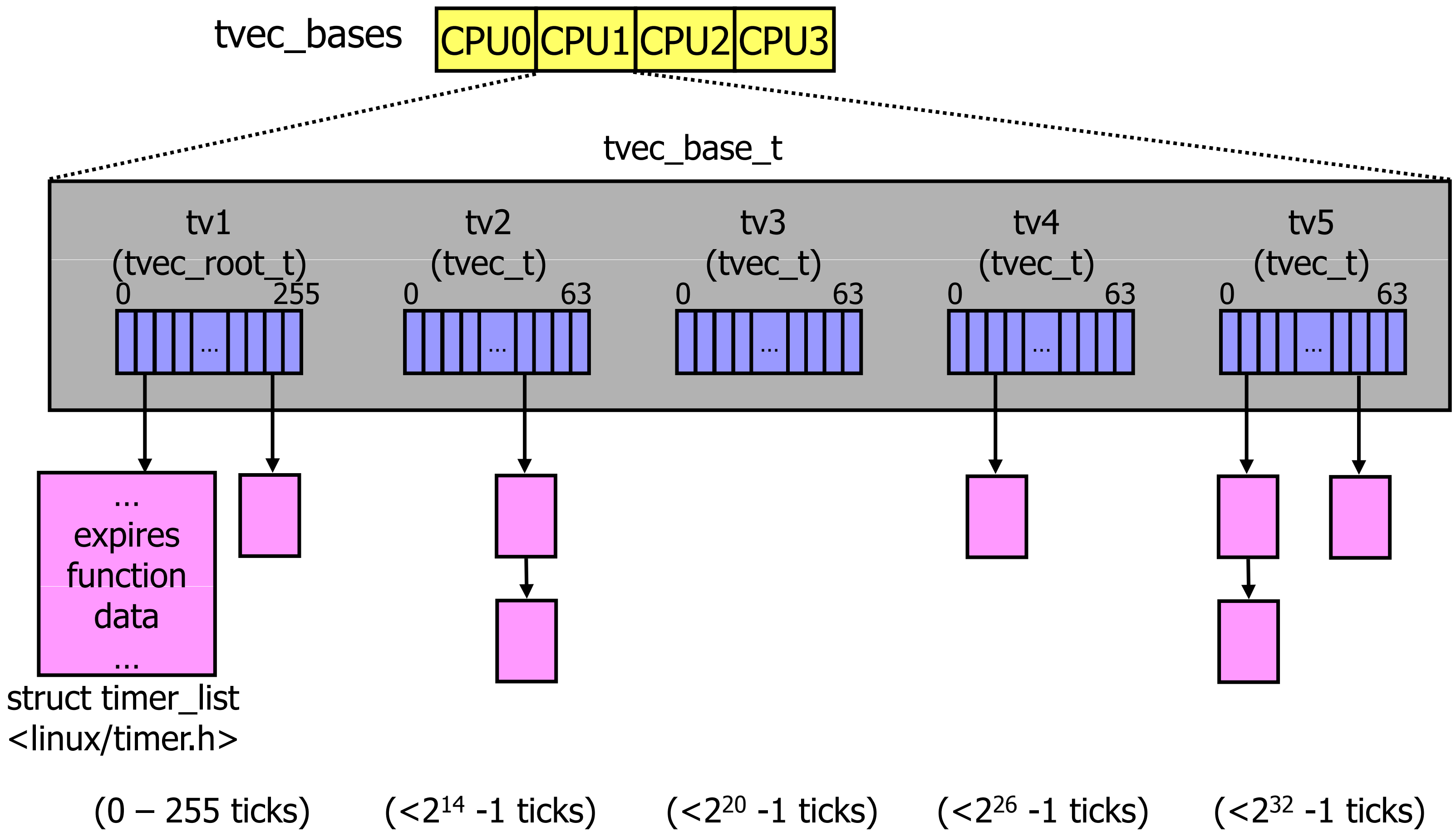
Timers (2)

- **Kernel data structures for timers**



Timers (3)

- **Kernel data structures for timers (cont'd)**



Delaying Execution (1)

- **Busy waiting or busy looping**
  - ```
    unsigned long delay = jiffies + 2*HZ;
    while (time_before(jiffies, delay)) ;
    ```
- **Reschedule the process**
  - ```
    unsigned long delay = jiffies + 5*HZ;
    while (time_before(jiffies, delay))
            cond_resched();
    ```
  - Cannot be used in interrupt handlers.
  - Conditionally invokes the scheduler if there is some more important task to run.

Delaying Execution (2)

- **Small delays**
  - **void udelay (unsigned long usecs);**
  - **void mdelay (unsigned long msecs);**
  - Implemented as busy looping using BogoMIPS
    - BogoMIPS: the number of busy loop iterations the processor can perform in a given period.

- **schedule_timeout()**
  - **set_current_state (TASK_INTERRUPTIBLE);**
    schedule_timeout (s * HZ);
  - Implemented using timers.