Advanced Operating Systems
MS degree in Computer Engineering
University of Rome Tor Vergata

Lecturer: Francesco Quaglia

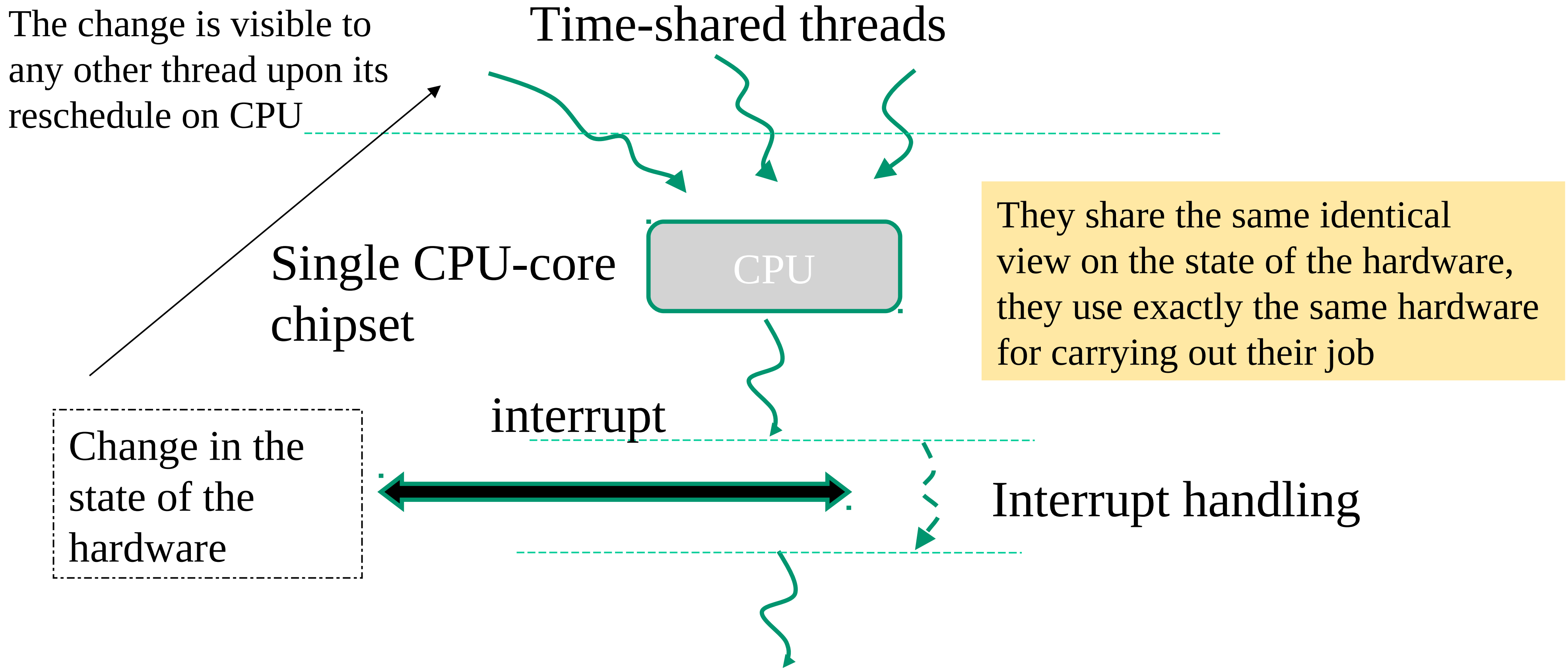
Trap/interrupt architecture

1. Architectural hints
2. Relations with software and its layering
3. Bindind to the Linux kernel internals

Single-core traditional concepts

- Traditional single-core machines only relied on
  - Traps (synchronous events wrt software execution)
  - Interrupts from external devices (asynchronous events)
- The classical way of handling the event has been based on running operating system code on the **unique CPU** in the system (single CPU systems) upon event acceptance
- This has been enough (in terms of consistency) even for concurrent (multi-thread) applications given that the state of the hardware was time-shared across threads

Some more insights



An example with traps (e.g. syscalls)

Application code

Processor state (e.g. TLB state) is A

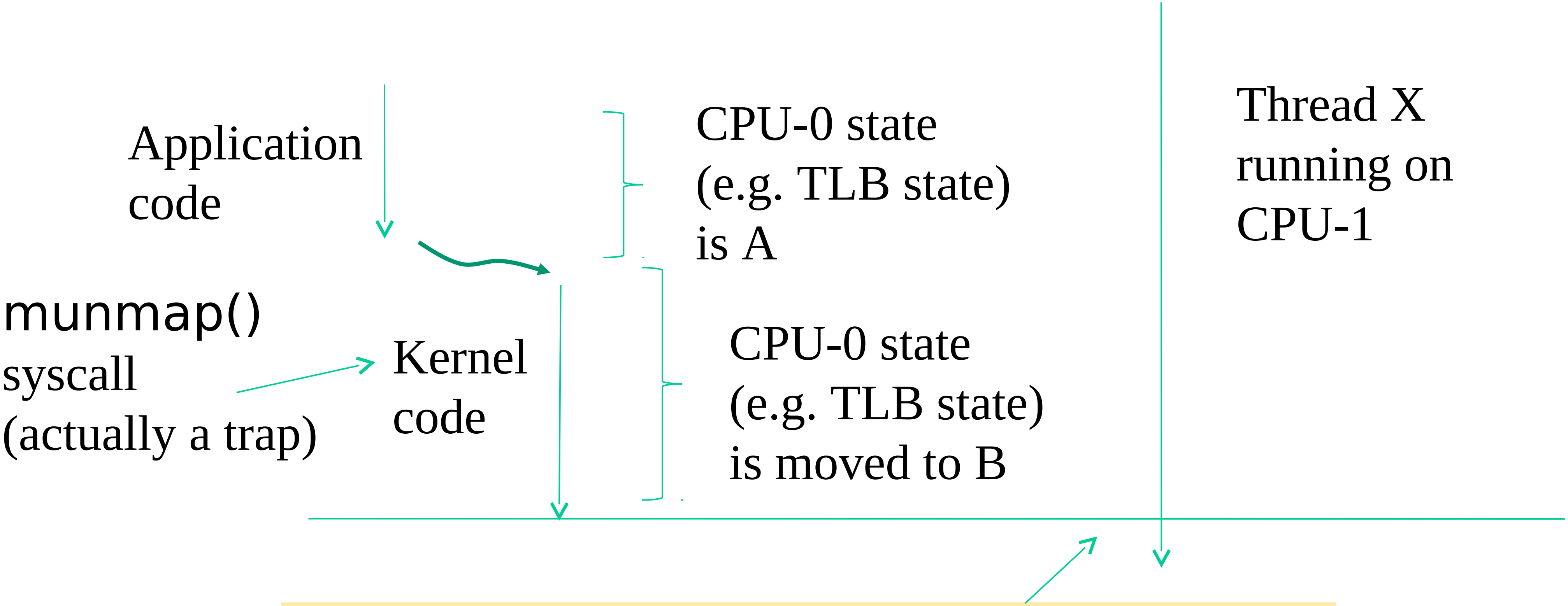
munmap()
Syscall
(actually a trap)

Kernel code

Processor state (e.g. TLB state) is moved to B

from this point any time-shared thread sees the correct final state as determined by trap handling

Moving to multi-core systems



This thread does not see state B – what if the TLB on CPU-1 caches the same page table (the same state portion) as the one of CPU-0??

Core issues

- If the system state is distributed/replicated within the hardware architecture we need mechanisms for allowing state changes by traps/interrupts to be propagated
- As an example, a trap on CPU-0 needs to be propagated to CPU-1 etc.
- In some cases this is addressed by pure firmware protocols (such as when the event **is bound to deterministic handling**)
- Otherwise we need mechanisms to propagate and handle the event at the operating system (software) level

The IPI (Inter Processor Interrupt) support

- IPI is a third type of event (beyond traps and classical interrupts) that may trigger the execution of specific operating system software on any CPU

- An IPI is a **synchronous event at the sender** CPU and an **asynchronous one at the recipient** CPU

- On the other hand, IPI is typically used to put in place cross CPU activities (e.g. request/reply protocols) allowing, e.g., a specific CPU to trigger a change in the state of another one

- Or to trigger a change on the hardware portion only observable by the other CPU

Priorities

- IPIs are generated via firmware support, <u>but are finally processed at software level (it becomes an OS matter)</u>
- Classically, at least two priority levels are admitted
  - ✓ High
  - ✓ Low
- High priority leads to immediate processing of the IPI at the recipient (a single IPI is accepted and stands out at any point in time)
- Low priority generally leads to queue the requests and process them via sequentialization

Actual support in x86 machines

- In x86 processors, the basic firmware support for interrupts is the so called APIC (Advanced Programmable Interrupt Controller)

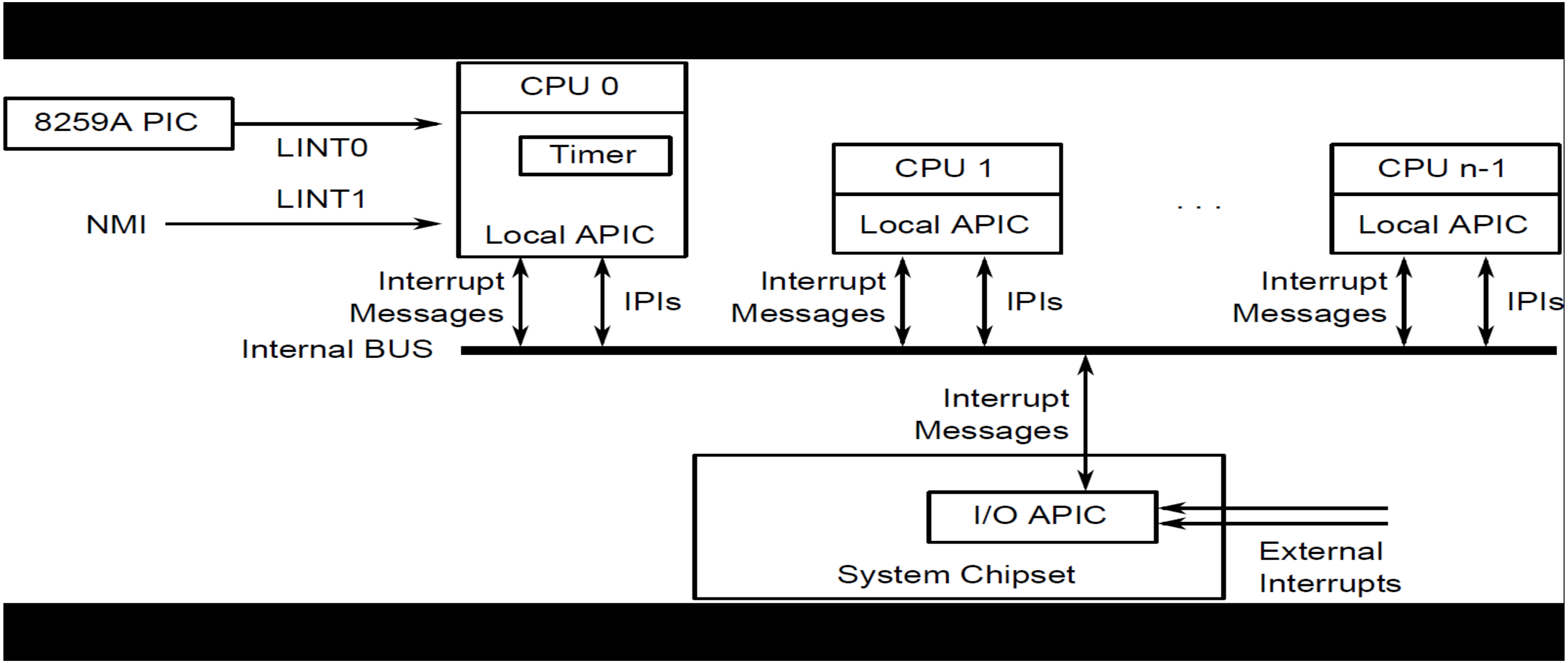
- This offers a local instance to any CPU (called LAPIC – Local APIC)

- As an example, LAPIC offers a “CPU-local” programmable timer (for time tracking and time-sharing purposes) …. the LAPIC-T we already met

- It also offers pseudo-registers to be used for posting IPI requests in the system

- IPI requests travel along an ad-hoc APIC bus

The architectural scheme



The architectural scheme evolution

PIC Intel 8259	IRQ0 - IRQ7
Two PIC Intel 8259	IRQ0 - IRQ15
IO-APIC	Max 255 physical hardware IRQ, typical system only around 24 total hardware lines

IRQ 0	System timer. Reserved for the system. Cannot be changed by a user.
IRQ 1	Keyboard. Reserved for the system. Cannot be altered even if no keyboard is present or needed.
IRQ 2	Second IRQ controller. See below for explanation.
IRQ 3	COM 2(Default) COM 4(User)
IRQ 4	COM 1(Default) COM 3(User)
IRQ 5	Sound card (Sound Blaster Pro or later) or LPT2(User)
IRQ 6	Floppy disk controller
IRQ 7	LPT1(Parallel port) or sound card (8-bit Sound Blaster and compatibles)
IRQ 8	Real time clock
IRQ 9	ACPI SCI or ISA MPU-401
IRQ 10	Free / Open interrupt / Available
IRQ 11	Free / Open interrupt / Available
IRQ 12	PS/2 connector Mouse / If no PS/2 connector mouse is used, this can be used for other peripherals
IRQ 13	Math co-processor. Cannot be changed
IRQ 14	Primary IDE. If no Primary IDE this can be changed
IRQ 15	Secondary IDE

Nomenclature

- IRQ is the actual code associated with the interrupt request (depending on hardware configuration)
- INT is the “interrupt line” as seen by the OS-kernel software
- In the essence INT = F(IRQ)
- The evaluation of the function F is typically hardware specific
- As it will be clear in a few slides, on x86 processors INT = IRQ+32
- This means that the first 32 INT lines are reserved for something else – these are the predefined traps of the hardware architecture

I/O APIC insights

- I/O APIC tracks how many CPUs are in the current chipset
- It can selectively direct interrupts to the different CPUs
- It uses so called local APIC-ID as an identifier of the CPU
- Fixed/physical operations
  - ✓ it sends interrupts from certain device to some single, predefined CPU
- Logical/low priority operations
  - ✓ it can deliver interrupts from certain device to multiple CPUs in a round robin fashion

The Linux interface for APIC

- /proc/interrupt tells the actual accounting of the interrupt delivery to the different CPUs

- /proc/irq/<IRQ#>/smp_affinity tells what is the affinity of interrupts to CPUs in the logical/low priority operating mode

- The actual setup of the I/O APIC working mode is hard-coded into kernel boot rules and is generally observable via the dmesg buffer

Linux core data structures - the IDT

- It is a table of entries that are used to describe the entry point (the GATE) for the handling of any interrupt
- x86 machines have IDTs formed by 256 entries (the max amount of IRQ vectors we can generate with the I/O APIC architecture)
- The actual size and structure of the entries depends on the type of machine we are working with (say 32 vs 64 bit machines)
- Here is a high level view of the actual usage of the entries …..

Linux IDT bindings

Back here in a while

Vector range	Use
0-19 (0x0-0x13)	Nonmaskable interrupts and exceptions
20-31 (0x14-0x1f)	Intel-reserved
32-127 (0x20-0x7f)	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls (segmented style)
129-238 (0x81-0xee)	External interrupts (IRQs)
239 (0xef)	**Local APIC timer interrupt**
240-250 (0xf0-0xfa)	Reserved by Linux for future use
251-255 (0xfb-0xff)	**Inter-processor interrupts**

The mixture changes with kernel releases (e.g. 255 is spurious)

What we already saw - idtr

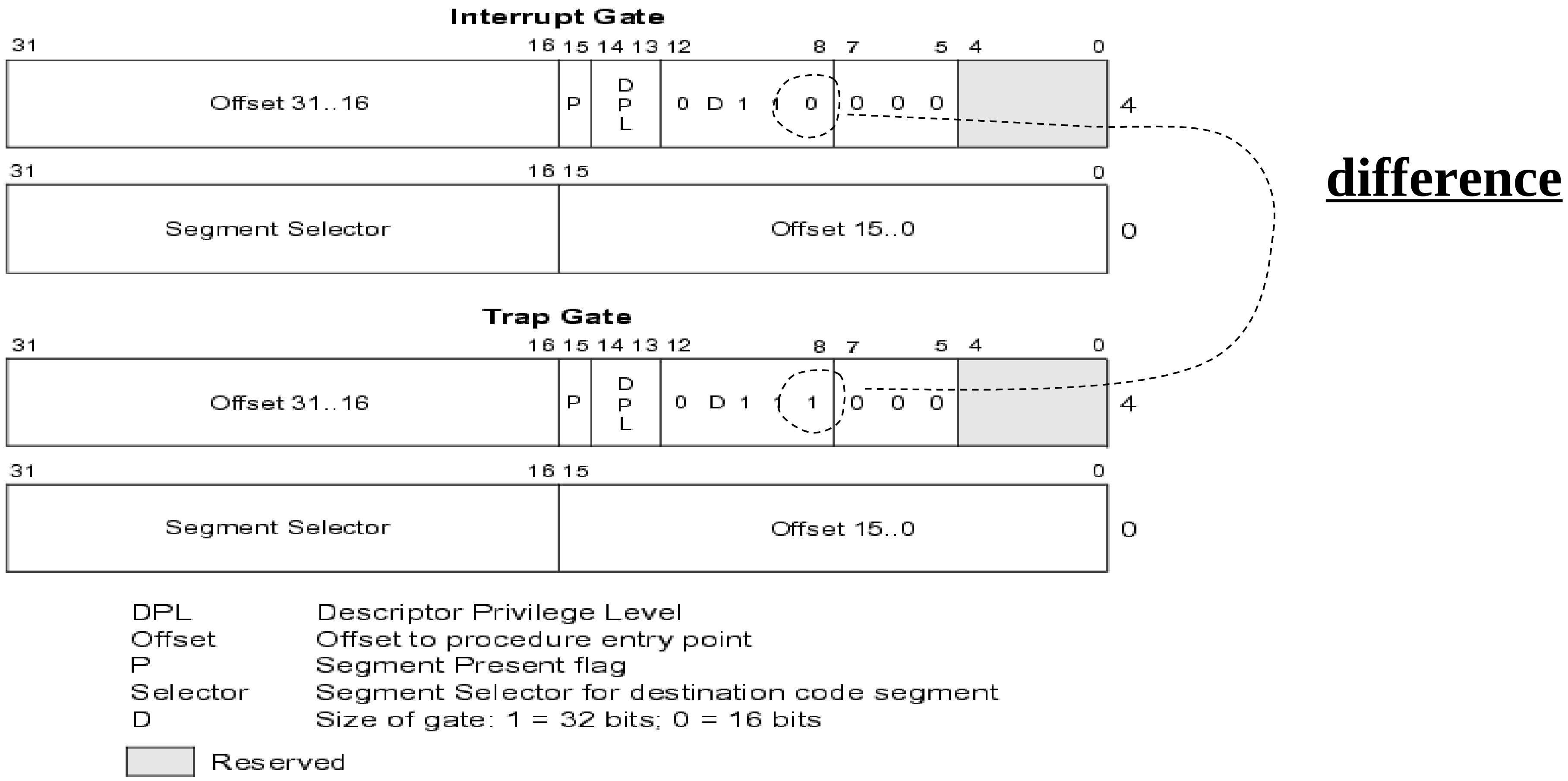
- The **idtr** register (interrupt descriptor table register) keeps on each CPU-core
  - ✓ the IDT **virtual address (expressed as up to 6 bytes – 48bit – linear address)**
  - ✓ **The number of entries currently present in the IDT (expressed as 2 bytes – up to 256)**
- This is a packed structure that we can manipulate with the LIDT (Load IDT) and SIDT (Store IDT) x86 machine instructions

x86 protected mode

- The elements of the IDT are made up by 32-bit data structures
- In more detail, the data stucture is of type `struct desc_struct`
- It is defined in `include/asm-i386/desc.h` as

```
struct desc_struct {
    unsigned long a,b;
}
```

Structure of the x86 protected mode IDT entry



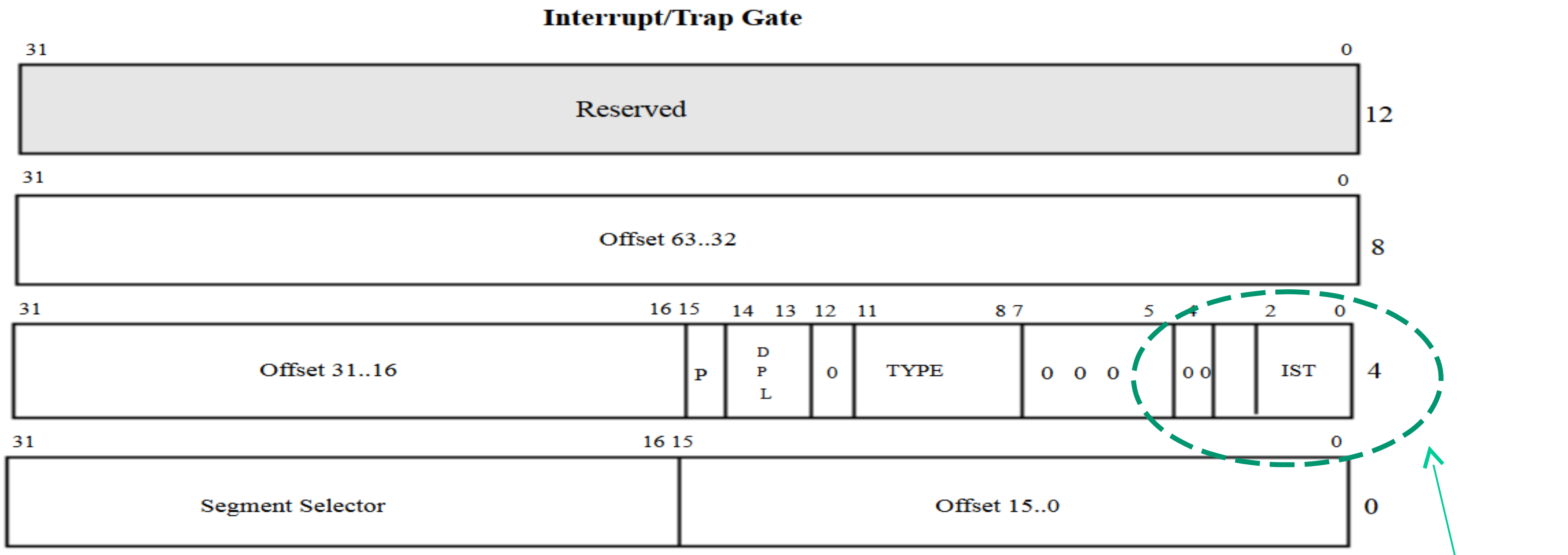
IDT entry, Interrupt Gates

<table>
<tr><th>Name</th><th>Bit</th><th>Full Name</th><th>Description</th></tr>
<tr><td>Offset</td><td>48..63</td><td>Offset 16..31</td><td>Higher part of the offset.</td></tr>
<tr><td>P</td><td>47</td><td>Present</td><td>can be set to 0 for unused interrupts or for Paging.</td></tr>
<tr><td>DPL</td><td>45,46</td><td>Descriptor Privilege Level</td><td>Gate call protection. Specifies which privilege Level the calling Descriptor minimum should have. So hardware and CPU interrupts can be protected from beeing called out of userspace.</td></tr>
<tr><td>S</td><td>44</td><td>Storage Segment</td><td>= 0 for interrupt gates.</td></tr>
<tr><td>Type</td><td>40..43</td><td>Gate Type 0..3</td><td>Possible IDT gate types :
<table>
<tr><td>0b0101</td><td>0x5</td><td>5</td><td>80386 32 bit Task gate</td></tr>
<tr><td>0b0110</td><td>0x6</td><td>6</td><td>80286 16-bit interrupt gate</td></tr>
<tr><td>0b0111</td><td>0x7</td><td>7</td><td>80286 16-bit trap gate</td></tr>
<tr><td>0b1110</td><td>0xE</td><td>14</td><td>80386 32-bit interrupt gate</td></tr>
<tr><td>0b1111</td><td>0xF</td><td>15</td><td>80386 32-bit trap gate</td></tr>
</table></td></tr>
<tr><td>0</td><td>32..39</td><td>Unused 0..7</td><td>Have to be 0.</td></tr>
<tr><td>Selector</td><td>16..31</td><td>Selector 0..15</td><td>Selector of the interrupt function (to make sense - the kernel's selector). The selector's descriptor's DPL field has to be 0.</td></tr>
<tr><td>Offset</td><td>0..15</td><td>Offset 0..15</td><td>Lower part of the interrupt function's offset address (also known as pointer).</td></tr>
</table>

Recap on relations with the GDT

- The segment identifier/selector allows accessing the entry of the GDT where we can find the base value for the target segment
- **NOTE**
  - ➢ As we already know, there are 4 valid data/code segments, all mapped to base 0x0
  - ➢ This is done in order to make **Linux portable on architectures offering no segmentation support** (i.e. only offering paging)
  - ➢ This is one reason why
    - ✓ Protection meta-data are also kept within page table entries
    - ✓ Setting up the offset for a GATE requires a **displacement referring to 0x0**, which can be denoted to the linker by the & operator

Long mode IDT entry structure



DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
IST	Interrupt Stack Table

Accessing the gate address - long mode

```
#define HML_TO_ADDR(h,m,l)   \
   ((unsigned long) (l) | ((unsigned long) (m) << 16) | \
   ((unsigned long) (h) << 32))
```

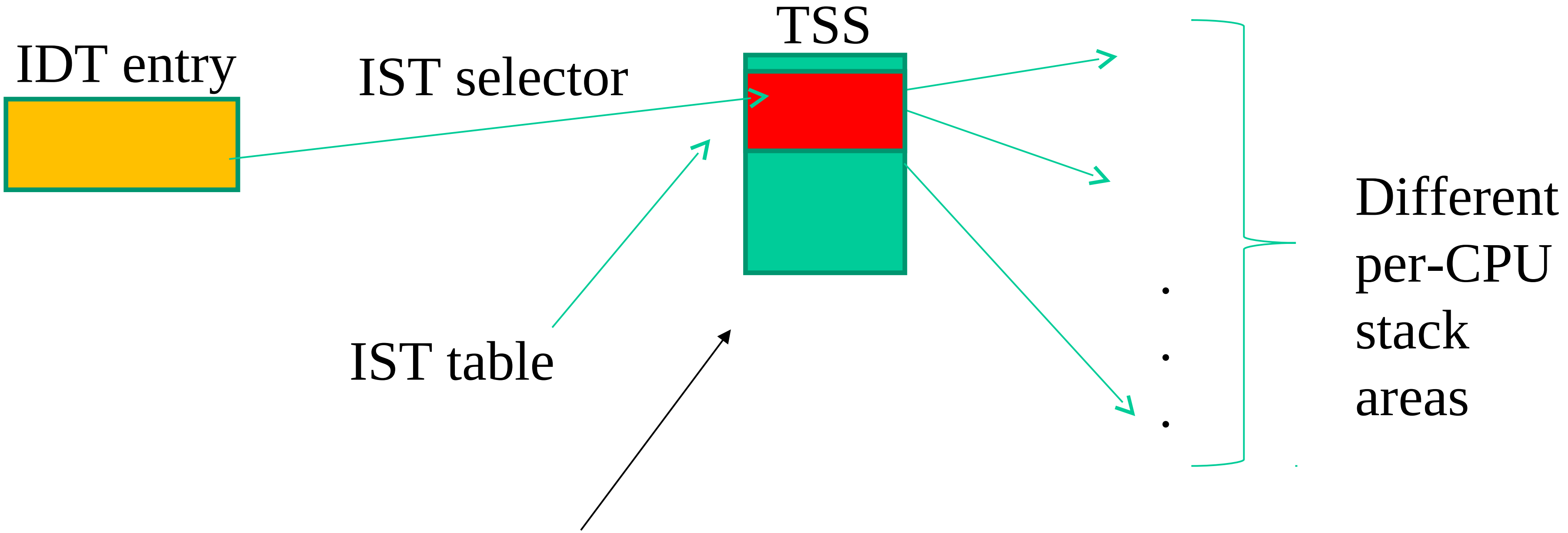
………

```
gate_desc *gate_ptr;


gate_ptr = ……;

HML_TO_ADDR(gate_ptr->offset_high, gate_ptr->offset_middle,
      gate_ptr->offset_low);
```

x86 long mode fully new concepts - IST

- The **Interrupt Stack Table (IST)** is available as an alternative to handle stack switch upon traps/interrupts
- This mechanism unconditionally switches stacks when it is enabled on each individual interrupt-vector basis using a field in the IDT entry
- This means that some interrupt vectors can selectively use the IST mechanism
- IST provides a method for specific interrupts (such as NMI, double-fault, and machine-check) to always execute on a known good stack
- The IST mechanism provides **up to seven IST pointers** in the TSS

A scheme



These are typically the primary stacks (possibly of different size) for processing a given trap/interrupts
Software will then switch to the classical kernel level stack of the running task if nothing prevents it (e.g. a double fault)

Macros for setting IDT entries - x86 protected mode

- within the `arch/i386/kernel/traps.c` file we can find the declaration of the following macros that can be used for setting up one entry of the IDT
  - `set_trap_gate(displacement,&symbol_name)`
  - `set_intr_gate(displacement,&symbol_name)`
  - `set_system_gate(displacement,&symbol_name)`
- `displacement` indicates the target entry of the IDT
- `&simbol_name` identifies the segment displacement (starting from 0x0) which determines the address of the software module to be invoked for handling the trap or the interrupt

Main differences among the modules

- The `set_trap_gate()` function initializes one IDT entry such in away to define the value 0 as the privilege level admitted for accessing the GATE via software

- Therefore we cannot rely on the INT assembly instruction unless we are already executing in kernel mode

- The `set_intr_gate()` function looks similar, however the handler activation relies on interrupt masking

- `set_system_gate()` is similar to `set_trap_gate()` however it defines the value 3 as the level of privilege admitted for accessing the GATE

i386/kernel-2.4 examples

Handler managing division errors

```
set_trap_gate(0,&divide_error)
```

Handler for non-maskable interrupts

```
set_intr_gate(2,&nmi)
```

Handler used for dispatching system calls

```
set_system_gate(SYSCALL_VECTOR,&system_call)
```

Variants for x86 long mode - kernel 3

CODE SNIPPET FROM desc.h

```
/*
 * This routine sets up an interrupt gate at directory privilege level 3.
 */
static inline void set_system_intr_gate(unsigned int n, void *addr)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_INTERRUPT, addr, 0x3, 0, __KERNEL_CS);
}

static inline void set_system_trap_gate(unsigned int n, void *addr)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_TRAP, addr, 0x3, 0, __KERNEL_CS);
}

static inline void set_trap_gate(unsigned int n, void *addr)
{
        BUG_ON((unsigned)n > 0xFF);
        _set_gate(n, GATE_TRAP, addr, 0, 0, __KERNEL_CS);
}
```

Variants for x86 long mode - kernel 4/5

```
#define write_ldt_entry(dt, entry, desc) \
        native_write_idt_entry(dt, entry, desc)

static inline void native_write_idt_entry(gate_desc *idt,
        int entry, const gate_desc *gate) {
            memcpy(&idt[entry], gate, sizeof(*gate));
}
```

Reserved vs available IDT entries (i)

- The entries from 0 to 31 are reserved for handlers that are used to manage specific (predefined) events/conditions (such as divide by 0 or page fault) or are already planned for future use … these are mostly traps

- This is based on hardware design/requirements

- All the other entries are available for system programming purposes

- As an example, the entry at displacement 0x80 has been traditionally used for kernel level access via system calls

- We note that for some of the reserved entries, microcode tasks generate a **so called error-code to be passed to the handler** ……

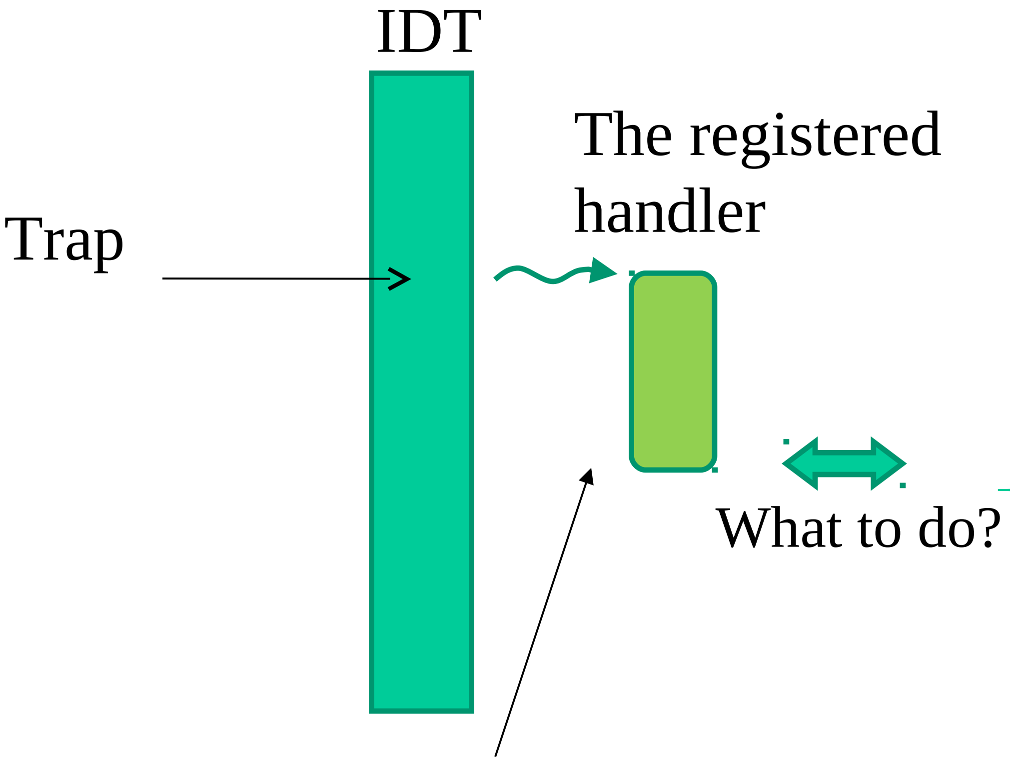
Reserved vs available IDT entries (ii)

- If needed, the handler needs to be structured such in a way to be aware of the production of the error-code

- Particularly, beyond exploiting the error-code value, it needs to remove it from, e.g., the stack right before returning from trap/interrupt (IRET)

- Non-reserved entries area managed by the microcode with no generation of any error-code value

Recap on actions of trap/interrupt handlers



CPU snapshot generation on the stack? YES

Management of the presence/absence of error code? YES

Additional stack change? YES/NO

Control passage to a second level handler? Typically YES (for sure if we are running a non top/bottom half task)

In modern kernels we also have the need for handling kernel isolation on page tables

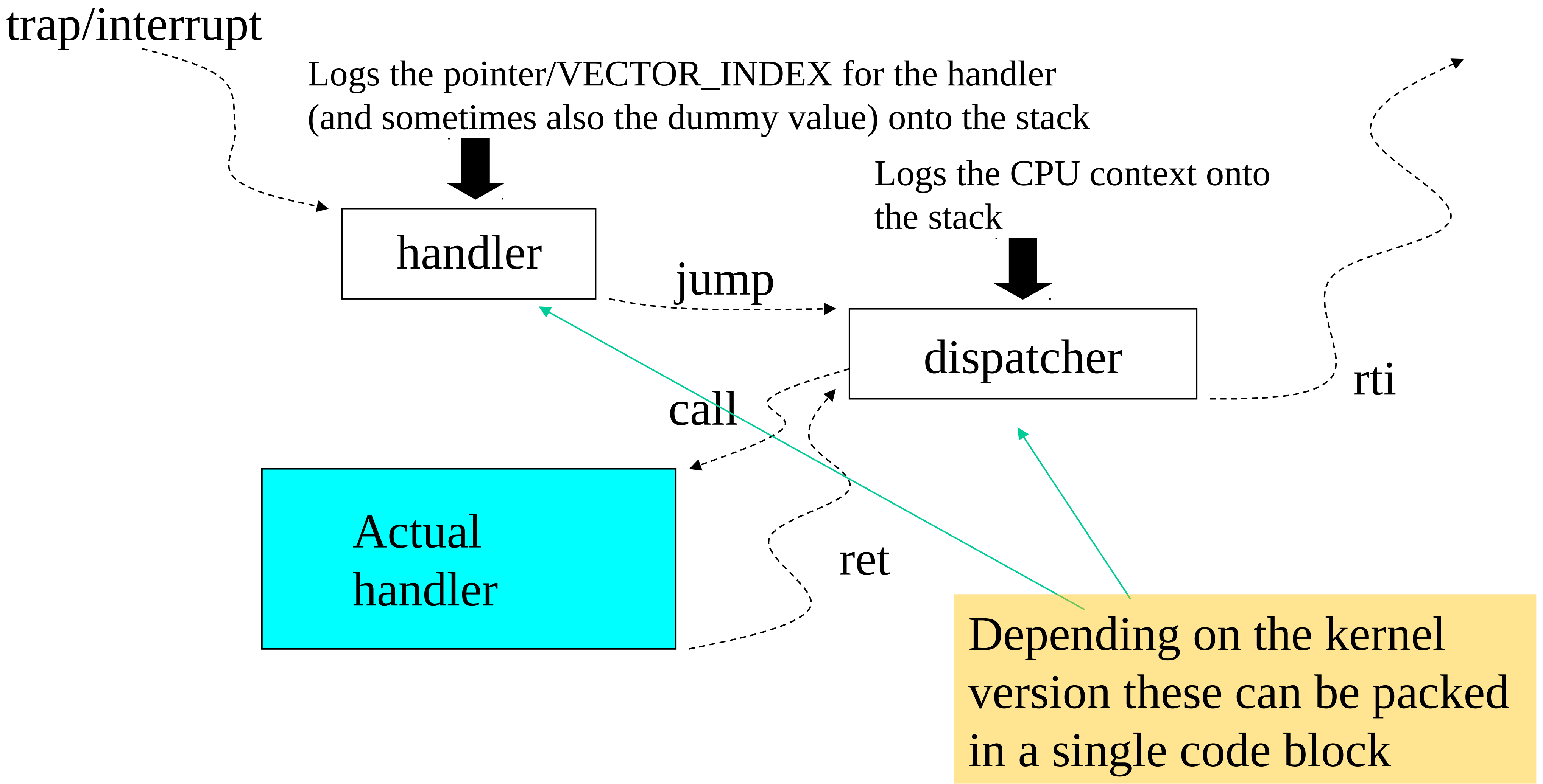
Modular handler management (i)

- The interrupt handlers are managed via an additional dispatcher
- Initially, each handler logs a dummy-value into the stack in case no error-code is generated in relation to the specific trap/interrupt
- Then it logs into the stack the address of the actual handler-function (typically written in C)
- In more modern versions we log a VECTOR_INDEX for access to the vector of function pointers …….

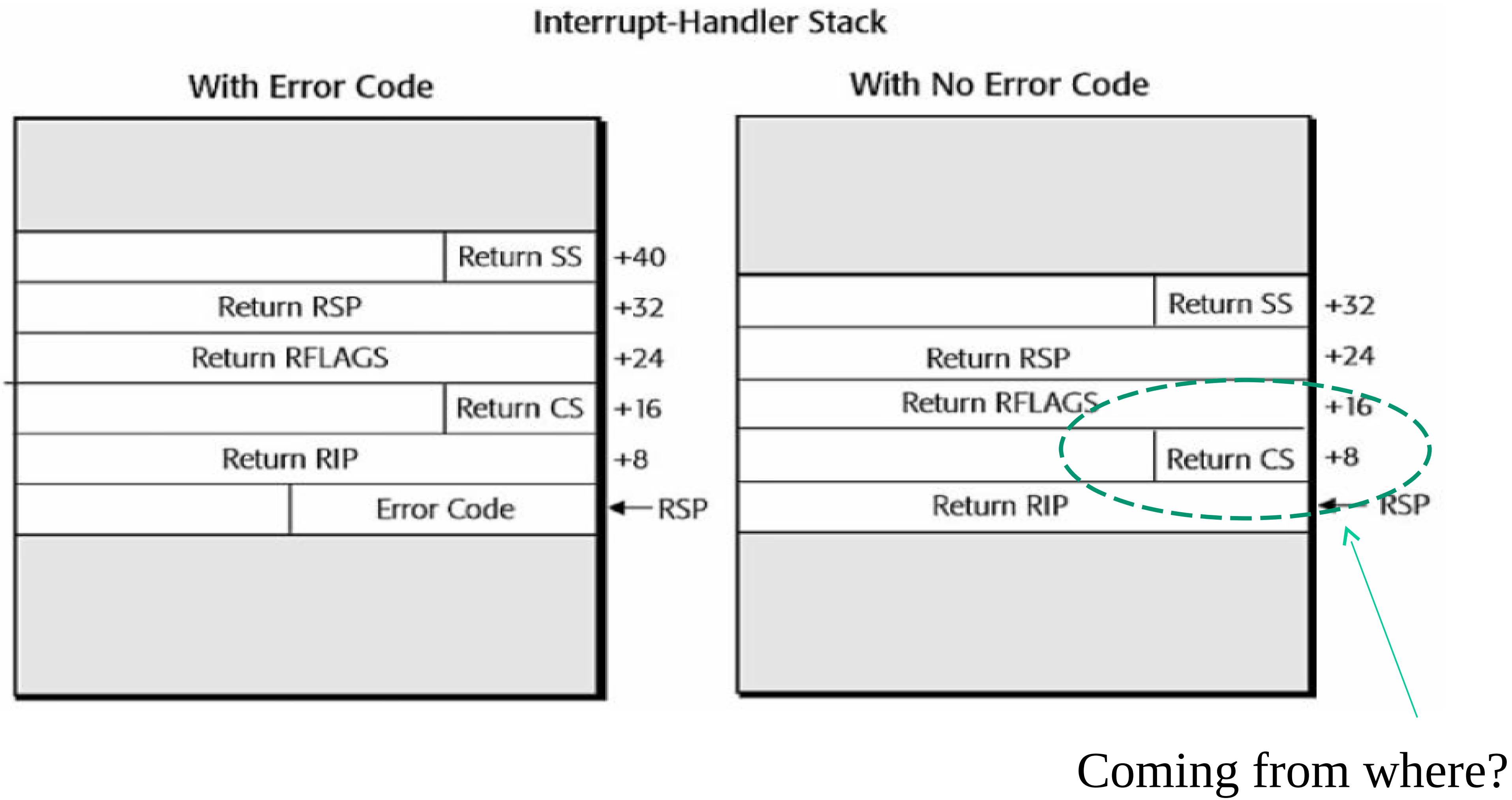
Modular handler management (ii)

- After, an assembly module, operating the dispatching, is activated
- This logs the CPU context and gives control to the handler via a conventional call
- Given that the input parameters are passed via the stack, the handlers will need to be compiled with `asmlinkage` directives (or more modern `dotraplinkage`)
- … in more modern Linux kernel flavors (**e.g. for x86 long mode**), the layering is a bit more articulated, but the basic concepts are the same
- One thing which is dealth with explicitly is IST and the stack frame redirection

The actual scheme



x86-64 early trap/interrupt stack layout details



Examples (dated)

```
ENTRY(overflow)
        pushl $0
        pushl $ SYMBOL_NAME(do_overflow)
        jmp error_code

ENTRY(general_protection)
        pushl $ SYMBOL_NAME(do_general_protection)
        jmp error_code

ENTRY(page_fault)
        pushl $ SYMBOL_NAME(do_page_fault)
        jmp error_code
```

No error code by firmware

Error code already posted firmware

The error_code block - still i386 case

- The assembler code block called `error_code` is in charge of logging the CPU context into the stack
- This is done by aligning the stack content with the following data structure defined in `include/asm-i386/ptrace.h`

```
struct pt_regs {
        long ebx;         long ecx;
        long edx; long esi;
        long edi; long ebp;
        long eax; int  xds; int  xes;
        long orig_eax; long eip; int  xcs;
        long eflags; long esp;    int  xss;
}
```

- The actual handler can take as input a `pt_regs*` pointer and, if needed, an `unsigned long` representing the error-code

struct pt_regs for x86 long mode

```
struct pt_regs {
        unsigned long r15;  …  unsigned long r12;
        unsigned long bp;
        unsigned long bx; /* arguments: non interrupts/non tracing syscalls only save
up to here*/
        unsigned long r11;  … unsigned long r8;
        unsigned long ax;
        unsigned long cx;
        unsigned long dx;
        unsigned long si;
        unsigned long di;
        unsigned long orig_ax; /* end of arguments */ /* cpu exception frame or
undefined */
        unsigned long ip;
        unsigned long cs;
        unsigned long flags;
        unsigned long sp;
        unsigned long ss; /* top of stack page */
}
```

The page fault handler - main features

- The page fault handler is `do_page_fault(struct pt_regs *regs, unsigned long error_code)`

- It takes as input the error-code determining the type of the occurred fault, which needs to be handled

- The fault type is specified via the three least significant bits of `error_code` according to the following rules

  - **bit 0 == 0 means no page found, 1 means protection fault**
  - **bit 1 == 0 means read, 1 means write**
  - **bit 2 == 0 means kernel, 1 means user-mode**

Back to IPI

- Immediate handling is allowed for the case in which there are no data structures that are shared across CPU-cores that need to be accessed for the handling (kind of stateless scenarios)

- An example is the system-halt (e.g. upon panic)

- Other classical usages of IPI are
  - ✓ Execution on a same function across all the CPU-cores (like the initialization of per-CPU variables)
  - ✓ Change of the state of hardware components across multiple CPU-cores in the system (e.g. the TLB state)
  - ✓ Ask some CPU to preempt the current thread

Actual IPI usage in Linux - a few examples

CALL_FUNCTION_VECTOR
Sent to all CPUs but the sender, forcing those CPUs to run a function passed by the sender. The corresponding interrupt handler is named call_function_interrupt( ). Usually this interrupt is sent to all CPUs except the CPU executing the calling function by means of the smp_call_function( ) facility function.

RESCHEDULE_VECTOR
When a CPU receives this type of interrupt, the corresponding handler limits itself to acknowledge the interrupt.

INVALIDATE_TLB_VECTOR
Sent to all CPUs but the sender, forcing them to invalidate their TLB.

Actual IPI API in the apic driver

`send_IPI_all( )`

Sends an IPI to all CPUs (including the sender)

`send_IPI_allbutself( )`

Sends an IPI to all CPUs except the sender

`send_IPI_self( )`

Sends an IPI to the sender CPU

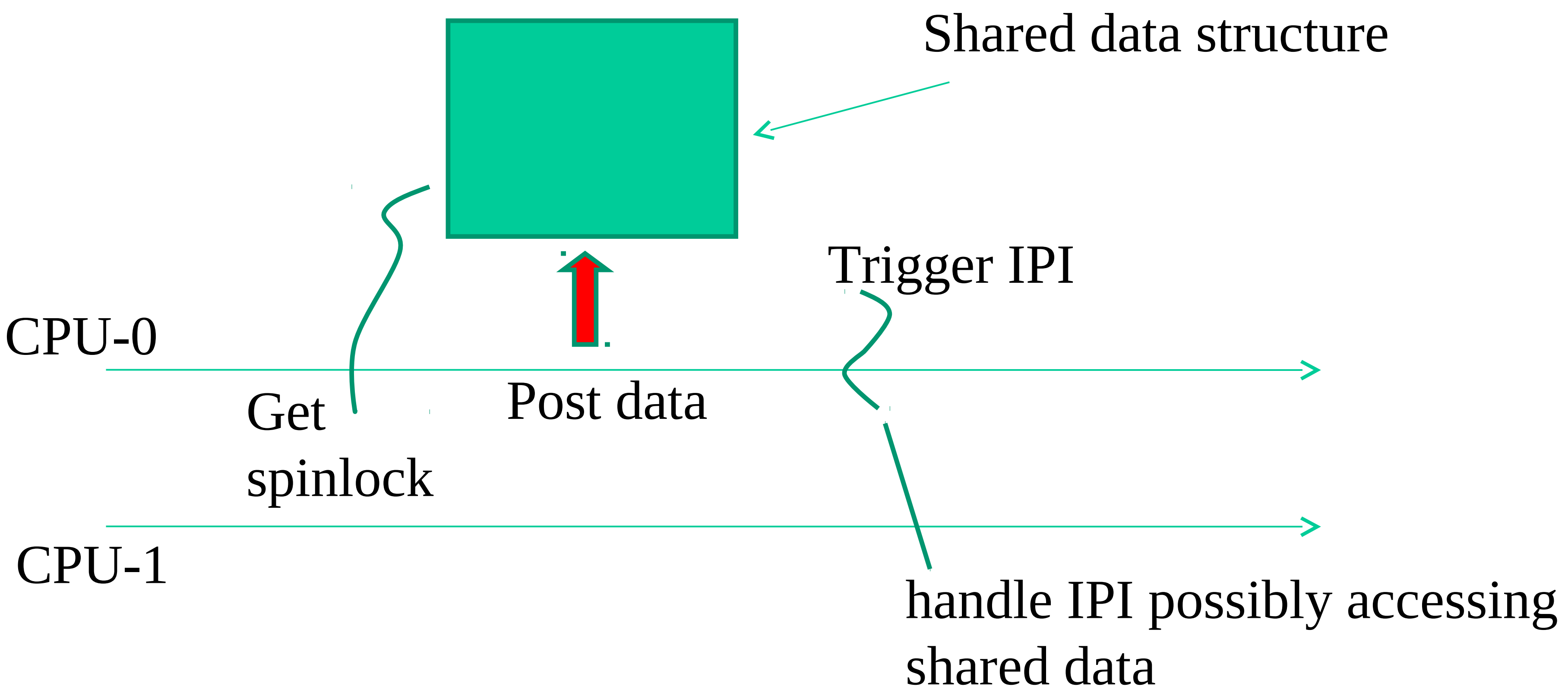
`send_IPI_mask( )`

Sends an IPI to a group of CPUs specified by a bit mask

Sequentialization of IPI management

- The sequentializing approach is used in case the IPI requires managing a shared data structure across the threads
- This is the typical case of an IPI that requires specific parameters for correct management
- These parameters are in fact passed into predetermined memory locations accessible to all the CPU-cores, whose position in memory is predetermined
- The classical case is the one of smp-call-function, whose function pointer and parameter are both passed into a global table

The scheme


Shared data structure
Trigger IPI
CPU-0
Get spinlock
Post data
CPU-1
handle IPI possibly accessing shared data

```
int smp_call_function(void (*_func)(void *info), void *_info, int wait)
{
………
	/* Can deadlock when called with interrupts disabled */
	WARN_ON(irqs_disabled());

	spin_lock_bh(&call_lock);
	atomic_set(&scf_started, 0);
	atomic_set(&scf_finished, 0);
	func = _func;
	info = _info;

	for_each_online_cpu(i)
		os_write_file(cpu_data[i].ipi_pipe[1], "C", 1);

	while (atomic_read(&scf_started) != cpus)
		barrier();

	if (wait)
		while (atomic_read(&scf_finished) != cpus)
			barrier();

	spin_unlock_bh(&call_lock);
	return 0;
```

← Beware this!!

IPI additional effects

- As noted before, one IPI used by Linux is the **reschedule** one
- This may lead to preemption of the task running on the CPU targeted by the IPI
- This may have effects on both
  - ✓ Correctness/consistency
  - ✓ Performance

Consistency aspects

- What about running a piece of code which is CPU-specific and preemption occurs??
- One example

```
struct _the_struct v[NR_CPUS];
v[smp_processor_id()] = some_value;

/* task is preempted here... */

something = v[smp_processor_id()];
```

We may be targeting different entries

Performance aspects

- `smp_call_function()` tipcally runs with interrupts allowed … just remember the deadlock issue!!
- But we cannot risk to have some `smp_call_function()` runner getting context switched off the CPU
- Otherwise the release of the `smp_call_function()` resources (e.g. the spinlock) might be delayed
- …. and we might even deadlock anyhow!!

How to run with interrupts but no actual preemption

- We use per-thread atomic counters (we already saw)
- If the counter is not zero then no preemption will take place (although we can be targeted by interrupts)
- The check is clearly done via software upon attempting to process the preemption interrupt
- Beware managing the preemption counter explicitly if required!!

Preemption enabling/disabling API recall

```
preempt_enable() // decrement the preempt counter

preempt_disable() // increment the preempt counter

preempt_enable_no_resched() decrement, but do not
immediately preempt

preempt_check_resched() // if needed, reschedule

preempt_count() return the preempt counter

put_cpu() /get_cpu()  //decrase/increase the counter
(enable/disable preemption)
```

Variants of each other

Preemption vs SMP function calls

```
int smp_call_function(void (*func) (void *info), void *info, int wait)
{
    preempt_disable();
    smp_call_function_many(cpu_online_mask, func, info, wait );
    preempt_enable();
    return 0;
}
```

Internal structure with preemption awareness

Be careful

- IPI is an extremely powerful technology
- However you need to consider scalability aspects
- This leads to conclude that IPI schemes involving large counts of CPU-cores need to be used only when mandatorily needed
- The classical example is when patching the kernel on line, e.g. upon mounting a module