

Interrupt and Time Management

Advanced Operating Systems and Virtualization

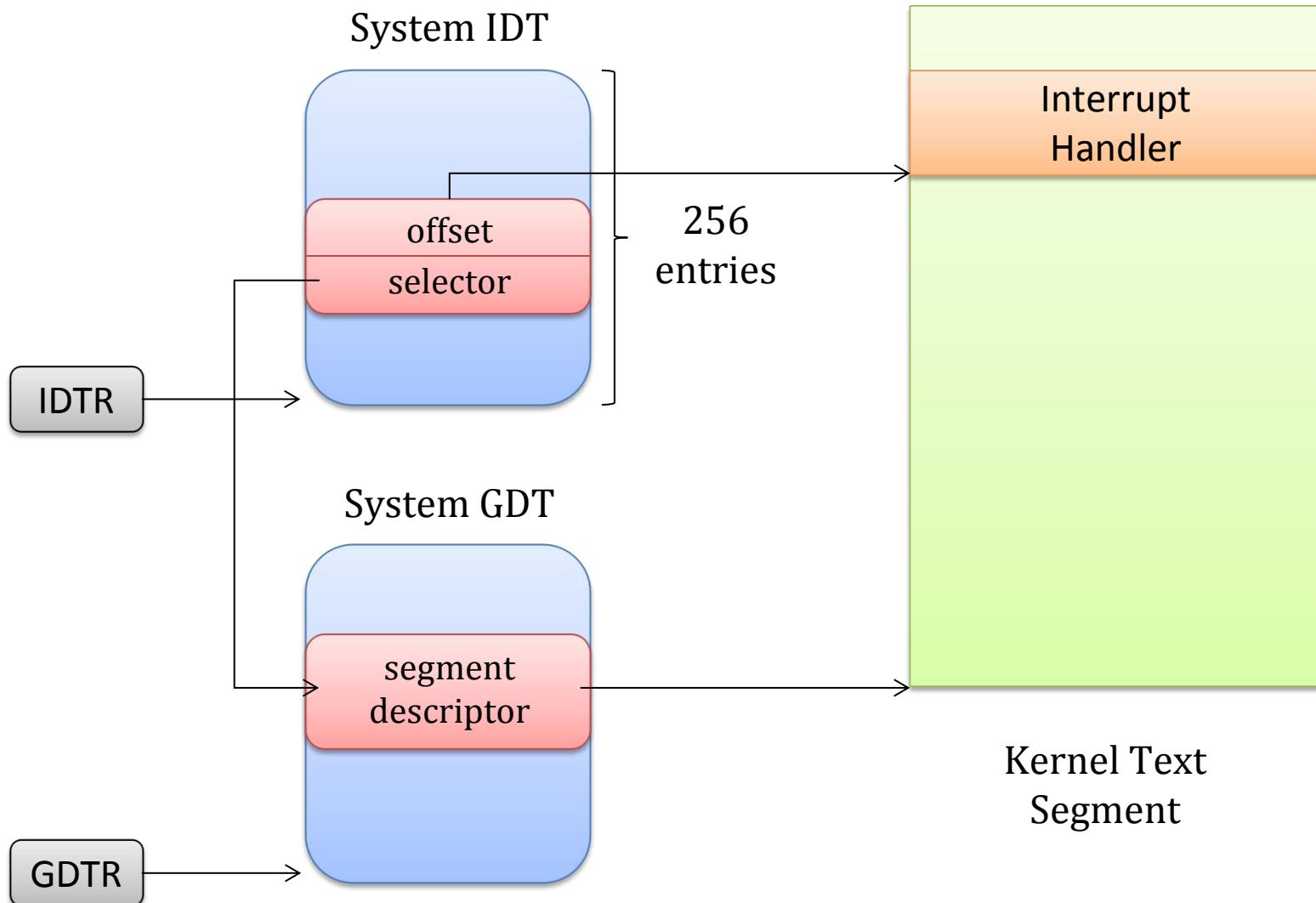
Alessandro Pellegrini

A.Y. 2019/2020

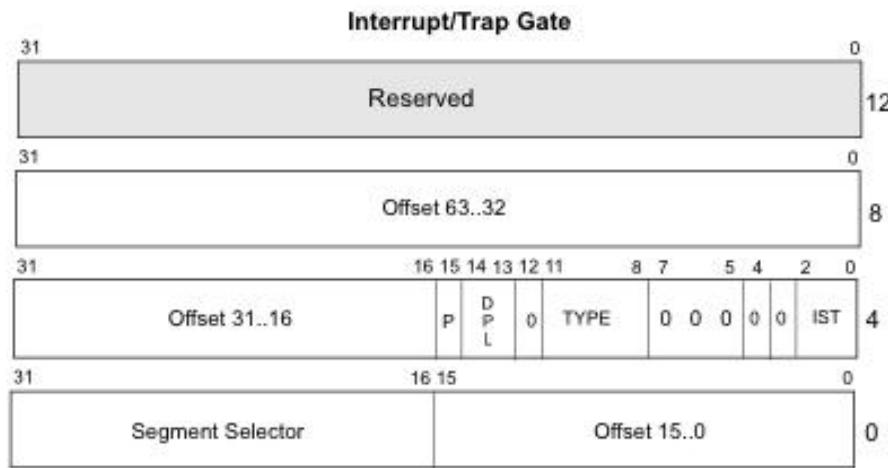


SAPIENZA
UNIVERSITÀ DI ROMA

IDT and GDT Relations



IDT Entries (x64)



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

```
struct idt_bits {  
    u16 ist : 3,  
    zero : 5,  
    type : 5,  
    dpl : 2,  
    p : 1;  
} __attribute__((packed));  
  
struct gate_struct {  
    u16 offset_low;  
    u16 segment;  
    struct idt_bits bits;  
    u16 offset_middle;  
#ifdef CONFIG_X86_64  
    u32 offset_high;  
    u32 reserved;  
#endif  
} __attribute__((packed));
```

IDT Entry Initialization

- /arch/x86/include/asm/desc.h:

```
static inline void pack_gate(gate_desc *gate, unsigned type,
unsigned long func, unsigned dpl, unsigned ist, unsigned seg) {
    gate->offset_low      = (u16) func;
    gate->bits.p          = 1;
    gate->bits.dpl        = dpl;
    gate->bits.zero       = 0;
    gate->bits.type       = type;
    gate->offset_middle   = (u16) (func >> 16);
#ifndef CONFIG_X86_64
    gate->segment         = __KERNEL_CS;
    gate->bits.ist         = ist;
    gate->reserved        = 0;
    gate->offset_high     = (u32) (func >> 32);
#else
    gate->segment         = seg;
    gate->bits.ist         = 0;
#endif
}
```



IDT Entries

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)	Interprocessor interrupts



Gate Descriptors

- A gate descriptor is a segment descriptor of type *system*:
 - Call-gate descriptors
 - Interrupt-gate descriptors
 - Trap-gate descriptors
 - Task-gate descriptors
- These are referenced by the **Interrupt Descriptor Table (IDT)**, pointed by the IDTR register



IDT Entry Types

- /arch/x86/include/asm/desc_defs.h:

```
enum {  
    GATE_INTERRUPT = 0xE,  
    GATE_TRAP = 0xF,  
    GATE_CALL = 0xC,  
    GATE_TASK = 0x5,  
};
```

Interrupts vs. Traps

- Interrupts are **asynchronous events** not related to the current CPU execution flow
- Interrupts are generated by external devices, and can be masked or not (NMI)
- Traps (or *exceptions*) are **synchronous events**, strictly related to the current CPU execution (e.g. division by zero)
- Traps were historically used to demand access to kernel mode (`int $0x80`)



Interrupts vs. Traps

- Differently from interrupts, trap management does not automatically reset the interruptible-state of a CPU core (IF)
- Critical sections in the trap handler must explicitly mask and then re-enable interrupts (`cli` and `sti` instructions)
- For SMP/multi-core machines this **might not be enough** to guarantee correctness (atomicity) while handling the trap
- The kernel uses spinlocks, based on atomic test-end-set primitives
 - We have already seen an example of CAS based on `cmpxchg`
 - Another option is the `xchg` instruction
 - Some RMW allow to atomically increment counters (`lock incl`)



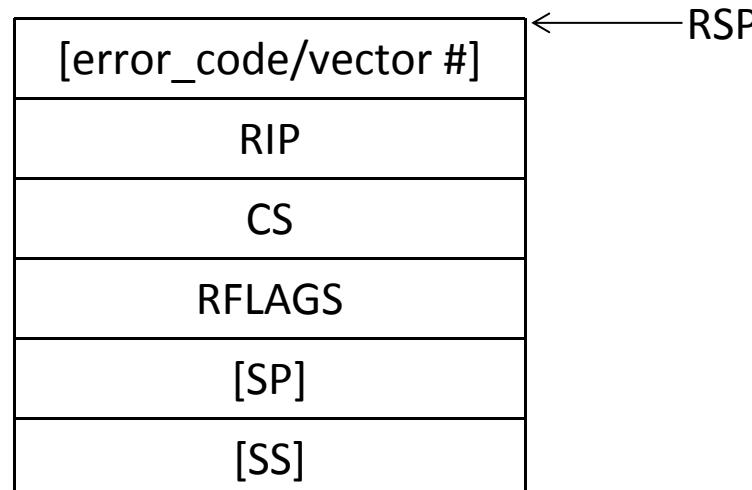
Interrupt Classification

- I/O Interrupts
 - This is received every time that an I/O device requests attention to the kernel
 - The interrupt handler must query the device to setup proper actions
- Timer Interrupts
 - The LAPIC timer has issued an interrupt
 - This notifies the kernel that some time has passed
- Interprocessor Interrupts (IPI)
 - On multicore systems, we must ensure that different cores synchronize with each other in some circumstances

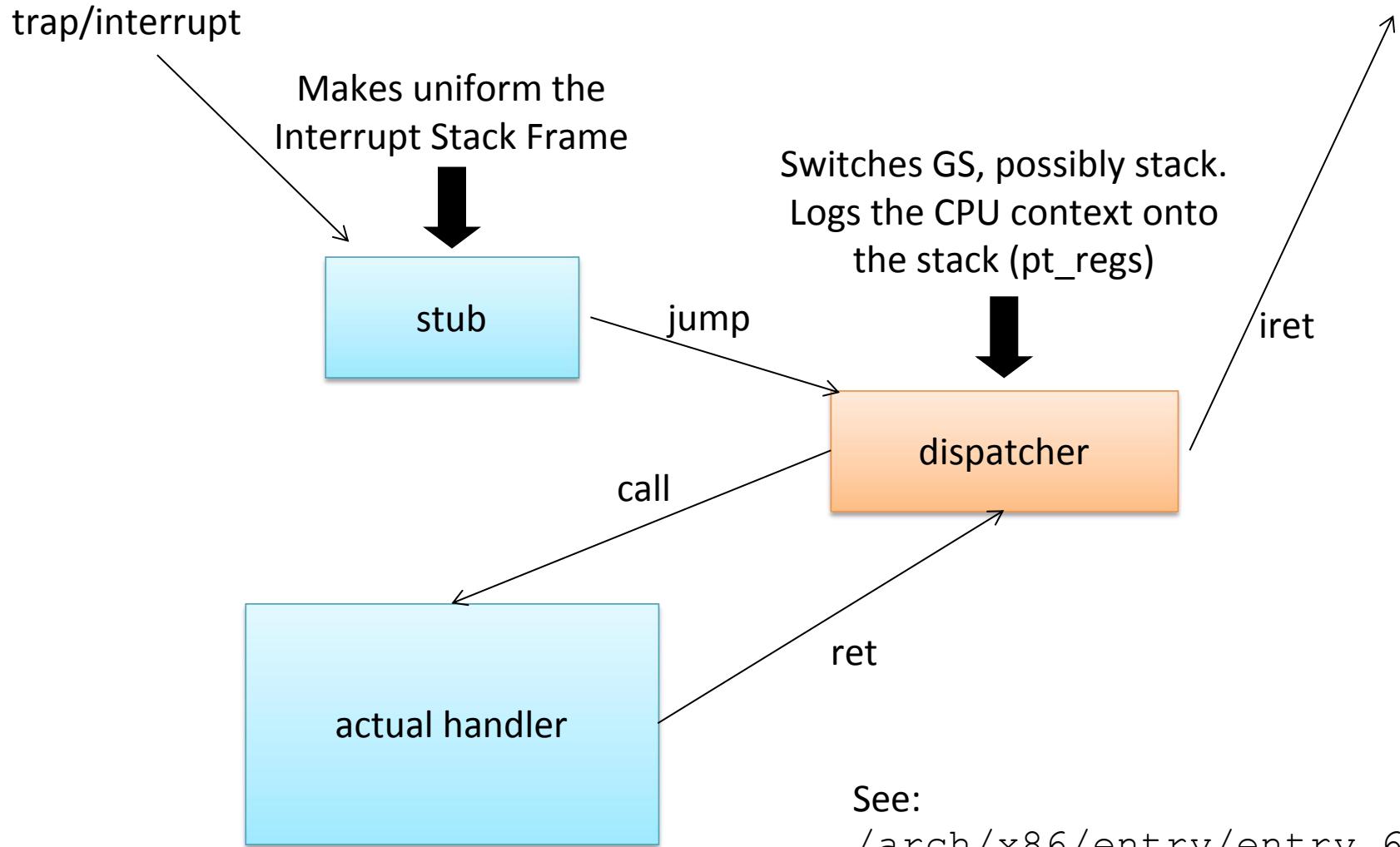


x86 Interrupt Frame

- Upon an interrupt, the firmware changes stack
 - If the IDT has an IST value different from 0 the corresponding stack from the TSS is taken
 - Otherwise, the stack corresponding to the destination Privilege Level is used
- On the new stack, the following Interrupt Stack Frame is packed by the firmware:



Global Activation Scheme



Interrupt Entry Points

- An additional push might take place:
 - If an exception occurred and no error code is placed by the firmware, a dummy -1 value is placed on stack
 - If it is an IRQ, the vector number is pushed on stack
- The correct dispatcher is then uniformly reached



Exception Examples

```
ENTRY(overflow)
```

```
    pushl $-1 // No syscall to restart  
    jmp dispatcher
```

```
ENTRY(general_protection)
```

```
    pushl $-1 // No syscall to restart  
    jmp dispatcher
```

```
ENTRY(page_fault)
```

```
    jmp dispatcher
```

This is a severe simplification: actual code is macro-generated and performs more actions



Interrupts: the “macro way” (5.0)

```
/*
 * Build the entry stubs with some assembler magic.
 * We pack 1 stub into every 8-byte block.
 */
.align 8
ENTRY(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
        UNWIND_HINT_IRET_REGS
        pushq  $(~vector+0x80) /* Note: always in signed byte range */
        jmp    common_interrupt
    .align 8
    vector=vector+1
.endr
END(irq_entries_start)
```



Exceptions: the “macro way” (5.0)

```
.macro idtentry sym do_sym has_error_code:req paranoid=0 shift_ist=-1
ENTRY(\sym)
    .if \has_error_code == 0
        pushq $-1                                /* ORIG_RAX: no syscall to restart */
    .endif

    movq %rsp, %rdi                            /* pt_regs pointer */

    .if \has_error_code
        movq ORIG_RAX(%rsp), %rsi               /* get error code */
        movq $-1, ORIG_RAX(%rsp)                /* no syscall to restart */
    .else
        xorl %esi, %esi                         /* no error code */
    .endif

    call \do_sym
    jmp error_exit

ASM_NOKPROBE(\sym)
END(\sym)
.endm

idtentry overflow    do_overflow    has_error_code=0
idtentry page_fault  do_page_fault has_error_code=1
```



swapgs

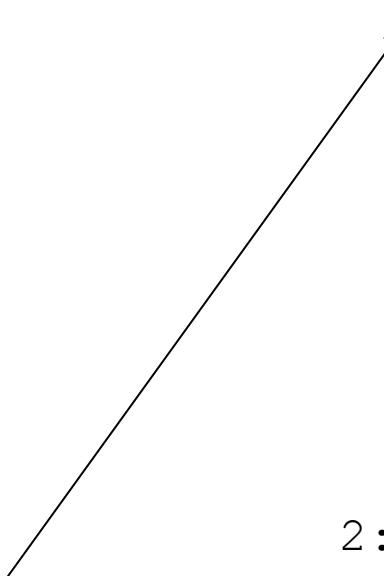
- swapgs is intended to be used by only the operating system to switch between two Model Specific Registers (MSRs)
 - IA32_GS_BASE: points to the user-mode per-thread data structure (on Windows, Linux uses FS)
 - IA32_KERNEL_GS_BASE: points to the kernel per-CPU data structure
- This allows the kernel to quickly gain access to internal, per-CPU data structures



Exception Dispatcher Skeleton

- Again a simplification , the actual code is again macro-assisted

dispatcher:

```
    cld
    testq $3, 16(%rsp)    // If coming from userspace, switch GS
    jz 1f
    swapgs
1:
    pushq %rdi
    pushq %rsi
    pushq %rdx
    pushq %rcx
    pushq %rax
    pushq %r8
    pushq %r9
    ...
    pushq %r15
    
    <prepare parameters>
    call actual_handler
    cli
    popq %r15
    ...
    popq %rdi
    testq $3, 16(%rsp)
    jz 2f
    swapgs
2:
    iretq
```



struct pt_regs

```
struct pt_regs {  
    unsigned long r15;  
    unsigned long r14;  
    unsigned long r13;  
    unsigned long r12;  
    unsigned long bp;  
    unsigned long bx;  
    unsigned long r11;  
    unsigned long r10;  
    unsigned long r9;  
    unsigned long r8;  
    unsigned long ax;  
    unsigned long cx;  
    unsigned long dx;  
    unsigned long si;  
    unsigned long di;  
    unsigned long orig_ax; // <- Syscall: syscall#. Exception: error code.  
    unsigned long ip;      // hw interrupt: IRQ number  
    unsigned long cs;  
    unsigned long flags;  
    unsigned long sp;  
    unsigned long ss;  
};
```



swapgs Speculative Execution

- In OOPs, also `swapgs` is executed speculatively
- An attacker can leak the address of kernel-level per-CPU data (on Intel CPUs)
- Two different attacks are then possible:
 - carry out a KASLR bypass attack
 - leak actual kernel memory
- This attack, anyhow, requires a specific gadget to be located in the kernel memory
- The branch predictor should also be faked into executing the `swapgs` instruction when it should not (it requires a Spectre attack variant)



swapgs Speculative Execution

- This is a code snippet from exception handlers in Windows

```
test byte ptr [nt!KiKvaShadow],1
jne skip_swapgs
swapgs
skip_swapgs:
    mov r10, qword ptr gs:[188h]
    mov rcx, qword ptr gs:[188h]
    mov rcx, qword ptr [rcx+220h]
    mov rcx, qword ptr [rcx+830h]
    mov qword ptr gs:[270h],rcx
```

Exception Example: Page Fault Handler

- The page fault handler is `do_page_fault(struct pt_regs *regs, unsigned long error_code)` defined in `linux/arch/x86/mm/fault.c`
- It takes in input the error code associated with the occurred fault
- 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 mode, 1 means user mode
- Unaccessible memory address is taken from CR2



Kernel Exception Handling

- When a process runs in kernel mode, it may have to access user memory passed by an untrusted process
 - `verify_area(int type, const void * addr, unsigned long size)`
 - `access_ok(int type, unsigned long addr, unsigned long size)`
- This may take an unnecessary large amount of time
- This operation takes place quite often



Kernel Exception Handling

- Linux exploits the MMU to take care of this
- If the kernel accesses an address which is not accessible, a page fault is generated
- The unaccessible address is taken from CR2
 - If the address is within the VA space of the process we either have to swap in the page or there was an access in write mode to a read-only page
- Otherwise, a jump to `bad_area` label tries to activate a *fixup*



Kernel Fixups

- In `bad_area`, the kernel uses the address in `regs->eip` to find a suitable place to recover execution
- This is done by replacing the content of `regs->eip` with the *fixup address*
- This must be executable code in kernel mode
- The fixup is defined by macros
- An example: `get_user(c, buf)` in `arch/x86/include/asm/uaccess.h` as called from `drivers/char/sysrq.c`



Fixup: Expanded Macro

```
(  
{  
    long __gu_err = - 14 , __gu_val = 0;  
    const __typeof__(*( ( buf ) )) * __gu_addr = ((buf));  
    if (((((0 + current_set[0])->tss.segment) == 0x18 ) ||  
        ((sizeof(* (buf))) <= 0xC0000000UL) &&  
        ((unsigned long) (__gu_addr ) <= 0xC0000000UL - (sizeof(* (buf))))))  
        do {  
            __gu_err = 0;  
            switch ((sizeof(* (buf)))) {  
                case 1:  
                    __asm__ __volatile__ (  
                        "1:    mov" "b" "%2,%" "b" "1\n"  
                        "2:\n"  
                        ".section .fixup,\\"ax\"\n"  
                        "3:    movl %3,%0\n"  
                        "        xor" "b" " %" "b" "1,%" "b" "1\n"  
                        "        jmp 2b\n"  
                        ".section __ex_table,\\"a\"\n"  
                        ".align 4\n"  
                        ".long 1b,3b\n"  
                        ".text"      : "=r"(__gu_err), "=q" (__gu_val): "m"((*(struct __large_struct *)  
                                ( __gu_addr ))) ), "i"(- 14 ), "0"( __gu_err ));  
                    break;  
                case 2:  
                    __asm__ __volatile__ (  
                        "1:    mov" "w" "%2,%" "w" "1\n"  
                        "2:\n"  
                        ".section .fixup,\\"ax\"\n"  
                        "3:    movl %3,%0\n"  
                        "        xor" "w" " %" "w" "1,%" "w" "1\n"  
                        "        jmp 2b\n"
```



Fixup: Expanded Macro

```
        ".section __ex_table, \"a\"\n"
        ".align 4\n"
        ".long 1b,3b\n"
        ".text"      : "=r"(__gu_err), "=r" (__gu_val) : "m"((*(struct __large_struct
*)(
        (    __gu_addr    )))), "i"(- 14 ), "0"(__gu_err));
        break;
case 4:
    __asm__ __volatile__(
        "1:    mov" "l" "%2,%" "" "1\n"
        "2:\n"
        ".section .fixup, \"ax\"\n"
        "3:    movl %3,%0\n"
        "        xor" "l" " %" "" "1,%" "" "1\n"
        "        jmp 2b\n"
        ".section __ex_table, \"a\"\n"
        ".align 4"           ".long 1b,3b\n"
        ".text"      : "=r"(__gu_err), "=r" (__gu_val) : "m"((*(struct __large_struct
*)(
        (    __gu_addr    )))), "i"(- 14 ), "0"(__gu_err));
        break;
default:
    (__gu_val) = __get_user_bad();
}
} while (0) ;
((c)) = (__typeof__(*((buf))))__gu_val;
__gu_err;
}
);
```



Fixup: Generated Assembly

```
xorl %edx,%edx
movl current_set,%eax
cmpl $24,788(%eax)
je .L1424
cmpl $-1073741825,64(%esp)
ja .L1423
.L1424:
    movl %edx,%eax
    movl 64(%esp),%ebx
1:   movb (%ebx),%dl /* this is the actual user access */
2:
.section .fixup,"ax"
3:   movl $-14,%eax
    xorb %dl,%dl
    jmp 2b
.section _ex_table,"a" ←
    .align 4
    .long 1b,3b
.text
.L1423:
    movzbl %dl,%esi
```

Non-standard Sections

Fixup: Linked Code

```
$ objdump --disassemble --section=.text vmlinu
```

```
c017e785 <do_con_write+c1> xorl    %edx, %edx
c017e787 <do_con_write+c3> movl    0xc01c7bec, %eax
c017e78c <do_con_write+c8> cmpl    $0x18, 0x314(%eax)
c017e793 <do_con_write+cf> je      c017e79f <do_con_write+db>
c017e795 <do_con_write+d1> cmpl    $0xbfffffff, 0x40(%esp, 1)
c017e79d <do_con_write+d9> ja      c017e7a7 <do_con_write+e3>
c017e79f <do_con_write+db> movl    %edx, %eax
c017e7a1 <do_con_write+dd> movl    0x40(%esp, 1), %ebx
c017e7a5 <do_con_write+e1> movb    (%ebx), %dl
c017e7a7 <do_con_write+e3> movzbl %dl, %esi
```

Fixup Sections

```
$ objdump --section-headers vmlinu
```

```
vmlinu:      file format elf32-i386
```

Sections:

Idx	Name	Size	VMA	LMA	File off	Align
0	.text	00098f40	c0100000	c0100000	00001000	2***4
		CONTENTS, ALLOC, LOAD, READONLY, CODE				
1	.fixup	000016bc	c0198f40	c0198f40	00099f40	2**0
		CONTENTS, ALLOC, LOAD, READONLY, CODE				
2	.rodata	0000f127	c019a5fc	c019a5fc	0009b5fc	2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA				
3	__ex_table	000015c0	c01a9724	c01a9724	000aa724	2**2
		CONTENTS, ALLOC, LOAD, READONLY, DATA				
4	.data	0000ea58	c01abcf0	c01abcf0	000abcf0	2***4
		CONTENTS, ALLOC, LOAD, DATA				
5	.bss	00018e21	c01ba748	c01ba748	000ba748	2**2
		ALLOC				
6	.comment	00000ec4	00000000	00000000	000ba748	2**0
		CONTENTS, READONLY				
7	.note	00001068	00000ec4	00000ec4	000bb60c	2**0
		CONTENTS, READONLY				



Fixup Non Standard Sections

```
$ objdump --disassemble --section=.fixup vmlinux
```

```
c0199ff5 <.fixup+10b5> movl    $0xffffffff2,%eax  
c0199ffa <.fixup+10ba> xorb    %dl,%dl  
c0199ffc <.fixup+10bc> jmp     c017e7a7 <do_con_write+e3>
```

```
$ objdump --full-contents --section=__ex_table vmlinux
```

```
c01aa7c4 93c017c0 e09f19c0 97c017c0 99c017c0 .....  
c01aa7d4 f6c217c0 e99f19c0 a5e717c0 f59f19c0 .....  
c01aa7e4 080a18c0 01a019c0 0a0a18c0 04a019c0 .....
```

- Remember x86 is little endian!

```
c01aa7c4 c017c093 c0199fe0 c017c097 c017c099 .....  
c01aa7d4 c017c2f6 c0199fe9 c017e7a5 c0199ff5 .....  
c01aa7e4 c0180a08 c019a001 c0180a0a c019a004 .....
```



Fixup Activation Steps

1. access to invalid address: c017e7a5 <do_con_write+e1> movb (%ebx), %dl
2. MMU generates exception
3. CPU calls do_page_fault
4. do page fault calls search_exception_table (regs->eip == c017e7a5);
5. search_exception_table looks up the address c017e7a5 in the exception table and returns the address of the associated fault handle code c0199ff5.
6. do_page_fault modifies its own return address to point to the fault handle code and returns.
7. execution continues in the fault handling code:
 - a) EAX becomes -EFAULT (== -14)
 - b) DL becomes zero (the value we "read" from user space)
 - c) execution continues at local label 2 (address of the instruction immediately after the faulting user access).

Fixup in 64-bit Kernels

- First possibility: expand the table to handle 64-bit addresses
- Second possibility: represent offsets from the table itself

.long 1b, 3b  .long (from) - .
 .long (to) - .

```
ex_insn_addr(const struct exception_table_entry
*x) {
    return (unsigned long)&x->insn + x->insn;
}
```

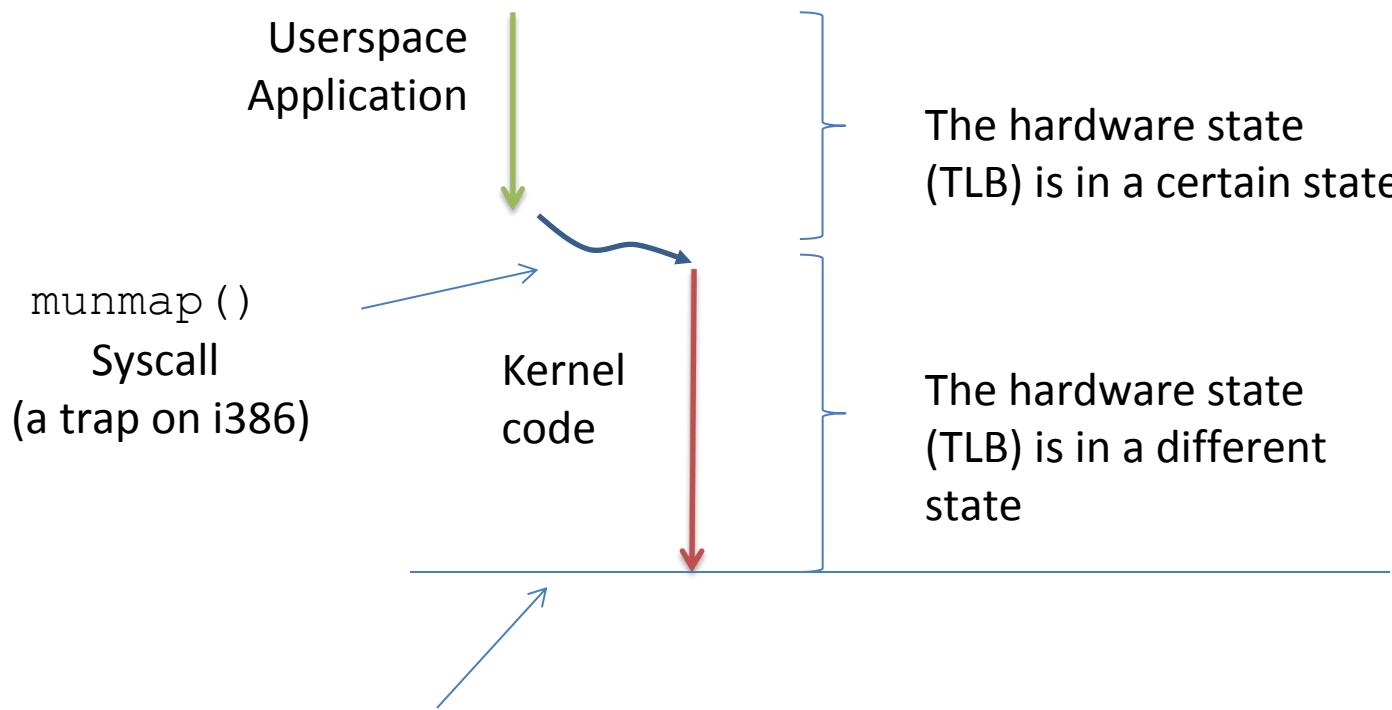


Interrupts on Multi-Core Machines

- On single core machines, interrupt/trap events are managed by running operating system code on the single core in the system
- This is sufficient to ensure consistency also in multithreaded applications
 - The hardware is time-shared across threads
- In multi-core systems, an interrupt/trap event is delivered to only one core
 - Other cores might be running other threads of the same application, though
- This can lead to race conditions or inconsistent state, due to the replication of hardware
 - We need a way to *propagate* an interrupt/trap event to other cores, if needed
- The same problem holds for synchronous requests from userspace implemented without using traps (e.g., via the vDSO)



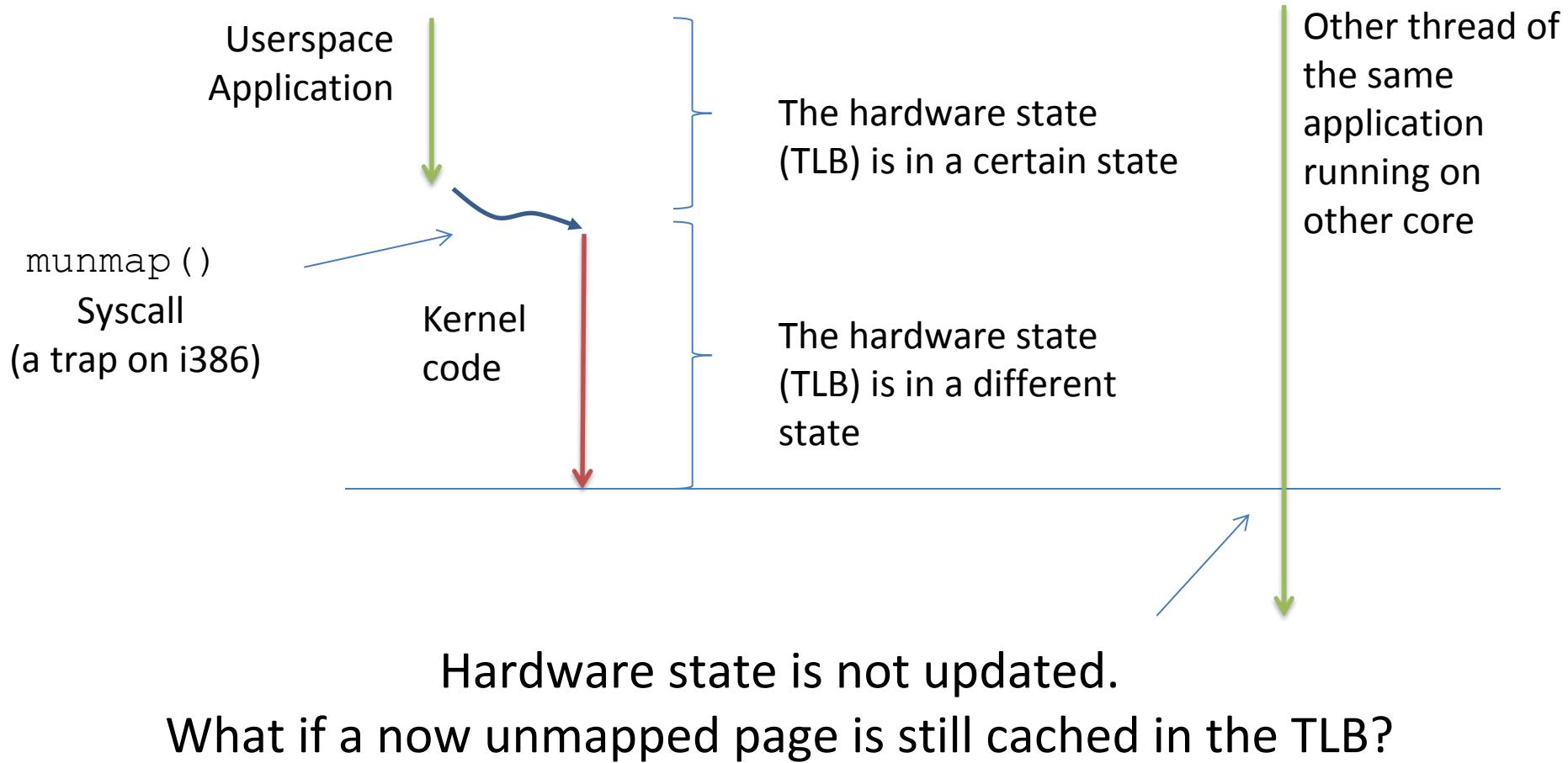
An Example: Memory Unmapping



from this time on, any time-shared thread sees a consistent state, as determined by the execution of kernel-level code



An Example: Memory Unmapping



Inter Processor Interrupts

- IPIs are interrupts also used to trigger the execution of specific operating-system functions on other cores
- IPI are used to enforce cross-core activities (e.g. request/reply protocols) allowing a specific core to trigger a change in the state of another
- IPIs are generated at firmware level, but are processed at software level
 - Synchronous at the sender, asynchronous at the receiver
- At least two priority levels are available: High and 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 lead to queueing the requests and process them in a serialized way



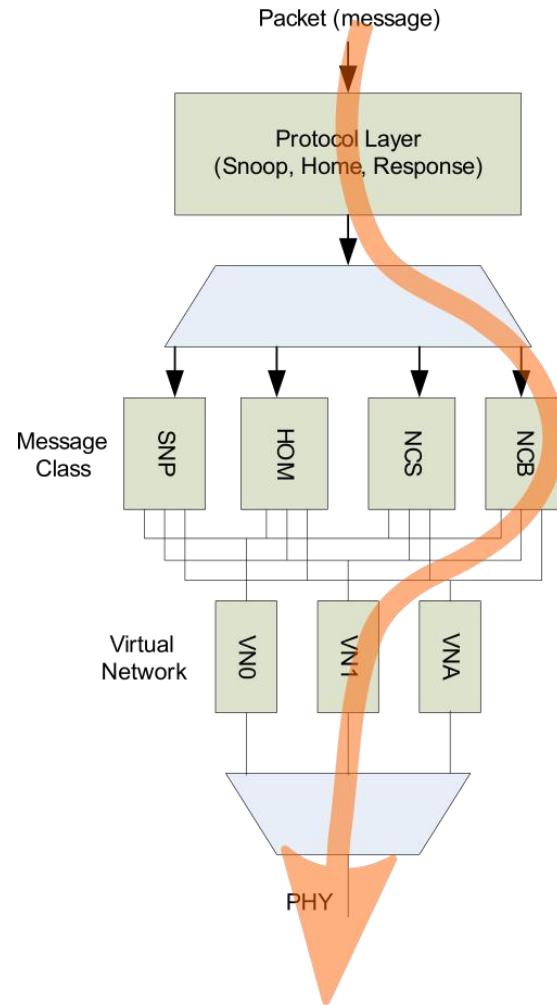
Inter Processor Interrupts

- IPIs are generated at firmware, but are processed at software
 - Synchronous at the sender, asynchronous at the receiver
- IPIs are interrupts also used to trigger the execution of specific operating-system functions on other cores
 - Software-based hardware-assisted cross-core protocols
- Two priorities: high and 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 IPIs are queued and processed in a serialized way



IPI Hardware Support on x86

- We have already seen the registers to trigger IPIs
 - They are also used to power up additional cores in a machine
- They are an interface to the APIC/LAPIC circuitry
- LAPIC offers an instance local to each core
- IPI requests travel along an ad-hoc APIC bus
 - On modern x86 architectures, this is the QuickPath Interconnect



IPI Software Management

- Immediate handling is allowed when there is no need to share data across cores (stateless processing)
- An example is *system halt* (e.g. upon panic)
- Other usages of IPI are:
 - Execution of the same function across all the CPU-cores (cross-core kernel synchronization)
 - Change of the hardware state across multiple cores in the system (e.g. the TLB)



IPI Vectors

- CALL_FUNCTION_VECTOR (*vector* 0xfb)
 - Sent to all CPUs but the sender, forcing those CPUs to run a function passed by the sender. The corresponding interrupt handler is `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 (*vector* 0xfc)
 - When a CPU receives this type of interrupt, the corresponding handler, named `reschedule_interrupt()`, just acknowledges the interrupt.
- INVALIDATE_TLB_VECTOR (*vector* 0xfd)
 - Sent to all CPUs but the sender, forcing them to invalidate their TLBs. The corresponding handler, named `invalidate_interrupt()` flushes some TLB entries of the processor.



IPIs' API (5.0)

- /arch/x86/kernel/apic/ipi.c
- These functions are wrapped in the struct apic data structure (which is to prefer on direct invocation)

default_send_IPI_all()

Sends an IPI to all CPUs (including the sender)

default_send_IPI_allbutself()

Sends an IPI to all CPUs except the sender

default_send_IPI_self()

Sends an IPI to the sender CPU

default_send_IPI_mask()

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

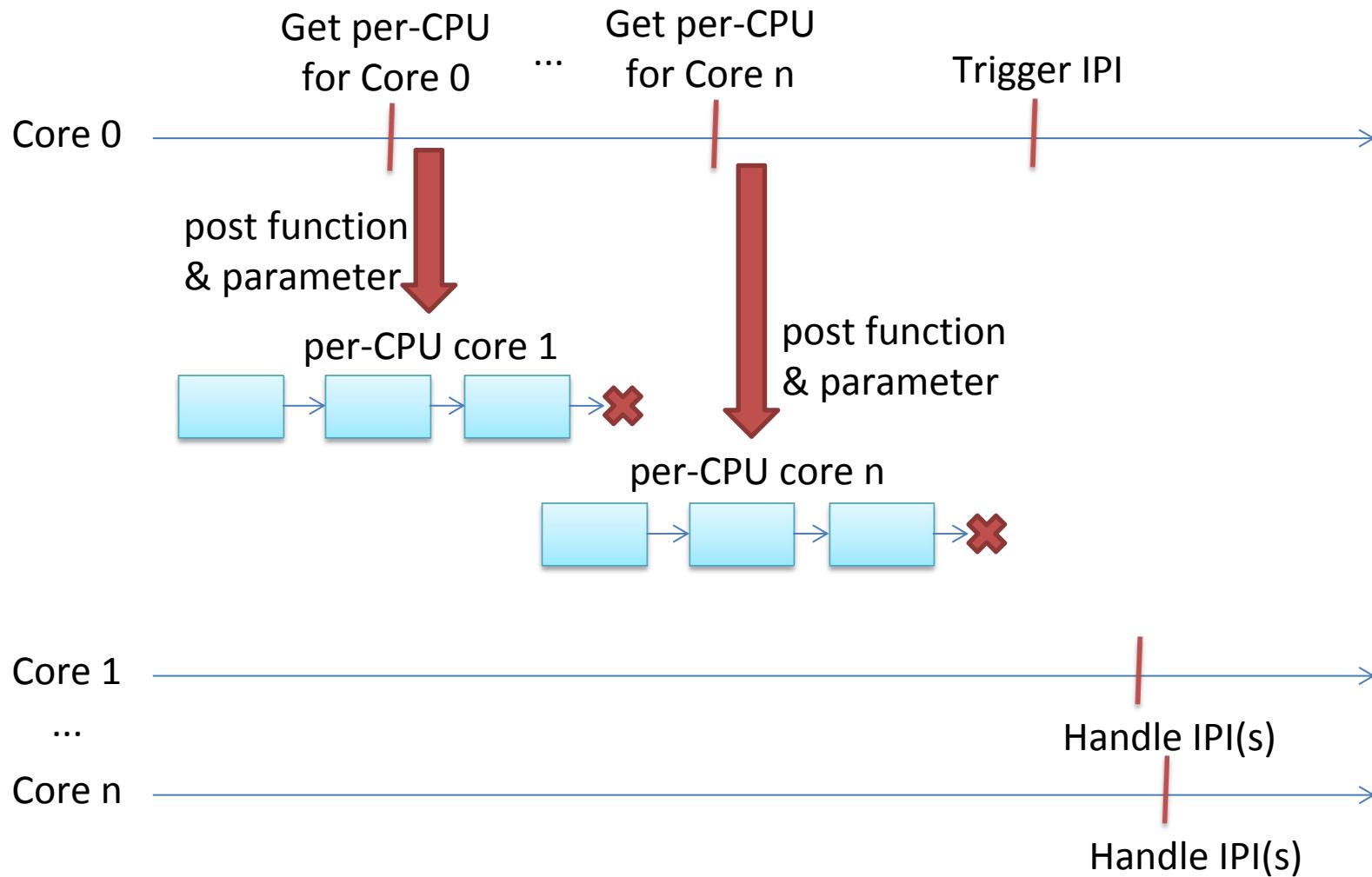


Registering IPI Functions

- IPIs are used to scheduled multiple cross-core tasks, but a single vector exists (`CALL_FUNCTION_VECTOR`)
- There is the need to register a specific action associated with the firing of an IPI
- Older version of the kernel were relying on a global data structure protected by a lock
 - This solution hampers scalability and performance
- In 5.0, there is a per-CPU linked list of registered functions and associated data to process
 - Concurrent access relies on the lock-free list



Registering IPI Functions



__call_single_data

```
struct __call_single_data {  
    struct llist_node llist;  
    smp_call_func_t func;  
    void *info;  
    unsigned int flags;  
};
```

- This is the definition of the list node
- Lists on each CPU are processed by `flush_smp_call_function_queue()`
 - It's invoked by the generic IPI handler
 - It is also invoked if a CPU is about to go offline



smp_call_function_many()

```
/* Preemption must be disabled when calling this function. */
void smp_call_function_many(const struct cpumask *mask,
                           smp_call_func_t func, void *info, bool wait)
{
    struct call_function_data *cfд;
    .....
    /*Can deadlock when called with interr. disabled*/
    WARN_ON_ONCE(/*...*/ && irqs_disabled());
    for_each_cpu(cpu, cfд->cpumask) {
        call_single_data_t *csd = per_cpu_ptr(cfд->csd, cpu);
        csd_lock(csd);
        if (!wait)
            csd->flags |= CSD_FLAG_SYNCHRONOUS;
        csd->func = func;
        csd->info = info;
        if (llist_add(&csd->llist, &per_cpu(call_single_queue, cpu)))
            __cpumask_set_cpu(cpu, cfд->cpumask_ipi);
    }
}
```



smp_call_function_many()

```
/* Send a message to all CPUs in the map */
arch_send_call_function_ipi_mask(cfd->cpumask_ipi);

if (wait) {
    for_each_cpu(cpu, cfd->cpumask) {
        call_single_data_t *csd;

        csd = per_cpu_ptr(cfd->csd, cpu);
        csd_lock_wait(csd);
    }
}
```



smp_call_function_many()

- smp_call_function_many()
 - arch_send_call_function_ipi_mask()
 - native_send_call_func_ipi()
 - apic->send_IPI_mask()
 - default_send_IPI_mask_logical()
 - __default_send_IPI_dest_field()

```
void __default_send_IPI_dest_field(unsigned int mask, int
                                    vector, unsigned int dest)
```

```
{  
    unsigned long cfg;  
    // [...]  
    cfg = __prepare_ICR2(mask);  
    native_apic_mem_write(APIC_ICR2, cfg);  
    cfg = __prepare_ICR(0, vector, dest);  
    native_apic_mem_write(APIC_ICR, cfg);  
}
```



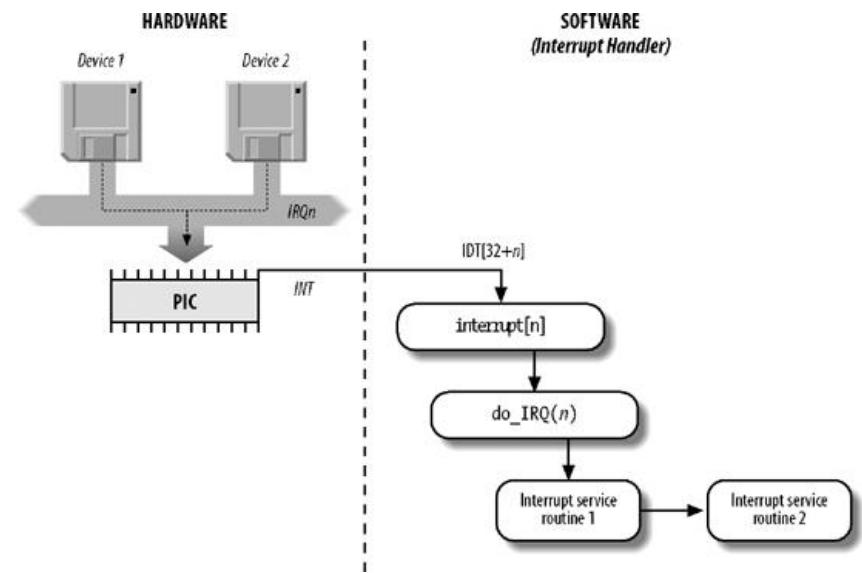
I/O Interrupt Management

- How much time does it take to handle an interrupt request?
 - We run the IRQ management with IF=0!
- Actions to be performed are split across:
 - **Critical Actions:** acknowledging the interrupt, reprogramming the device, exchanging data with the device. Executed in the handlers with IF=0.
 - **Non-Critical Actions:** any management of data structures in the kernel which are not shared with the device. These are usually quick, executed in the handler with IF=1.
 - **Non-Critical Deferrable Actions:** anything else (e.g., copying data to userspace). This is done eventually.

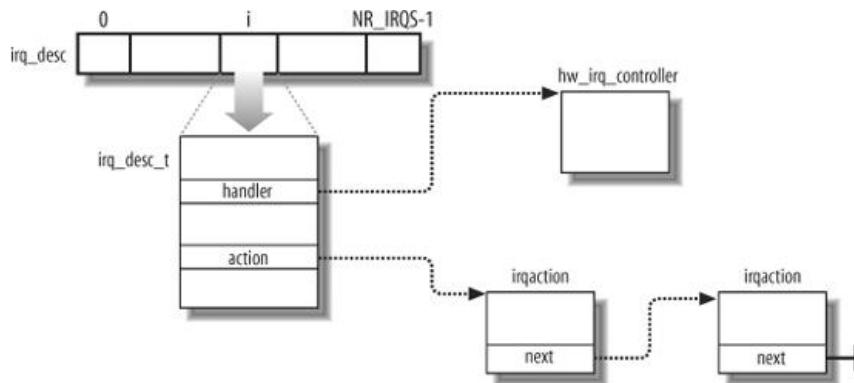


Performing Critical Actions

- Save IRQ value on stack
- Acknowledge the IRQ to the device
 - It is thus allowed to send additional IRQs
- Execute the Interrupt Service Routines (ISRs)
- iret



Locating ISRs



- The same IRQ# can be shared with multiple devices
- This is why we have multiple ISRs for an IRQ
- Hardware Geometry allows ISRs to find out what is the correct routine to process the request
- ISRs are the way to bridge device drivers and IRQ management

Private Thread Stack & IRQ Context

```
dispatcher:  
    cld  
    testb $0x3, 0x16(%rsp)  
    jz    1f  
    swapgs  
    push  %rdi  
    mov   %rsp,%rdi // Switch stack but keep the current pointer  
    mov   PER_CPU_VAR(cpu_current_top_of_stack),%rsp // Hard IRQ stack  
    pushq 0x38(%rdi) // Copy interrupt frame  
    pushq 0x30(%rdi)  
    pushq 0x28(%rdi)  
    pushq 0x20(%rdi)  
    pushq 0x18(%rdi)  
    pushq 0x10(%rdi)  
    pushq 0x8(%rdi)  
    mov   (%rdi),%rdi // Restore RDI  
  
1: push  %rsi   // Get vector number as parameter  
    mov   0x8(%rsp),%rsi  
    mov   %rdi,0x8(%rsp)  
    <save registers>  
  
    call do_IRQ // Interrupts are off here
```

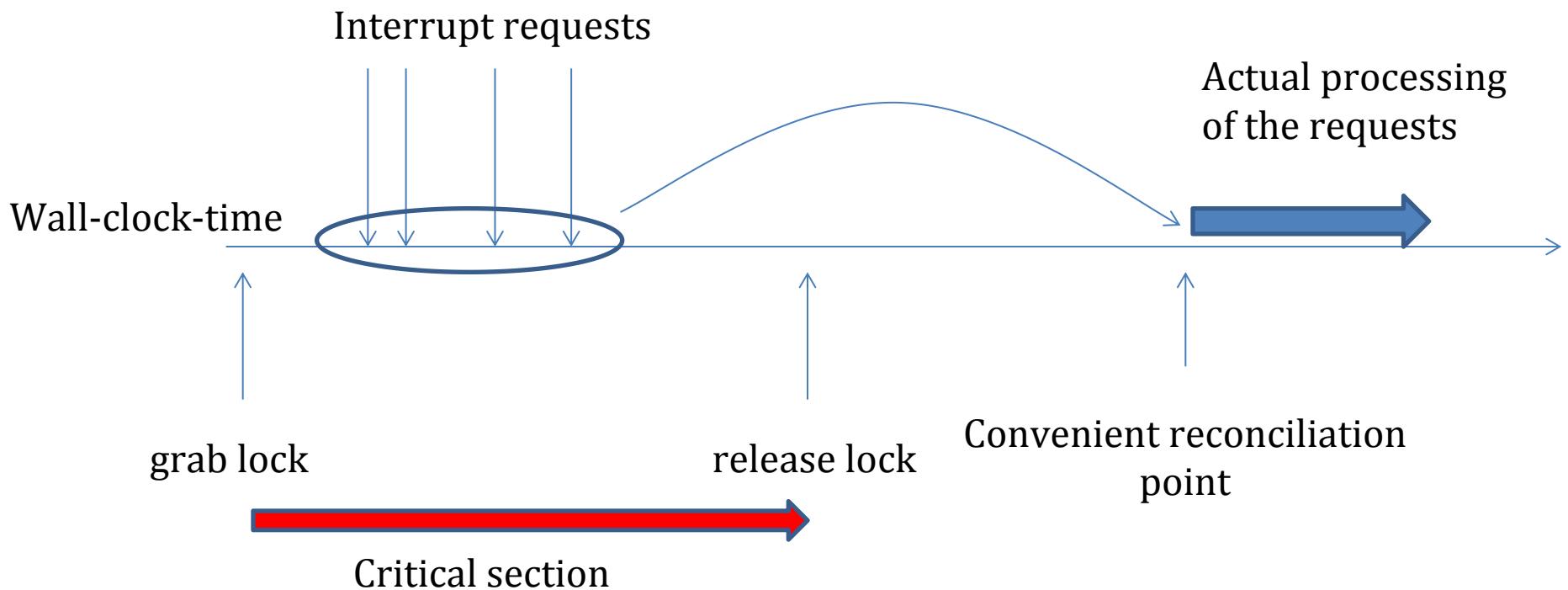


Deferred Work

- Longest part non-critical management of the interrupt can be deferred to a later time
 - when to do that work exactly?
- **Temporal reconciliation**
 - Interrupt management is mapped to regular execution contexts, and therefore shifted in time
 - Management of events in the system can be aggregated (many-to-one aggregation)
 - Care must be taken not to induce starvation



Deferred Work

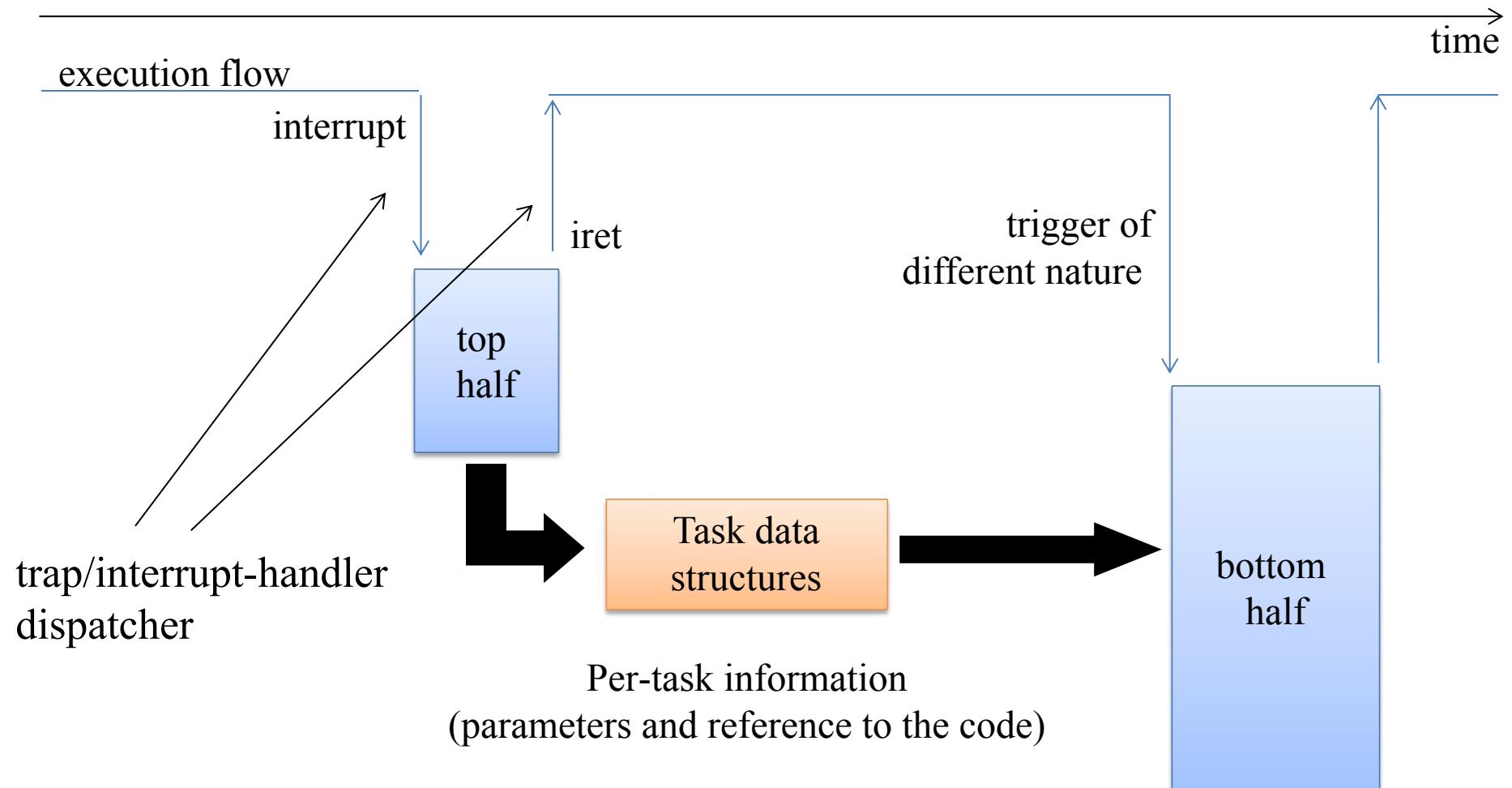


Basic Idea: Top/Bottom Halves

- Management of work comes at two levels: top half and bottom halves
- The top-half executes a minimal amount of work which is mandatory to later finalize the whole interrupt management
- The top-half code is managed according to a non-interruptible scheme
- The finalization of the work takes place via the bottom-half level
- The top-half takes care of *scheduling* the bottom-half task by queuing a record into a proper data structure



Basic Idea: Top/Bottom Halves



Deferred Work Main Steps

- Initialization
 - *Deferrable functions* are limited in number
 - They are initialized at kernel startup/module load
- Activation
 - A deferrable function is marked as “pending”
 - It will be run at the next reconciliation point
- Masking
 - Single deferrable functions can be selectively disabled
- Execution
 - Executed on the same CPU on which it was activated
 - Motivated by cache locality
 - Can be the cause of severe load unbalance

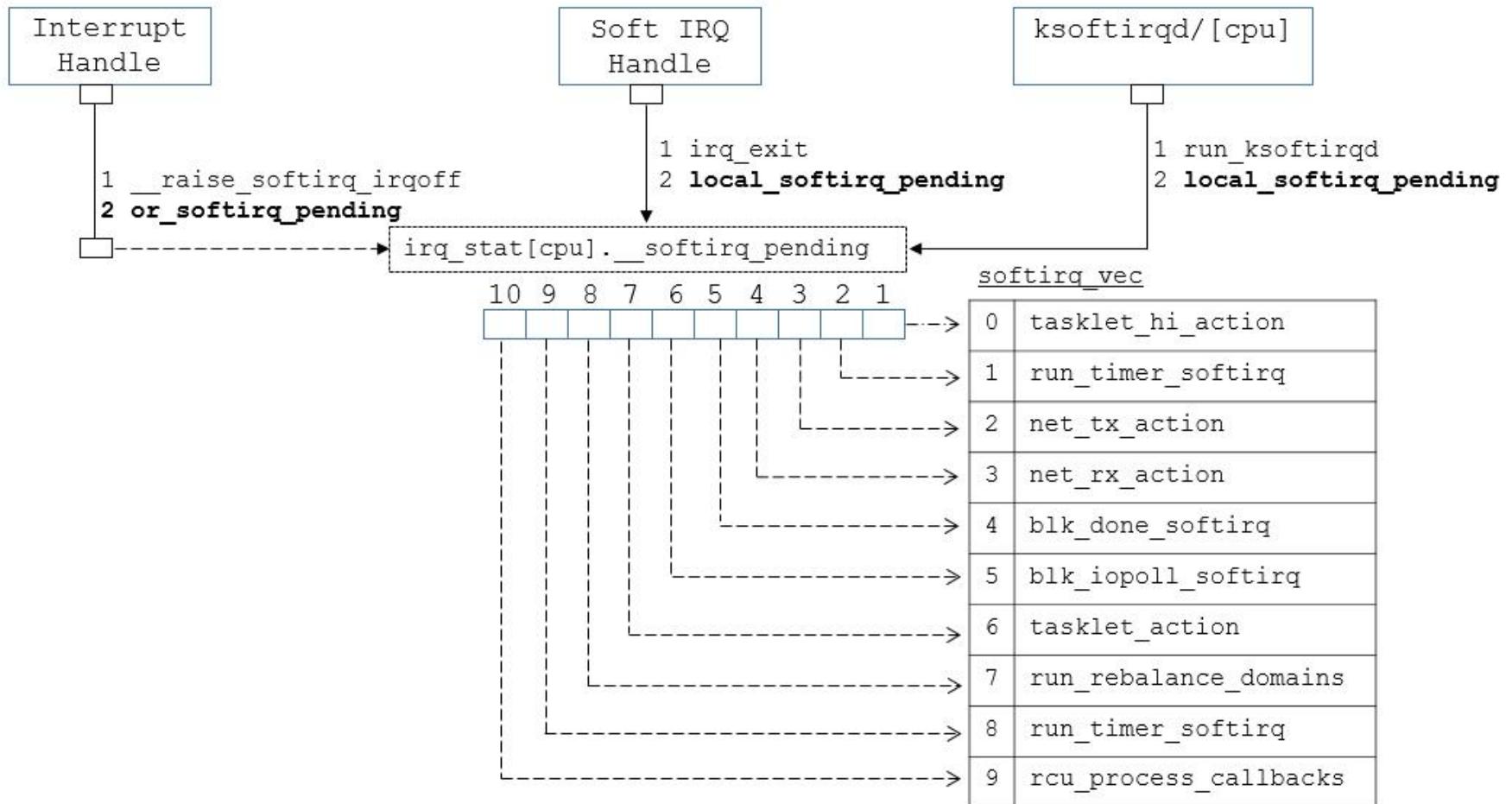


SoftIRQs

- This is the “Software Interrupt” Linux mechanism
 - «*it's a conglomerate of mostly unrelated jobs, which run in the context of a randomly chosen victim w/o the ability to put any control on them*»—Thomas Gleixner
- They are fired at specific reconciliation points
 - Coming back from a hard IRQ (with IF = 1)
 - Coming back from a syscall
 - At specific points in code (e.g., `spin_unlock_bh()`)
- They are interruptible, so they must be reentrant
 - `local_irq_save(unsigned long flags)`
 - `local_irq_restore(unsigned long flags)`
- They are rarely used directly (Tasklets)



SoftIRQs



do_softIRQ()

1. Checks if invoked in interrupt context
 - In this case it returns
2. Calls `local_irq_save()`
3. Switches to a private stack (similarly to HardIRQ management)
4. Processes IRQs (`_do_softIRQ()`)
5. Calls `local_irq_restore()`



__do_softIRQ()

- This function activates the pending actions set in the local SoftIRQ bitmask
- Other deferrable functions are deactivated locally while processing actions (`local_bh_disable()`)
- Local HardIRQs are re-enabled
- During the execution of SoftIRQs, other SoftIRQs can be activated (the bitmask is locally copied)
- After a certain numbers of iterations, it activated the SoftIRQ Daemon (with low priority)



ksoftirqd

```
for(;;) {  
    set_current_state(TASK_INTERRUPTIBLE );  
    schedule( );  
    /* now in TASK_RUNNING state */  
    while (local_softirq_pending( )) {  
        preempt_disable();  
        do_softirq( );  
        preempt_enable();  
        cond_resched( );  
    }  
}
```



Tasklets

- Tasklets are data structures used to track a specific task, related to the execution of a specific function in the kernel
- They are the preferred way to implement deferrable work
- The function accepts a parameter (an `unsigned long`) and is of type `void`
- Tasklets are declared as
(`include/linux/interrupt.h`):
 - `DECLARE_TASKLET(tasklet, function, data)`
 - `DECLARE_TASKLET_DISABLED(tasklet, function, data)`
- If declared as disabled, tasks will not be executed until enabled



Enabling and Running Tasklets

```
tasklet_enable(struct tasklet_struct *tasklet)
tasklet_hi_enable( struct tasklet_struct * );
tasklet_disable(struct tasklet_struct *tasklet)
void tasklet_schedule(struct tasklet_struct *tasklet)
```

- Each tasklet represents a single task
- Unless a tasklet reactivates itself, every tasklet activation triggers at most one execution of the tasklet function
- Management of tasklets is such that a tasklet of the same kind cannot be run concurrently on two different cores
 - If a core is running a tasklet and another core attempts to run it, it is again deferred to a later time



How Tasklets are Run

- Tasklets are run using Soft IRQs
- Enable functions are mapped to Soft IRQs lines:
 - `tasklet_enable()` mapped to `TASKLET_SOFTIRQ`
 - `tasklet_hi_enable()` mapped to `HI_SOFTIRQ`
 - No real difference between the two, except that `do_softirq()` processes `HI_SOFTIRQ` before `TASKLET_SOFTIRQ`
- All non-disabled Tasklets are executed, before the corresponding SoftIRQ action completes
 - Disabled Tasklets are put back in the corresponding list
- Remember that they are run with HardIRQs enabled



Work Queues

- More recent deferral mechanisms introduced in 2.5.41
- Similar in spirit to Tasklets, but they are run by ad-hoc kernel-level *worker threads*
- Work Queues are always run in process context
 - They can perform blocking operations
- This does not mean that they can access userspace address space



Work Queue Main Datastructure

- This is defined in linux/workqueue.h as:

```
struct work_struct {  
    atomic_long_t data;  
    struct list_head entry;  
    work_func_t func;  
};
```

```
typedef void (*work_func_t)(struct work_struct  
*work);
```



Work Queues Main API Function

```
INIT_WORK( work, func );  
INIT_DELAYED_WORK( work, func );  
INIT_DELAYED_WORK_DEFERRABLE( work, func );  
  
struct workqueue_struct *create_workqueue( name );  
void destroy_workqueue( struct workqueue_struct * );  
int schedule_work( struct work_struct *work );  
int schedule_work_on( int cpu, struct work_struct  
*work );  
  
int scheduled_delayed_work( struct delayed_work *dwork,  
unsigned long delay );  
int scheduled_delayed_work_on(int cpu, struct  
delayed_work *dwork, unsigned long delay );
```



struct delayed_work

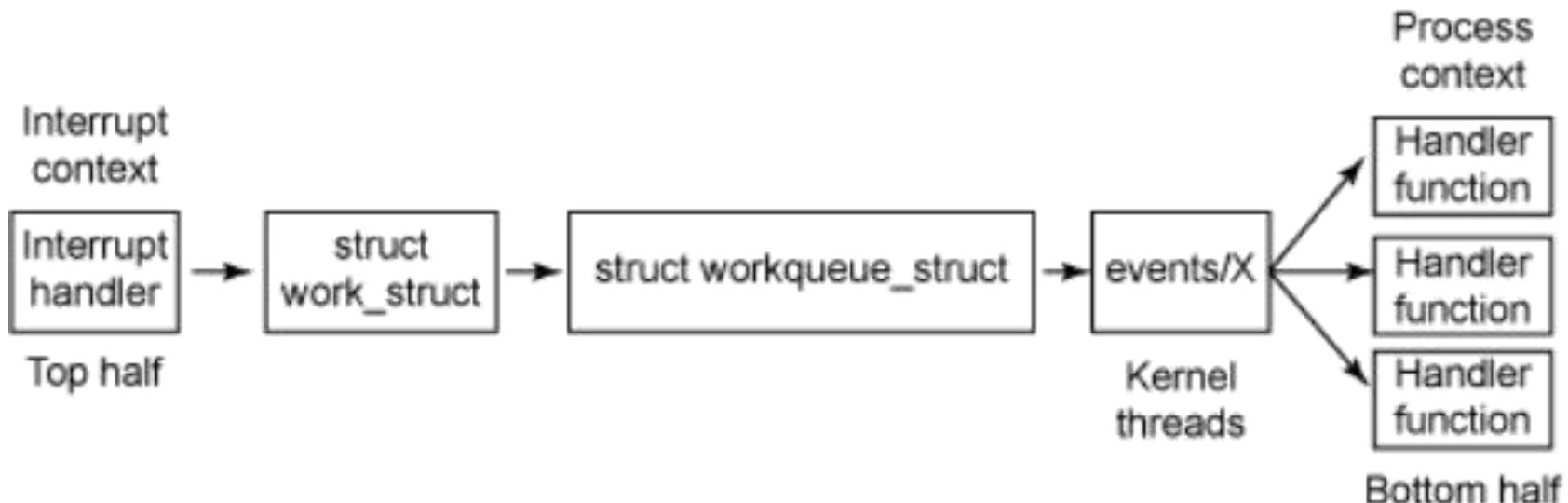
```
struct delayed_work {  
    struct work_struct work;  
    struct timer_list timer;  
  
    /* target workqueue and CPU ->timer uses  
to queue ->work */  
    struct workqueue_struct *wq;  
    int cpu;  
};
```



struct workqueue_struct

```
struct workqueue_struct {  
    struct list_head pwqs; /* WR: all pwqs of this wq */  
    struct list_head list; /* PR: list of all workqueues */  
    struct mutex mutex; /* protects this wq */  
    int work_color; /* WQ: current work color */  
    ...  
    struct list_head maydays; /* MD: pwqs requesting rescue */  
    struct worker *rescuer; /* I: rescue worker */  
    char name[WQ_NAME_LEN]; /* I: workqueue name */  
    ...  
    struct pool_workqueue __percpu *cpu_pwqs;  
    ...  
};
```

Work Queue Summary



Predefined Work Queues

- Spawning a set of worker threads to run a function is an overkill
- There is the *events* predefined work queue in the kernel
 - `schedule_work(w)` → `queue_work(keventd_wq, w)`
 - `schedule_delayed_work(w, d)` →
`queue_delayed_work(keventd_wq, w, d)` (on any CPU)
 - `schedule_delayed_work_on(cpu, w, d)` →
`queue_delayed_work(keventd_wq, w, d)` (on a given CPU)
 - `flush_scheduled_work()` → `flush_workqueue(keventd_wq)`
- Don't block for a long time: functions are serialized on each CPU



Timekeeping

- A computer would be useless if programs would not have the possibility to keep track of time passing
- This fundamental facility is handled by the kernel
- There are multiple hardware and software facilities to keep track of time
 - They provide different granularity and precision



Kernel-Level Timekeeping Concepts

- **Timeouts:** used primarily by networking and device drivers to detect when an event (e.g., I/O completion) does not occur as expected.
 - Low resolution requirements
 - Almost always removed before they actually expire
- **Timers:** used to sequence ongoing events.
 - They can have high resolution requirements, and usually expire



Hardware Clock Sources

- Real Time Clock (RTC)
 - Available on all PCs
 - Can trigger an interrupt periodically or when the counter reaches a certain value
- Time Stamp Counter (TSC)
 - Available on x86 CPUs
 - Counts the number of CPU clocks
 - Can be explicitly read (`rdtsc`)
- Programmable Interval Timer (PIT)
 - Can be programmed explicitly
 - Sends an interrupt periodically (the *timer interrupt*) to all CPUs
- CPU Local Timer (LAPIC)
 - Available on x86 CPUs
 - Delivers a timer interrupt only to the local CPU
- High Precision Event Timer (HPET)
 - Offers multiple timers which can be programmed independently



Clock Events

- They are an abstraction introduced in 2.6
- Clock Events are generated by Clock Event Devices
- This interface allows to drive hardware which can be programmed to send interrupts at different grains (e.g. PITs, HRETs)



Linux Timekeeping Architecture

- Five fundamental goals
 1. Update elapsed time since system startup
 2. Update time and date
 3. Manage process scheduling (*time quantum*)
 4. Update resource usage statistics
 5. Check if some software timer has to fire
- The kernel must adapt to the available hardware and to type of system (unicore vs multicore)



Tracking Elapsed Time

- In Linux, time is measured by a global variable named `jiffies`, which identifies the number of ticks that have occurred since the system was booted (in `kernel/time/jiffies.c`)
- The `jiffies` global variable is used broadly in the kernel for a number of purposes
- One purpose is the current absolute time to calculate the time-out value for a timer



Timer Interrupts Management on 2.4

- They are handled according to the top/bottom half paradigm (using Task Queues, which have now been removed from the Kernel)
- The top half executes the following actions:
 - Registers the bottom half
 - Increments jiffies
 - Checks whether the CPU scheduler needs to be activated, and in the positive case flags `need_resched` (more on this later)



Timer Interrupt Activation on 2.4

```
Linux Timer IRQ
IRQ 0 [Timer]
|
\|/
|IRQ0x00_interrupt          //    wrapper IRQ handler
|  |
|  |SAVE_ALL
|  |do_IRQ                   |    wrapper routines
|  |  |
|  |  \Handle_IRQ_event     ---|
|  |  |handler() -> timer_interrupt // registered IRQ 0 handler
|  |  |do_timer_interrupt
|  |  |  |
|  |  |  \do_timer
|  |  |  |jiffies++;
|  |  |  |update_process_times
|  |  |  |if (--counter <= 0) { // if time slice ended then
|  |  |  |  |counter = 0;        //    reset counter
|  |  |  |  |need_resched = 1; //    prepare to reschedule
|  |  |  |
|  |  |}
|  |  |do_softirq
|  |while (need_resched) { // if necessary
|  |  |schedule           //    reschedule
|  |  |handle_softirq
|  |}
|  |RESTORE_ALL
```



High-Resolution Timers

- They are based on the `ktime_t` type (nanosecond scalar representation) rather than jiffies

```
struct hrtimer {  
    struct timerqueue_node          node;  
    ktime_t                         _softexpires;  
    enum hrtimer_restart           (*function)(struct hrtimer *);  
    struct hrtimer_clock_base      *base;  
    u8                             state;  
    u8                             is_rel;  
};
```



High-Resolution Timers API

- `void hrtimer_init(struct hrtimer *time, clockid_t which_clock, enum hrtimer_mode mode);`
- `int hrtimer_start(struct hrtimer *timer, ktime_t time, const enum hrtimer_mode mode);`
- `int hrtimer_cancel(struct hrtimer *timer);`
- `int hrtimer_try_to_cancel(struct hrtimer *timer);`
- `int hrtimer_callback_running(struct hrtimer *timer);`



Kernel Timers

- A facility to allow a generic function to be activated at a later time (time out instant)
 - Fundamental for applications (e.g., `alarm()`)
 - Widely used by device drivers (e.g., to detect anomalous conditions)
- Timers are associated with *deferrable functions*
 - Linux does not guarantee that activation takes place at *exact* time
 - They are not appropriate for hard real-time applications



Dynamic Kernel Timers

- They can be dynamically created and destroyed
- Defined in `include/linux/timer.h`

```
struct timer_list {  
    /*  
     * All fields that change during normal runtime  
     * grouped to the same cacheline  
     */  
    struct hlist_node entry;  
    unsigned long expires;  
    void (*function)(struct timer_list *);  
    u32 flags;  
};
```



Dynamic Kernel Timer API

- `void init_timer(struct timer_list *timer);`
- `void setup_timer(struct timer_list *timer,
void (*function)(unsigned long), unsigned long data);`
- `int mod_timer(struct timer_list *timer, unsigned long
expires);`
- `void del_timer(struct timer_list *timer);`
- `int timer_pending(const struct timer_list *timer);`

- Timers are prone to race conditions (e.g., if resources are released)
- They should be deleted *before* releasing the resources

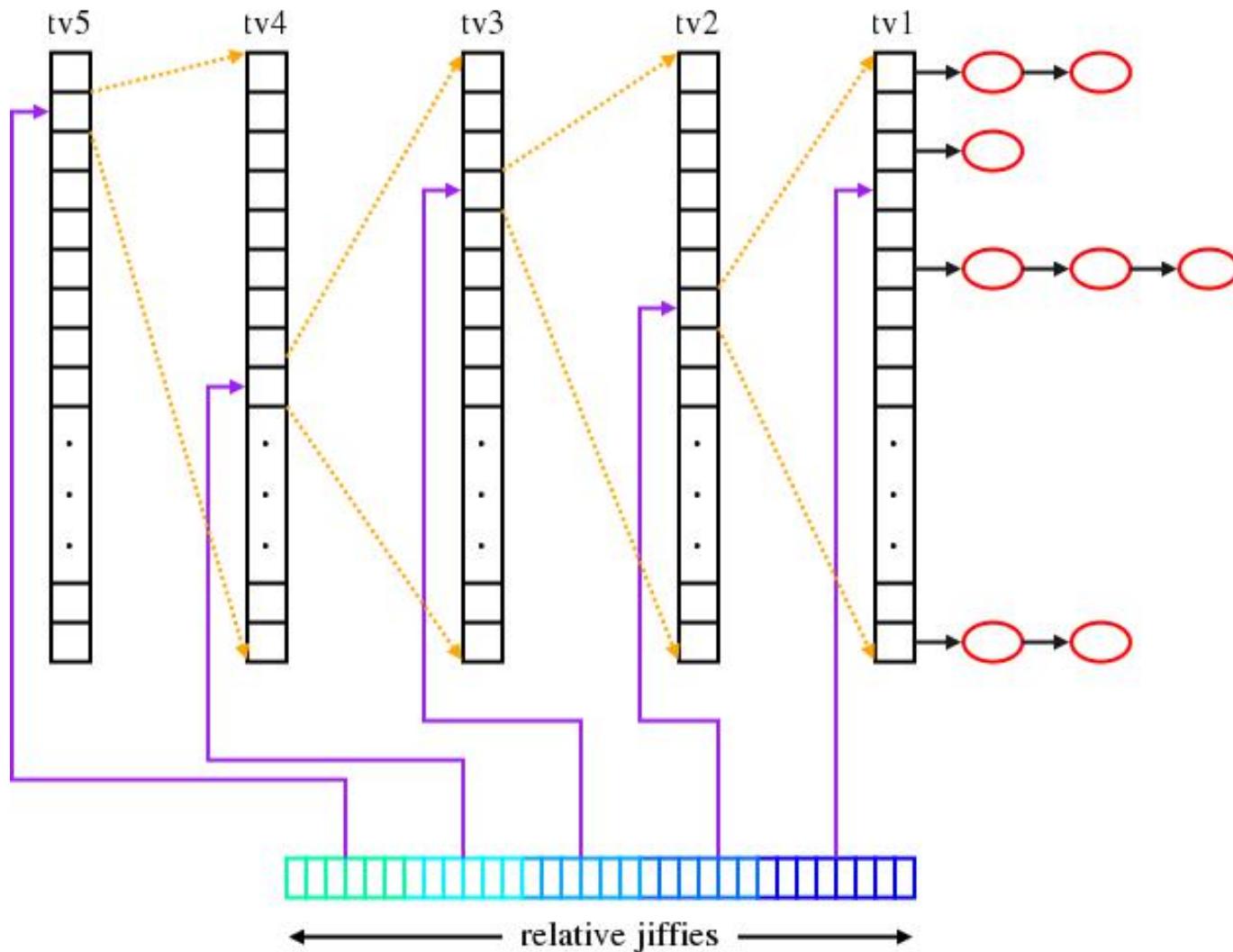


Kernel Timer Management

- Early Linux implementations had timers organized in a single list with nodes (slightly) ordered according to expiration time
- This was significantly unreliable and inefficient
- The *Timer Wheel*
 - A nested structure



The Timer Wheel (2005)



Timer Interrupt Activation on ≥2.6

```
__visible void smp_apic_timer_interrupt(struct pt_regs *regs) {
    struct pt_regs *old_regs = set_irq_regs(regs);

    /*
     * NOTE! We'd better ACK the irq immediately,
     * because timer handling can be slow.
     *
     * update_process_times() expects us to have
     * done irq_enter().
     * Besides, if we don't timer interrupts ignore the global
     * interrupt lock, which is the WrongThing (tm) to do.
     */
    entering_ack_irq();
    trace_local_timer_entry(LOCAL_TIMER_VECTOR);
    local_apic_timer_interrupt();
    trace_local_timer_exit(LOCAL_TIMER_VECTOR);
    exiting_irq();

    set_irq_regs(old_regs);
}
```



Timer Interrupt Activation on ≥2.6

- In arch/x86/kernel/apic/apic.c

```
static DEFINE_PER_CPU(struct clock_event_device, lapic_events);  
  
static void local_apic_timer_interrupt(void)  
{  
    struct clock_event_device *evt =  
        this_cpu_ptr(&lapic_events);  
    ...  
    inc_irq_stat(apic_timer_irqs);  
  
    evt->event_handler(evt);  
}
```

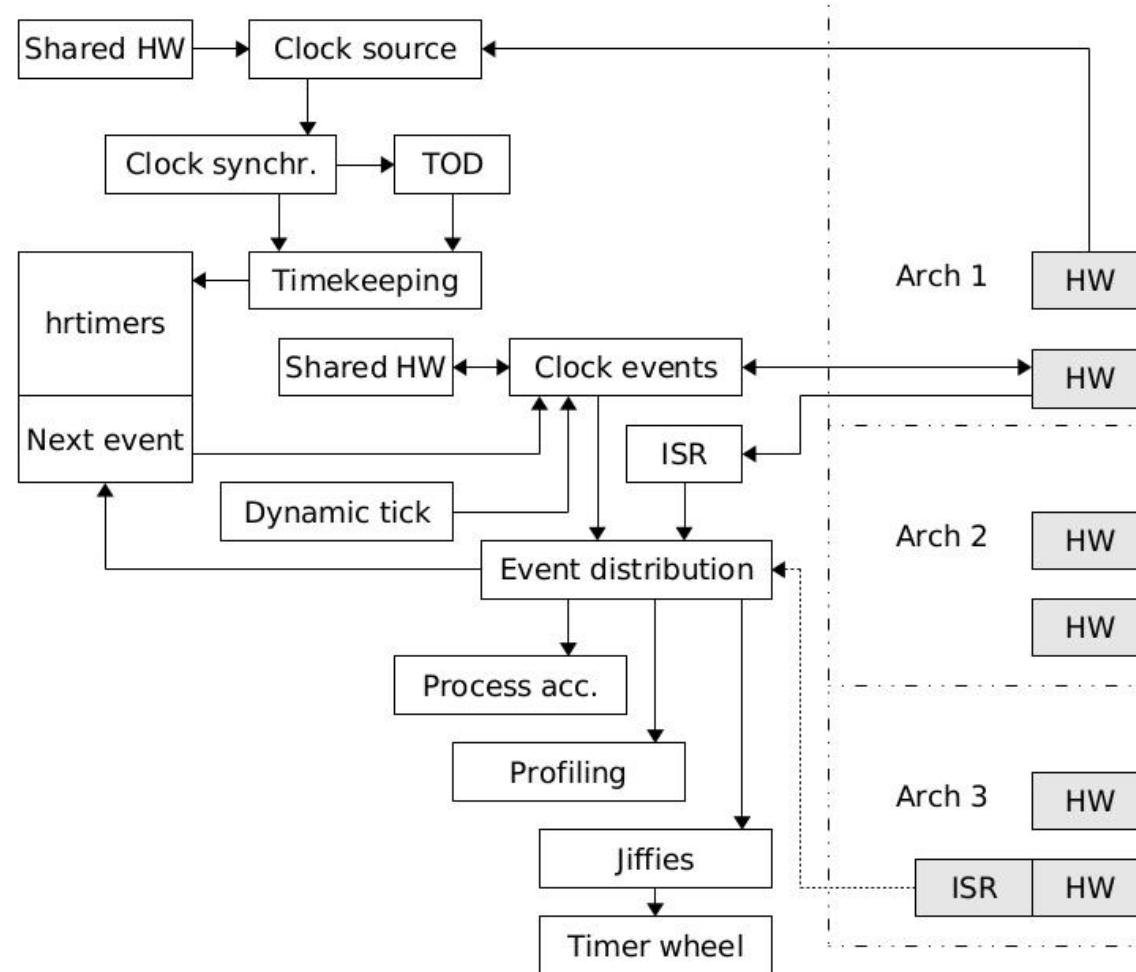


POSIX Clocks

- `CLOCK_REALTIME`: This clock provides a best effort estimate of UTC in a way that is backwards compatible with existing practice. Very little is guaranteed for this clock. It will never show leap seconds
- `CLOCK_UTC`: This clock is only available when the system knows with high assurance Coordinated Universal Time (UTC) with an estimated accuracy of at least 1 s
- `CLOCK_TAI`: This clock is only available when the system knows International Atomic Time (TAI) with at least an accuracy of 1s
- `CLOCK_MONOTONIC`: This clock never jumps, it is guaranteed to be available all the time right after system startup, and its frequency never varies by more than 500 ppm
- `CLOCK_THREAD`: This clock started its Epoch when the current thread was created and runs only when the current thread is running on the CPU
- `CLOCK_PROCESS`: This clock starts its Epoch when the current process was created and runs only when a thread of the current process is running on the CPU



Overall Timekeeping Architecture



Linux Watchdog

- A watchdog is a component that monitors a system for “normal” behaviour and if it fails, it performs a system reset to hopefully recover normal operation.
- This is a last resort to maintain system availability or to allow sysadmins to remotely log after a restart and check what happened
- In Linux, this is implemented in two parts:
 - A kernel-level module which is able to perform a hard reset
 - A user-space background daemon that refreshes the timer



Linux Whatchdog

- At kernel level, this is implemented using a Non-Maskable Interrupt (NMI)
- The userspace daemon will notify the kernel watchdog module via the `/dev/watchdog` special device file that userspace is still alive

```
while (1) {  
    ioctl(fd, WDIOC_KEEPALIVE, 0);  
    sleep(10);  
}
```

