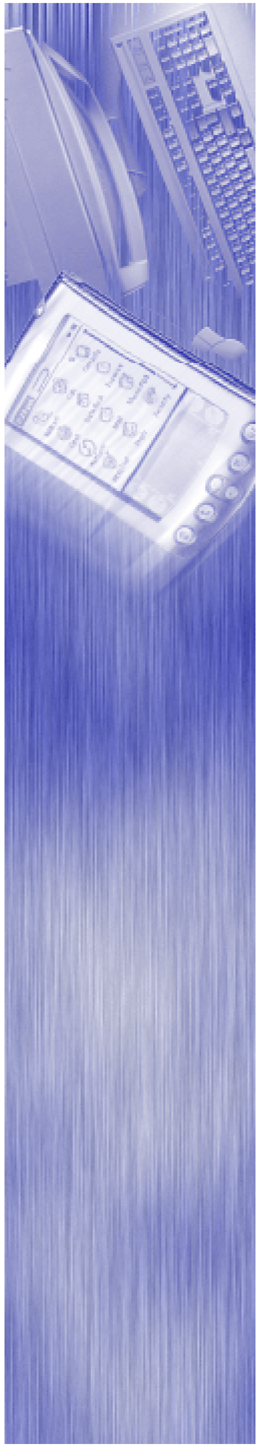


CS632/SEP564: Embedded Operating Systems (Fall 2008)

System Calls

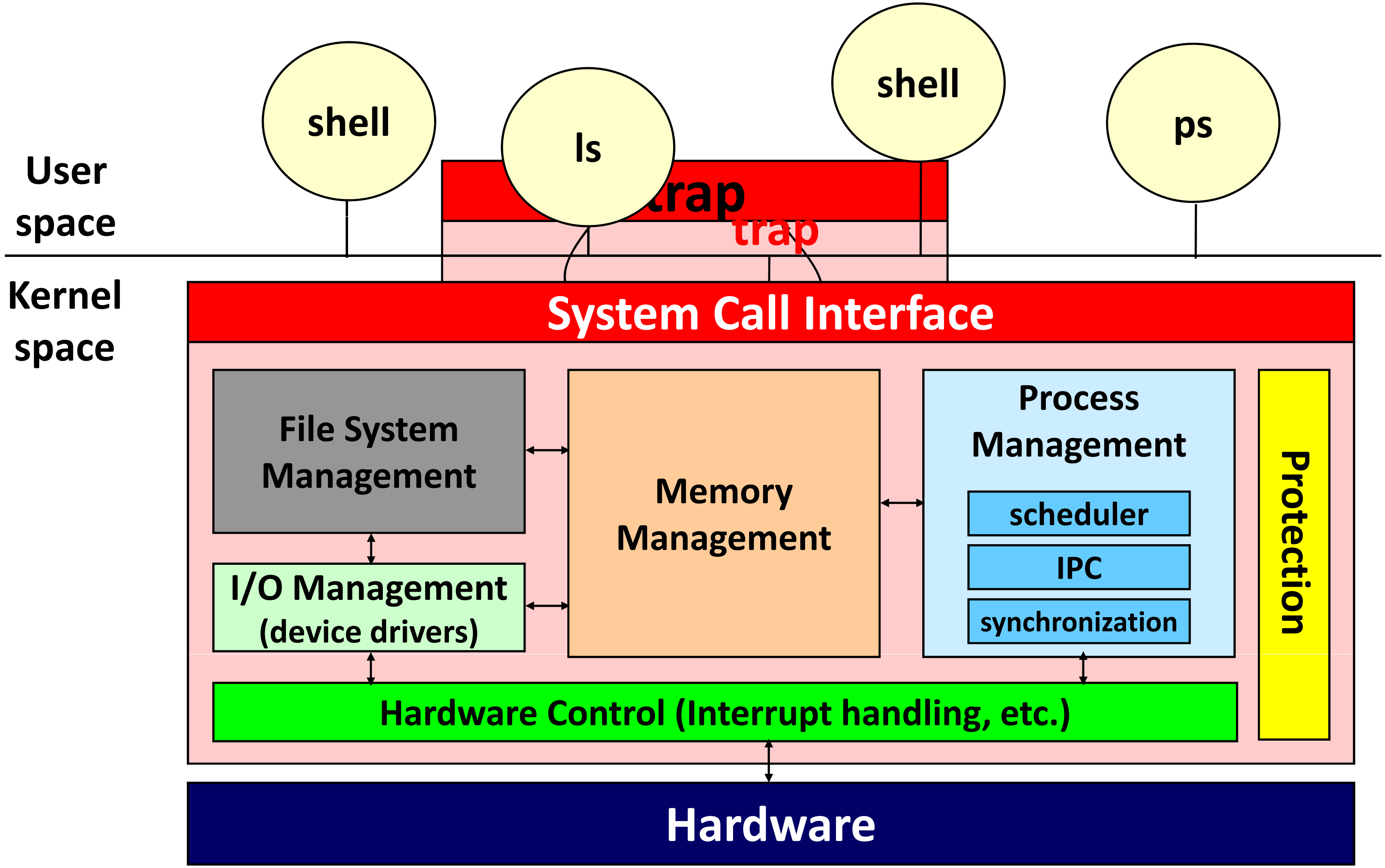
Interrupts and Exceptions (1)

- **Interrupts**
  - Generated by hardware devices
    - Triggered by a signal in INTR or NMI pins (x86)
  - Asynchronous
- **Exceptions**
  - Generated by software executing instructions
    - INT instruction in x86
  - Synchronous

Interrupts and Exceptions (2)

- **Further classification of exceptions**
  - Traps
    - Intentional
    - System calls, breakpoint traps, special instructions, etc.
    - Return control to “next” instruction
  - Faults
    - Unintentional but possibly recoverable
    - Page faults (recoverable), protection faults (unrecoverable), etc.
    - Either re-execute faulting (“current”) instruction or abort
  - Aborts
    - Unintentional and unrecoverable
    - Parity error, machine check, etc.
    - Abort the current program.

OS Structure Revisited



Introduction (1)

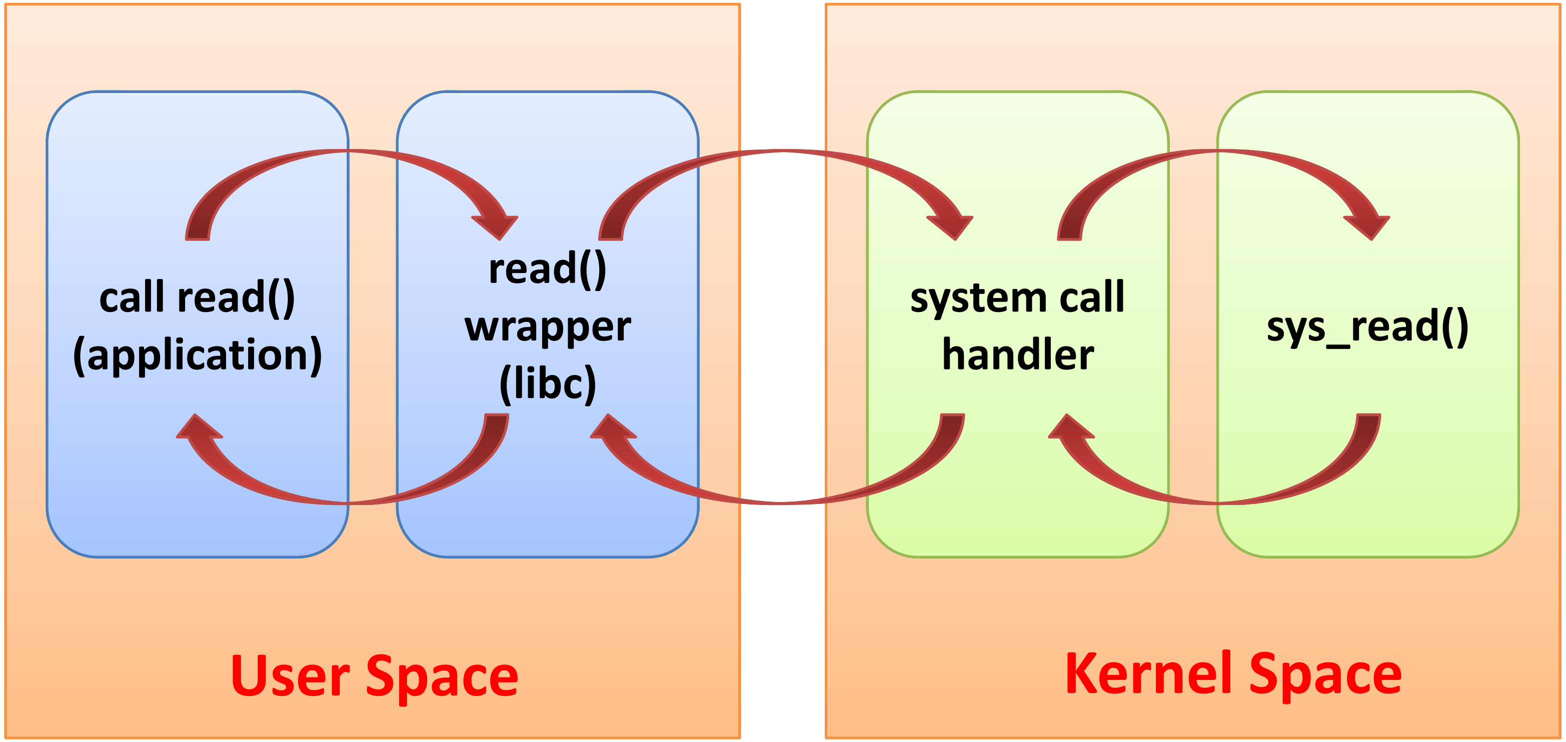
- **System calls**
  - A layer between the hardware and user-space processes
  - Roles:
    - Provide an abstract hardware interface for user-space.
    - Ensure system security and stability.
    - Single common layer allows for the virtualized system provided to processes.

- **APIs vs. system calls**

Introduction (2)

- **POSIX**
  - A series of standards from the IEEE that aim to provide a portable operating system standard.
    - POSIX 1. Core Services (IEEE Std 1003.1-1988)
    - POSIX 1b. Real-time extensions (IEEE Std. 1003.1b-1993)
    - POSIX 1c. Threads extensions (IEEE Std. 1003.1c-1995)
  - POSIX APIs have a strong correlation to the Unix system calls.
  - Microsoft Windows NT offers POSIX-compatible libraries.
    - Microsoft Windows Services for UNIX 3.5
    - (cf) Cygwin: Partial POSIX compliance for certain Microsoft Windows products.

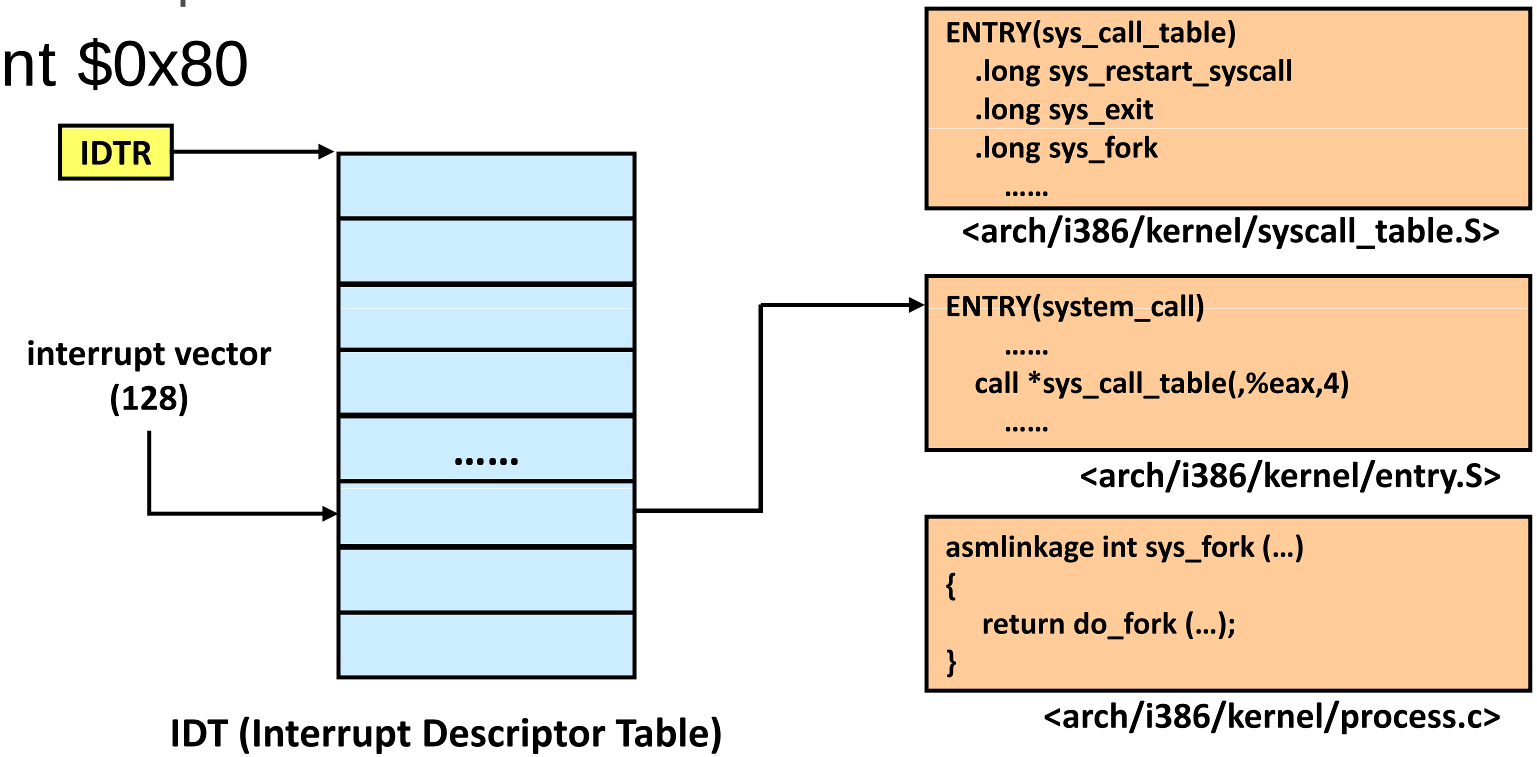
Introduction (2)



Implementation (1)

▪ x86 implementation

- Signal the kernel via software interrupt.
  - Incur an exception, switch to the kernel mode, and execute the exception handler
- int $0x80

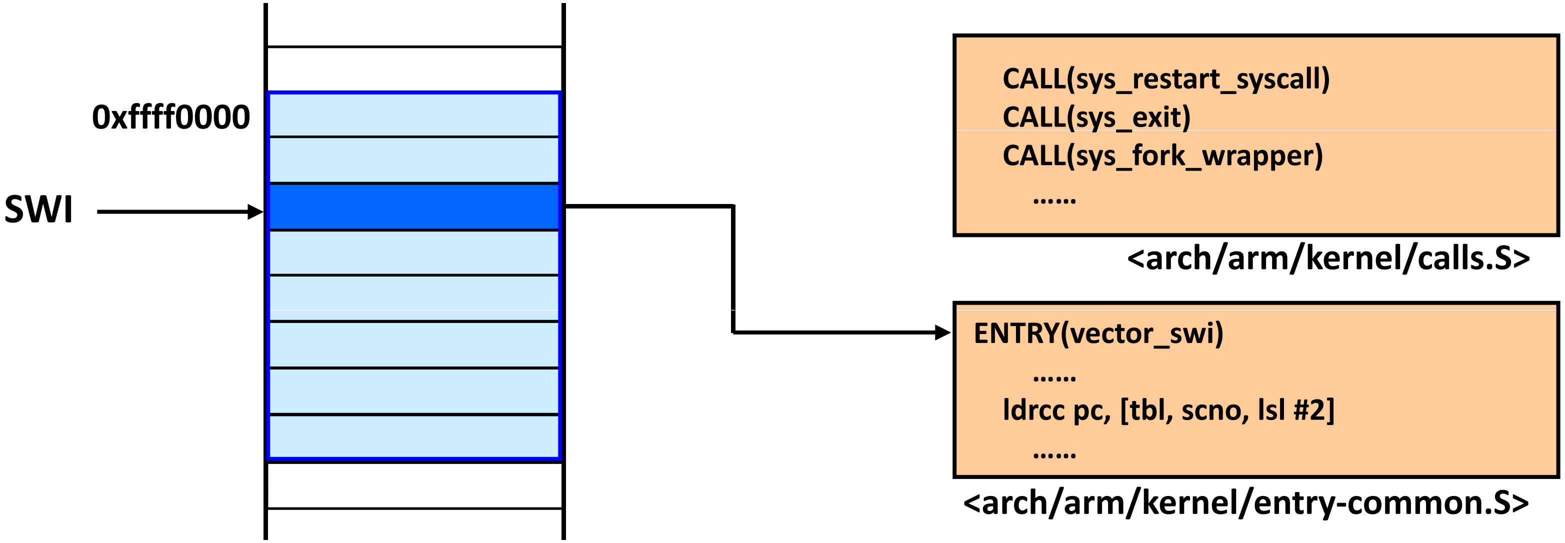


Implementation (2)

▪ ARM implementation

- SWI *n*
- Vector table
  - 0x00000000 (default) or 0xffff0000

Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c



Implementation (3)

- **Passing information**
  - System call number
    - x86: eax register
    - ARM: immediate operand in SWI instruction
      - » System call handler needs to parse the system call number.
  - Parameters: via registers
    - x86: ebx, ecx, edx, esi, edi, and ebp registers
    - ARM: r0, r1, r2, r3, r4, and r5 registers
  - Return value
    - x86: eax register
    - ARM: r0 register

Implementation (4)

- **Adding a system call**

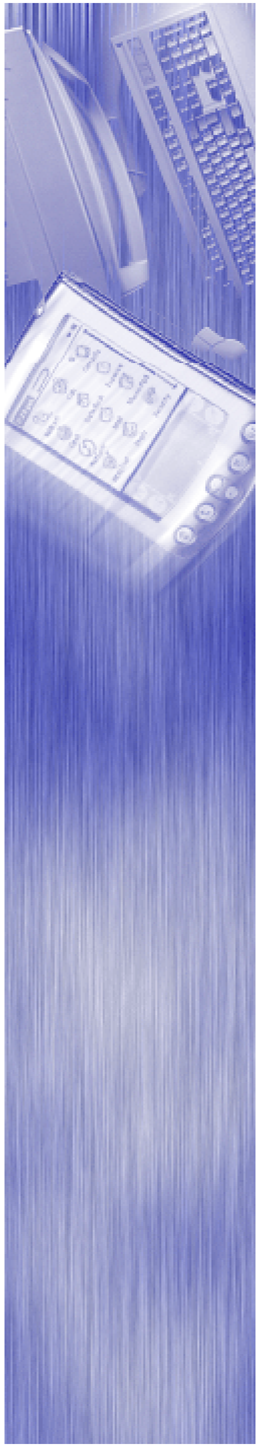
```
#include <sys/syscall.h>
#include <unistd.h>

int syscall (int number, ...);
```

```
...                                             <asm/unistd.h>
#define __NR_exit           1
#define __NR_fork           2
...
```

```
#include <sys/syscall.h>
#include <unistd.h>
#define __NR_foo            310
int foo()
{
        return syscall (__NR_foo);
}
```

Summary

- **Pros**
  - Simple to implement and easy to use
  - Fast performance on Linux
- **Cons**
  - System call number should be officially assigned.
  - Once it is in a stable series kernel, it cannot change without breaking user-space applications.
  - Each architecture needs to separately register the system call and support it.
  - System calls are not easily used from scripts and cannot be accessed directly from the filesystem.
  - For simple exchange of information, it is overkill.

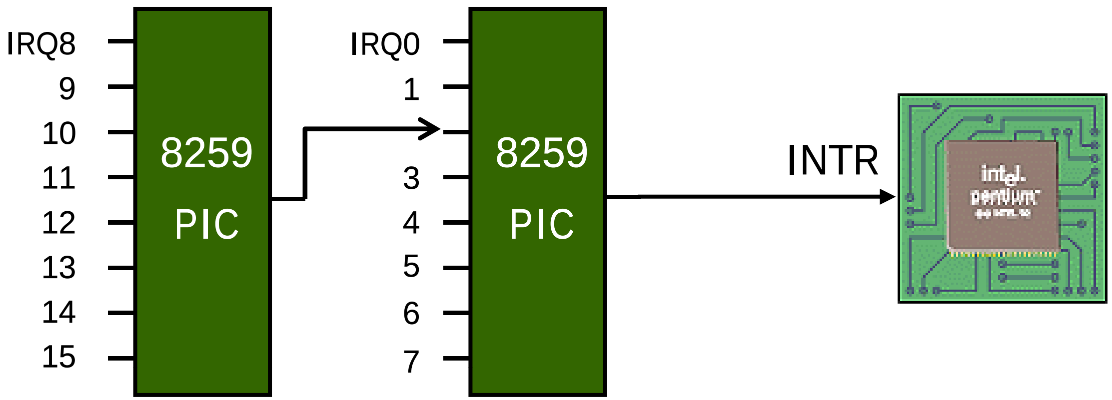


CS632/SEP564: Embedded Operating Systems (Fall 2008)

Interrupts

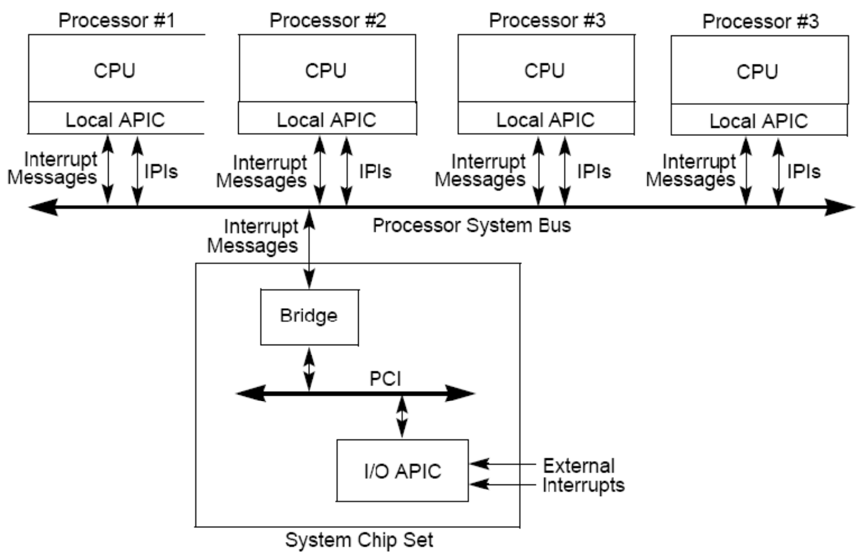
Interrupt Hardware (1)

- **Programmable Interrupt Controller (PIC)**
  - Two cascaded Intel 8259A PICs: IRQ0 ~ IRQ15
  - Intel's default vector associated with IRQn is n+32.
  - Each IRQ line can be selectively disabled.
    - Disabled interrupts are not lost.
  - cli/sti: global control of maskable interrupts (x86)



Interrupt Hardware (2)

- **Variations**
  - Interrupt priority
  - Static vs. dynamic allocation
  - Interrupt sharing
  - Interrupt coalescing
  - Distributed architecture for SMP



Interrupt Vectors (1)

- **An example of IRQ assignments (IA32)**

IRQ	INT	Hardware Device
0	32	Timer
1	33	Keyboard
2	34	PIC cascading
3	35	Second serial port (COM2)
4	36	First serial port (COM1)
6	38	Floppy disk
8	40	System clock
10	42	Network interface
11	43	USB port, sound card
12	44	PS/2 mouse
13	45	Mathematical coprocessor
14	46	EIDE disk controller’s first chain
15	47	EIDE disk controller’s second chain

Interrupt Vectors (2)

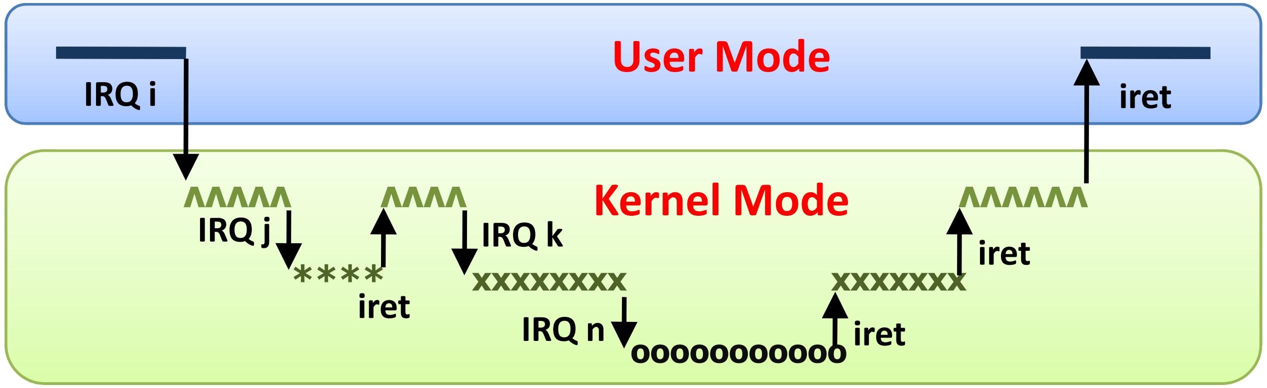
- /proc/interrupts

cafe.kaist.ac.kr - PuTTY

```
[kernel:/u/jinsoo-4] cat /proc/interrupts
           CPU0
  0: 3651631242          XT-PIC  timer
  1:       8324          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  5:          0          XT-PIC  uhci_hcd:usb4
  8:          1          XT-PIC  rtc
  9:    3973280          XT-PIC  CMI8738-MC6, ehci_hcd:usb1, ohci_hcd:usb2, ohci
_hcd:usb3, uhci_hcd:usb5, ohci1394, eth1
 10:  303073663          XT-PIC  eth0
 11:  596065717          XT-PIC  mga@pci:0000:01:00.0
 12:      23621          XT-PIC  i8042
 14:   11521708          XT-PIC  ide0
 15:         22          XT-PIC  ide1
NMI:          0
ERR:          0
[kernel:/u/jinsoo-5]
```

Nested Interrupts

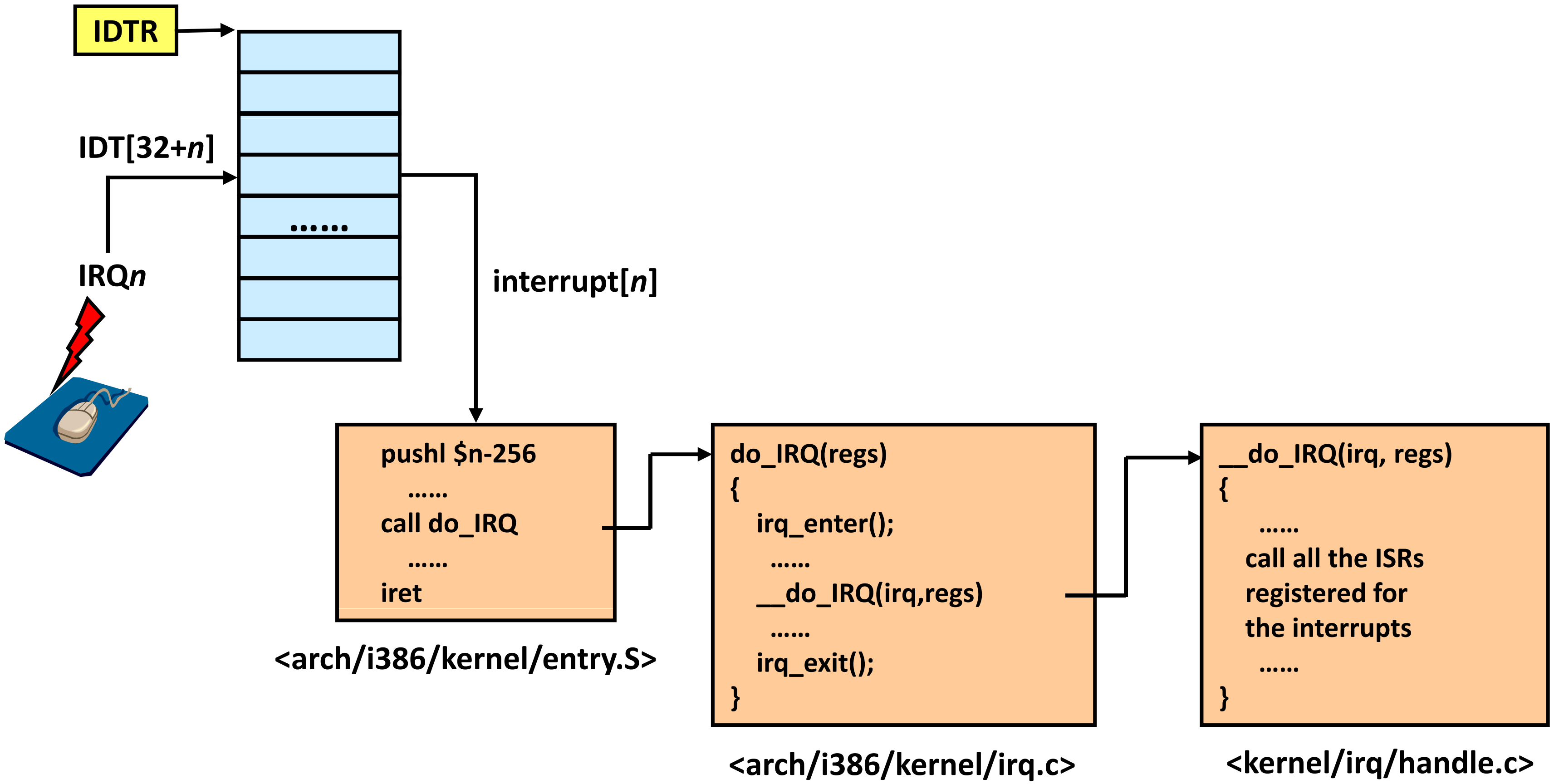
- **Nested interrupts and kernel control paths**
  - An interrupt handler may be interrupted.



    - The execution of kernel control paths can be nested.
    - An interrupt handler must never block.
      - » No process switch can take place until an interrupt handler is running.
      - » Interrupt handlers never generate page faults.
  - Interrupt handlers in Linux need not be reentrant.

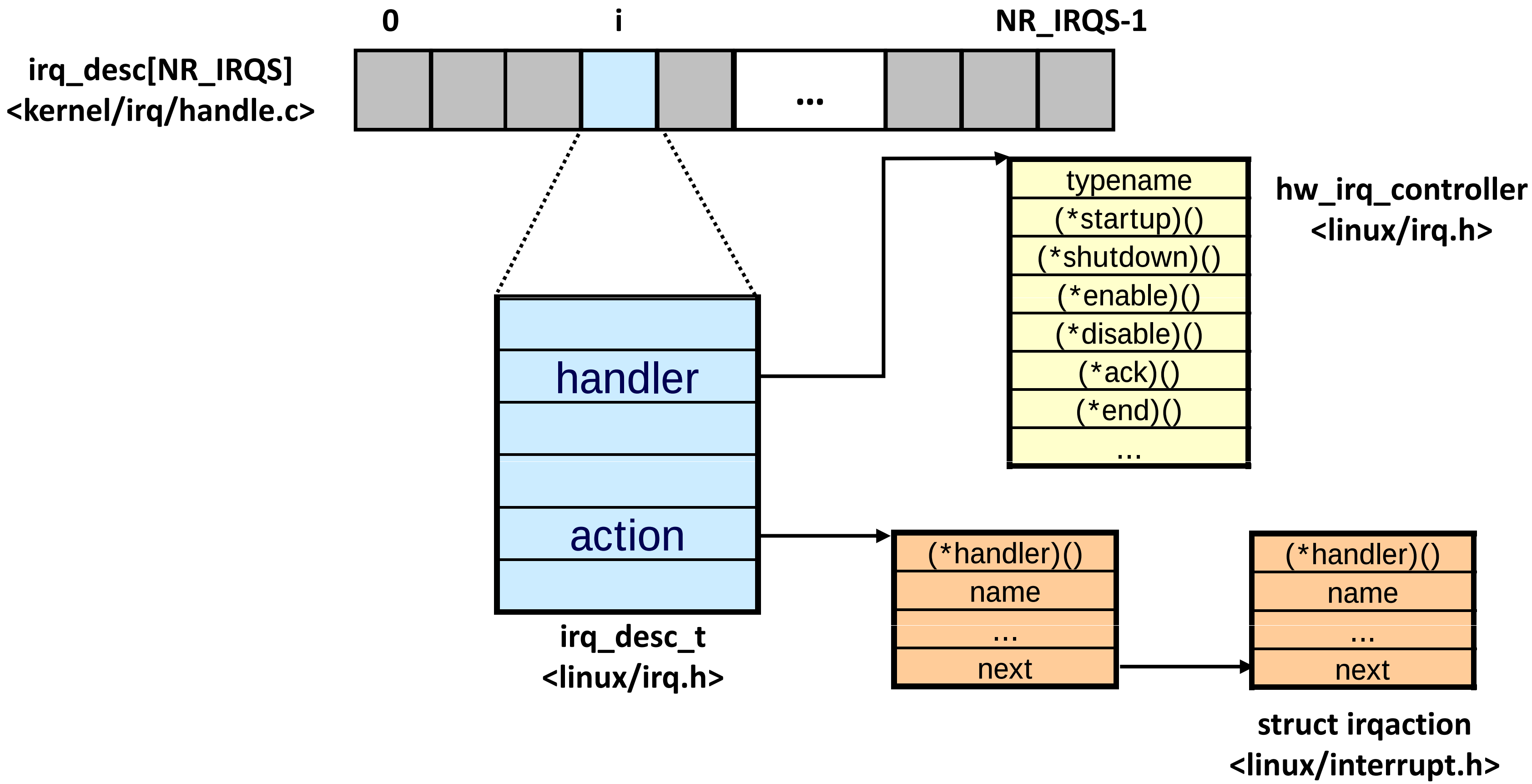
Interrupt Handling (1)

IDT (Interrupt Descriptor Table)



Interrupt Handling (2)

- **IRQ data structures**



Related Kernel Functions (1)

- **Registering/freeing an interrupt handler**
  - int request_irq(unsigned int irq,
    irqreturn_t (*handler)(int, void*, struct pt_regs *),
    unsigned long irqflags, const char *devname, void *dev_id);
  - int free_irq(unsigned int irq, void *dev_id);
  - irqflags
    - SA_INTERRUPT: “fast interrupt handler”
      - » Run with all interrupts disabled on the local processor
      - » Timer interrupt
    - SA_SAMPLE_RANDOM
      - » Interrupts contribute to the kernel entropy pool.
      - » Should not be enabled for periodic interrupts.
    - SA_SHIRQ
      - » The interrupt line can be shared.

Related Kernel Functions (2)

- **Disabling/enabling interrupts**
  - local_irq_disable();

    ...

    local_irq_enable();

    Dangerous! Why?

  - local_irq_save(unsigned long flags);

    ...

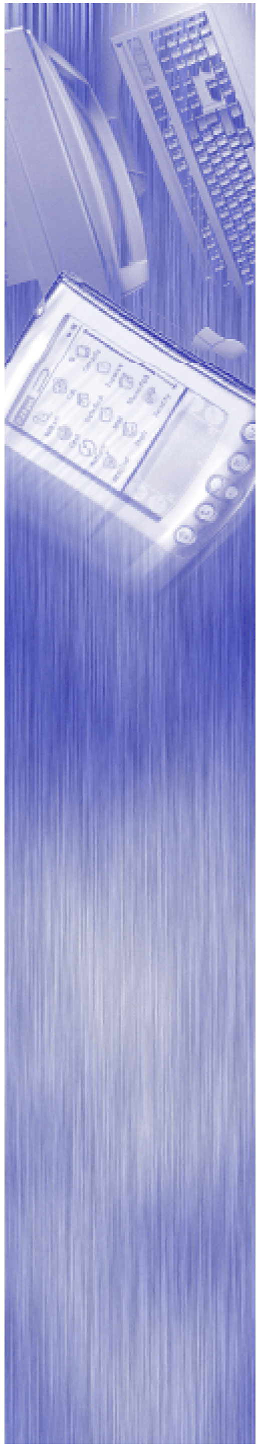
    local_irq_restore(unsigned long flags);

Related Kernel Functions (3)

- **Controlling a specific interrupt line**
  - void disable_irq(unsigned int irq);
    - Disable the given interrupt line and ensure no handler on the line is executing before returning
  - void disable_irq_nosync(unsigned int irq);
    - Disable the given interrupt line
  - void enable_irq(unsigned int irq);
    - Enable the given interrupt line
  - void synchronize_irq(unsigned int irq);
    - Waits for a specific interrupt handler to exit, if it is executing
  - int irqs_disabled(void);
    - Nonzero if local interrupt delivery is disabled. Otherwise, zero.

Related Kernel Functions (4)

- **Status of the interrupt system**
  - in_interrupt()
    - Nonzero if the kernel is in interrupt context.
    - Either executing an interrupt handler or a bottom half handler.
  - in_irq()
    - Nonzero only if the kernel is specifically executing an interrupt handler.



CS632/SEP564: Embedded Operating Systems (Fall 2008)

Bottom Halves

KAIST

Deferring Work (1)

- **Why?**
  - Interrupt handlers need to run as quickly as possible.
    - It may interrupt other potentially important code, including other interrupt handlers.
    - Interrupts at the same level (without SA_INTERRUPT), or all the other interrupts (with SA_INTERRUPT) are disabled while an interrupt handler is running.
  - Interrupt handlers are often very timing critical because they deal with hardware.
  - Interrupt handlers do not run in process context, therefore they cannot block.

Deferring Work (2)

- **General steps for interrupt handling**

Critical actions

: Acknowledge an interrupt to the PIC.
: Reprogram the PIC or the device controller.
: Update data structures shared by multiple devices

Reenable interrupts

Noncritical actions

: Exchange command/data/status with the device (e.g., reading the scan code from the keyboard)

Return from interrupts

Noncritical deferred actions

: Actions may be delayed.
: Copy buffer contents into the address space of some process (e.g., sending the keyboard line buffer to the terminal handler process).
: Bottom half (Softirqs, Tasklets, and Work Queues)

Deferring Work (3)

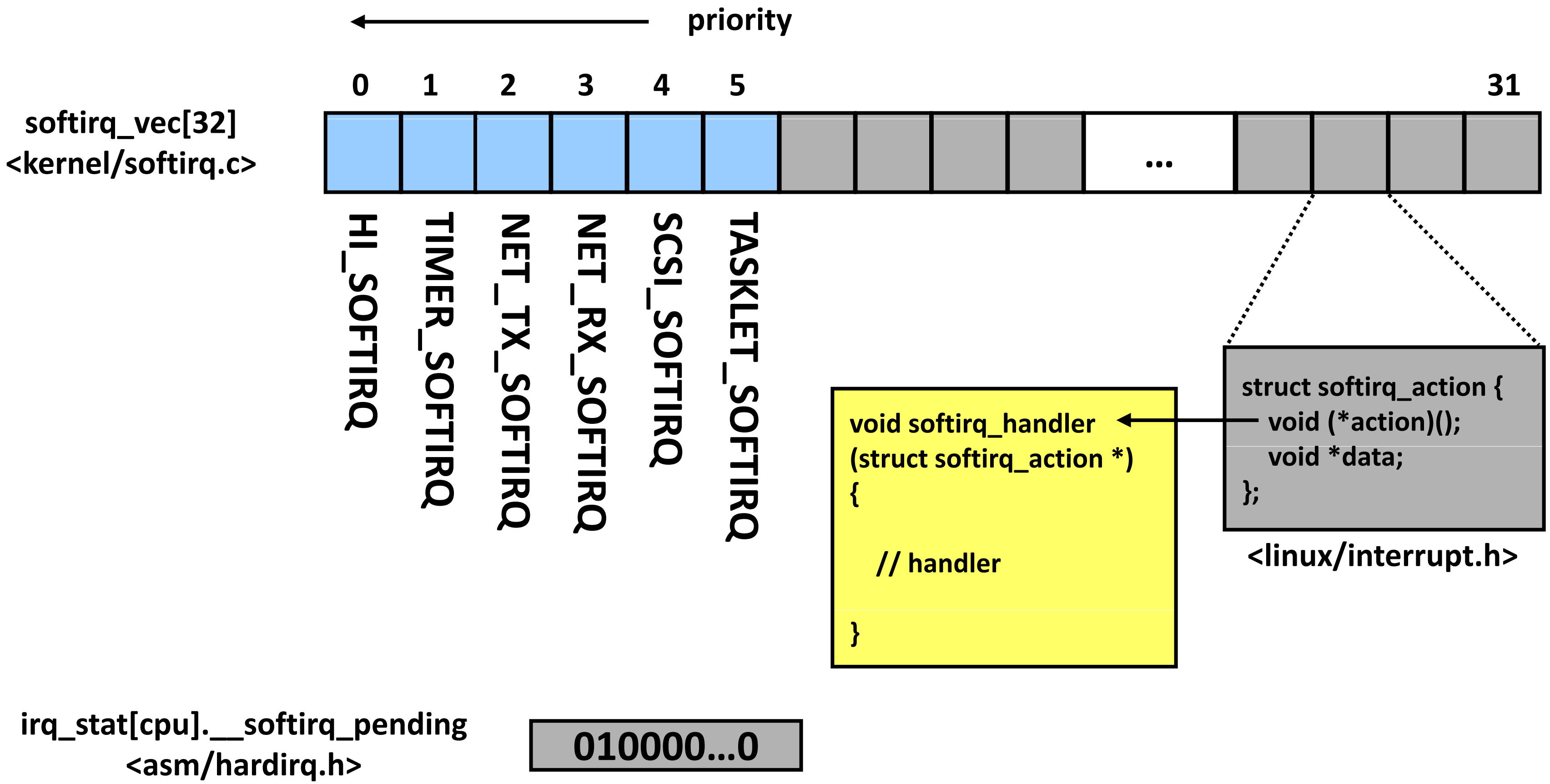
- **Bottom halves in Linux**
  - BH (Bottom Half), Task queues
    - Removed in 2.5
  - Softirqs, Tasklets
    - Available since 2.3
  - Work queues
    - Available since 2.5

Deferring Work (4)

- **Softirqs**
  - A set of 32 statically allocated bottom halves
    - Softirqs must be registered statically at compile-time.
  - Softirqs of the same type can run concurrently on several CPUs.
    - Softirqs need to be reentrant.
- **Tasklets**
  - Tasklets are implemented on top of softirqs.
  - Two different tasklets can run concurrently on different processors, but tasklets of the same type are always serialized.
  - Tasklets can be allocated and initialized at runtime.

Softirqs (1)

- **Kernel data structures for softirqs**



Softirqs (2)

- **Registering a handler**
  - void open_softirq(int nr,
    void (*action)(struct softirq_action *), void *data);
  - The softirq handler should be reentrant!
- **Raising a handler**
  - void raise_softirq(unsigned int nr);
    - Disables interrupts prior to actually raising the softirq, and then restores them to their previous state.
  - void raise_softirq_irqoff(unsigned int nr);
    - Used when the interrupts are already off.
  - Softirqs are most often raised from within interrupt handlers.

Softirqs (3)

- **Executing softirqs**
  - In the return from hardware interrupt code
    - When the do_IRQ() finishes handling an I/O interrupt, it invokes irq_exit()
    - irq_exit() then calls do_softirq() if it is not in_interrupt() and local softirqs are pending.
  - In the ksoftirqd kernel thread
  - In any code that explicitly checks for and executes pending softirqs
    - e.g., networking subsystem
  - When the bottom half processing is enabled
    - local_bh_enable();
  - ...

Softirqs (4)

- **do_softirq() → __do_softirq()** <kernel/softirq.c>
  - New pending softirqs might pop up.
    - Iterate maximum MAX_SOFTIRQ_RESTART (10) times to handle new softirqs.
    - Prevent user mode processes from severe delaying.
    - The remaining softirqs are handled by ksoftirqd.

```
void __do_softirq(void)
{
   ...
   pending = local_softirq_pending();
restart:
   h = softirq_vec;
   do {
     if (pending & 1) {
       h->action(h);
     }
     h++;
     pending >>= 1;
   while (pending);
   pending = local_softirq_pending();
   if (pending && --max_restart)
     goto restart;
   if (pending)
     wakeup_softirqd();
   ...
}
```

Softirqs (5)

- **ksoftirqd**
  - Each CPU has its own ksoftirqd/n kernel thread.
  - Trade-off between softirq latency vs. user process performance when softirqs are activated at very high frequency.

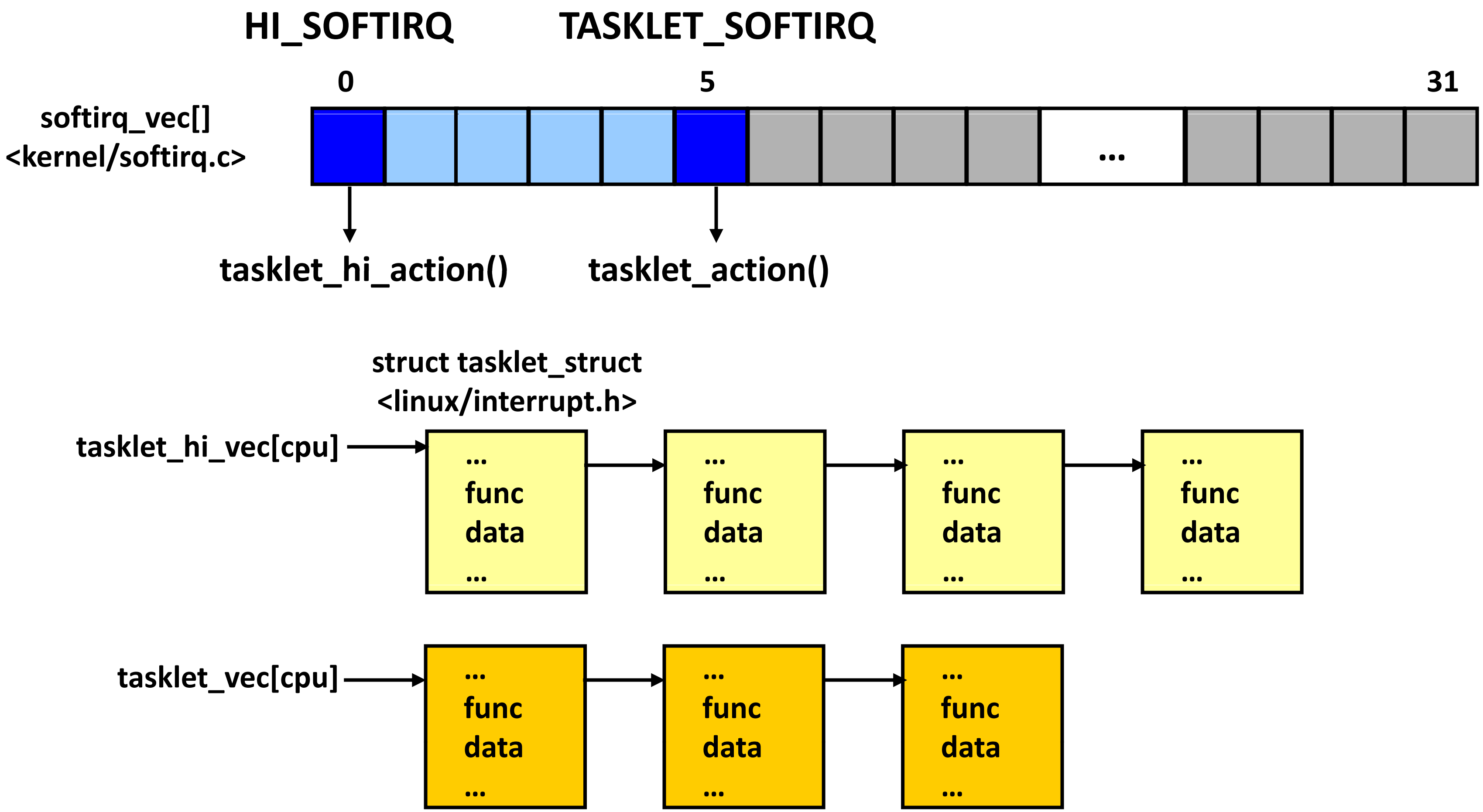
```
ksoftirqd()
{
   ...
   for (;;) {
     set_current_state (TASK_INTERRUPTIBLE);
     schedule();
     while (local_softirq_pending()) {
       preempt_disable();
       do_softirq();
       preempt_enable();
       cond_resched();
     }
}
```

Tasklets (1)

- **Characteristics**
  - Tasklets have nothing to do with tasks. (bad naming!)
  - Tasklets are built on top of softirqs.
  - Tasklets work just fine for the vast majority of cases.
    - Softirqs are required only for very high-frequency and highly threaded uses.
  - Only one tasklet of a given type is running at the same time.
  - As with softirqs, tasklets cannot sleep.

Tasklets (2)

- **Kernel data structures for tasklets**



Tasklets (3)

- **Creating a tasklet**
  - DECLARE_TASKLET(name, func, data)
    - struct tasklet_struct name =
      {NULL, 0, ATOMIC_INIT(0), func, data }
  - void tasklet_init(struct tasklet_struct *t,
    void (*func)(unsigned long), unsigned long data);
- **Scheduling tasklets**
  - void tasklet_schedule(struct tasklet_struct *t);
  - void tasklet_hi_schedule(struct tasklet_struct *t);
  - Raise tasklets
    - Add the tasklet to tasklet_vec or tasklet_hi_vec.
    - Raise the TASKLET_SOFTIRQ or HI_SOFTIRQ softirq.

Tasklets (4)

- **Enabling/disabling tasklets**
  - Enabling/disabling is controlled by an atomic counter.
    - If it's nonzero, the tasklet is disabled and cannot run.
  - void tasklet_disable(struct tasklet_struct *t);
    - If the tasklet is currently running, the function will not return until it finishes executing.
  - void tasklet_disable_nosync(struct tasklet_struct *t);
    - Does not wait for the tasklet to complete prior to returning.
  - void tasklet_enable(struct tasklet_struct *t);
  - void tasklet_kill(struct tasklet_struct *t);
    - Waits for the tasklet to finish executing and then removes the tasklet from the queue.

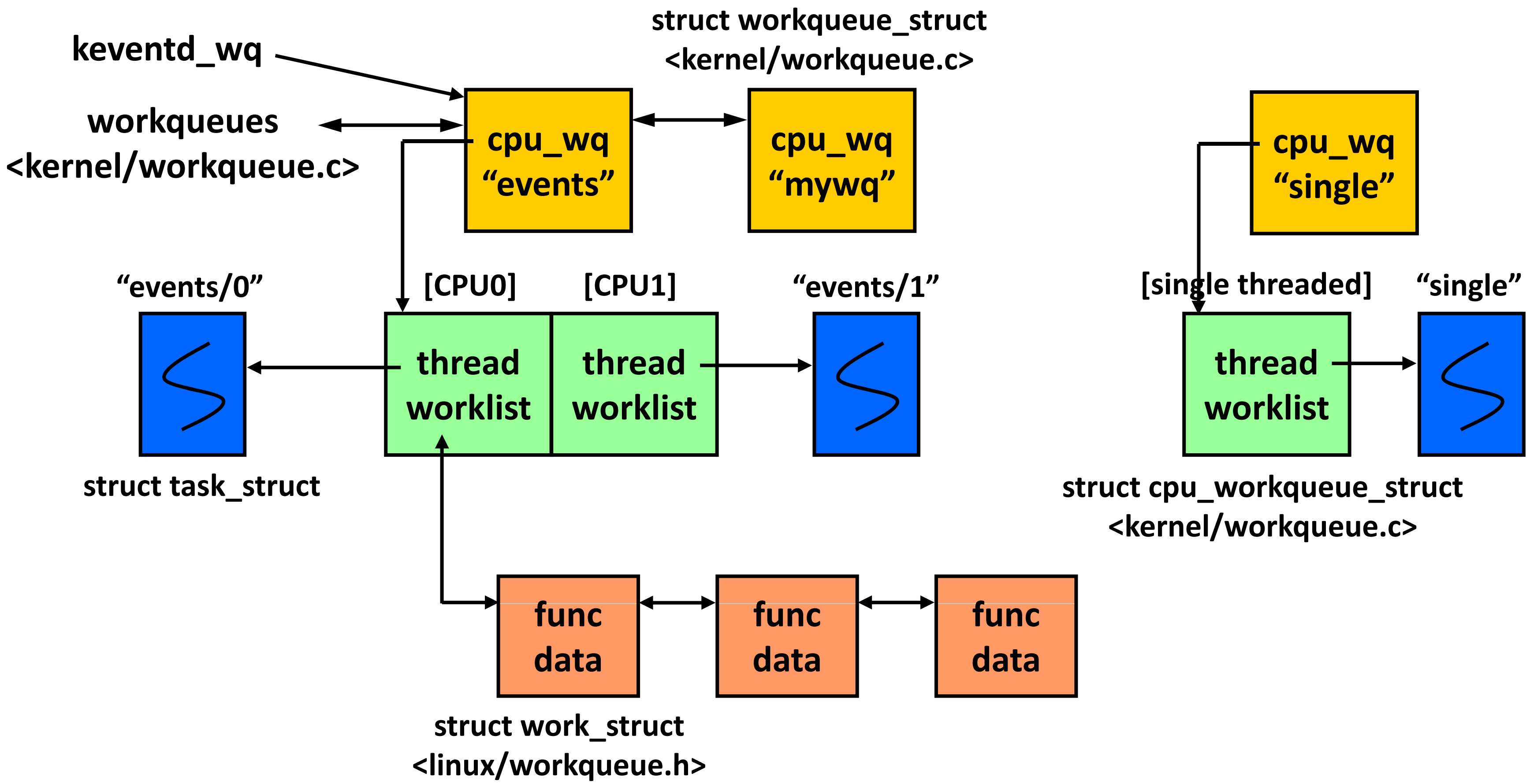
Work Queues (1)

▪ Characteristics

- Work queues defer work into a kernel thread.
- The only bottom-half mechanism that runs in process context.
- Work queues are schedulable and can therefore sleep.
- Useful for situations where you need to
  - allocate a lot of memory
  - obtain a semaphore
  - perform block I/O, etc.
- The default worker threads are called events/n.
- You can create your own worker thread for processor-intense and performance-critical work.

Work Queues (2)

- **Kernel data structures for work queues**



Work Queues (3)

- **Worker thread events**
  - run_workqueue() calls the function specified in the work_struct.
  - The corresponding entry is removed from the worklist.

```
worker_thread ()
{
   ....
   for (;;) {
     set_task_state (current, TASK_INTERRUPTIBLE);
     add_wait_queue (&cwq->more_work, &wait);
     if (list_empty (&cwq->worklist))
       schedule();
     else
       set_task_state (current, TASK_RUNNING);
     remove_wait_queue (&cwq->more_work, &wait);
     if (!list_empty (&cwq->worklist))
       run_workqueue (cwq);
}
```

Work Queues (4)

- **Creating/destroying new work queues**
  - struct workqueue_struct *create_workqueue
    (const char *name);
  - struct workqueue_struct *create_singlethread_workqueue
    (const char *name);
  - void destroy_workqueue(struct workqueue_struct *wq);

- **Creating work**
  - DECLARE_WORK(name, void (*func)(void *), void *data);
  - INIT_WORK(struct work_struct *work, void (*func)(void *),
    void *data);

Work Queues (5)

- **Scheduling work to eventd**
  - int schedule_work (struct work_struct *work);
  - int schedule_delayed_work(struct work_struct *work, unsigned long delay);
    - Not execute for at least delay timer ticks into the future.
  - int schedule_delayed_work_on(int cpu, struct work_struct *work, unsigned long delay);
  - void flush_scheduled_work(void);
    - Waits until all entries in the queue are executed before returning.
  - int cancel_delayed_work(struct work_struct *work);
    - Cancels the pending work

Work Queues (6)

- **Scheduling work to general work queues**
  - int queue_work(struct workqueue_struct *wq,
    struct work_struct *work);
  - int queue_delayed_work(struct workqueue_struct *wq,
    struct work_struct *work, unsigned long delay);
  - void flush_workqueue(struct workqueue_struct *wq);

Summary

- **Which to use?**

Bottom Half	Context	Serialization
Softirqs	Interrupt	None
Tasklets	Interrupt	Against the same tasklet
Work queues	Process (may sleep)	None