ECE391
Computer System Engineering

Lecture 12

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Fall 2018

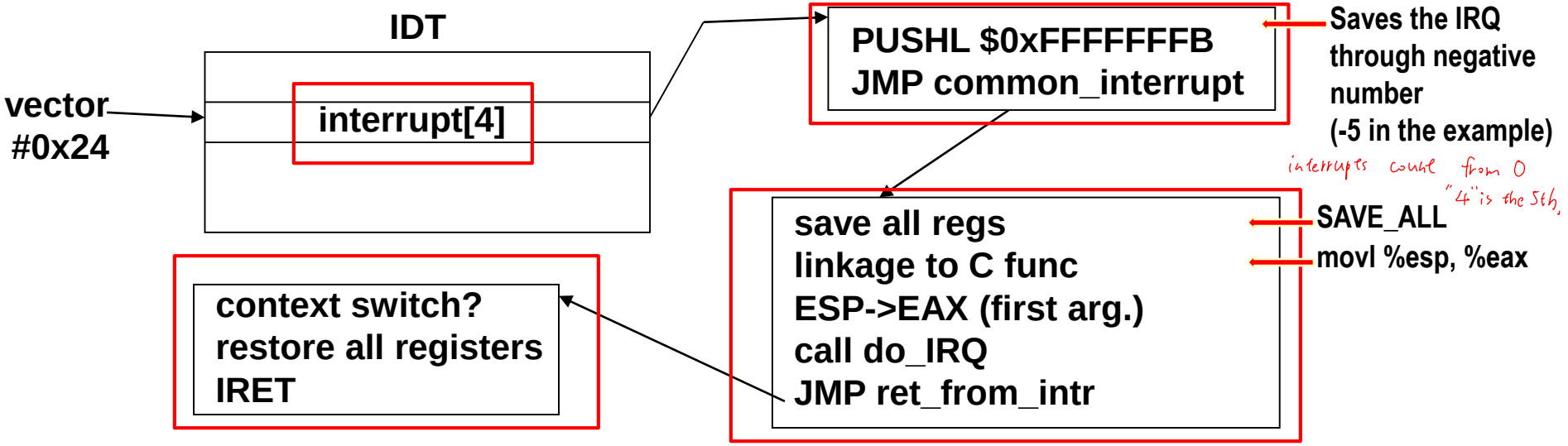
Lecture Topics

- Linux interrupt system
  - data structures
  - handler installation & removal
  - invocation
  - execution
  - soft interrupts (tasklets) in Linux

Aministrivia

- **MP2 Checkpoint 1**
  - **Due by 6:00pm Monday, October 8**
- **MP3 Teams**
  - **Due by end of Friday, October 12**

Interrupt Invocation

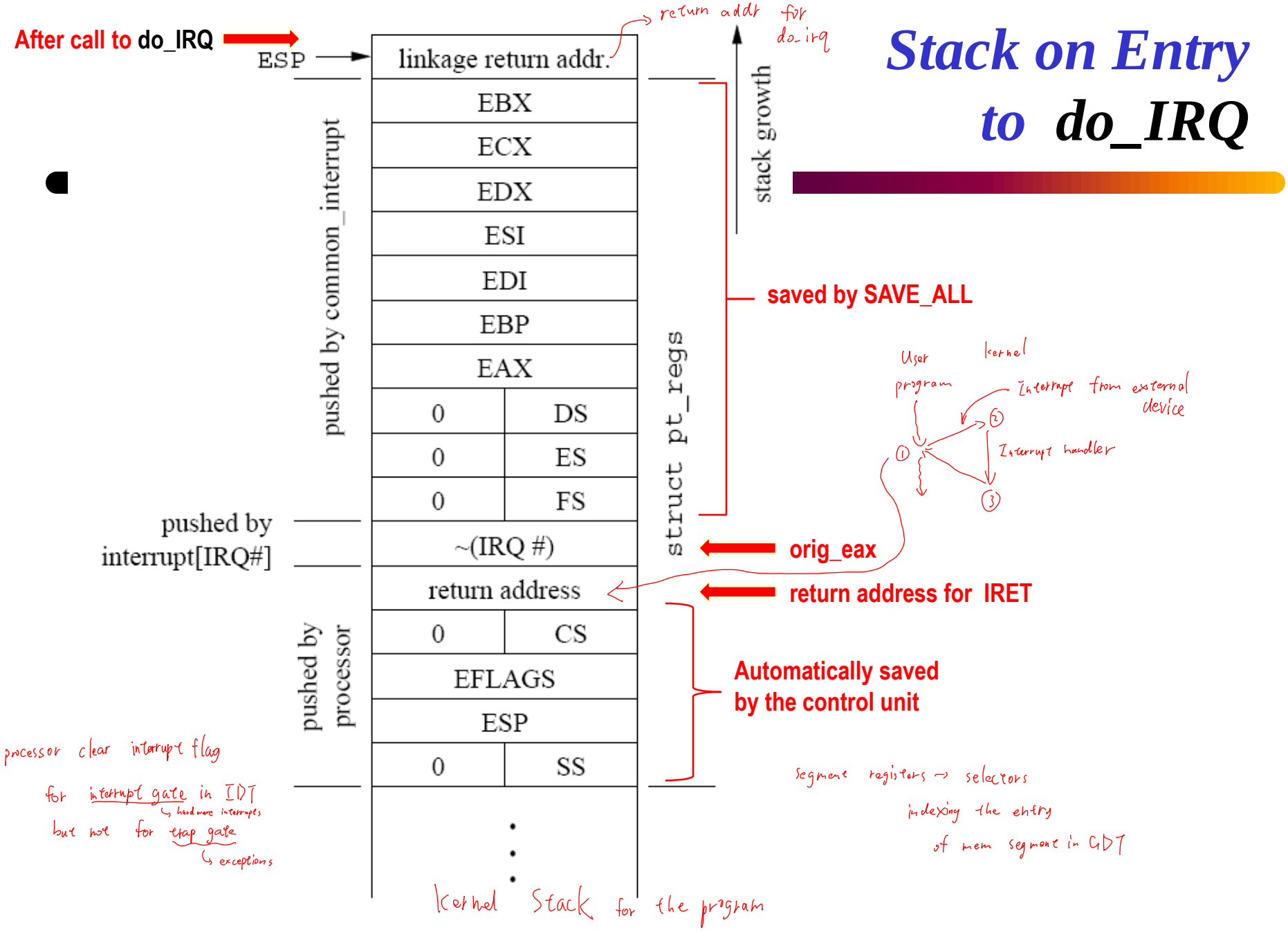


- Why funnel all IRQs into one piece of code(instead of producing one function per IRQ) ?
  - kernel code/size versus speed tradeoff
    - perhaps no big deal for 16
    - keep in mind that you're saving a tiny number of instructions
    - APIC used by most SMPs has 256 vectors

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



Linux do_IRQ

arch/i386/kernel/irq.c

```
fastcall unsigned int
do_IRQ (struct pt_regs* regs)
{
    struct pt_regs* old_regs;
    /* high bit used in ret_from_ code */
    int irq = ~regs->orig_eax;
    struct irq_desc* desc = irq_desc + irq;

    if ((unsigned)irq >= NR_IRQS) {
        printk (KERN_EMERG "%s: cannot handle IRQ %d\n", __FUNCTION__, irq);
        BUG ();
    }

    old_regs = set_irq_regs (regs);
    irq_enter ();

    /* for 8259A interrupts, handle_irq is set to handle_level_irq */
    desc->handle_irq (irq, desc);

    irq_exit ();
    set_irq_regs (old_regs);
    return 1;
}
```

fastcall convention used to pass arguments in registers; reduces the number of memory accesses required for the call

gets the interrupt number from the orig_eax filed in the regs structure

Record registers at time of interrupt

Increments counter of nested interrupt handlers

call interrupt flow handler handle_level_irq for all 8259A interrupts

Exit irq context and process softirqs if needed

Interrupt Invocation (cont.)

- Signature for do_IRQ

```
fastcall unsigned int do_IRQ
    (struct pt_regs* regs);
```

  - **`fastcall`** macro in Linux tells gcc to pass args in EAX, EDX, ECX
  - EAX points to saved registers (**`regs`** argument)
  - saved registers already on stack
  - note that processor clears IF when it takes an interrupt
  - EFLAGS stored by processor include original IF value

Comments on `do_irq`

- Value pushed by irq-specific code re-converted to find irq #
- Start with a sanity check
- **`set_irq_regs`** calls record registers at time of interrupt
  - per-CPU storage

Comments on `do_irq` (cont.)

- **`irq_enter` / `irq_exit`**
  - necessary for proper priority
    - e.g., second hard interrupt occurs
    - after new interrupt handled, should return to first
    - must delay processing of soft interrupts
  - **`irq_exit`** processes soft interrupts when appropriate

Comments on `do_irq` (cont.)

- Central component
  - call interrupt flow handler
  - **`handle_level_irq`** for all 8259A interrupts

- Return value ignored (probably intended to indicate interrupt handled)

Linux handle_level_irq

kernel/irq/chip.c

```
fastcall void
handle_level_irq (unsigned int irq, struct irq_desc* desc)
{
    unsigned int      cpu = smp_processor_id ();
    struct irqaction* action;
    irqreturn_t       action_ret;

    spin_lock (&desc->lock);
    mask_ack_irq (desc, irq);

    if (desc->status & IRQ_INPROGRESS)
        goto out_unlock;
    desc->status &= ~(IRQ_REPLAY | IRQ_WAITING);
    kstat_cpu (cpu).irqs[irq]++;

    /*
     * If its disabled or no action available
     * keep it masked and get out of here
     */
    action = desc->action;
    if (!action || (desc->status & IRQ_DISABLED)) {
        desc->status |= IRQ_PENDING;
        goto out_unlock;
    }

    desc->status |= IRQ_INPROGRESS;
    desc->status &= ~IRQ_PENDING;
    spin_unlock (&desc->lock);

    action_ret = handle_IRQ_event (irq, action);
    if (!noirqdebug)
        note_interrupt (irq, desc, action_ret);

    spin_lock(&desc->lock);
    desc->status &= ~IRQ_INPROGRESS;
    if (!(desc->status & IRQ_DISABLED) && desc->chip->unmask)
        desc->chip->unmask (irq);

out_unlock:
    spin_unlock (&desc->lock);
}
```

Critical section begins

Call PIC's mask_ack function

If interrupt already in progress, do nothing

Atomically decide whether to execute handlers

Set status for execution: in-progress and no longer pending

Critical section ends

Handler execution done via handle_IRQ_event

Critical section begins

When done, remove in-progress flag and unmask on PIC (unless disabled)

Critical section ends

Comments on `handle_level_irq`

- Critical section starts
  - to read descriptor status and action (handler) list
  - IF=0 at this point (set by processor when taking interrupt)
- Immediately call PIC's mask_ack function (via a wrapper function)
- If interrupt already in progress, do nothing
  - Interrupt already re-masked and re-acknowledged on PIC
- Remove software replay and autoprobe flags
- Kernel statistics track # of interrupts seen (see /proc/interrupts)

Comments on `handle_level_irq` (cont.)

- Atomically decide whether to execute handlers immediately
  - do so if handler defined and not disabled by software
  - if not, skip to end
    - mark as pending and end interrupt handling
    - replayed after last nested **`enable_irq`** call

- Set status for execution: in-progress, and no longer pending

Comments on `handle_level_irq` (cont.)

- Handler execution done via **`handle_IRQ_event`**
  - done without descriptor lock
    - allows handlers to use infrastructure
    - e.g., **`enable_irq`**/**`disable_irq`**, **`request_irq`**/**`free_irq`**
  - usually done with IF=1 (changed inside **`handle_IRQ_event`**)
    - done without descriptor lock
    - otherwise this code can deadlock with self

- When done, remove in-progress flag and unmask on PIC (unless disabled)

Linux handle_IRQ_event

kernel/irq/handle.c

```
irqreturn_t
handle_IRQ_event (unsigned int irq, struct irqaction* action)
{
    irqreturn_t  ret;
    irqreturn_t  retval = IRQ_NONE;
    unsigned int status = 0;

    /* call specific to ARM processor (to disambiguate timer ticks) */
    handle_dynamic_tick (action);

    if (!(action->flags & IRQF_DISABLED))
        local_irq_enable_in_hardirq ();

    do {
        ret = action->handler (irq, action->dev_id);
        if (ret == IRQ_HANDLED)
            status |= action->flags;
        retval |= ret;
        action = action->next;
    } while (action);

    if (status & IRQF_SAMPLE_RANDOM)
        add_interrupt_randomness (irq);
    local_irq_disable ();

    return retval;
}
```

call translates basically to STI
(local means "on this CPU")

Walk through list; call each handler
(return value 1 means handled)

Turn interrupts back off (IF=0 / CLI)

Comments on `handle_IRQ_event`

- Set IF=1 unless the first handler asked to be executed with IF=0
  - indicated by IRQF_DISABLED flag
  - call translates basically to STI (local means “on this CPU”)
- Walk through list
  - call each handler
  - return value 1 means handled
- Generate random numbers if desired
- Turn interrupts back off (IF=0 / CLI)

Summary on Descriptor Flags

- IRQ_PENDING—interrupt raised by hardware; waiting to be executed
- IRQ_INPROGRESS—some processor is executing handlers
- IRQ_DISABLED—interrupt disabled in software; postpone execution
- IRQ_REPLAY—software replay of previously postponed execution

Soft Interrupts in Linux

linux/interrupt.h and kernel/softirq.c

- When are soft interrupts executed?
  - after a hard interrupt completes
  - periodically by a daemon in the kernel

- How?
  - seven or eight prioritized types, including high and low tasklet priorities
  - linked list for each tasklet priority
  - run on processor on which interrupt is scheduled
  - each handler atomic with respect to itself (only)

Soft Interrupts in Linux (cont.)

linux/interrupt.h and *kernel/softirq.c*

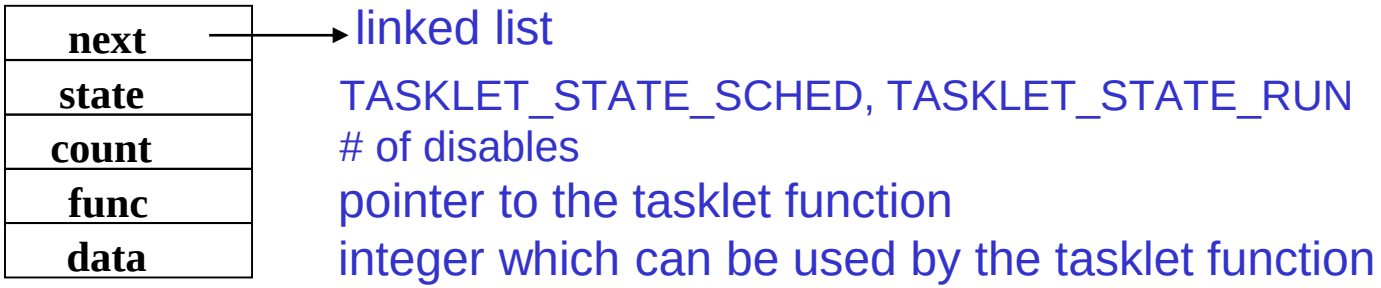
Declaring a handler

name for **`tasklet_struct`**

an unsigned long

```
DECLARE_TASKLET (name, func, data);
```

void (*func) (unsigned long);



Tasklet Scheduling

- The following two calls schedule a tasklet for execution

```
void tasklet_schedule (struct tasklet_struct* t);
void tasklet_hi_schedule (struct tasklet_struct* t);
```

- First form
  - schedules tasklet at low priority
  - on the executing processor
- Second form schedules at high priority
- Enable and disable calls analogous to hard interrupts (including nesting)

Tasklet Execution

- **`do_softirq`** call
  - checks per-processor bit vector of pending priorities (high, low, etc.)
  - executes action for each priority [**`softirq_vec`**]
  - **`tasklet_action`** walks through linked list
    [**`tasklet_hi_action`** walks through high-priority list]
  - repeats up to 10 times or until no softirqs are raised

Tasklet Execution Atomicity

- Two bits in state changed atomically
  - TASKLET_STATE_SCHED – tasklet scheduled for execution
  - TASKLET_STATE_RUN – tasklet executing on some processor

- When scheduling
  - set TASKLET_STATE_SCHED atomically
  - if already set, schedule call does nothing

Tasklet Execution Atomicity (cont.)

When executing, for each tasklet in linked list (at either priority)

A. set TASKLET_STATE_RUN atomically (if already set, stop)

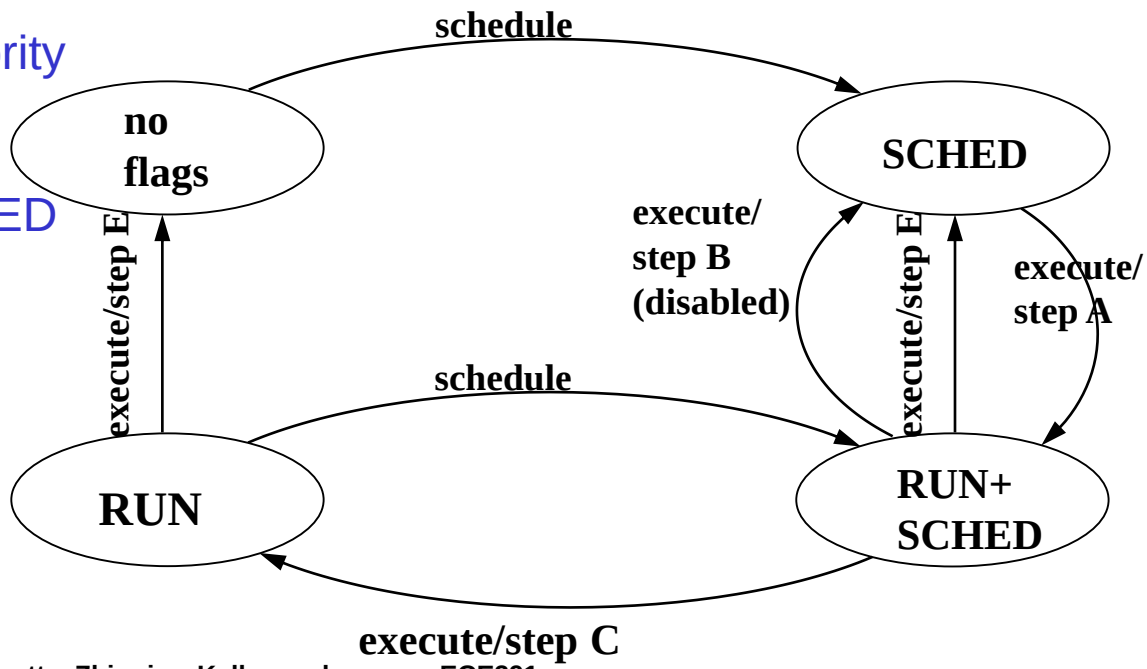
B. check if tasklet is software disabled (count field)

- if so, clear TASKLET_STATE_RUN
- leave the tasklet in the linked list for this priority
- set the pending bit for this priority
- daemon will try again later

C. clear TASKLET_STATE_SCHED

D. execute handler

E. clear TASKLET_STATE_RUN



execution (execute/step D) occurs in lower two states (and execute/step A fails in these states)