ECE391
Computer System Engineering

Lecture 12

Dr. Zbigniew Kalbarczyk

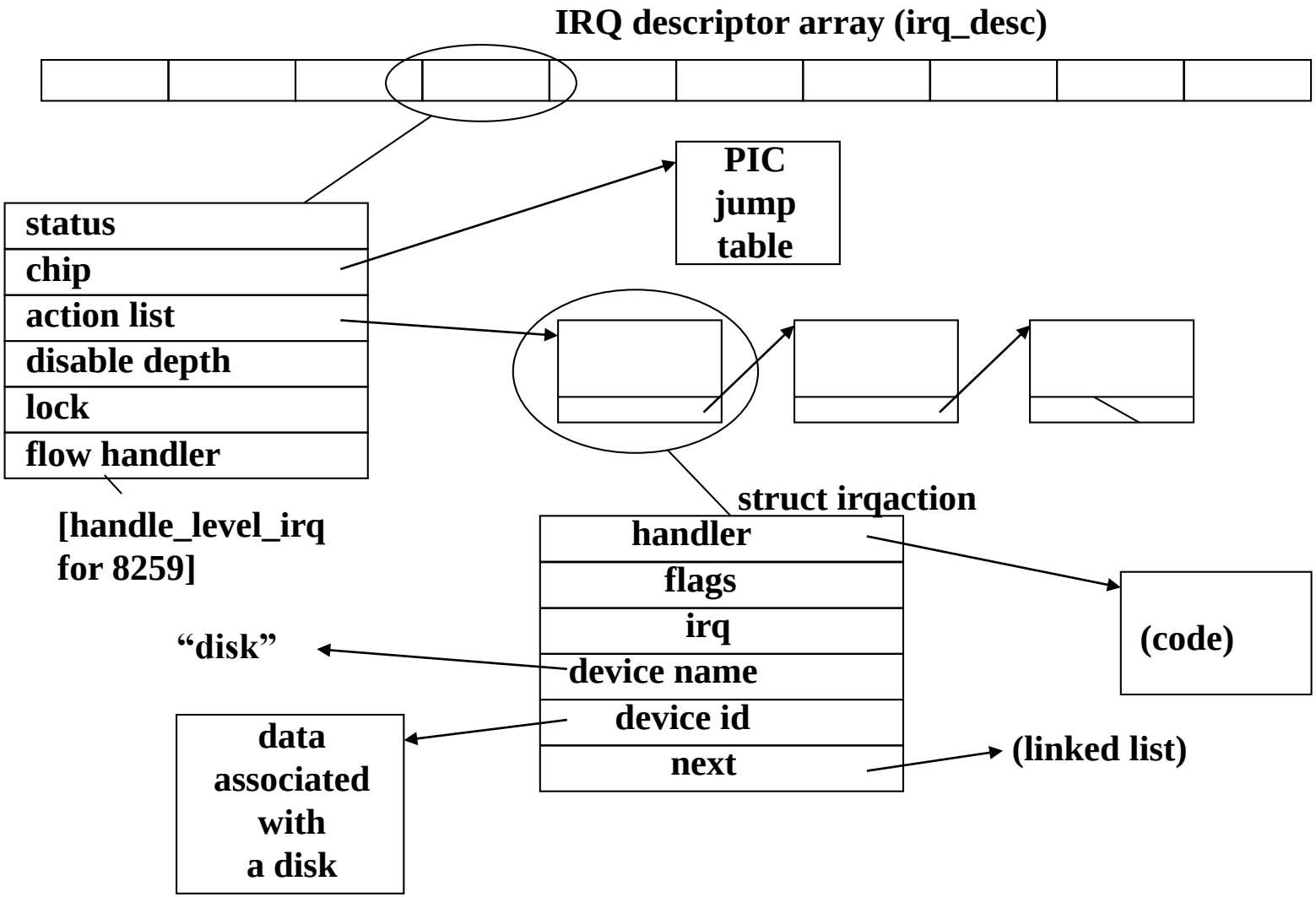
University of Illinois at Urbana- Champaign

Spring 2021

Lecture Topics

- Linux interrupt system
  - data structures
  - handler installation & removal
  - invocation
  - execution

Linux' Interrupt Data Structures



Linux' request_irq (kernel/irq/manage.c)

```
int
request_irq (unsigned int irq,
             irq_handler_t handler,
             unsigned long irqflags,
             const char* devname,
             void* dev_id)
{
    struct irqaction* action;
    int retval;

    if (irq >= NR_IRQS)
        return -EINVAL;
    if (!handler)
        return -EINVAL;

    action = kmalloc (sizeof (struct irqaction), GFP_ATOMIC);
    if (!action)
        return -ENOMEM;

    action->handler = handler;
    action->flags   = irqflags;
    cpus_clear (action->mask);
    action->name    = devname;
    action->next    = NULL;
    action->dev_id  = dev_id;

    select_smp_affinity (irq);

    retval = setup_irq (irq, action);
    if (retval)
        kfree (action);

    return retval;
}
```

irq # (0-15 for PIC)

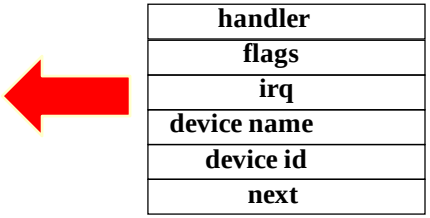
pointer to interrupt handler

bit vector

human-readable device name

pointer to device-specific data

Dynamically allocate new action structure for linked list of actions



Fill in the new action structure

Add the new action structure to the link list of actions

Upon failure free the action structure

Comments on `request_irq` – Arguments

- **`irq`** – irq # (0-15 for PIC)
- **`handler`** – pointer to interrupt handler with following arguments
  - irq #
  - dev_id pointer (type void*)
  - handlers should return 1 if handled, 0 if not
- **`irqflags`** – bit vector
  - IRQF_SHARED – interrupt chaining is allowed
  - IRQF_DISABLED – execute handlers with IF=0
  - IRQF_SAMPLE_RANDOM – use device timing as source of random numbers
- **`devname`** – human-readable device name (visible in /proc/interrupts)
- **`dev_id`** – pointer to device-specific data (returned to handler when called)

Comments on `request_irq` (cont.)

- Start by checking values
  (two checks from the shown code)
- Dynamically allocate new action structure for linked list of actions
- Fill in the new action structure
- Try to add it
  - using **`setup_irq`**
  - if call fails, free the action structure

Linux setup_irq

(kernel/irq/manage.c)

```
int
setup_irq (unsigned int irq, struct irqaction* new)
{
    struct irq_desc* desc = irq_desc + irq;
    struct irqaction* old;
    struct irqaction** p;
    unsigned long flags;
    int shared = 0;

    if (irq >= NR_IRQS)
        return -EINVAL;
    if (desc->chip == &no_irq_chip)
        return -ENOSYS;

    if (new->flags & IRQF_SAMPLE_RANDOM)
        rand_initialize_irq (irq);

    spin_lock_irqsave (&desc->lock, flags);
    p = &desc->action;
    if ((old = *p) != NULL) {;
        /* Can't share interrupts unless both agree to and are same type. */
        if (!((old->flags & new->flags) & IRQF_SHARED) ||
            ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK)) {
            spin_unlock_irqrestore (&desc->lock, flags);
            return -EBUSY;
        }

        /* add new interrupt at end of irq queue */
        do {
            p = &old->next;
            old = *p;
        } while (old);
        shared = 1;
    }

    *p = new;

    if (!shared) {
        irq_chip_set_defaults (desc->chip);
        desc->status &= ~(IRQ_AUTODETECT | IRQ_WAITING | IRQ_INPROGRESS);
        if (!(desc->status & IRQ_NOAUTOEN)) {
            desc->depth = 0;
            desc->status &= ~IRQ_DISABLED;
            desc->chip->startup (irq);
        } else {
            /* Undo nested disables: */
            desc->depth = 1;
        }
    }

    spin_unlock_irqrestore (&desc->lock, flags);

    new->irq = irq;
    register_irq_proc (irq);
    new->dir = NULL;
    register_handler_proc (irq, new);

    return 0;
}
```

Critical section begins

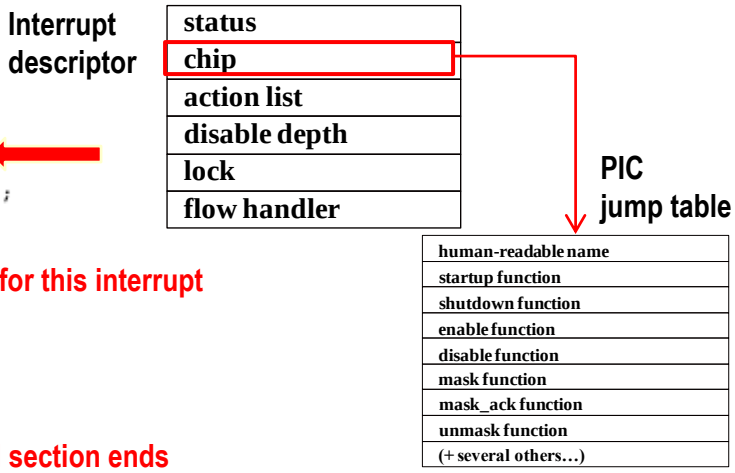
New action is added at the end of the link list

Startup PIC for this interrupt

Critical section ends

Create new /proc/irq/<irq#> directory

Add handler subdirectory in /proc/irq/<irq#>/<action name>



Comments on `stup_irq`

- Start with a couple of sanity checks

- If random sampling flag is set
  - initialize as source of random numbers
  - good random numbers are hard to find!
  - devices such as disks can be used because of “random” rotational latency to read data (for example)

Comments on `stup_irq` (cont.)

- Critical section (most of function)
  - blocks other activity on descriptor
  - uses irqsave/restore to allow handlers to be added from any context
- Note that new action goes at the end of the linked list, not the start
- Interrupt chaining only allowed
  - if all handlers agree to it
  - otherwise only first handler is successfully added

Comments on `stup_irq` (cont.)

- If this handler is the first
  - make sure that PIC jump table has proper default functions
  - clear some status flags
  - clear any previous software disablement (depth)
  - call the PIC startup function

- After critical section
  - create **`/proc/irq/`**<irq#> directory if necessary
  - add handler subdirectory in **`/proc/irq/`**<irq #>/<action name>

Linux free_irq
(kernel/irq/manage.c)

```
void
free_irq (unsigned int irq, void* dev_id)
{
    struct irq_desc* desc;
    struct irqaction** p;
    unsigned long flags;

    if (irq >= NR_IRQS)
        return;

    desc = irq_desc + irq;
    spin_lock_irqsave (&desc->lock, flags);
    p = &desc->action;
    for (;;) {
        struct irqaction* action = *p;

        if (action) {
            struct irqaction** pp = p;

            p = &action->next;
            if (action->dev_id != dev_id)
                continue;

            /* Found it - now remove it from the list of entries */
            *pp = action->next;

            if (!desc->action) {
                desc->status |= IRQ_DISABLED;
                desc->chip->shutdown (irq);
            }
            spin_unlock_irqrestore (&desc->lock, flags);

            /* Remove from /proc/irq/<irq #> */
            unregister_handler_proc (irq, action);

            /* Make sure it's not being used on another CPU */
            synchronize_irq (irq);

            kfree (action);
            return;
        }
        printk (KERN_ERR "Trying to free already-free IRQ %d\n", irq);
        spin_unlock_irqrestore (&desc->lock, flags);
        return;
    }
}
```

Find the correct irq descriptor

Critical section begins

Search for action in list

The “continue” causes the rest of the statement body in the loop to be skipped.

No more handlers for the IRQ; you can invoke PIC shutdown

Critical section ends

Remove handler subdirectory in /proc/irq/<irq#>/<action name>

Free the action structure

Critical section ends

Comments on `free_irq`

- Arguments
  - **`irq`** – irq # (0-15 for PIC)
  - **`dev_id`** – MUST match pointer value used in **`request_irq`** call
- Start by checking arguments
- Critical section starts to prevent manipulations of descriptor
  - blocks interrupt from starting to execute
  - (interrupt may already be executing on another processor)

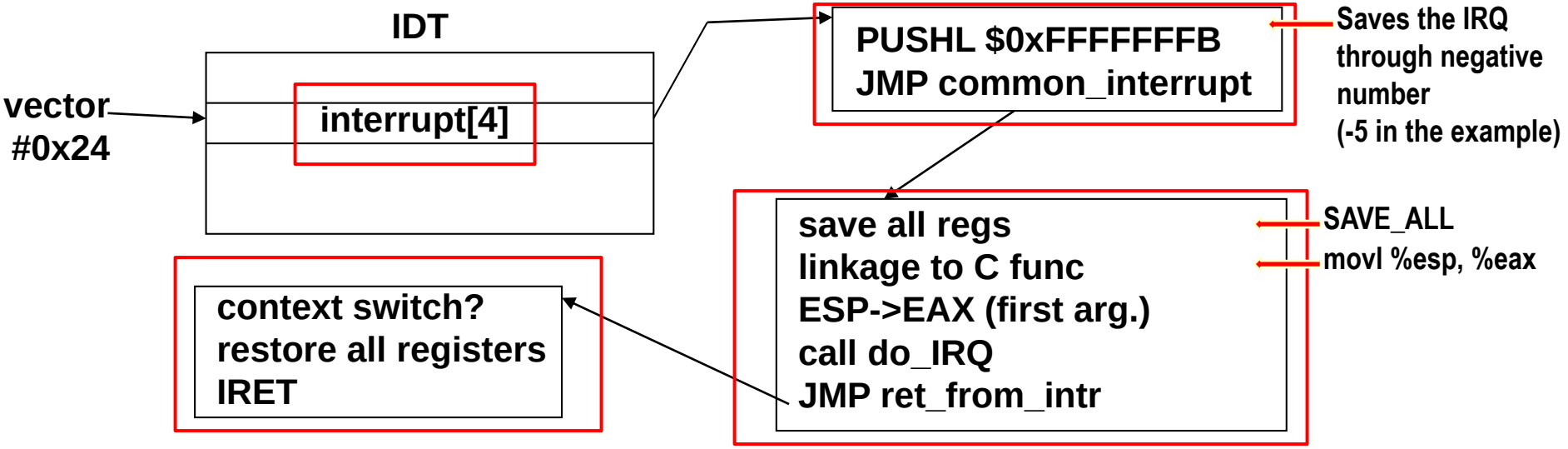
Comments on `free_irq` (cont.)

- Search for action in list
  - based on **`dev_id`** pointer
  - hence need for matching pointer
  - passing NULL **`dev_id`** not allowed with chained interrupts
- Once found (notice that most of the loop executes exactly once)
  - remove the action
  - if this action/handler was the last for this interrupt,
    - turn on software disablement
    - (doing so signals interrupts waiting for descriptor lock to abort once they obtain the lock)
    - call the PIC shutdown function

Comments on `free_irq` (cont.)

- Remove the **`/proc/irq/`**<irq #>/<action name> entry
- On an SMP
  - interrupt may be executing on another processor
  - need to wait for it to finish after releasing descriptor lock
- Finally, free the action structure

Interrupt Invocation



- Why funnel all IRQs into one piece of code(instead of producing one function per IRQ) ?
  - kernel code/size versus speed tradeoff
    - perhaps no big deal for 16
    - keep in mind that you're saving a tiny number of instructions
    - APIC used by most SMPs has 256 vectors

Interrupt Descriptor Table

0x00–0x1F defined by Intel	0x00	division error	
	⋮		
	0x02	NMI (non-maskable interrupt)	
	0x03	breakpoint (used by KGDB)	
	0x04	overflow	
	⋮		
	0x0B	segment not present	
	0x0C	stack segment fault	
	0x0D	general protection fault	
	0x0E	page fault	
	⋮		
0x20–0x27 master 8259 PIC	0x20	IRQ0 — timer chip	example of possible settings
	0x21	IRQ1 — keyboard	
	0x22	IRQ2 — (cascade to slave)	
	0x23	IRQ3	
	0x24	IRQ4 — serial port (KGDB)	
	0x25	IRQ5	
	0x26	IRQ6	
	0x27	IRQ7	
0x28–0x2F slave 8259 PIC	0x28	IRQ8 — real time clock	
	0x29	IRQ9	
	0x2A	IRQ10	
	0x2B	IRQ11 — eth0 (network)	
	0x2C	IRQ12 — PS/2 mouse	
	0x2D	IRQ13	
	0x2E	IRQ14 — ide0 (hard drive)	
	0x2F	IRQ15	
0x30–0x7F	⋮	APIC vectors available to device drivers	
0x80	0x80	system call vector (INT 0x80)	
0x81–0xEE	⋮	more APIC vectors available to device drivers	
0xEF	0xEF	local APIC timer	
0xF0–0xFF	⋮	symmetric multiprocessor (SMP) communication vectors	

Stack on Entry to do_IRQ