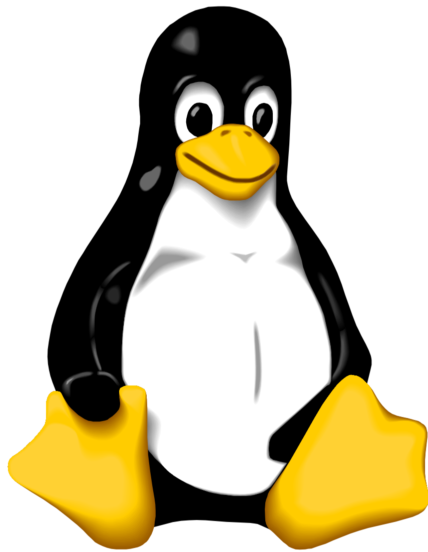
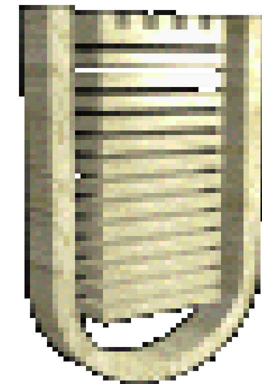
Linux Kernel Hacking Free Course

3rd edition

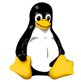
G.Grilli, University of Rome “Tor Vergata”

IRQ DISTRIBUTION IN MULTIPROCESSOR SYSTEMS

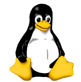




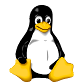
Contents:

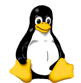
 What is an interrupt

- Synchronous and asynchronous interrupts
- Interrupts in uniprocessor and smp achitectures
- The advanced programmable interrupt controller (APIC)

 IRQs' distribution problem starting with Intel® Pentium 4 architecture

- Negative impact on massively parallel applications

 An experimental kernel with a new irq balancing capability

 Hint: using smp_affinity() to bind irq lines to specific processors

What is an interrupt?

- An interrupt is defined as an event that alters the sequence of instructions executed by a processor.

- Such events correspond to electrical signals generated by hardware circuits both inside and outside the CPU chip.

- Interrupts are often divided into "*synchronous*" and "*asynchronous*" interrupts

Interrupts and Exceptions (Intel® classification)

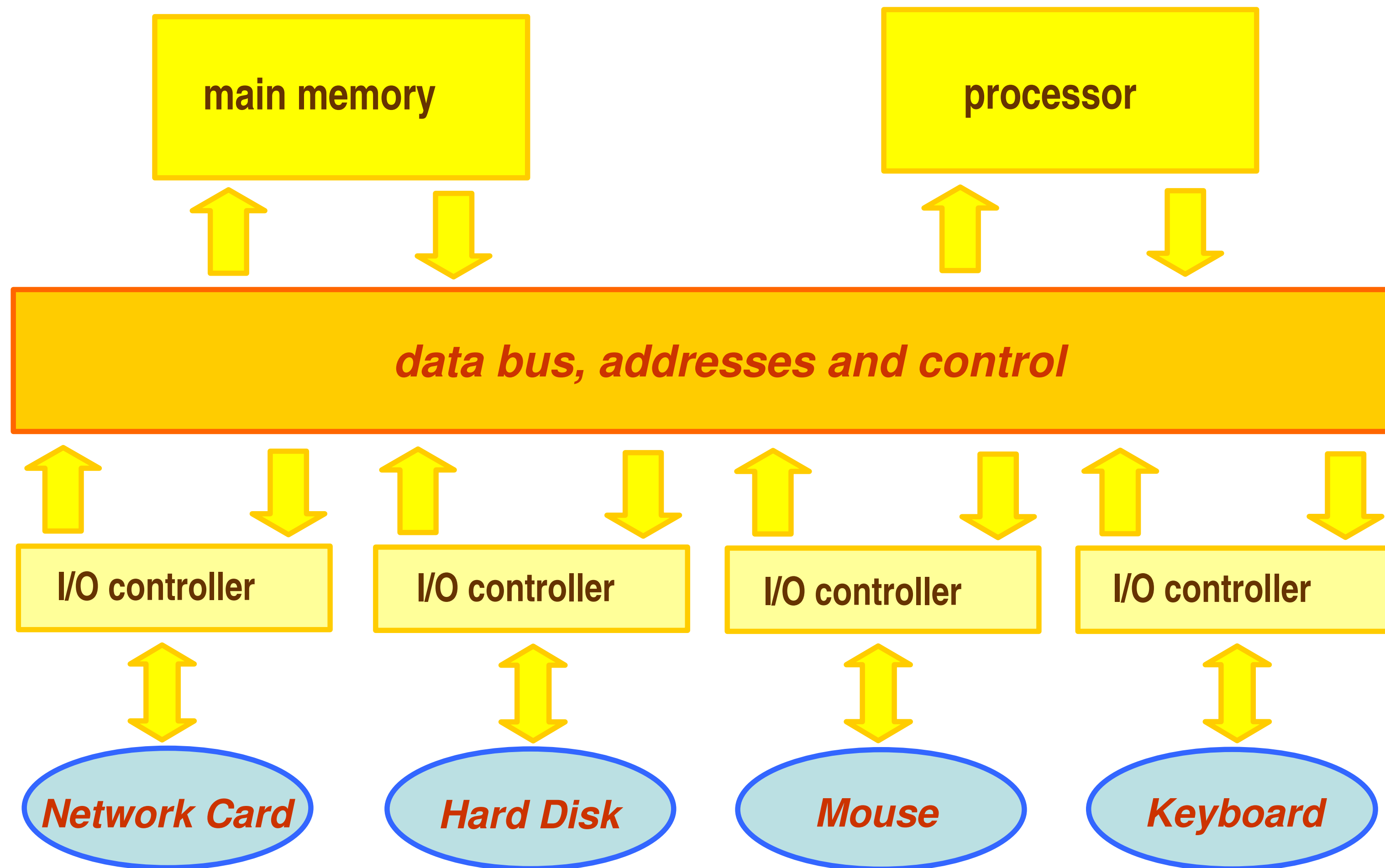
synchronous interrupts ("interrupts")

They are produced by the CPU control unit (CU) while executing instructions. "Synchronous" because the CU issues them only after terminating the execution of an instruction (programming errors or anomalous conditions).

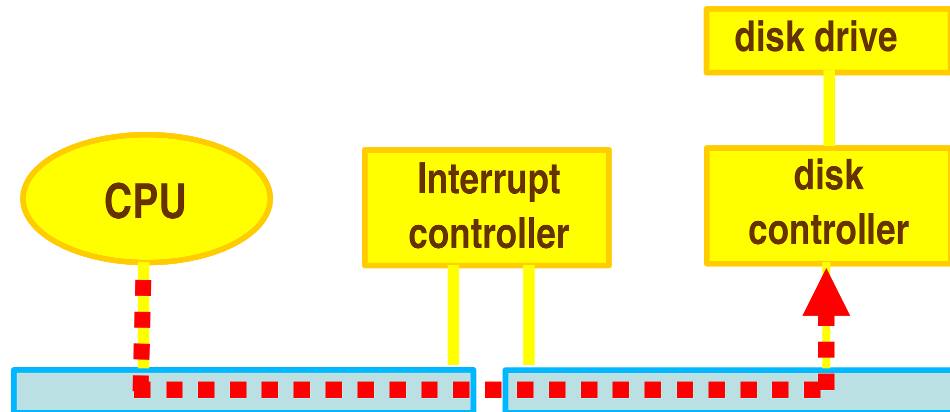
asynchronous interrupts ("exceptions")

They are generated by other hardware devices at arbitrary times with respect to the CPU clock signals (interval timers and I/O devices).

Interrupts (overview schema)

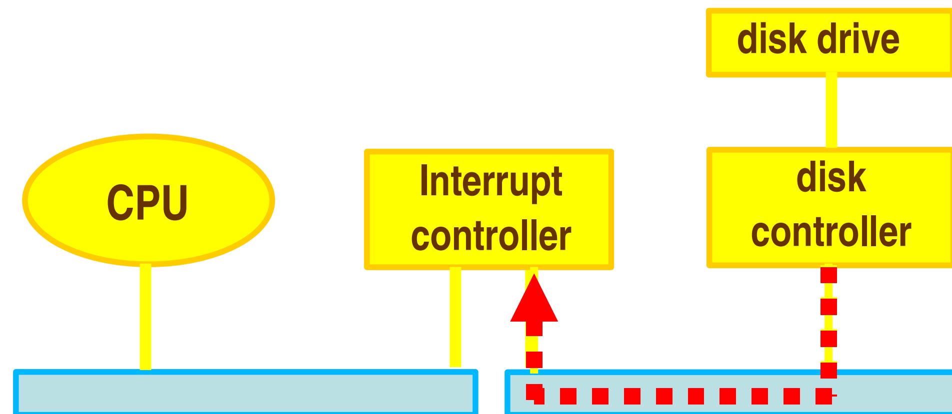


Interrupts – uniprocessor scenario (1)



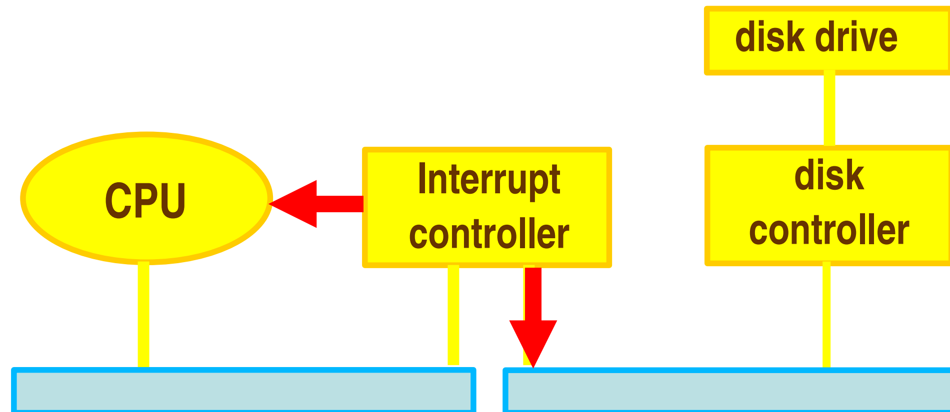
Phase 1: CPU asks the disk controller to perform some operations

Interrupts – uniprocessor scenario (2)



Phase 2: the disk controller has completes its task and raises an interrupt on the bus using a specific irq line.

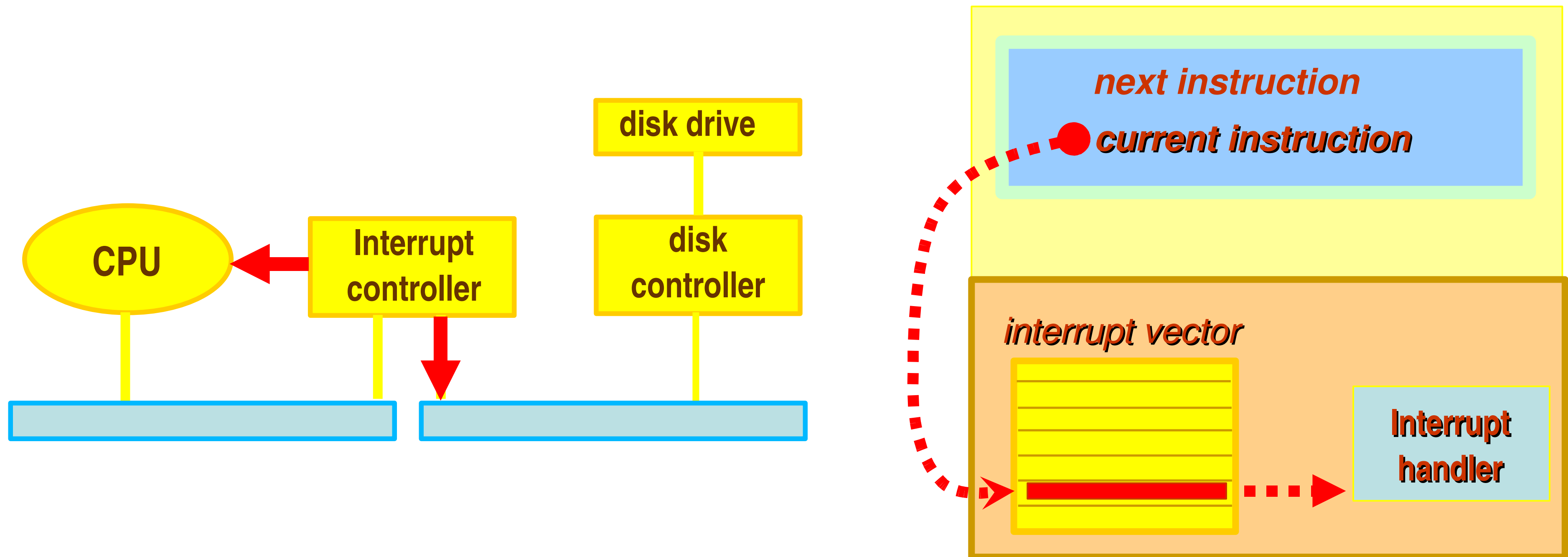
Interrupts – uniprocessor scenario (3)



Phase 3: the interrupt controller sets the interrupt signal to be handled by the cpu

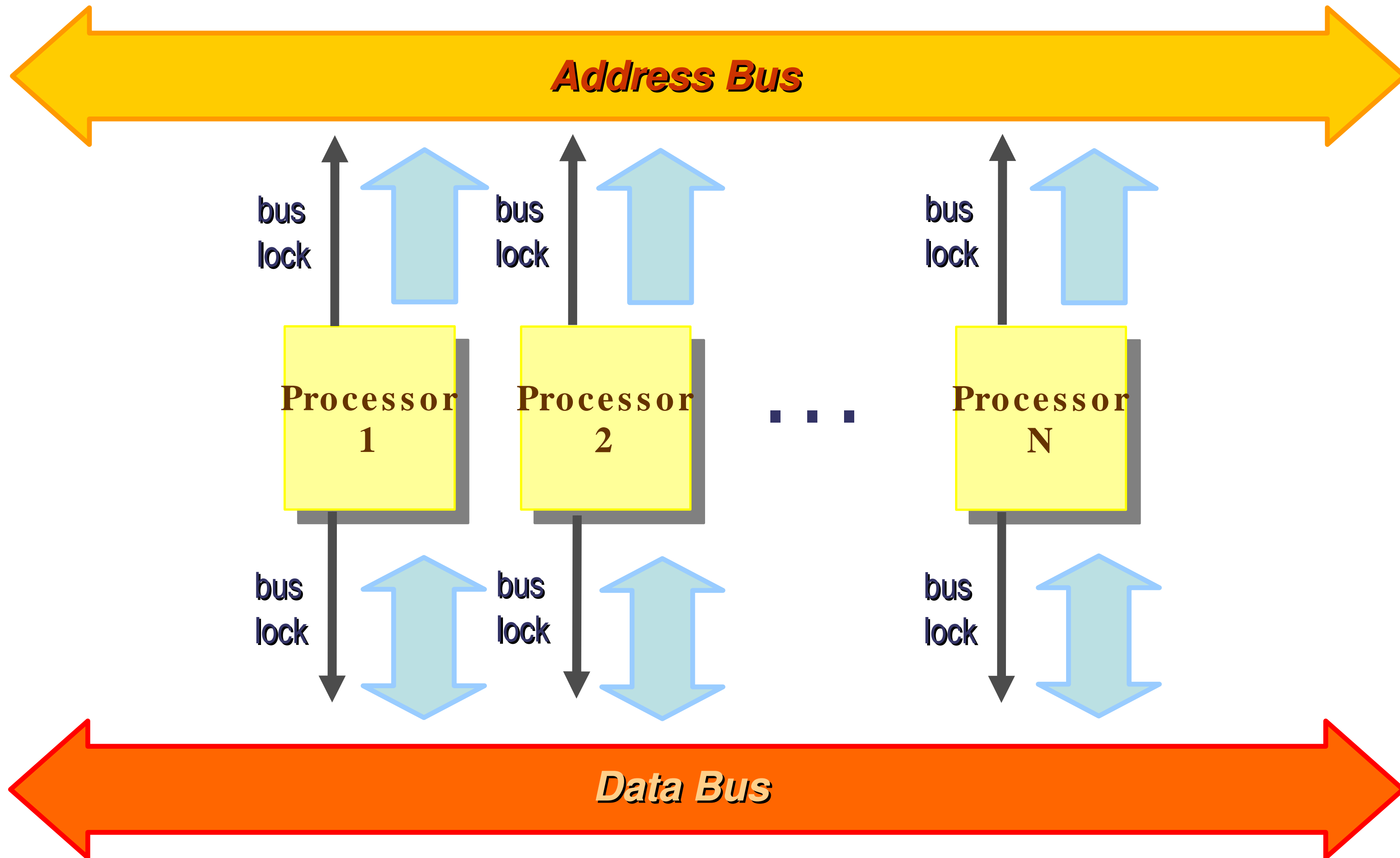
Phase 4: the interrupt controller sends on the bus the number which identifies the device raising the interrupt

Interrupts – uniprocessor scenario (4)



Phase 5: the CPU receiving the interrupt request changes the current instructions' flow by jumping to the proper interrupt handler

Symmetrical MultiProcessing architecture (SMP)



The Advanced Programmable Interrupt Controller (APIC) - (1)

In order to deliver interrupts to each CPU in the system (granting the parallelism of a smp architecture), Intel® introduced starting from Pentium III a new component, the *I/O APIC*.

In multiprocessor systems based on 80x86 architecture, each processor includes a *local APIC*.

Each l*ocal APIC* has 32 bit registers, an internal clock, a local timer device and two additional IRQ lines (LINT0 and LINT1) reserved for *local APIC* interrupts.

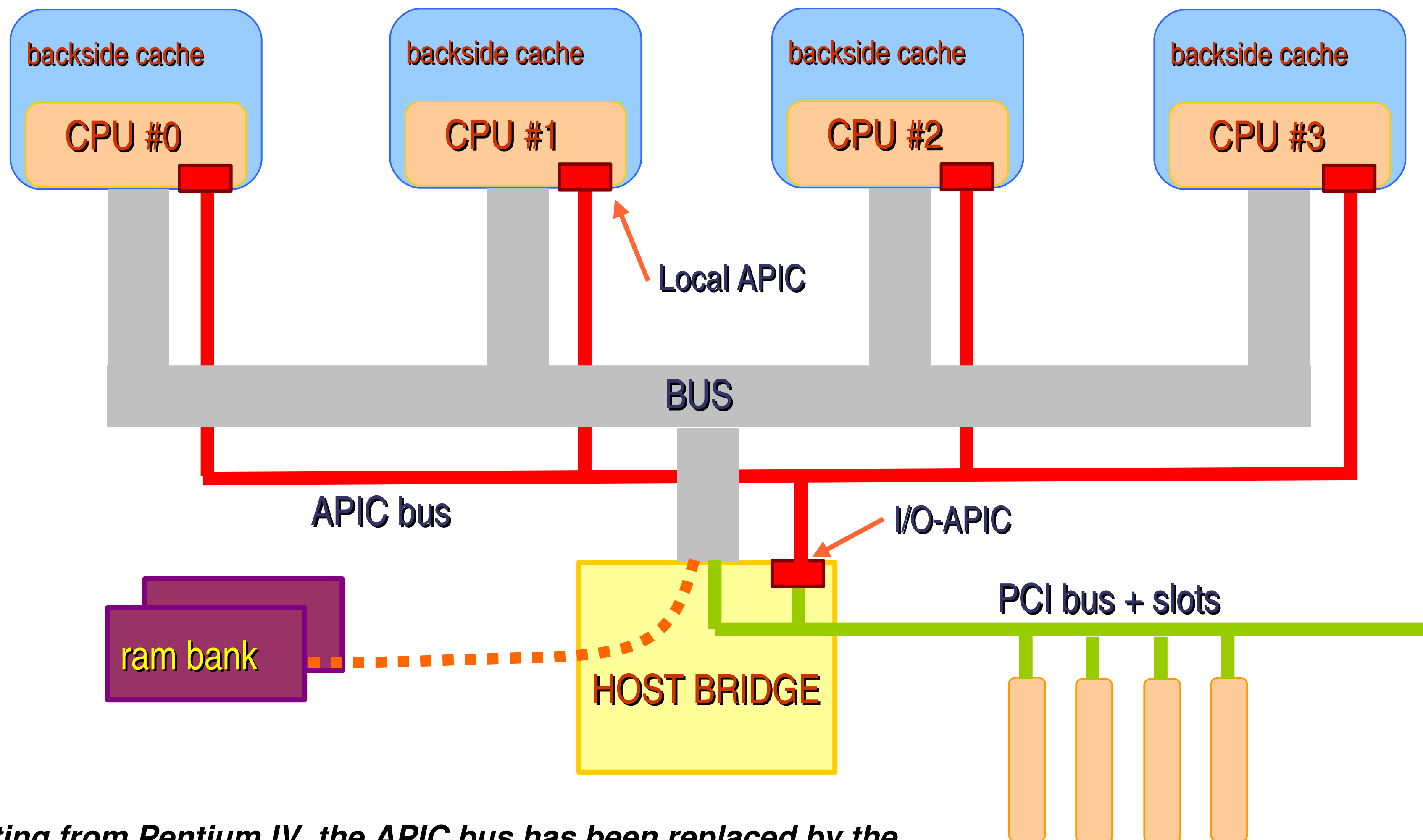
The Advanced Programmable Interrupt Controller (APIC) - (2)

The I/O APIC consists of a set of 24 IRQ lines, a 24-entry Interrupt Redirection Table, programmable registers and a message unit for sending and receiving APIC messages over the APIC bus.

Any entry in the redirection table can be individually programmed to indicate the interrupt vector and priority, the destination processor and how the processor is selected.

Generally speaking, the I/O APIC acts like an “IRQ router” with respect to the Local APICs.

APIC Overview in a quad processor architecture



**** starting from Pentium IV, the APIC bus has been replaced by the system bus***

How the I/O APIC ditributes irqs among the CPUs

Static distribution

The I/O APIC sends the IRQ signal according to the redirection table. The interrupt can be delivered to one specific CPU, to a subset of CPUs, or to all CPUs at once (broadcast mode).

Dynamic distribution

The IRQ signal is delivered to the local APIC of the processor that is executing the process with the lowest priority.

The I/O APIC consists of a set of 24 IRQ lines, a 24-entry Interrupt Redirection Table, programmable registers and a message unit for sending and receiving APIC messages over the APIC bus.

How the I/O APIC ditributes irqs among the CPUs - (cont.)

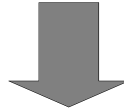
Every Local APIC has a programmable task priority register (TPR), which is used to compute the priority of the currently running process. Intel expects this register to be modified in an operating system kernel at every task switch.

If two or more CPUs share the lowest priority, the load is distributed between them using the ***arbitration technique***:

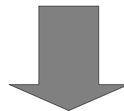
- each CPU has an arbitration priority ranging from 0 to 15 (highest)
- everytime an interrupt is delivered to a CPU, its priority is set to 0 while the priority of the other CPUs is increased by 1
- when the arbitration priority register becomes greater than 15, it is set to the previous arbitration priority of the winning CPU increased by 1

Problems start with Intel® Pentium 4 processor family

The Pentium 4 local Apic doesn't have an arbitration priority register and the mechanism is hidden in the bus arbitration circuitry



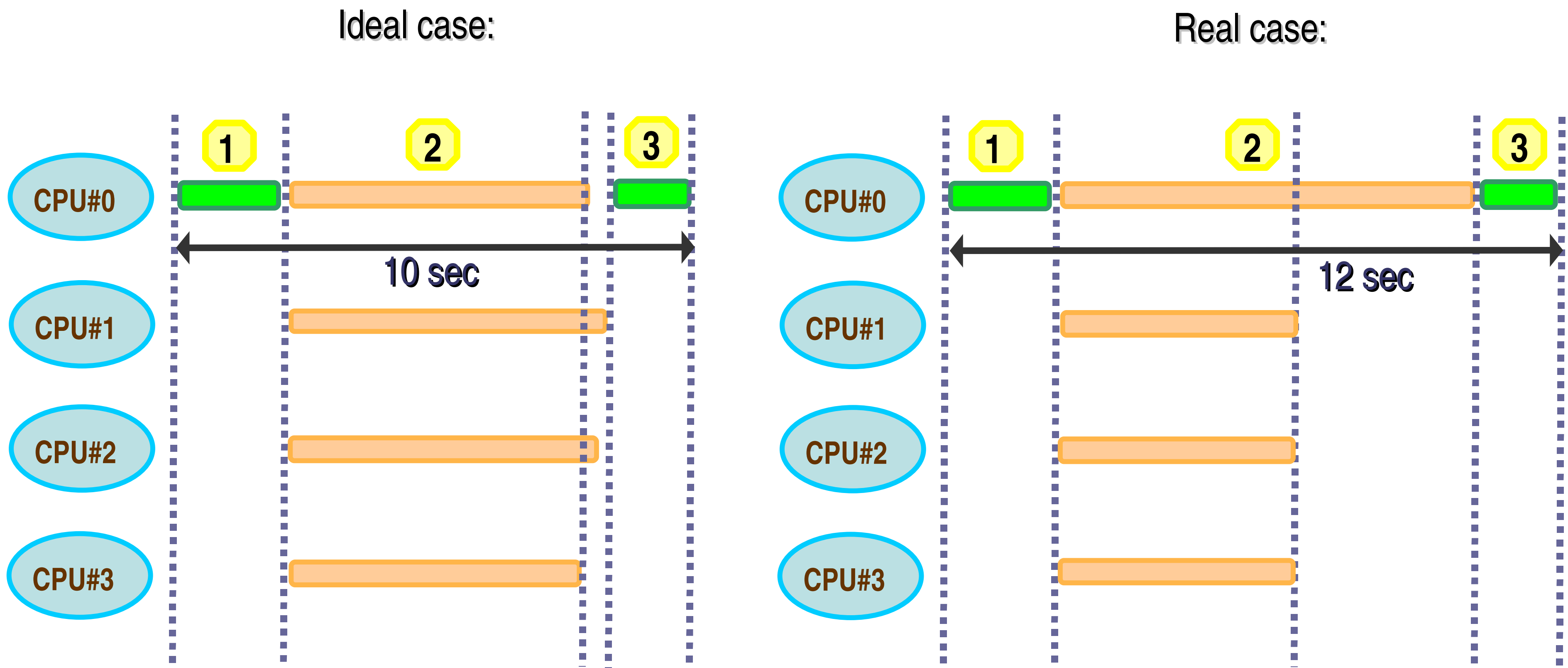
If the Operating System kernel does not regularly update the TPRs, performance may be suboptimal because interrupts might always be served by the same CPU!



Integrating this mechanism inside the kernel can be a critical task

The impact of load imbalance in massively parallel applications

In massively parallel application the delay in a job execution slows down the entire calculus due to a synchronization phase.



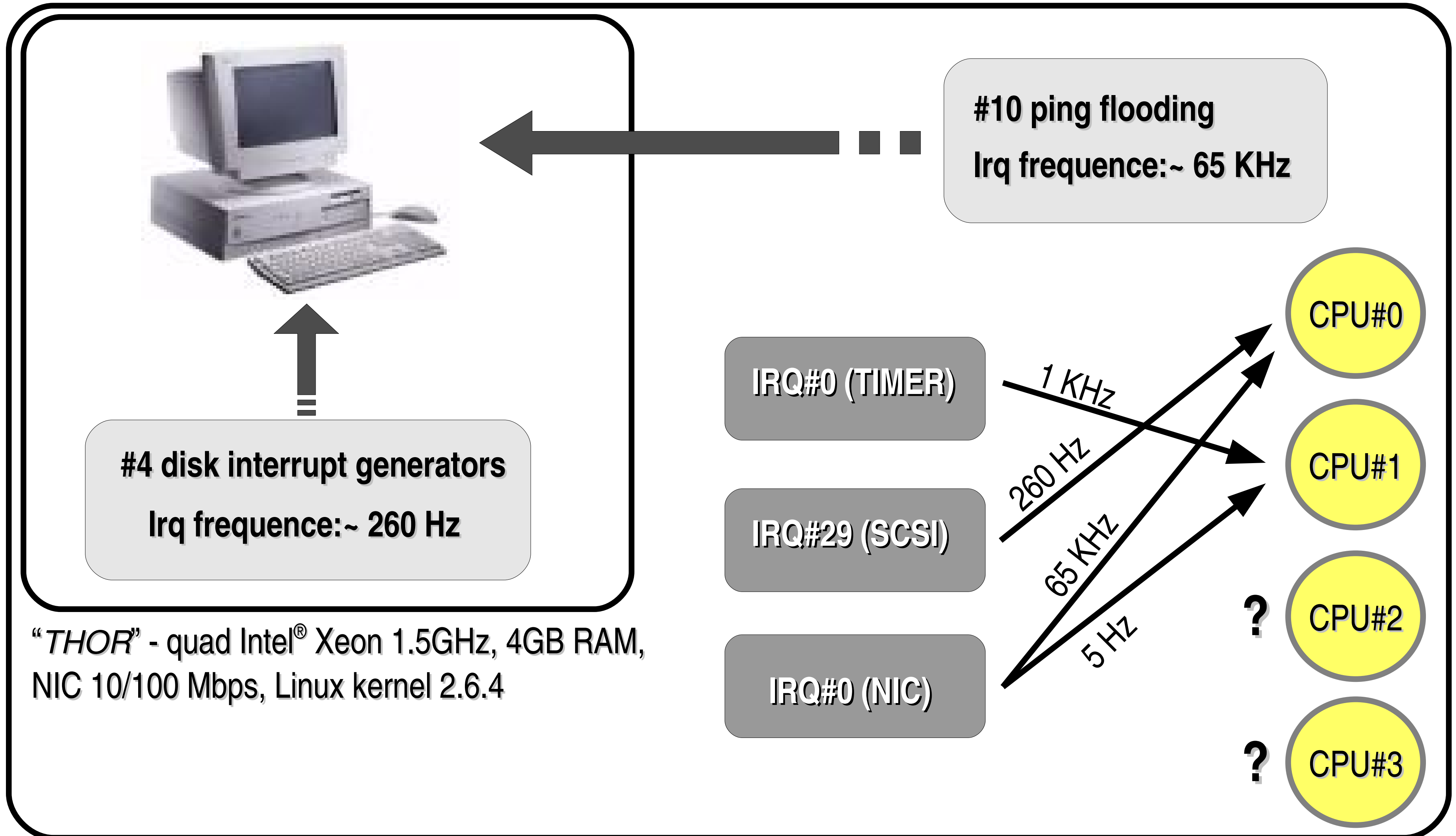
The current kernel implementation

- Developed and submitted by Nitin Kamble (Intel® Corporation)
- Integrated into the kernel starting from version 2.5.52 as a kernel thread named “irqd”

Problems related to this mechanism:

- *IRQs are not migrated if the interrupt rate is below an high treshold (usually, hundreds of interrupts per second)*
- *Even when the threshold is reached, irq balancing is suboptimal (see the next slide)*

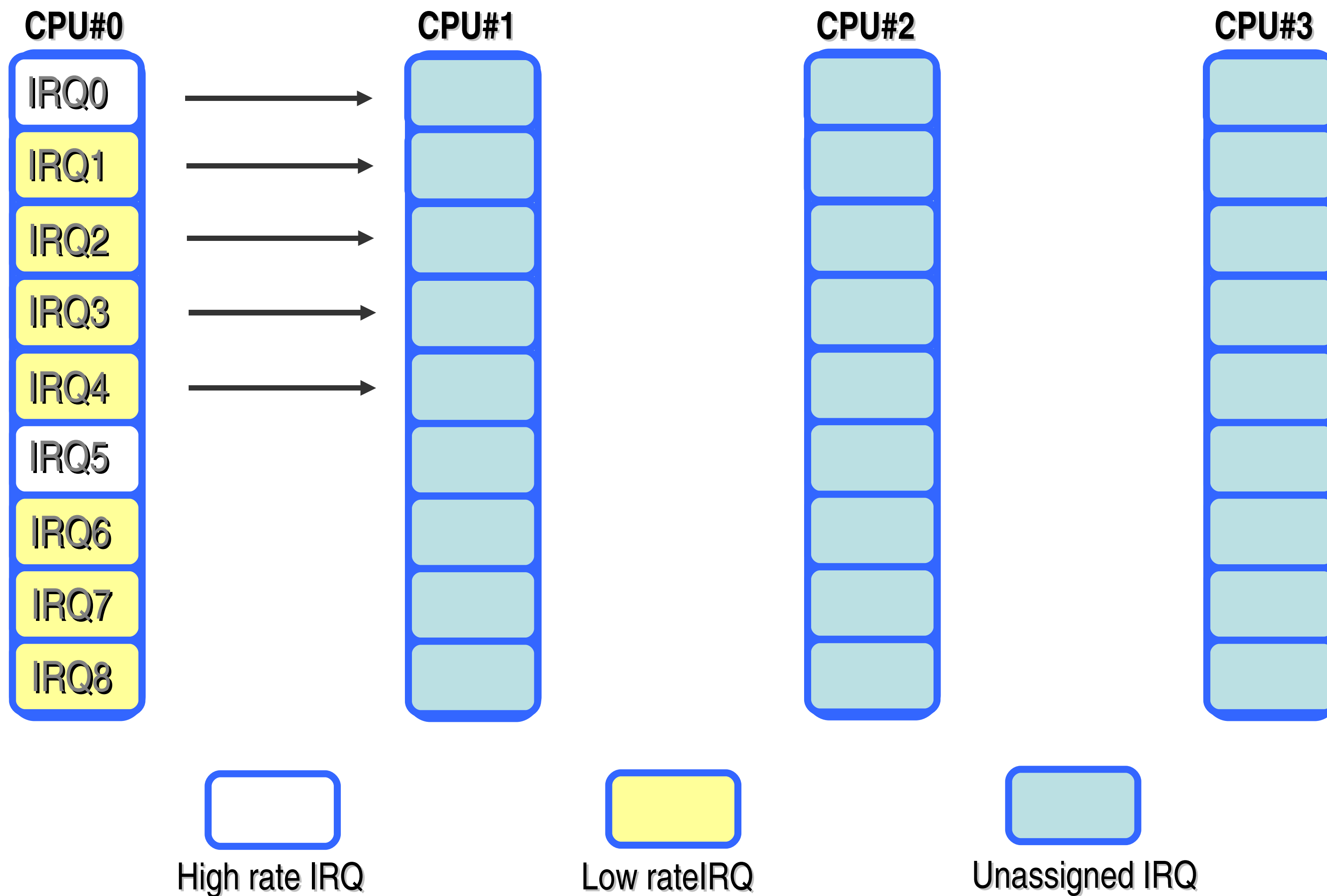
Example of bad irq distribution even under heavy irq load



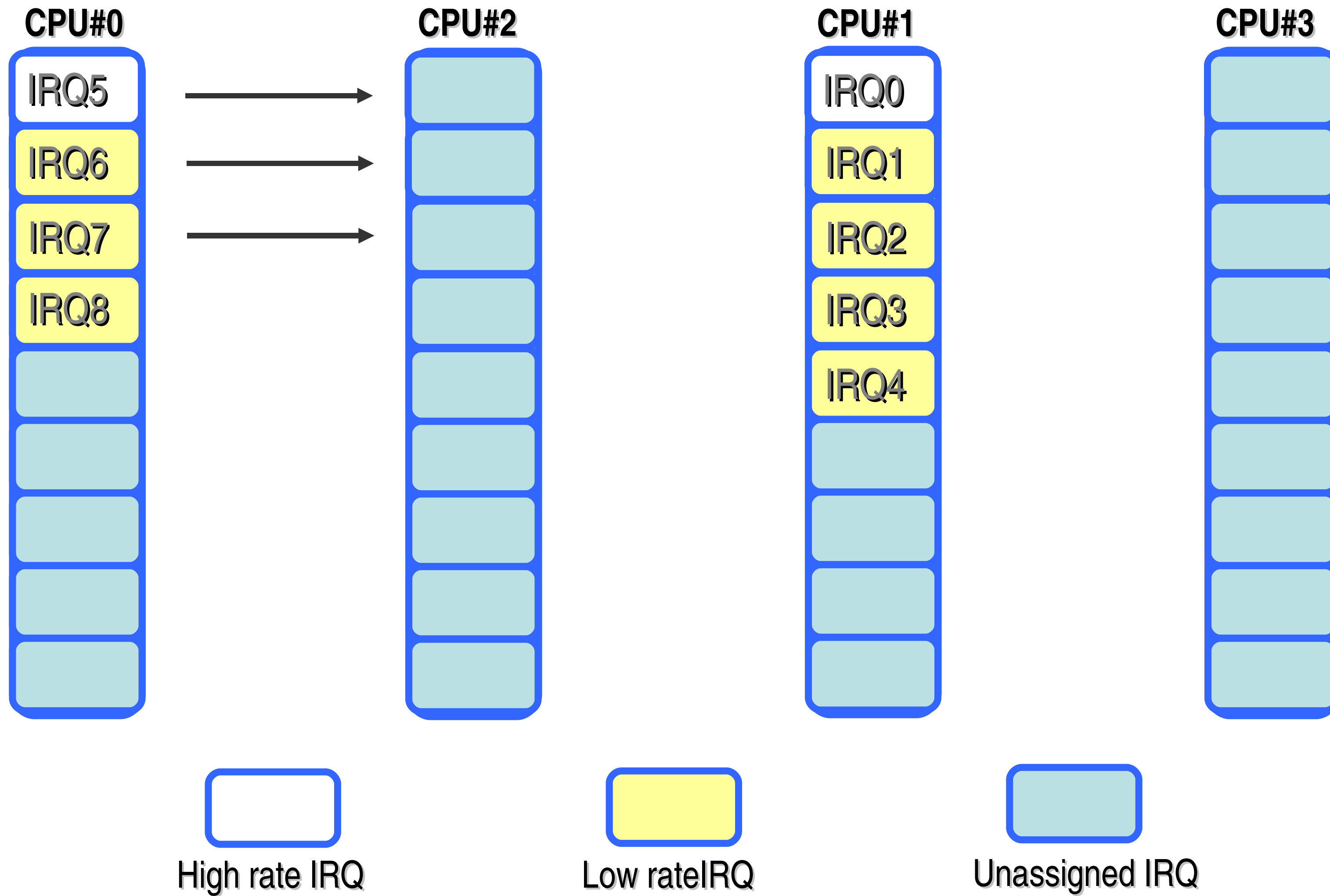
New irq distribution mechanism (experimental)

- Implemented as kernel thread in Linux kernel 2.6.4
- Intel Hyperthreading technology aware (physical CPU is seen by the operating system as a couple of logical CPUs).
- At every execution, the kernel thread updates the data structures and tries to find out the most and the least loaded CPU.
- An heuristic function is used in order to evaluate the cpu load related to irq traffic. This function checks both the interrupt requests raised till the last thread execution and the global ones, raised from the mechanism's startup time.
- If the most loaded CPU has “N” irq lines, the kernel thread tries to migrate the first “N/2 + 1” lines to the least loaded CPU.

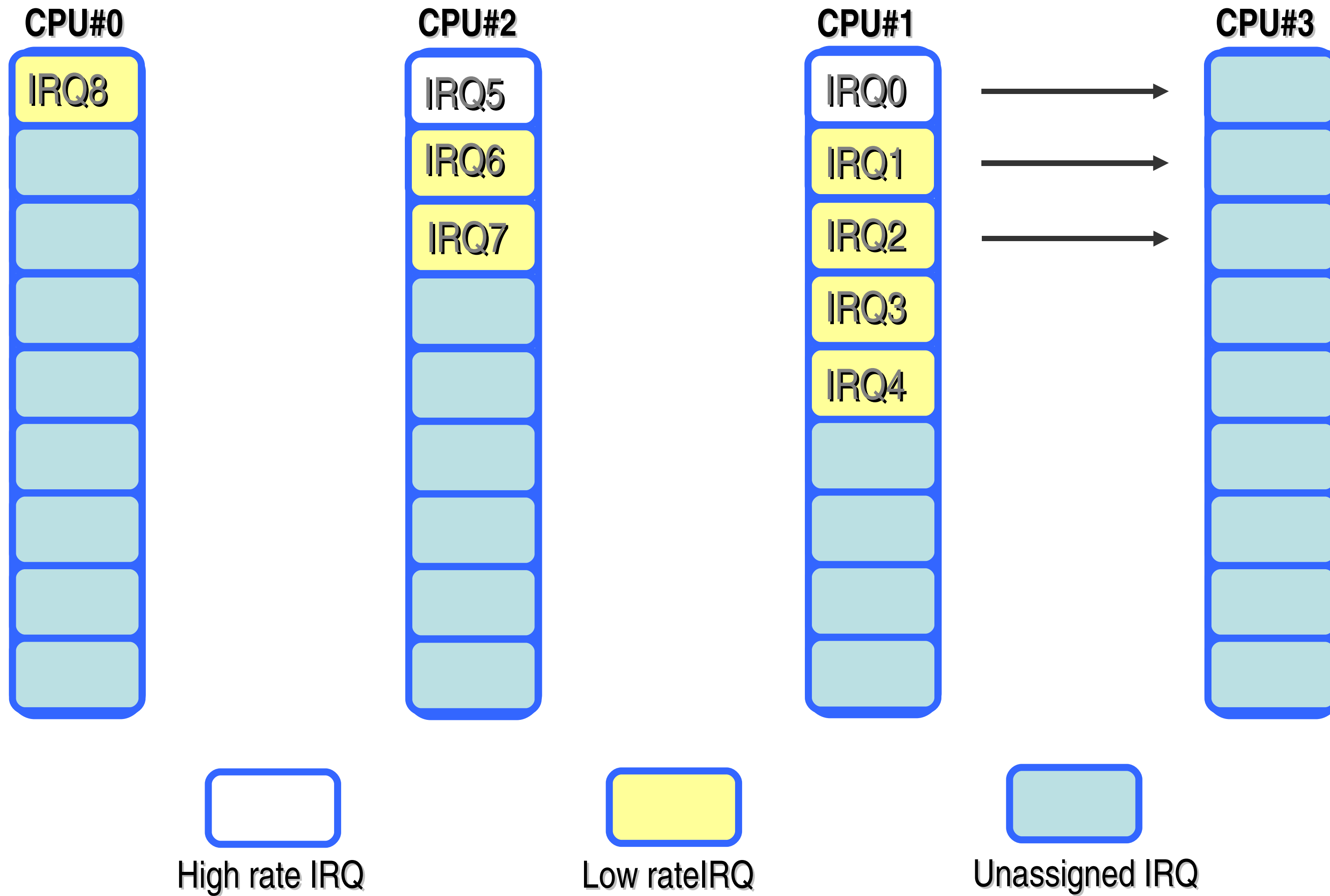
Algorithm example - (1)



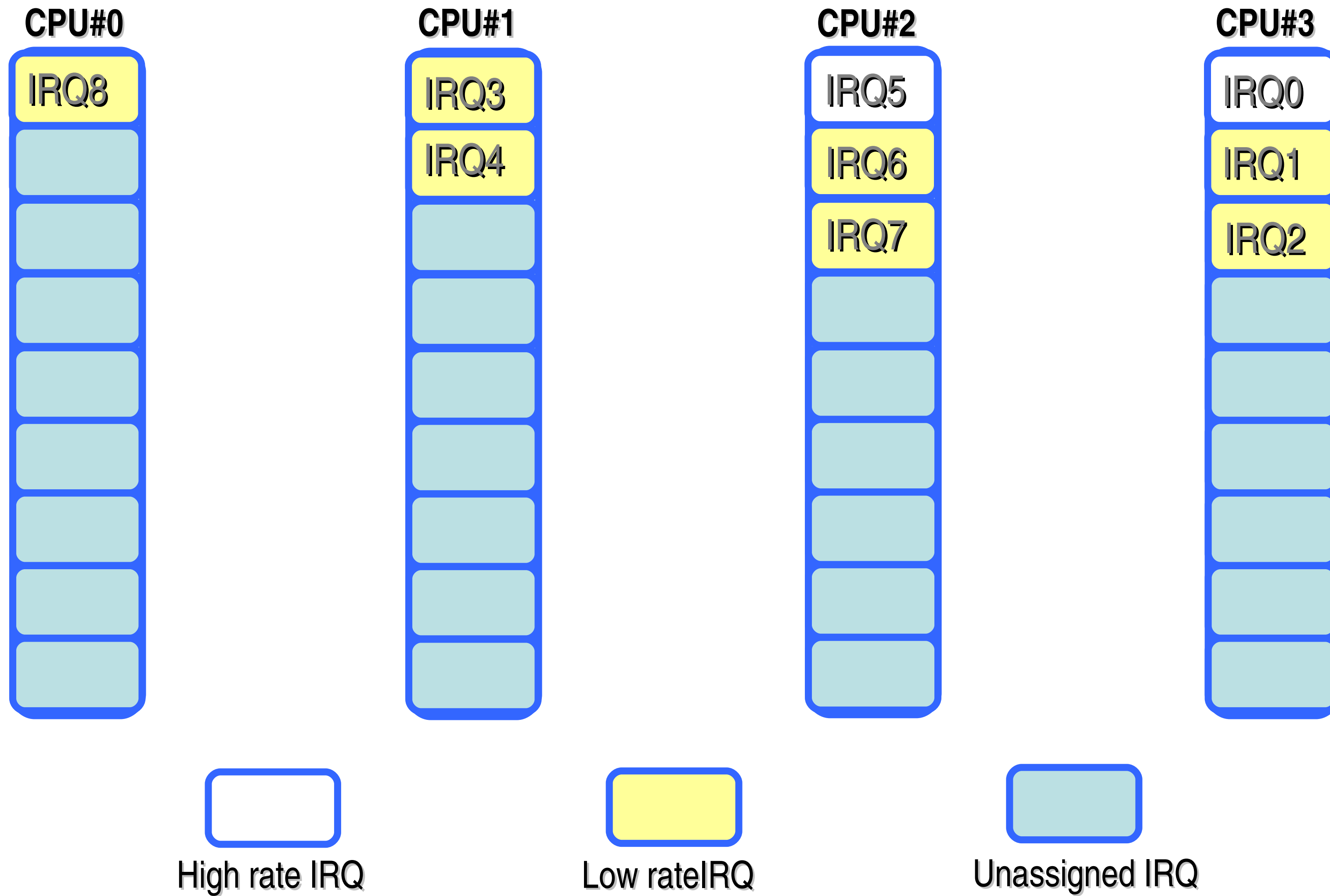
Algorithm example - (2)



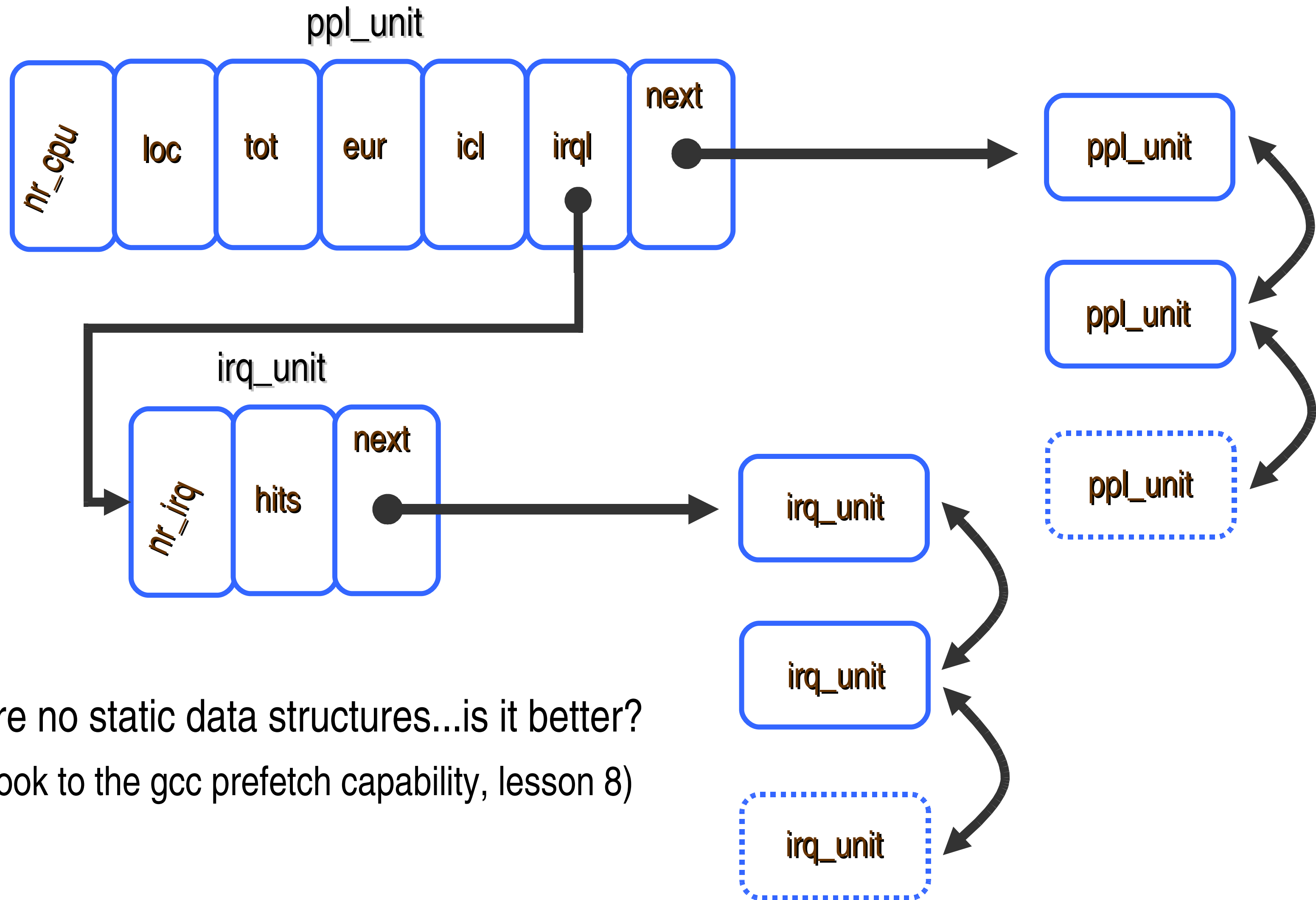
Algorithm example - (3)



Algorithm example - (4)



The new data structure

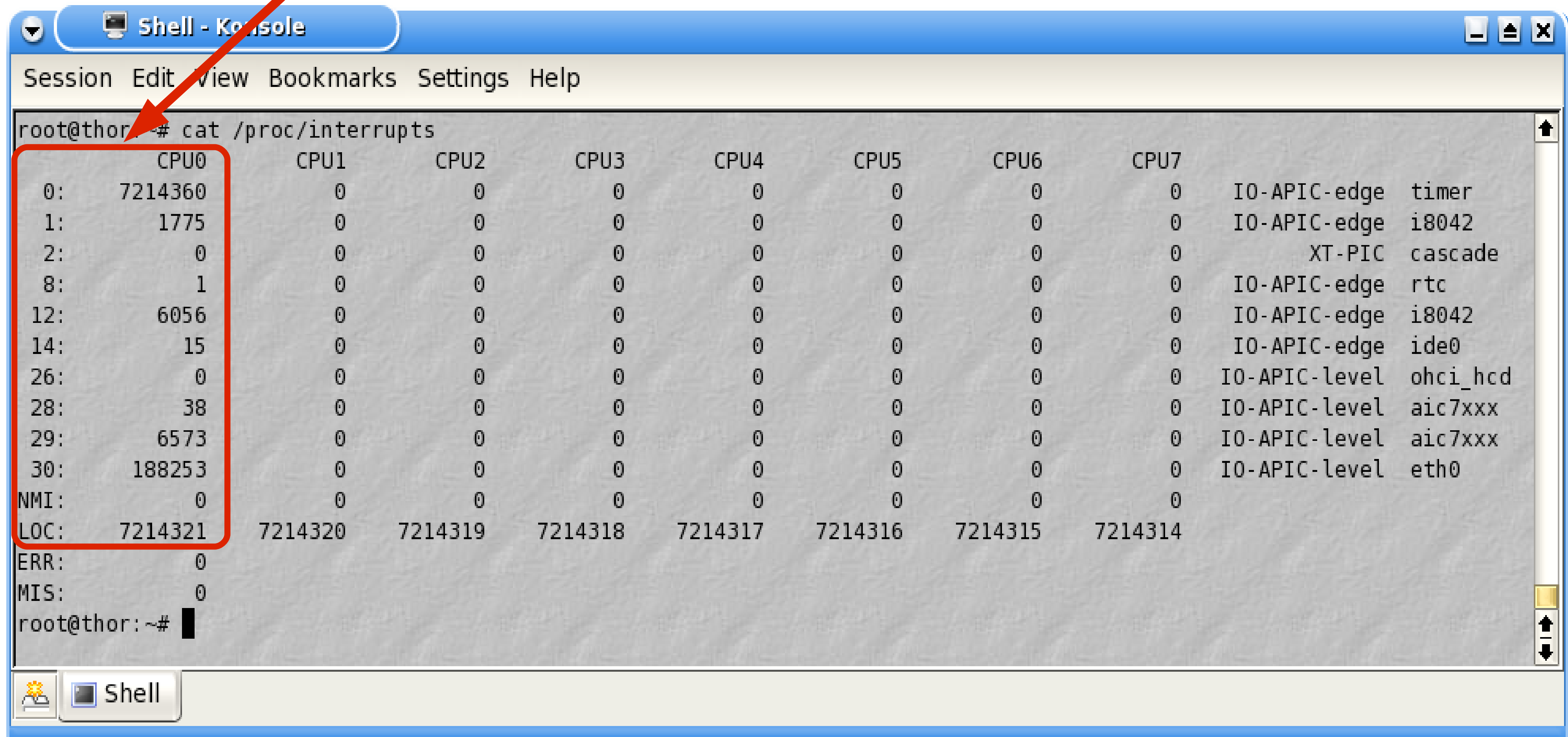


There are no static data structures...is it better?

(give a look to the gcc prefetch capability, lesson 8)

Irq distribution in kernel 2.6.4

Irq lines are binded to CPU0



Irq distribution in kernel 2.6.4irqd

Irq lines are distributed among the cpus

sibling cpus (hyperthreading enabled)

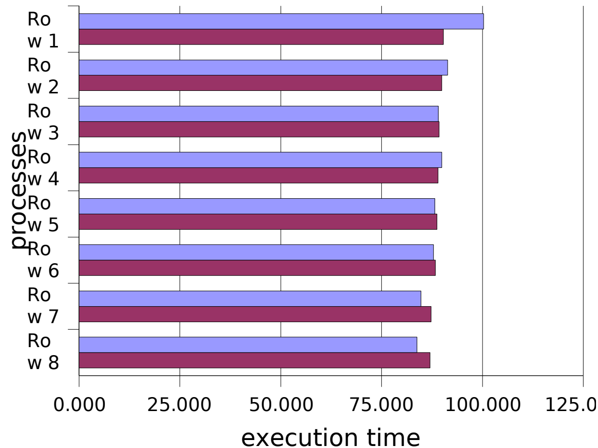
Shell - Konsole

Session Edit View Bookmarks Settings Help

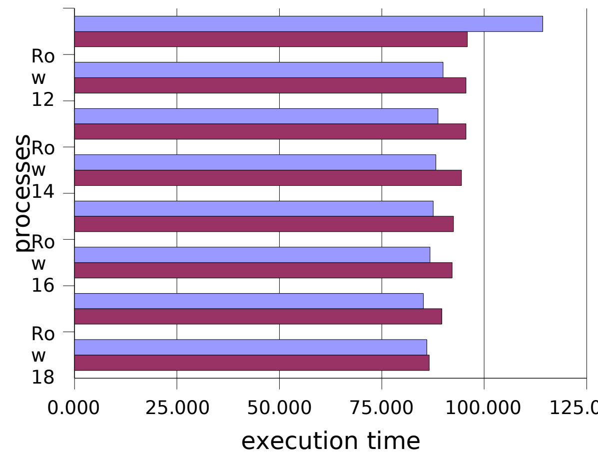
```
root@thor:~# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
  0:    2491529    2440509    2509909    2495530          0          0          0          0    IO-APIC-edge  timer
  1:        531        482        544        506          0          0          0          0    IO-APIC-edge  i8042
  2:          0          0          0          0          0          0          0          0          XT-PIC  cascade
  8:          1          0          0          0          0          0          0          0    IO-APIC-edge  rtc
 12:      12230      19026      11781      14186          0          0          0          0    IO-APIC-edge  i8042
 14:         15          0          0          0          0          0          0          0    IO-APIC-edge  ide0
 26:          0          0          0          0          0          0          0          0   IO-APIC-level  ohci_hcd
 28:          1         37          0          0          0          0          0          0   IO-APIC-level  aic7xxx
 29:        915       3588       2602       1795          0          0          0          0   IO-APIC-level  aic7xxx
 30:      73697      73856      73482      73621          0          0          0          0   IO-APIC-level  eth0
NMI:          0          0          0          0          0          0          0          0
LOC:    9936205    9936204    9936203    9936202    9936201    9936200    9936199    9936198
ERR:          0
MIS:          0
IRQD Kernel Running!-
root@thor:~#
```

Shell

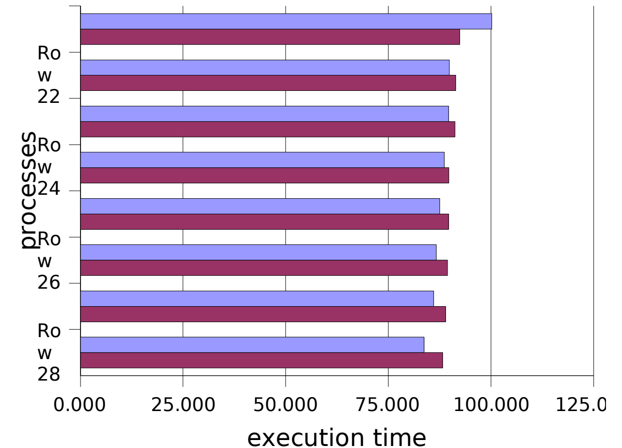
Benchmarks results for kernel 2.6.4 and 2.6.4irqd (ad-hoc mpp benchmark)



Kernel 2.6.4	Kernel 2.6.4irqd
100,290	90,280
91,382	89,911
89,080	89,278
89,883	89,002
88,180	88,715
87,844	88,358
84,781	87,254
83,794	87,020



Kernel 2.6.4	Kernel 2.6.4irqd
114,240	95,890
89,940	95,573
88,744	95,560
88,220	94,457
87,539	92,518
86,745	92,139
85,113	89,632
85,970	86,557



Kernel 2.6.4	Kernel 2.6.4irqd
100,204	92,426
89,894	91,456
89,645	91,230
88,657	89,745
87,513	89,654
86,673	89,440
86,044	88,972
83,721	88,270

Kernel 2.6.4

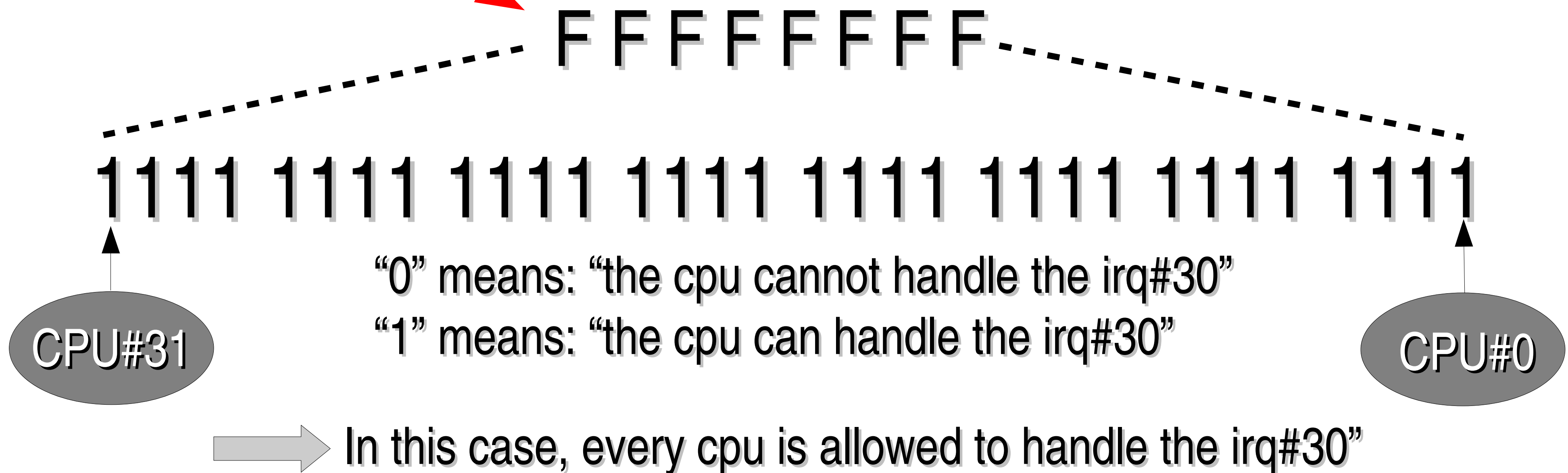
Kernel 2.6.4irqd

Hint: the smp_affinity() utility

- The `smp_affinity()` is used to assign IRQs to specific processors (or groups of processors)
- It allows you to control how your system will respond to various hardware events
- In this way you can easily redistribute the work load related to I/O devices
- started by Ingo Molnar in kernel 2.4
- some more informations related to SMP IRQ AFFINITY mechanism can be found here: `/usr/src/linux-2.X.X/Documentation/IRQ-affinity.txt`

Hint: the smp_affinity() - example (1)

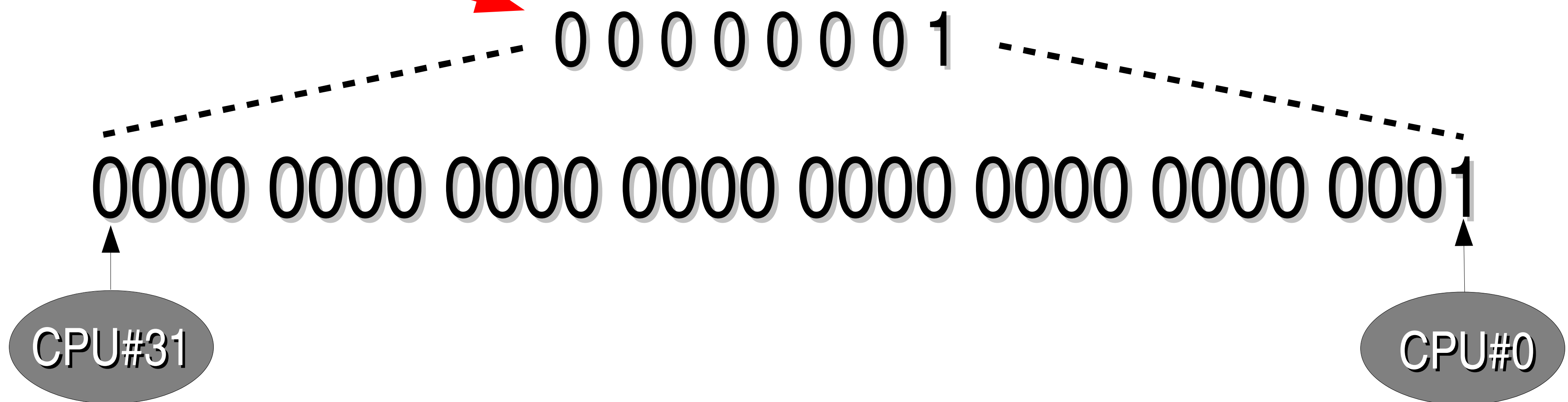
The number held in the "smp_affinity" file is presented in hexadecimal format and represents which processors any interrupts on certain irq line should be routed to.



Hint: the smp_affinity() - example (2)

Let's try to change the value stored in the smp_affinity bitmask in order to allow only cpu#0 to handle irq#30:

```
[root@archimedes /proc]# echo 1 > /proc/irq/30/smp_affinity
[root@archimedes /proc]# cat /proc/irq/30/smp_affinity
 00000001
```



Now only CPU#0 is allowed to handle the irq#30”