ECE391
Computer System Engineering

Lecture 10

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2018

Lecture Topics

- Linux abstraction of PIC
- General interrupt abstractions
- Linux interrupt system
  - data structures
  - handler installation & removal
  - invocation
  - execution
  - tasklets

ECE391 EXAM 1

- **EXAM I – October 2; 7:00pm-9:00pm**
  - **Seating:**
    - ECEB 1002: last name starting with A to W
    - ECEB 1013: last name starting with X to Z
- **Conflict Exam**
  - Tuesday October 2, 2:00pm
  - Location: ECEB 2015
- **NO Lecture on Tuesday, October 2**
- **Review Session**
  - Saturday September 29, 4:00pm
  - Location: ECEB 1002

Linux Abstraction of PICs

- Uses a jump table
  - same as vector table (array of function pointers)

PIC object

- Table is *hw_irq_controller* structure (or struct irq_chip)
  - each vector # associated with a table
  - table used to interact with appropriate PIC (e.g., 8259A, or Advanced PIC)

human-readable name
startup function
shutdown function
enable function
disable function
mask function
mask_ack function
unmask function
(+ several others…)

Linux Abstraction of PICs

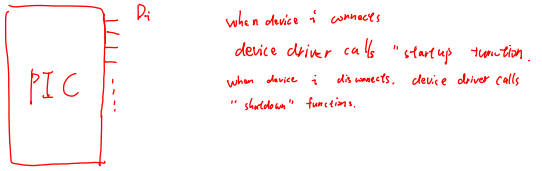
- *hw_irq_controller* structure definition
  - IRQs are #'d 0-15 (correspond to vector # - 0x20)

```
const char* name;
unsigned int (*startup)(unsigned int irq);
void (*shutdown)(unsigned int irq);
void (*enable)...
void (*disable)...
void (*ack)...
void (*end)...
/* we'll ignore the others... */
```

8259A's human-readable name is "XT-PIC"; see /proc/interrupts

PIC Functions in Jump Table: Explanation

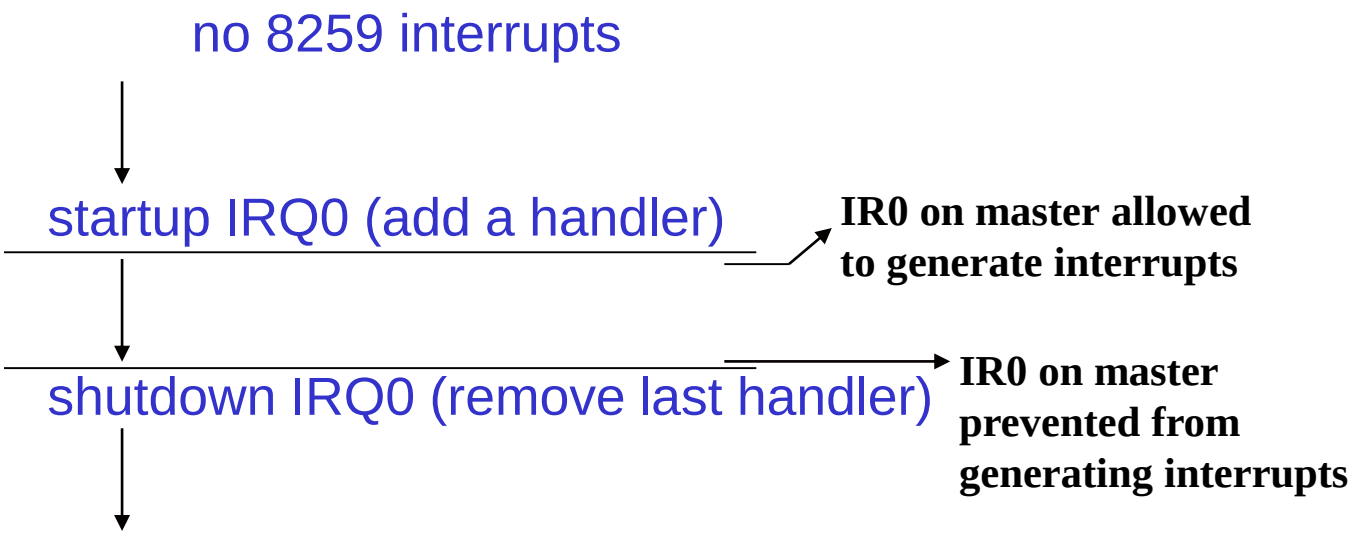
- Initially, all 8259A interrupts are masked out using mask on 8259A



- *startup* and *shutdown* functions
  - startup is called when first handler is installed for an interrupt
  - shutdown is called after last handler is removed for an interrupt
  - both functions change the corresponding mask bit in 8259A implementaion

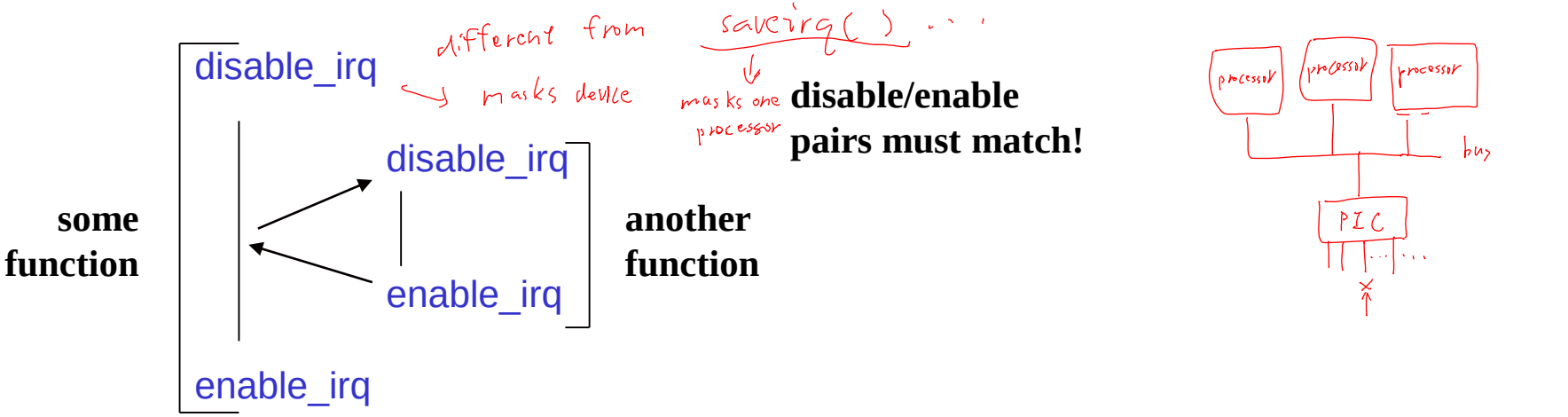
PIC Functions in Jump Table: Explanation (cont.)

- Example



PIC Functions in Jump Table (cont.)

- disable/enable functions
  - used to support nestable interrupt masking (disable_irq, enable_irq)



  - on 8259
    - first *disable_irq* calls jump table disable, which masks interrupt on PIC
    - last *enable_irq* calls jump table enable, which unmasks interrupt on PIC

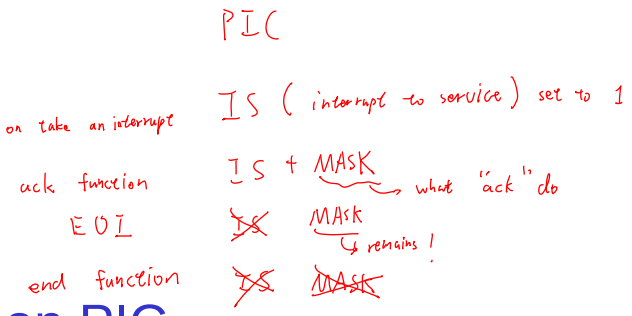
PIC Functions in Jump Table (cont.)

- ack function

  acknowledgement

  - called at start of interrupt handling to ack receipt of the interrupt
  - on 8259 (mask and ack) , masks interrupt on PIC, then sends EOI to PIC

- end function
  - called at end of interrupt handling
  - on 8259, enables interrupt (unmasks it) on PIC

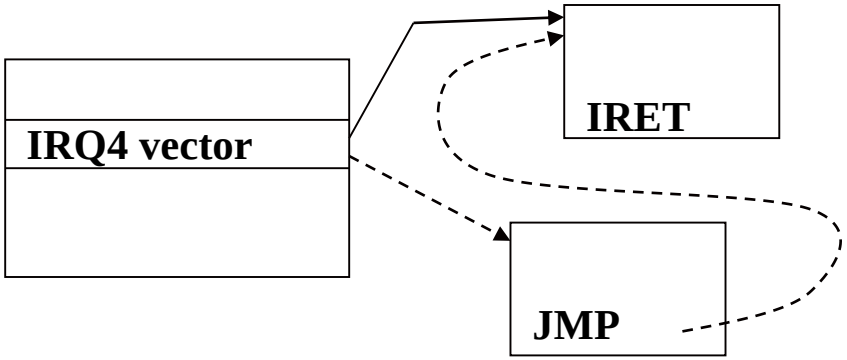


General Interrupt Abstractions: Interrupt Chaining

- Hardware view: 1 interrupt $\rightarrow$ 1 handler
- Problems
  - may have > 15 devices
  - > 1 software routines may want to act in response to device
  - examples:
    - hotkeys for various functions
    - move mouse to lower-right corner to start screen-saver

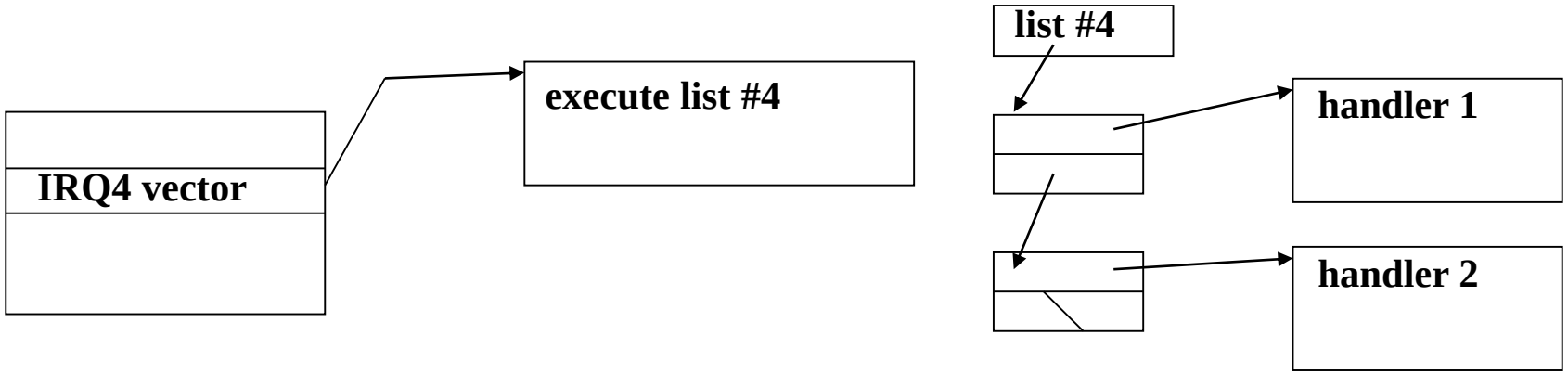
General Interrupt Abstractions: Interrupt Chaining (cont.)

- One approach
  - used by terminate and stay resident (TSR) programs in DOS
  - form linked list (chain) of handlers using JMP instructions
  - not very clean
    - no way to remove self
    - unless you're first in list
  - to be fair
    - TSR program not designed for removal



General Interrupt Abstractions
Interrupt Chaining (cont.)

- Solution
  - interrupt chaining with linked list data structure
  - (not list embedded into code!)



General Interrupt Abstractions: Interrupt Chaining (cont.)

- Drawbacks of chaining
  - for > 1 device
    - must query devices to see if they raised interrupt
    - not always possible
  - for 1 device
    - must avoid stealing data/confusing device
    - example
      - by sending two characters to serial port
      - in response to interrupt declaring port ready for one char.
    - another example
      - reading mouse location twice
      - if device protocol specifies reading once per interrupt

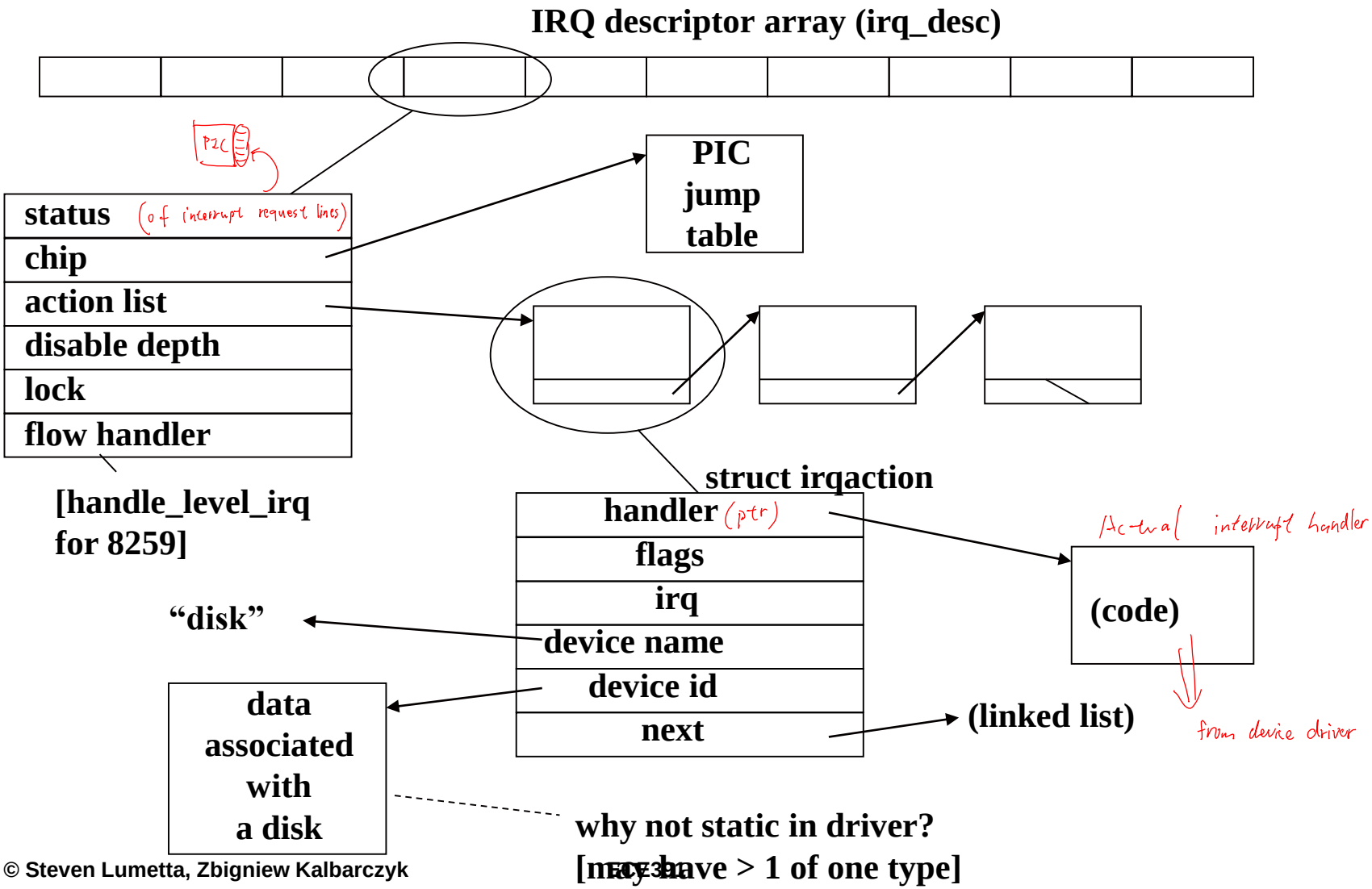
General Interrupt Abstractions: Soft Interrupts (cont.)

- Recall: why support interrupts?
  - slow device gets timely attention from fast processor
  - processor gets device responses without repeatedly asking for them
- A useful concept in software
  - example: network encryption/decryption
  - packet arrives, given to decrypter
  - when decrypter (software program) is done
    - want to interrupt program
    - to transfer data from packet
  - but has no access to INTR pin

General Interrupt Abstractions: Soft Interrupts (cont.)

- Solution
  - software-generated (soft) interrupt
  - (similarly, but later, signals—user-level soft interrupts)
  - runs at priority between program and hard interrupts
  - usually generated
    - by hard interrupt handlers
    - to do work not involving device
- Linux version is called tasklets
  - used by code provided to you for MP1
  - discussed later

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)

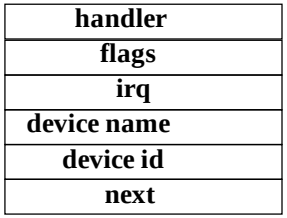
pointer to interrupt handler

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

Interrupt descriptor

status
chip
action list
disable depth
lock
flow handler

PIC jump table

human-readable name
startup function
shutdown function
enable function
disable function
mask function
mask_ack function
unmask function
(+several others…)

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