CSE 306 Operating Systems
Interrupts and Interrupt Handlers

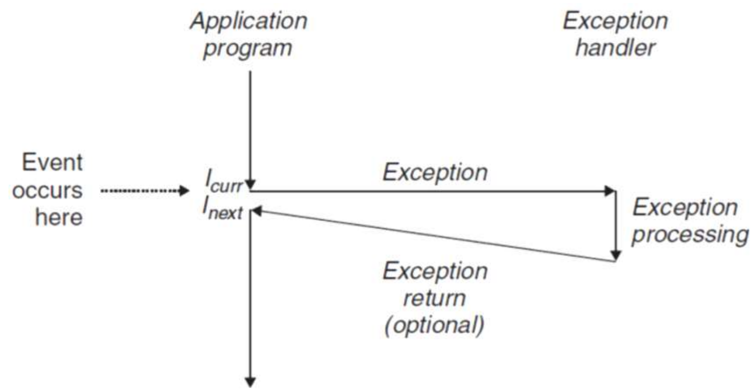
YoungMin Kwon

Interrupts

- Interrupts
  - An event that alters the sequence of instructions executed by a processor
- Two kinds of interrupts
  - Exceptions: synchronous events
    - Interrupts are produced by the CPU control unit
    - Generated after terminating the instruction
  - Interrupts: asynchronous events
    - Interrupts are produced by other hardware devices
    - Generated at arbitrary time

Exceptions

- Processor detected exceptions
  - Faults
    - Can be corrected (e.g. page faults)
    - Return to the instruction that caused the fault
  - Traps
    - Mainly used for debugging
    - Reported immediately following the execution of the instruction
  - Aborts
    - Caused by serious errors
    - Hardware failure



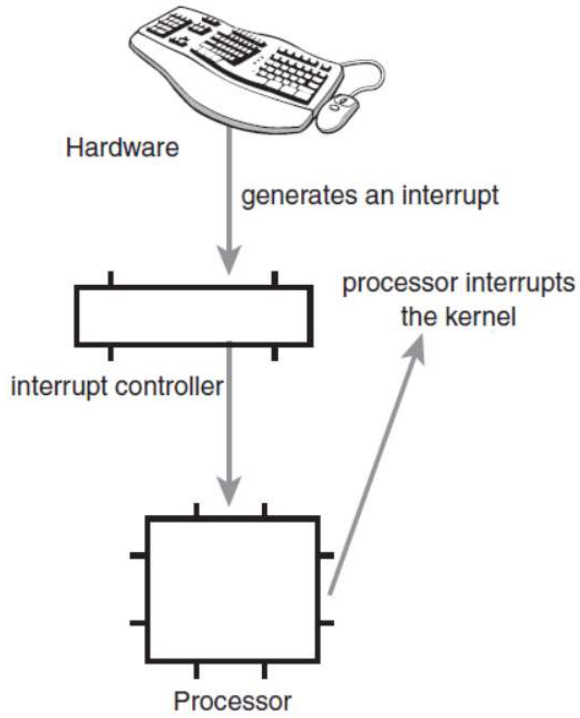
Exceptions

- Programmed exception
  - Software interrupts (handled as traps)
  - Triggered by int or int3 instructions
  - Mainly used to implement system calls or to notify a debugger of a specific event

Interrupts

- An issue in managing hardware
  - Processors can be orders of magnitudes faster than hardware

- Working with hardware
  - Polling: periodically check the status of hardware
  - Interrupt: make hardware signal the processor when attentions are needed



Interrupt Handlers

- Interrupt handler
  - The function that the kernel runs in response to a specific interrupt
  - A normal C function that matches a specific prototype
  - Handlers should run quickly and resume the interrupted code ASAP

Interrupt Handlers

- Two goals of an interrupt handler
  - Execute quickly
  - Perform a large amount of work
  - Top half and bottom half design
- Top half
  - Run immediately on receipt of the interrupt
  - Perform only the time-critical work (e.g. Ack of Int)
- Bottom half
  - Run in the future, at a more convenient time, with all interrupts enabled
  - Do what can be performed later

Top Halves

- Registering an interrupt handler

```
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev);
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

  - irq: interrupt number
  - handler: interrupt handler function
  - flags: options
    - IRQF_SHARED: the irq can be shared by multiple handlers
  - name: string representation of the device
  - dev: identifies the handler, like a cookie

Top Halves

- Freeing an interrupt handler

```
const void *free_irq(unsigned int irq, void *dev);
```

- Example: registering an interrupt handler

```
if (request_irq( irqn,
                 my_interrupt,
                 IRQF_SHARED,
                 "my_device",
                 my_dev)) {
    printk("error: request_irq\n");
    return -EIO;
}
```

Top Halves (handler example)

- Handler example

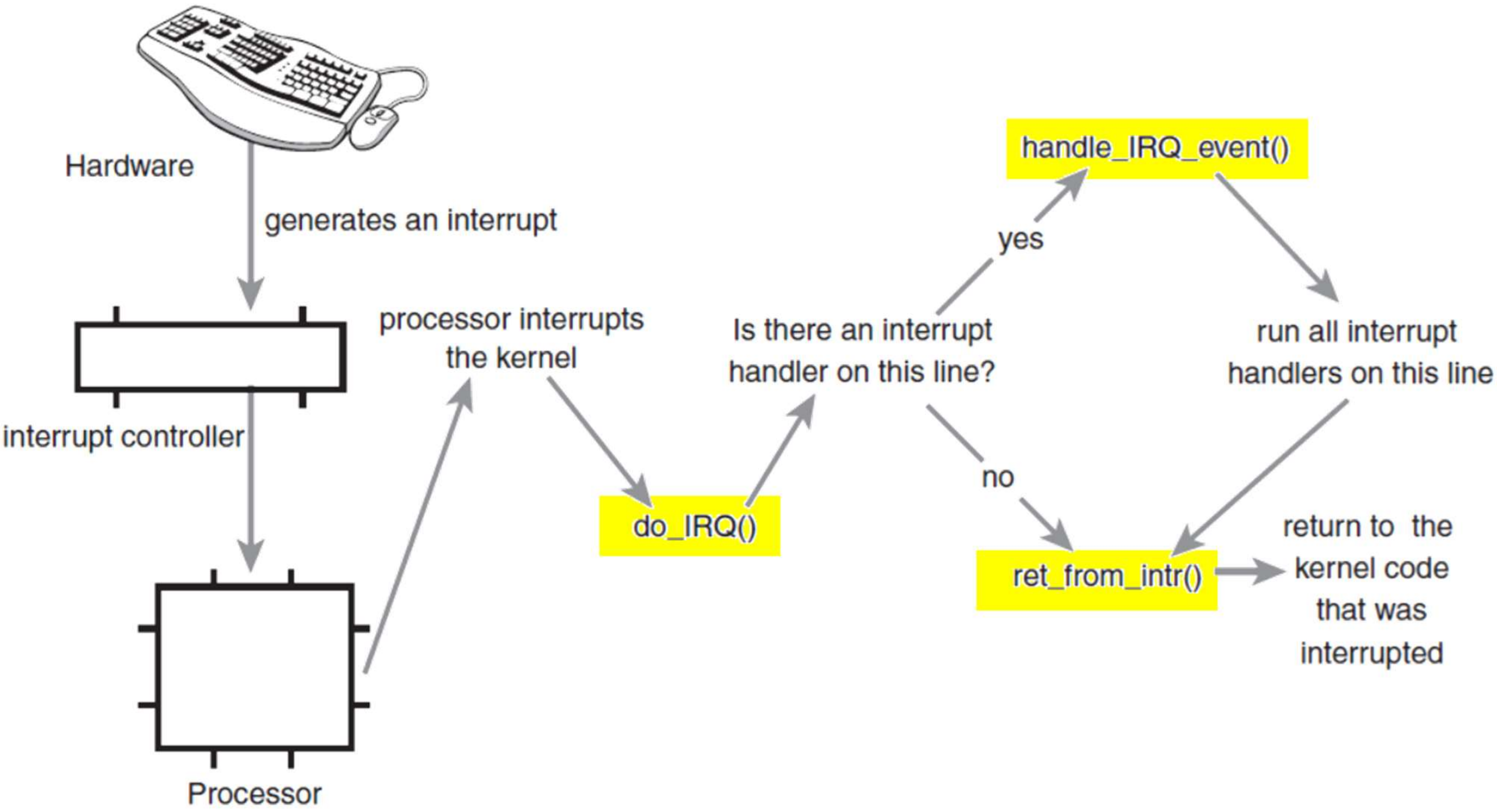
```
static DEFINE_SPINLOCK(rtc_lock);

static irqreturn_t my_interrupt(int irq, void *dev)
{
    spin_lock(&rtc_lock);
    rtc_irq_data += 0x100;
    …
    spin_unlock(&rtc_lock);
    …
    return IRQ_HANDLED; //or IRQ_NONE
}
```

Interrupt Context

- Interrupt context
  - While the kernel is executing an interrupt handler
  - No backing process
  - current macro is not valid
  - Interrupt context cannot sleep

Handling Interrupts



Handling Interrupts

- `do_IRQ()`
  - `arch/x86/kernel/irq.c`
  - Acknowledges the interrupt
  - Disables the interrupt on the line
- `handle_irq_event()`
  - `kernel/irq/handle.c`
  - Run all registered interrupt handlers for the line
    - `IRQF_SHARED`: possibly more than one handlers

Handling Interrupts

- `ret_from_intr()`
  - `arch/x86/entry/entry_64.S,`
  - `kernel/sched/core.c :preempt_schedule_irq(void)`

  - When returning to user space
    - `schedule()` if reschedule is pending (`need_resched` is set)
  - When returning to kernel space
    - `schedule()` only if `preempt_count` is zero

/proc/interrupts

- Statistics related to interrupts

```
$ cat /proc/interrupts
           CPU0
  0:        128   IO-APIC    2-edge      timer
  1:          9   IO-APIC    1-edge      i8042
  4:       3032   IO-APIC    4-edge      ttyS0
  8:          1   IO-APIC    8-edge      rtc0
  9:          0   IO-APIC    9-fasteoi   acpi
 11:        130   IO-APIC   11-fasteoi   enp0s3
 12:        125   IO-APIC   12-edge      i8042
 14:       7016   IO-APIC   14-edge      ata_piix
 15:        112   IO-APIC   15-edge      ata_piix
NMI:          0   Non-maskable interrupts
LOC:      11862   Local timer interrupts
…
```

we will use irq 8

Bottom Halves

- Deferring work
  - Softirqs
    - Statically defined (at compile time) bottom halves
    - Running the same softirqs is blocked on the same processor
    - Other processor can run the same softirq (handler must be reentrant)
      - Within a softirq accessing a global data needs a critical section
    - Cannot sleep

Bottom Halves

- Deferring work (cont'd)
  - Tasklets
    - Dynamically created (at run time) bottom halves
    - Built on top of softirqs
    - Running the same tasklets is blocked on any processor (handler does not need to be reentrant)
    - Cannot sleep
  - Work queues
    - Queuing work to be performed later in a process context
    - Can sleep

Synchronization

- Blocking preemption (preempt_count > 0)
  - Per CPU data is safe (not SMP safe)
  - Interrupt is still enabled
    - Potential synchronization issues with interrupt handlers
- Disabling interrupts
  - Per CPU data is safe (not SMP safe)
  - No concurrency with interrupt handlers
- Sleeping lock (semaphore)
  - Data is safe across multiple CPUs (SMP safe)
  - Should run in a process context

Implementing Softirqs

- softirq_vec: handlers are statically allocated at compile time

```
//in include/linux/interrupt.h
enum {
    HI_SOFTIRQ=0,    TIMER_SOFTIRQ,    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,  BLOCK_SOFTIRQ,    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ, SCHED_SOFTIRQ,    HRTIMER_SOFTIRQ,
    RCU_SOFTIRQ,     NR_SOFTIRQS
};

struct softirq_action {
    void(*action)(struct softirq_action *);
};

//in kernel/softirq.c
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

Implementing Softirqs

- Executing softirqs
  - Usually, an interrupt handler marks its softirq before returning
  - At a suitable time, the softirq runs
- Pending softirqs are checked and executed
  - In the return from hardware interrupt code path
    - Runs in the interrupt context
    - May also be handled in ksoftirqd about 2 msec later
  - In the ksoftirqd kernel thread
    - Runs in a process context
  - do_softirq explicitly checks and executes pending softirqs

Implementing Softirq

- do_softirq: invokes the handlers

```
void __do_softirq() {
    u32 pending;
…
    pending = local_softirq_pending();
    if (pending) {
        struct softirq_action *h;
        set_softirq_pending(0); //reset the pending bitmask

        h = softirq_vec;
        do {
            if (pending & 1)
                h->action(h); //invoking the handler
            h++;
            pending >>= 1;
        } while (pending);
    }
    //hand over to ksoftirqd after 2+ msec
}
```

Using Softirqs

- Assigning an index

Softirq Types

Tasklet	Priority	Softirq Description
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
BLOCK_SOFTIRQ	4	Block devices
TASKLET_SOFTIRQ	5	Normal priority tasklets
SCHED_SOFTIRQ	6	Scheduler
HRTIMER_SOFTIRQ	7	High-resolution timers
RCU_SOFTIRQ	8	RCU locking

Using Softirqs

- Registering handler

```
//e.g. in net/core/dev.c
static int net_dev_init(void) {
...
    open_softirq(NET_TX_SOFTIRQ, net_tx_action);
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);
...
}
```

- Raising softirq

```
raise_softirq(NET_TX_SOFTIRQ); //mark it as pending
```

Tasklets

- Tasklets
  - Built on top of softirqs
    - HI_SOFTIRQ and TASKLET_SOFTIRQ
  - Similar to softirqs, but with simpler interface and relaxed locking rules
    - Tasklets do not need to be reentrant
  - Have nothing to do with tasks

Implementing Tasklets

- Tasklet structure

```
struct tasklet_struct {
    struct tasklet_struct *next; /* next tasklet in the list */
    unsigned long state;         /* state of the tasklet */
    atomic_t count;              /* reference counter */
    void (*func)(unsigned long); /* tasklet handler function */
    unsigned long data;          /* argument to the tasklet function */
};
```

```
next:  scheduled tasks are stored in tasket_vec and tasklet_hi_vec lists
func:  tasklet handler
data:  argument to func
state: one of 0, TASKLET_STATE_SCHED, TASKLET_STATE_RUN
count: nonzero: disabled,
          zero: enabled
```

Implementing Tasklets

- Scheduling Tasklets

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

  - Simply return if the state is TASKLET_STATE_SCHED (already scheduled case)
  - Call __tasklet_schedule()
    - See __tasklet_schedule_common in kernel/softirq.c
    - Save the state of the interrupt system and disable local interrupts (nothing on this processor will interfere)
    - Add tasklet to tasklet_vec or tasklet_hi_vec
    - Raise TASKLET_SOFTIRQ or HI_SOFTIRQ
    - Restore the interrupts to their previous state

Using Tasklets

```
struct tasklet_struct {
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

- Declaring/initializing a tasklet

```
#define DECLARE_TASKLET(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }

#define DECLARE_TASKLET_DISABLED(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }

void tasklet_init( struct tasklet_struct *t,
                   void (*func)(unsigned long),
                   unsigned long data );
```

- Writing a tasklet

```
void tasklet_handler(unsigned long data) {
    ...
}
```

Using Tasklets

- Scheduling a tasklet

```
/* mark my_tasklet as pending */
tasklet_schedule(&my_tasklet);

/* disable tasklet */
tasklet_disable(&my_tasklet);

/* enable tasklet */
tasklet_enable(&my_tasklet);
```

Using Work Queues

- Work Queues
  - Defer work into a kernel thread
  - Runs in a process context
    - Schedulable (can sleep)
- Softirqs/tasklets vs work queues
  - Deferred work needs to sleep → use work queues
  - Deferred work need not sleep → use softirqs/tasklets

Implementing Work Queues

- Worker threads
  - Create kernel threads to handle work queued
  - Worker threads are called event/n, where n is the processor number

Implementing Work Queues

- Data structures

```
struct workqueue_struct {          /* one per type of worker thread */
    struct cpu_workqueue_struct[NR_CPUS];
    struct list_head list;         /* list of all workqueues */
    …
};

struct cpu_workqueue_struct {      /* one per cpu */
    spinlock_t lock;               /* lock protecting this structure */
    struct list_head worklist;     /* list of work_struct */
    wait_queue_head_t more_work;   /* when blocked, task will be moved to */
    struct work_struct *current_struct;
    struct workqueue_struct *wq;   /* associated workqueue_struct */
    task_t *thread;                /* associated thread */
};

struct work_struct {               /* one per deferrable function */
    atomic_long_t data;
    struct list_head entry;        /* linked list */
    work_func_t func;
};
```

Implementing Work Queues

- worker_thread() function

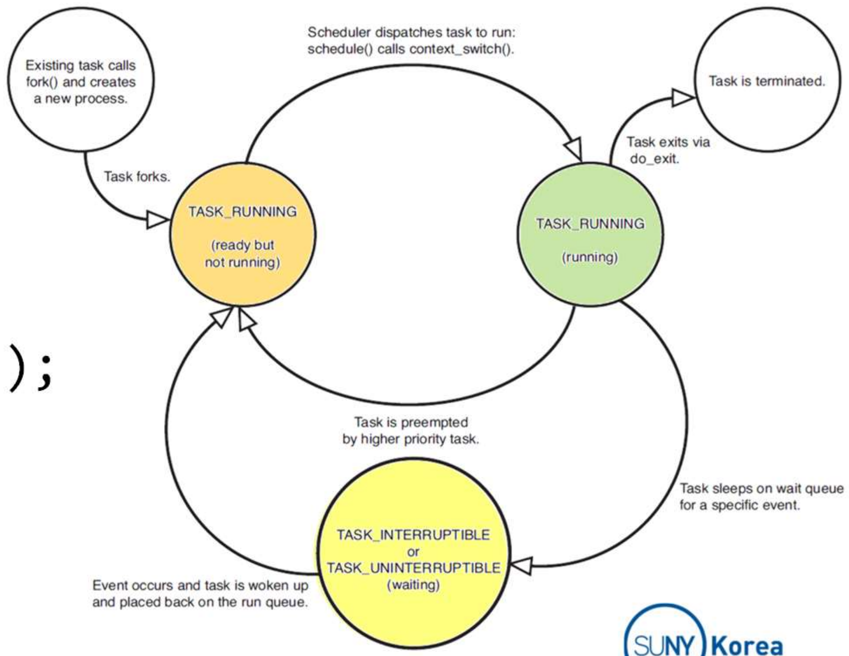
  __cancel_work_timer() in kernel/workqueue.c

```
for (;;) {
    //add current to wait and add wait to more_work
    prepare_to_wait(&cwq->more_work, &wait, TASK_INTERRUPTIBLE);

    if (list_empty(&cwq->worklist))
        schedule(); //context switch

    //remove wait from more_work and
    //add current to run queue
    finish_wait(&cwq->more_work, &wait);

    run_workqueue(cwq);
}
```



Implementing Work Queues

- run_workqueue() function

```
while (!list_empty(&cwq->worklist)) {
    struct work_struct *work;
    work_func_t f;
    void *data;

    work = list_entry(cwq->worklist.next,
                      struct work_struct, entry);
    f = work->func;
    list_del_init(cwq->worklist.next);
    work_clear_pending(work);

    f(work);
}
```

Using Work Queues

- Creating work

```
#include <linux/workqueue.h>

//to create a structure
DECLARE_WORK(name, void(*func)(void *));

//to create work via a pointer
INIT_WORK(struct work_struct *work, void(*func)(void *));
```

- Work queue handler

```
void work_handler(struct work_struct *work)
```

  - Handler runs in a process context

Using Work Queues

- Scheduling work

```
//to schedule immediately
schedule_work(&work);

//to schedule after delay
schedule_delayed_work(&work, delay);
```

- Flushing work

```
//to wait until all entries in the queue are executed
void flush_scheduled_work(void);

//to cancel the delayed work
int cancel_delayed_work(struct work_struct *work);
```

Using Work Queues

- Creating a new work queue

```
struct workqueue_struct *create_workqueue(const char *name);

//example
struct workqueue_struct *keventd_wq;
keventd_wq = create_workqueue("events");
```

- Scheduling on the created work queue

```
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);
```

SUNY Korea

Assignment 3

- Using this assignment, we will practice top-half and bottom-half interrupt handlers
  - Due date 4/11/2024
  - Create a work_struct rtc_work with a handler

```
static int wq_count;
static void work_queue_rtc_handler(struct work_struct *dummy);
//TODO: increase wq_count in a critical section
//      use a semaphore for the critical section
//TODO: printk("rtc: work_queue_rtc_handler: %d\n", wq_count);
```

Assignment 3

- Create a tasklet_struct rtc_tasklet with a handler

```
static int tl_count;
static void tasklet_rtc_handler(unsigned long dummy);
//TODO: increase tl_count in a critical section
//      use a spinlock for the critical section
//TODO: printk("rtc: tasket_rtc_handler: %d\n", tl_count);
//TODO: schedule rtc_work
```

- Write an irq handler for RTC

```
static irqreturn_t irq_rtc_handler(int irq, void *dev);
//TODO: increase rtc_count in a critical section
//      use a spinlock for the critical section
//TODO: printk("rtc: irq_rtc_handler: %d\n", rtc_count);
//TODO: schedule rtc_tasklet
//TODO: return IRQ_HANDLED
```

Assignment 3

- Correct errors in threadfn

```
#define DELAY {\
    long i;\
    for(i = 0; i < 10L*1000*1000/*10 million*/; i++)\
        /*do nothing*/;\
}
static int thr_done = 0;
static int threadfn(void *unused) {
    thr_done = 0;
    while (!thr_done) {
        rtc_count++; //use spin_lock_irqsave for the critical section
        DELAY;
        rtc_count--;

        tl_count++;  //use spin_lock_irqsave for the critical section
        DELAY;
        tl_count--;

        wq_count++;  //use semaphore for the critical section
        DELAY;
        wq_count--;

        schedule();
    }
    return 0;
}
```

It disables interrupt: otherwise, deadlock from the interrupt handler

Assignment 3

- Write a system call that registers (if on is true) or unregisters (if on is false) irq_rtc_handler

```
SYSCALL_DEFINE1(handle_rtc, int, on)
    if (on) {
        //register irq_rtc_handler
        //- irq: 8,
        //- flag: IRQF_SHARED,
        //- name: "my_rtc",
        //- dev: (void*) 1
        if (thr_done)
            //run threadfn
    }
    else {
        //unregister irq_rtc_handler
        if (!thr_done)
            //stop threadfn by setting thr_done = 1
    }
    return 0;
}
```

Assignment 3

- To register rtc interrupt handler, change the following param in drivers/rtc/rtc-cmos.c

```
retval = request_irq(rtc_irq, rtc_cmos_int_handler,
            0, dev_name(&cmos_rtc.rtc->dev),
            cmos_rtc.rtc);
```

to

```
retval = request_irq(rtc_irq, rtc_cmos_int_handler,
            IRQF_SHARED, dev_name(&cmos_rtc.rtc->dev),
            cmos_rtc.rtc);
```

Assignment 3

- User space program
  - Write rtc_on.c that registers the rtc handler

```
> rtc_on
> cat /proc/interrupts
               CPU0
...
  4:        15372    IO-APIC    4-edge        ttyS0
  8:          149    IO-APIC    8-edge        rtc0, my_rtc
```

  - Write rtc_off.c that unregisters the rtc handler

```
> rtc_off
> cat /proc/interrupts
               CPU0
...
  4:        15372    IO-APIC    4-edge        ttyS0
  8:          149    IO-APIC    8-edge        rtc0
```

Assignment 3

- To generate rtc interrupts

```
> sudo chmod ugo+w /sys/class/rtc/rtc0/wakealarm
> echo +1 > /sys/class/rtc/rtc0/wakealarm
> dmesg
…
[   31.701752] rtc: irq_rtc_handler: 1
[   31.701754] rtc: tasket_rtc_handler: 1
[   31.701757] rtc: work_queue_rtc_handler: 1
[   31.717374] rtc: irq_rtc_handler: 2
[   31.717375] rtc: tasket_rtc_handler: 2
[   31.717378] rtc: work_queue_rtc_handler: 2
[   31.733025] rtc: irq_rtc_handler: 3
[   31.733026] rtc: tasket_rtc_handler: 3
[   31.733084] rtc: work_queue_rtc_handler: 3
…
```