Interrupts & Deferred Work

Have a look

- Before getting into **Interrupts & Deferred Work**, please take a look at
    - LK_Bird's Eye View session,

      S8 Synch,Lock,Interrupt, Exception,Softirq,Tasklet,WorkQueue.

Interrupts

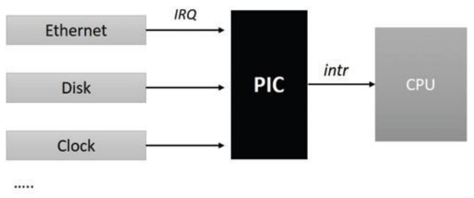
- Each hardware device controller capable of issuing interrupt requests usually has a single output line designated as the **Interrupt Request (IRQ)** line.

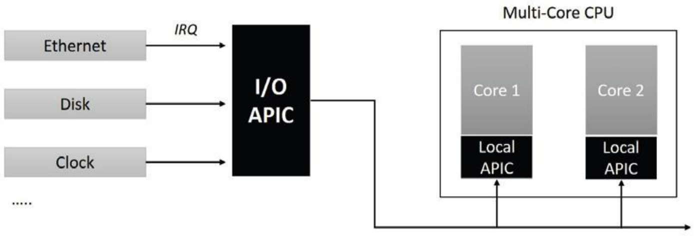
- All existing IRQ lines are connected to the input pins of a HW circuit called the **Programmable Interrupt Controller (PIC)**.

- At multi-core CPUs, there would be like **I/O APIC** at that time acts as a **router** with respect to the local APIC's.

- In response to the occurrence of an interrupt event, CPUs are preempted the current instruction sequence or thread of execution, and execute a special function called **interrupt service routine (ISR)**.

- To locate the appropriate ISR that corresponds to an interrupt event, **Interrupt vector tables** are used.





Interrupts

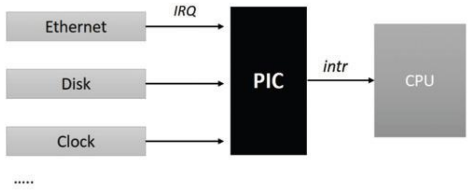
- i.e x86 interrupts and exceptions classificatons
  - Interrupts :
    Maskable interrupts: Most of the interrupts.
    Nonmaskable interrupts: i.e Reset interrupt.

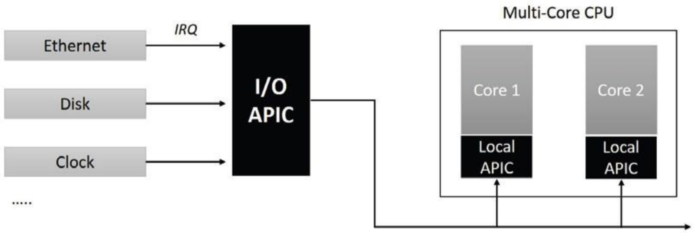
  - Exceptions :
    Processor-detected exceptions :
    Faults: i.e Page Fault Exception.

    Traps: Used in debugging, i.e a **breakpoint** has been reached within a program.

    Aborts: Report severe errors, i.e **hardware failures**.

    Programmed exceptions : User triggers an intended exception and this is used in i.e system calls assembly instruction, ***int3***.





Interrupts

- Interrupt controller operations
  - ***irq_chip*** structure declares a set of function pointers to account for all peculiarities of IRQ chips found across various hardware platforms.

  - A particular instance of the structure defined by **board-specific** code usually supports only a subset of possible operations.
    i.e. x86 multicore I/O APIC.

/arch/x86/kernel/apic/io_apic.c

```
static struct irq_chip ioapic_chip __read_mostly = {
    .name               = "IO-APIC",
    .irq_startup        = startup_ioapic_irq,
    .irq_mask           = mask_ioapic_irq,
    .irq_unmask         = unmask_ioapic_irq,
    .irq_ack            = irq_chip_ack_parent,
    .irq_eoi            = ioapic_ack_level,
    .irq_set_affinity   = ioapic_set_affinity,
    .irq_retrigger      = irq_chip_retrigger_hierarchy,
    .irq_get_irqchip_state  = ioapic_irq_get_chip_state,
    .flags              = IRQCHIP_SKIP_SET_WAKE |
                          IRQCHIP_AFFINITY_PRE_STARTUP,
};
```

/include/linux/irq.h

```
struct irq_chip {
    struct device   *parent_device;
    const char  *name;
    unsigned int    (*irq_startup)(struct irq_data *data);
    void        (*irq_shutdown)(struct irq_data *data);
    void        (*irq_enable)(struct irq_data *data);
    void        (*irq_disable)(struct irq_data *data);

    void        (*irq_ack)(struct irq_data *data);
    void        (*irq_mask)(struct irq_data *data);
    void        (*irq_mask_ack)(struct irq_data *data);
    void        (*irq_unmask)(struct irq_data *data);
    void        (*irq_eoi)(struct irq_data *data);

    int     (*irq_set_affinity)(struct irq_data *data, const struct
cpumask *dest, bool force);
    int     (*irq_retrigger)(struct irq_data *data);
    int     (*irq_set_type)(struct irq_data *data, unsigned int
flow_type);
    int     (*irq_set_wake)(struct irq_data *data, unsigned int on);

    void        (*irq_bus_lock)(struct irq_data *data);
    void        (*irq_bus_sync_unlock)(struct irq_data *data);

#ifdef CONFIG_DEPRECATED_IRQ_CPU_ONOFFLINE
    void        (*irq_cpu_online)(struct irq_data *data);
    void        (*irq_cpu_offline)(struct irq_data *data);
#endif
    void        (*irq_suspend)(struct irq_data *data);
    void        (*irq_resume)(struct irq_data *data);
    void        (*irq_pm_shutdown)(struct irq_data *data);

    ...
};
```

Interrupts

- IRQ descriptor table (IDT)
  - Interrupt controllers identify each IRQ source with a unique hardware **IRQ** number.
  - The kernel's generic interrupt-management layer maps each hardware IRQ to a unique identifier called **Linux IRQ**; these numbers abstract hardware IRQs, thereby ensuring **portability** of kernel code.
  - All of the peripheral device drivers are programmed to use the **Linux IRQ** number to bind or register their interrupt handlers.
  - Linux IRQ represented by ***irq_desc*** structure.
    The list of IRQ descriptors is maintained by ***irq_desc*** array,
    NR_IRQS is **arch-dependent**.
  - ***irq_set_handler()*** : used to set the ***handle_iq.***

    irq_set_chip_and_handler : used more frequent in drivers to set both chip ***(desc->irq_data.chip = chip;)*** and handler for any irq needed.

/include/linux/irqdesc.h

```
struct irq_desc {
    struct irq_common_data  irq_common_data;
    struct irq_data         irq_data;
    unsigned int __percpu   *kstat_irqs;
    irq_flow_handler_t  handle_irq;
    struct irqaction    *action;    /* IRQ action list */
    ...
};
```

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
    [0 ... NR_IRQS-1] = {
        .handle_irq = handle_bad_irq,
        .depth      = 1,
        .lock       = __RAW_SPIN_LOCK_UNLOCKED(irq_desc->lock),
    }
};
```

/include/linux/irq.h

```
static inline void irq_set_chip_and_handler(unsigned int irq,
struct irq_chip *chip,
                        irq_flow_handler_t handle)
{
    irq_set_chip_and_handler_name(irq, chip, handle, NULL);
}

...

static inline void
irq_set_handler(unsigned int irq, irq_flow_handler_t handle)
{
    __irq_set_handler(irq, handle, 0, NULL);
}
```

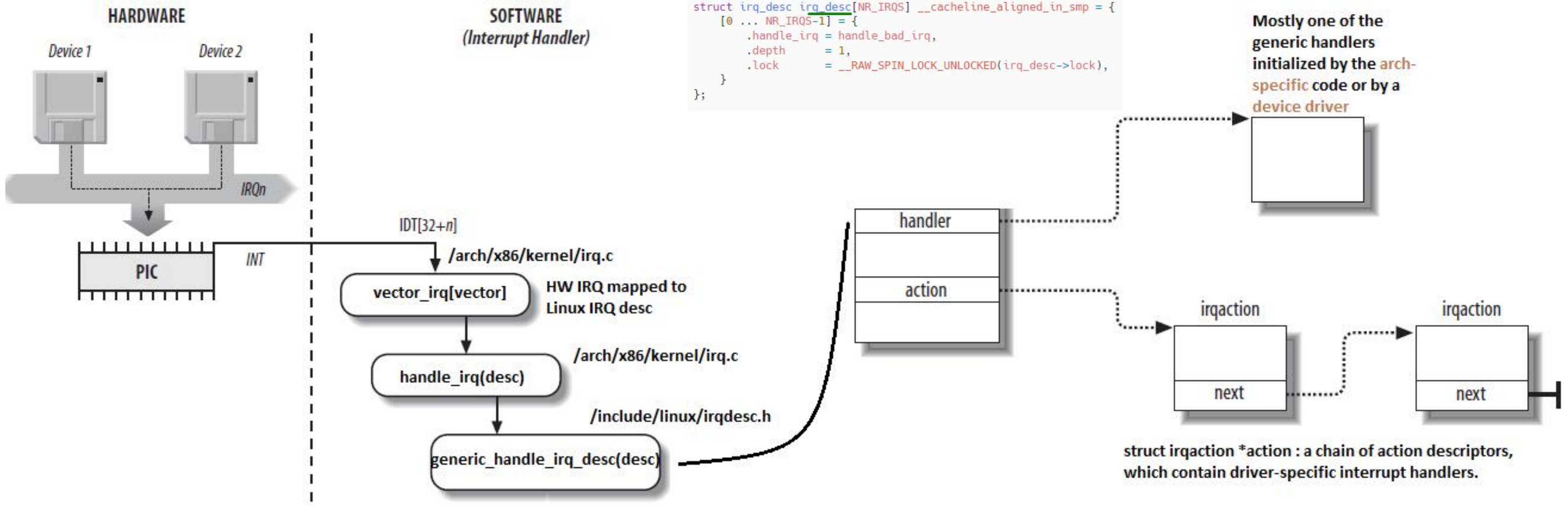
Interrupts

- IRQ descriptor table (IDT), *Cont’d*
  - Interrupt Handling
    Kernel core codes provide generic handlers and used by drivers or arch-specific code exist at ***/kernel/irq/chip.c***

    Generally at initialization , interrupt or exception vector early handlers is setup, i.e. **x86 ,** ***/arch/x86/kernel/idt.c*** : setup interrupts, exceptions, traps,... handlers.

```
/*
 * Built-in IRQ handlers for various IRQ types,
 * callable via desc->handle_irq()
 */
extern void handle_level_irq(struct irq_desc *desc);
extern void handle_fasteoi_irq(struct irq_desc *desc);
extern void handle_edge_irq(struct irq_desc *desc);
extern void handle_edge_eoi_irq(struct irq_desc *desc);
extern void handle_simple_irq(struct irq_desc *desc);
extern void handle_untracked_irq(struct irq_desc *desc);
extern void handle_percpu_irq(struct irq_desc *desc);
extern void handle_percpu_devid_irq(struct irq_desc *desc);
extern void handle_bad_irq(struct irq_desc *desc);
extern void handle_nested_irq(unsigned int irq);
...
```

```
struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
    [0 ... NR_IRQS-1] = {
        .handle_irq = handle_bad_irq,
        .depth      = 1,
        .lock       = __RAW_SPIN_LOCK_UNLOCKED(irq_desc->lock),
    }
};
```



Interrupts

- Register/Free an interrupt handler
  - ***request_irq()*** : instantiates an ***irqaction*** object with values passed as parameters and binds it to the ***irq_desc*** specified as the first (irq) parameter.

    This call allocates interrupt resources and enables the interrupt line and IRQ handling.

    hanlder is a function pointer of type ***irq_handler_t*** ,which takes the address of a driver-specific interrupt handler routine.

  - ***free_irq()*** : free an interrupt allocated with ***request_irq()***.

/include/linux/interrupt.

```
static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler,
unsigned long flags,
        const char *name, void *dev)
{
    ...
}
```

```
struct irqaction {
    irq_handler_t        handler;
    void               *dev_id;
    void __percpu        *percpu_dev_id;
    struct irqaction     *next;
    irq_handler_t        thread_fn;
    struct task_struct   *thread;
    struct irqaction     *secondary;
    unsigned int         irq;
    unsigned int         flags;
    unsigned long        thread_flags;
    unsigned long        thread_mask;
    const char         *name;
    struct proc_dir_entry   *dir;
} ____cacheline_internodealigned_in_smp;
```

/kernel/irq/manage.c

```
const void *free_irq(unsigned int irq, void *dev_id)
{
    ...
}
```

Interrupts

- Threaded interrupt handlers
  - Handlers registered through ***request_irq()*** are executed by the interrupt-handling path of the kernel running in a **interrupt context**.

  - These handler routines should be short and atomic (non blocking), to ensure **responsiveness** of the system.

  - However, not all hardware interrupt handlers can be short and atomic: there are necessary routines whose responses involve **complex variable-time** operations.

  - For such cases, the kernel intodouces **split-handler** design for the interrupt handler to <u>**top half**</u> and **bottom half**.

Interrupts

- Threaded interrupt handlers*, Cont'd*
  - **Top half** routines are invoked in hard interrupt context, and these functions are programmed to execute interrupt critical operations, such as physical I/O on the hardware registers, and schedule the **bottom half** for deferred execution.
  - **Bottom half** routines are usually programmed to deal with the rest of the interrupt non-critical and deferrable work, such as processing of data generated by the **top half**, interacting with process context, and accessing user address space. ***(Deferred Work)***
  - As an alternative to using bottom-half mechanisms, the kernel supports setting up interrupt handlers that can execute in a **thread context**, called **threaded interrupt handlers**.

    As the biggest advantage is reducing complexity by simplifying or avoiding locking between the "hard/top" and "soft/deferred" parts of interrupt handling.

- Drivers can set up threaded interrupt handlers through an alternate interface routine called. ***request_threaded_irq()***

/kernel/irq/manage.c

```c
/**
 *  request_threaded_irq - allocate an interrupt line
 *  @irq: Interrupt line to allocate
 *  @handler: Function to be called when the IRQ occurs.
 *        Primary handler for threaded interrupts.
 *        If handler is NULL and thread_fn != NULL
 *        the default primary handler is installed.
 *  @thread_fn: Function called from the irq handler thread
 *          If NULL, no irq thread is created
 *  ...
 */
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
            irq_handler_t thread_fn, unsigned long irqflags,
            const char *devname, void *dev_id)
{
    ...
}
```

```c
struct irqaction {
    irq_handler_t         handler;
    void                *dev_id;
    void __percpu         *percpu_dev_id;
    struct irqaction      *next;
    irq_handler_t         thread_fn;
    struct task_struct    *thread;
    struct irqaction      *secondary;
    unsigned int          irq;
    unsigned int          flags;
    unsigned long         thread_flags;
    unsigned long         thread_mask;
    const char          *name;
    struct proc_dir_entry   *dir;
} ____cacheline_internodealigned_in_smp;
```

Interrupts

- Control interfaces
  - ***enable_irq()*** : Enable IRQ.
  - ***disable_irq()*** : Disable IRQ.
  - ***local_irq_enable()***: Enables interrupts for the local processor.
  - ***local_irq_disable()*** : To disable interrupts on the local processor.
  - ***local_irq_save()*** : Disables interrupts on the local CPU by saving current interrupt state in flags.
  - ***local_irq_restore()*** : Enables interrupts on the local CPU by restoring interrupts to a previous state.

/kernel/irq/manage.c

```
void enable_irq(unsigned int irq)
{
    ...
}

void disable_irq(unsigned int irq)
{
    ...
}
```

/include/linux/irqflags.h

```
#define local_irq_enable()                          \
    do {                                            \
        trace_hardirqs_on();                        \
        raw_local_irq_enable();                     \
    } while (0)

#define local_irq_disable()                         \
    do {                                            \
        bool was_disabled = raw_irqs_disabled();\
        raw_local_irq_disable();                    \
        if (!was_disabled)                          \
            trace_hardirqs_off();                   \
    } while (0)

#define local_irq_save(flags)                       \
    do {                                            \
        raw_local_irq_save(flags);                  \
        if (!raw_irqs_disabled_flags(flags))        \
            trace_hardirqs_off();                   \
    } while (0)

#define local_irq_restore(flags)                    \
    do {                                            \
        if (!raw_irqs_disabled_flags(flags))        \
            trace_hardirqs_on();                    \
        raw_local_irq_restore(flags);               \
    } while (0)
```

Interrupts

- IRQ Stacks
  - Generally, interrupt handlers shared the kernel stack of the running process that was interrupted.
  - However, the kernel stack might not always be enough for kernel work and IRQ processing routines.
  - To address this, the kernel build (for a few **architectures**) is configured by default to set up an additional **per-CPU hard IRQ stack** for use by interrupt handlers.

/arch/x86/include/asm/processor.h

```
/* Per CPU interrupt stacks */
struct irq_stack {
    char        stack[IRQ_STACK_SIZE];
} __aligned(IRQ_STACK_SIZE);
...

DECLARE_PER_CPU(struct irq_stack *, hardirq_stack_ptr);
```

Deferred Work

- Deferred Wok Frameworks
  - Softirqs
  - Tasklets
  - Work Queues

- Softirqs
  - Deferred routines managed by this framework are executed at a high priority but with hard interrupt lines enabled.

  - Softirqs can preempt all other tasks except **hard interrupt handlers**.

  - Each softirq is represented through an instance of ***softirq_action.***

  - Usage of softirqs is restricted to static kernel code and this mechanism is not available for **dynamic kernel modules**, currently kernel has only **10 softirqs**.

/include/linux/interrupt.h

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

```
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
};
```

Deferred Work

- Softirqs, *Cont'd*
  - ***softirq_vec[]*** : Define the different softirqs.
  - ***open_softirq()*** : initialize the softirq instance with the corresponding **bottom-half** routine.
  - ***raise_softirq()*** : trigger the execution of softirq handlers, by waking up the *ksoftirqd* thread.
  - i.e. ***/kernel/time/timer.c***

```c
void __init init_timers(void)
{
    init_timer_cpus();
    posix_cputimers_init_work();
    open_softirq(TIMER_SOFTIRQ, run_timer_softirq);
}

static void run_local_timers(void)
{
    ...
    raise_softirq(TIMER_SOFTIRQ);
}
```

/kernel/softirq.c

```c
static struct softirq_action softirq_vec[NR_SOFTIRQS]
__cacheline_aligned_in_smp;

const char * const softirq_to_name[NR_SOFTIRQS] = {
    "HI", "TIMER", "NET_TX", "NET_RX", "BLOCK", "IRQ_POLL",
    "TASKLET", "SCHED", "HRTIMER", "RCU"
};

void open_softirq(int nr, void (*action)(struct softirq_action *))
{
    softirq_vec[nr].action = action;
}

void raise_softirq(unsigned int nr)
{
    unsigned long flags;

    local_irq_save(flags);
    raise_softirq_irqoff(nr);
    local_irq_restore(flags);
}
```

/kernel/softirq.c

```c
static void wakeup_softirqd(void)
{
    /* Interrupts are disabled: no need to stop preemption */
    struct task_struct *tsk = __this_cpu_read(ksoftirqd);

    if (tsk)
        wake_up_process(tsk);
}
```

Deferred Work

- Tasklets
  - Considered as a wrapper around the softirq framework; in fact, tasklet handlers are executed by **softirqs**.
  - However , 2 differences
    1) tasklets are not reentrant, which guarantees that the same tasklet handler can never run concurrently.
    This helps minimize overall **latencies**.

    2) unlike softirqs (which are restricted), any kernel code can use tasklets, and this includes **dynamically linked** services.
  - ***DECLARE_TASKLET()*** : Instantiate a new tasklet statically.
  - The kernel maintains two per-CPU tasklet lists for queuing scheduled tasklets,
    tasklet_vec : Normal list, all tasklets are run by ***TASKLET_SOFTIRQ*** softirq.

    tasklet_hi_vec : high-priority tasklet list are run by ***HI_SOFTIRQ*** softirq.

/include/linux/interrupt.h

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    bool use_callback;
    union {
        void (*func)(unsigned long data);
        void (*callback)(struct tasklet_struct *t);
    };
    unsigned long data;
};

#define DECLARE_TASKLET(name, _callback)        \
struct tasklet_struct name = {                  \
    .count = ATOMIC_INIT(0),                    \
    .callback = _callback,                      \
    .use_callback = true,                       \
}
```

/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 tasklet_head {
    struct tasklet_struct *head;
    struct tasklet_struct **tail;
};

static DEFINE_PER_CPU(struct tasklet_head, tasklet_vec);
static DEFINE_PER_CPU(struct tasklet_head, tasklet_hi_vec);
```

Deferred Work

- Tasklets, *Cont'd*
  - ***tasklet_disable()*** : disables the specified tasklet by incrementing its disable count.
    The tasklet may still be scheduled, but it is not executed until it has been enabled again.
  - ***tasklet_enable()*** : attempts to enable a tasklet that had been previously disabled by decrementing its disable count.
  - ***tasklet_kill()*** : kill the given tasklet, to ensure that the it cannot be scheduled to run again.

/include/linux/interrupt.h

```
static inline void tasklet_disable(struct tasklet_struct *t)
{
    ...
}

static inline void tasklet_enable(struct tasklet_struct *t)
{
    ...
}
```

/kernel/softirq.c

```
void tasklet_kill(struct tasklet_struct *t)
{
    ...
}
```

Deferred Work

- Workqueues
  - Kernel workqueue is a list of work items, each containing a function pointer that takes the address of a routine to be executed asynchronously in a process context.
  - Each work item in the queue is represented by an instance ***work_struct.***
  - ***DECLARE_WORK()*** : create and initialize Wq.
  - ***schedule_work()*** : schedule a work into a workqueue.
  - ***schedule_work_on()*** : mark a work item for execution on a specific CPU,while scheduling it into the queue.
  - Another API allows the caller to queue work tasks whose execution is guaranteed to be delayed at least until a specified timeout using ***delayed_work.***

/include/linux/workqueue.h

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};

#define DECLARE_WORK(n, f)                          \
    struct work_struct n = __WORK_INITIALIZER(n, f)

static inline bool schedule_work(struct work_struct *work)
{
    return queue_work(system_wq, work);
}

static inline bool schedule_work_on(int cpu, struct work_struct *work)
{
    return queue_work_on(cpu, system_wq, work);
}
```

```
struct delayed_work {
    struct work_struct work;
    struct timer_list timer;
    struct workqueue_struct *wq;
    int cpu;
};
```