Ausgewählte Betriebssysteme 2004

Interrupt handling
Kernel activities

Agenda

- Interrupts
- Exceptions
  - Syscalls
- Deferred execution contexts
  - Soft IRQ
  - Tasklets
  - Bottom halfs
  - Task queues

Transfer of control

- Kernel needs to be able to regain control under defined conditions
- Exception
  - Error induced
    - Divide by zero, invalid opcode
  - Page fault
  - Syscall
- Interrupt
  - External devices
  - Timer for scheduling

Interrupts

- Hardware Interrupt
  - Limited number
    - Old style PIC (programmable interrupt contr.): 15
    - SMP capable APIC (Advanced PIC): 24 +
    - Sharing may alleviate the problem
  - Asynchronous notification
    - Not associated with current activity
    - No access to process specific data that are arbitrary

Interrupts (2)

```
peter@arsen$ cat /proc/interrupts
           CPU0
  0:  113940238          XT-PIC  timer
  1:     420372          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  8:          4          XT-PIC  rtc
 11:   16556165          XT-PIC  usb-uhci, usb-uhci, usb-uhci,
eth0, Intel 82801CA-ICH3
 12:   11099606          XT-PIC  PS/2 Mouse
 14:    1389405          XT-PIC  ide0
 15:          3          XT-PIC  ide1
NMI:          0
LOC:          0
ERR:          0
MIS:          0
```

Exception

- CPU signals abnormal programm execution
  - Current activity is responsible
    - Deterministic with respect to control flow in user code
    - Transformation into user visible signal for certain events
      - e.g. `SIGSEGV` when accessing not allocated memory
- Defined by processor architecture
  - Max. 32 on IA32

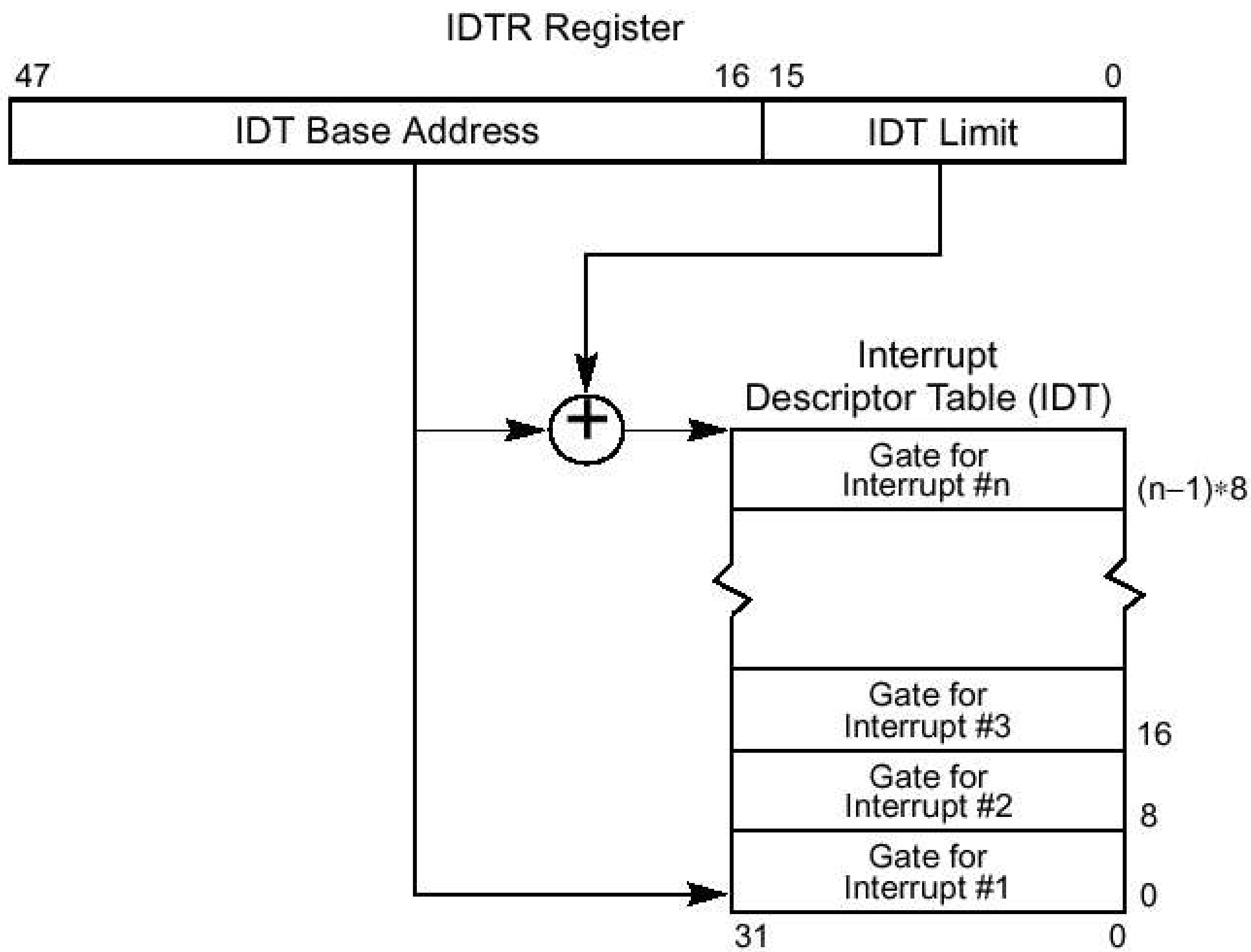
Exception (2)

#	Exception	Exception handler	Signal
0	Divide error	`divide_error()`	SIGFPE
1	Debug	`Debug()`	SIGTRAP
2	NMI	`Nmi()`	None
3	Breakpoint	`Int3()`	SIGTRAP
4	Overflow	`Overflow()`	SIGSEGV
5	Bounds check	`Bounds()`	SIGSEGV
6	Invalid opcode	`invalid_op()`	SIGILL
7	Device not available	`device_not_available()`	SIGSEGV
8	Double Fault	`double_fault()`	SIGSEGV
9	Coprocessor segment overrun	`coprocessor_segment_overrun()`	SIGFPE
10	Invalid TSS	`invalid_tss()`	SIGSEGV
11	Segment not present	`Segment_not_present()`	SIGBUS
12	Stack exception	`setack_segment()`	SIGBUS
13	General protection	`general_protection()`	SIGSEGV
14	Page fault	`page_fault()`	SIGSEGV
15	Reserved	None	None
16	Floating point error	`coprocessor_error()`	SIGFPE
17	Alignment check	`alignment_check()`	SIGBUS
18	Machine check	`machine_check()`	None
19	SIMD floating point	`simd_coprocessor_error()`	SIGFPE

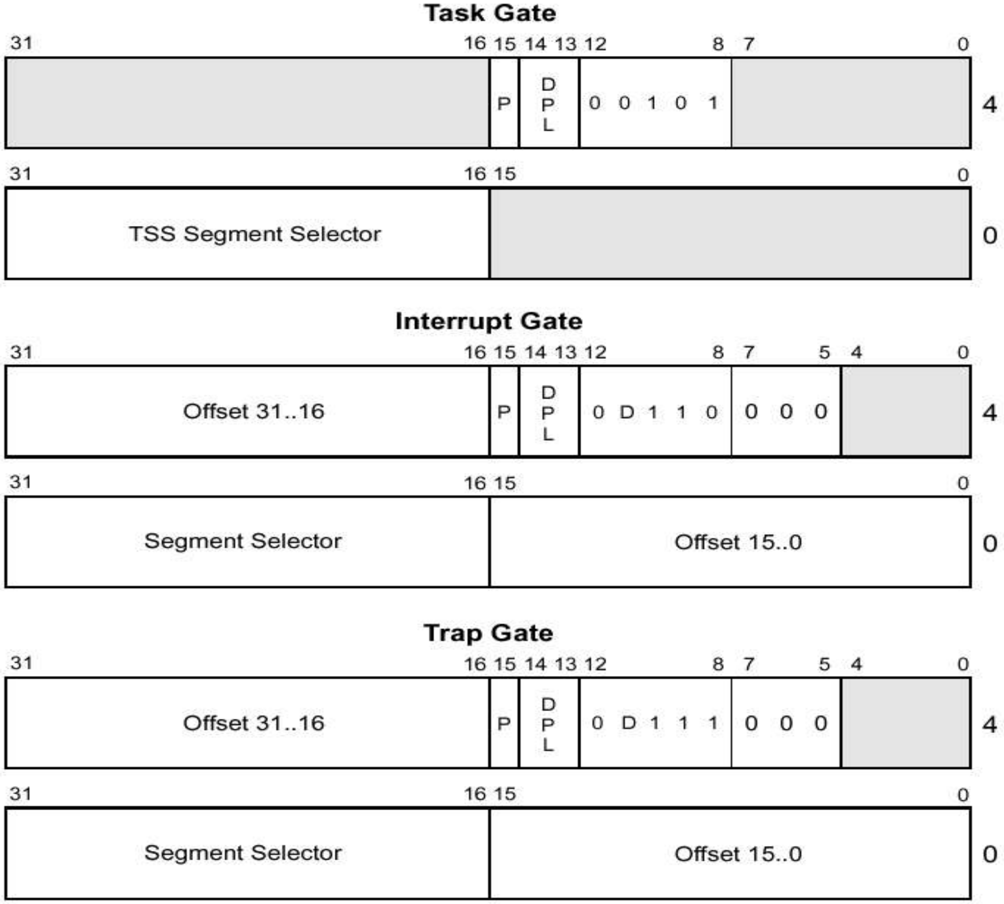
Interrupt descriptor table

- Entry points for interrupts and exceptions
  - Max. 256 entries
  - First 32 fixed for exceptions
  - Interrupt vectors programmable with external controller
- Data structure
  - Global variable `struct desc_struct idt_table[256]`
  - Access functions: `set_intr_gate`, `set_system_gate`, `set_trap_gate`

IDT (2)

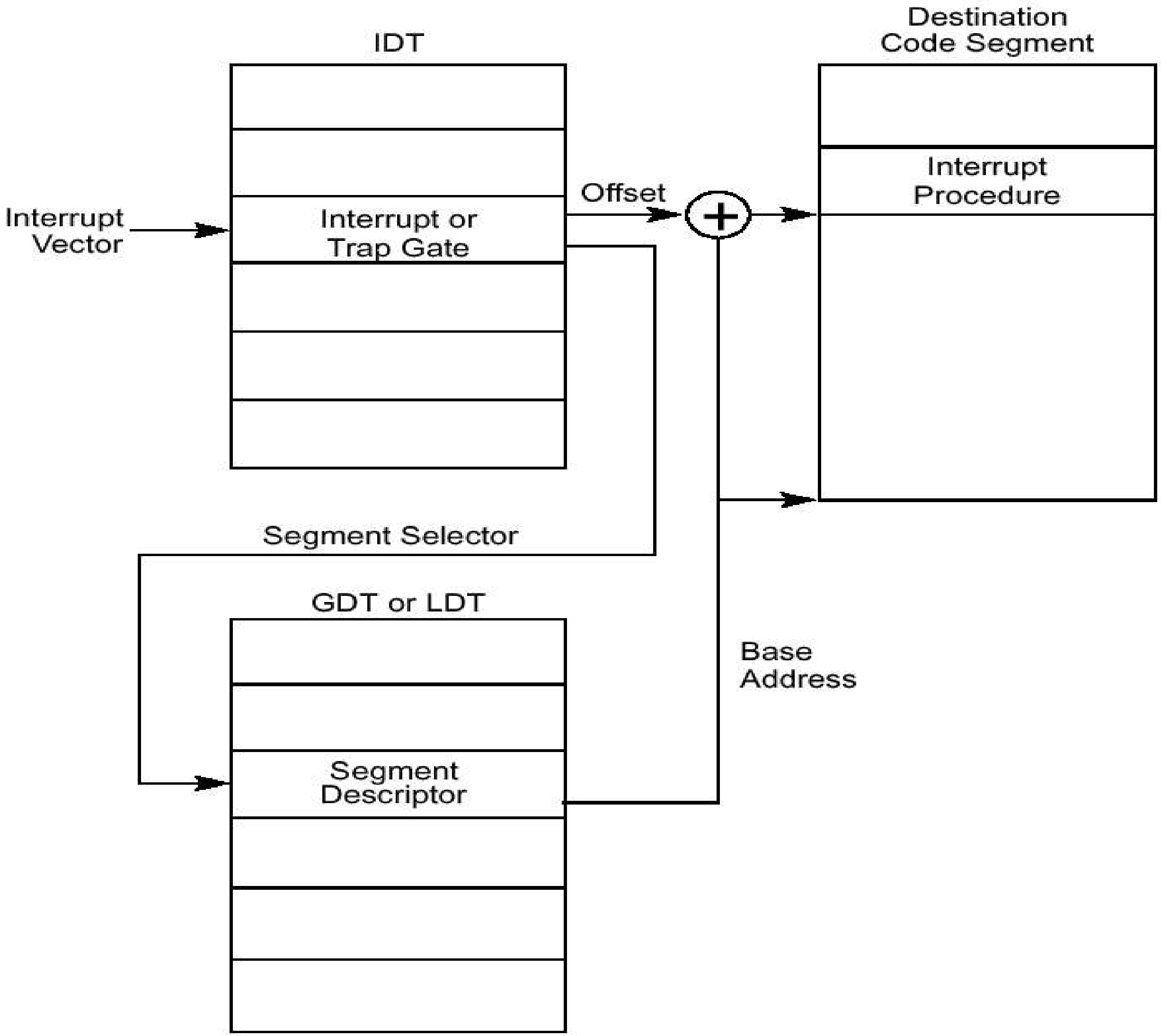

IDTR Register
47
16 15
0
IDT Base Address
IDT Limit
Interrupt
Descriptor Table (IDT)
Gate for
Interrupt #n
(n–1)∗8
Gate for
Interrupt #3
16
Gate for
Interrupt #2
8
Gate for
Interrupt #1
0
31
0

IDT (3)

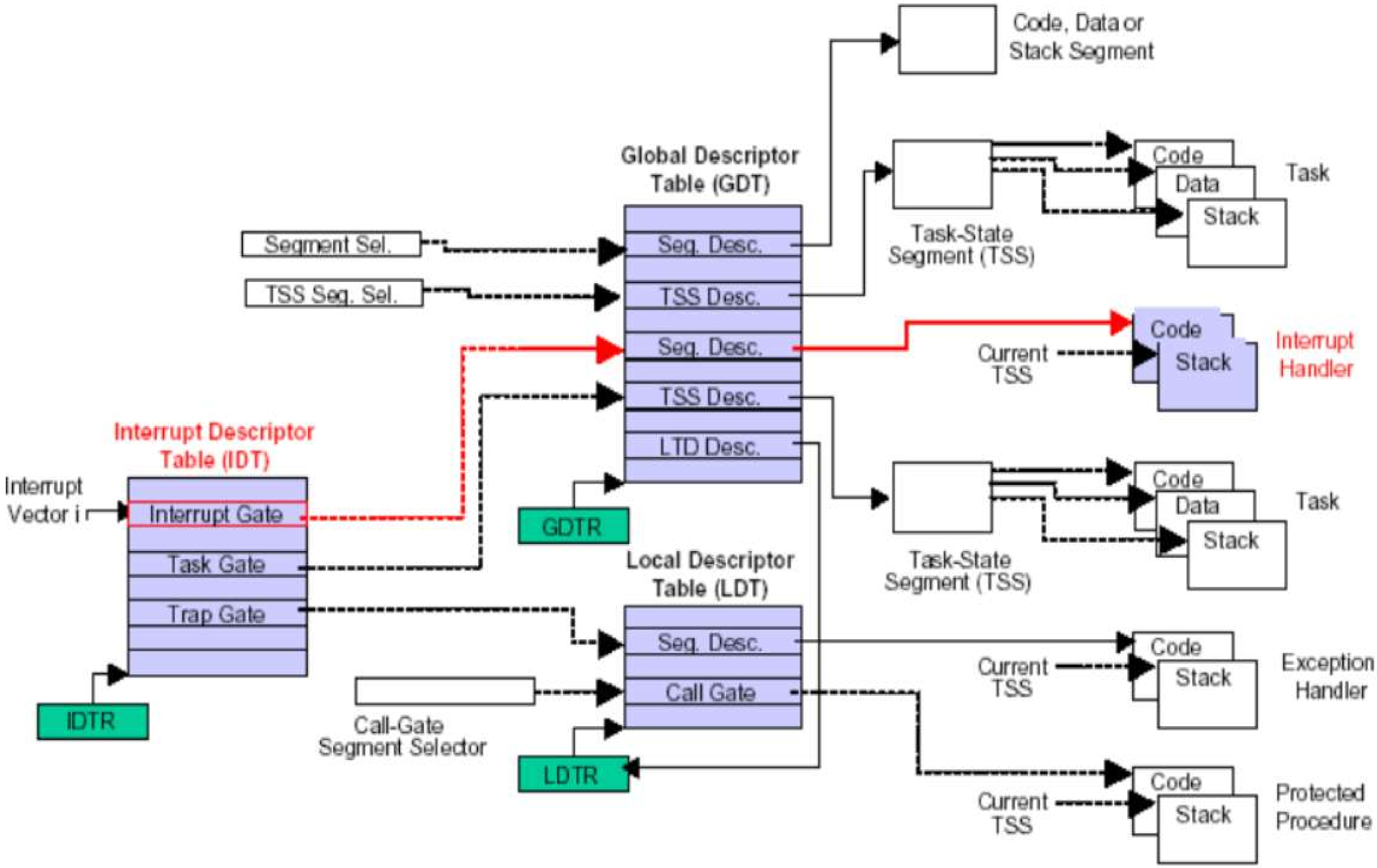


DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
D	Size of gate: 1 = 32 bits; 0 = 16 bits

IDT (4)


IDT
Destination
Code Segment
Interrupt
Procedure
Offset
Interrupt
Vector
Interrupt or
Trap Gate
Segment Selector
GDT or LDT
Base
Address
Segment
Descriptor

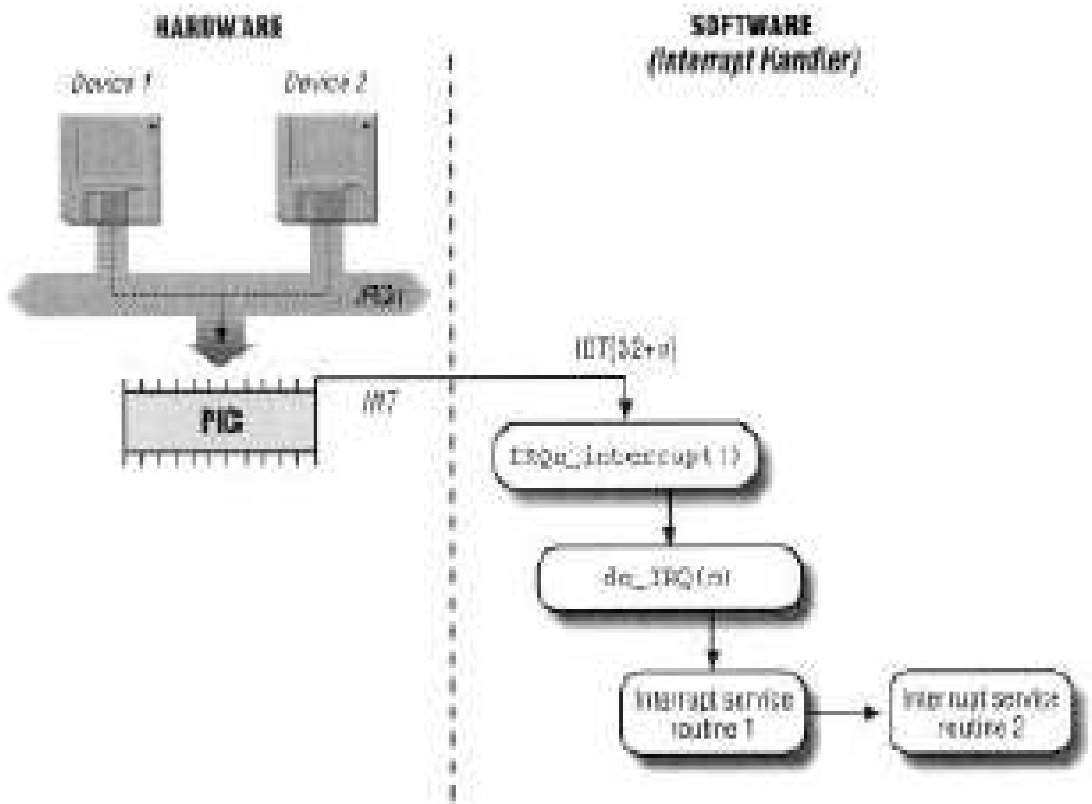
The whole picture



Interrupt handler

- For basic actions
  - Save the current state
  - Acknowledge interrupt at the controller
  - Execute interrupt service routines
    - Possibly schedule work for delayed execution
  - Return to previous execution context
    - Run delayed work before returning to non-interrupt context

Interrupt Handler (2)



IRQ entry

- HW
  - Continue execution in well defined environment
    - Switch to kernel mode
    - `eip` and `cs` from IDT
    - `esp` from TSS
- SW (Linux)
  - Low level assembly code generated with `BUILD_IRQ` macro
  - Finally call C-Function `do_IRQ`
    - Code for several irq controller (acknowledge irq etc.)
    - Invoke registered irq handlers from drivers
    - schedule deferred activities (`do_softirq`)

IRQ descriptor

- Linux abstraction for IRQ
  - IDT is architecture specific
  - `irq_desc_t` contains data for general abstraction
- Support for IRQ sharing
  - One irq can support several interrupt handler

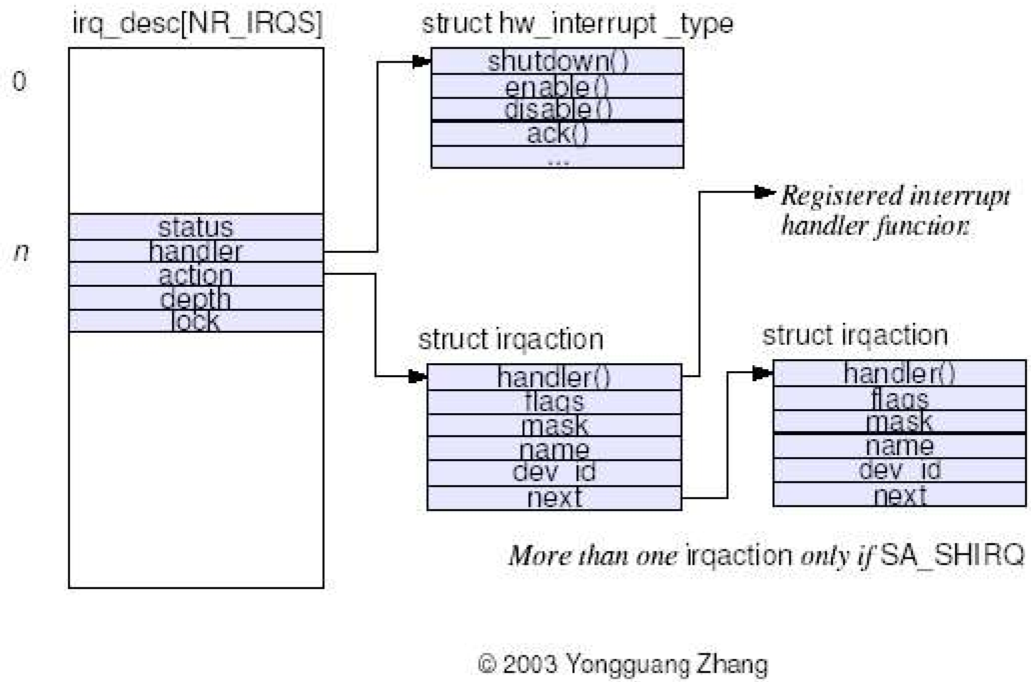
IRQ descriptor (2)

```
/* include/linux/irq.h */

typedef struct {
   unsigned int status;    /* IRQ status */
   hw_irq_controller *handler;
   struct irqaction *action;  /* IRQ action list */
   unsigned int depth;     /* nested irq disables */
   spinlock_t lock;
} ____cacheline_aligned irq_desc_t;

extern irq_desc_t irq_desc [NR_IRQS];
```

IRQ descriptor (3)



IRQ handler

- Driver can register irq handler

```
int request_irq(unsigned int irq,
        void (*handler)(int, void *, struct pt_regs *),
        unsigned long irqflags,
        const char * devname,
        void *dev_id)
```

- and unregister

```
void free_irq(unsigned int irq, void *dev_id)
```

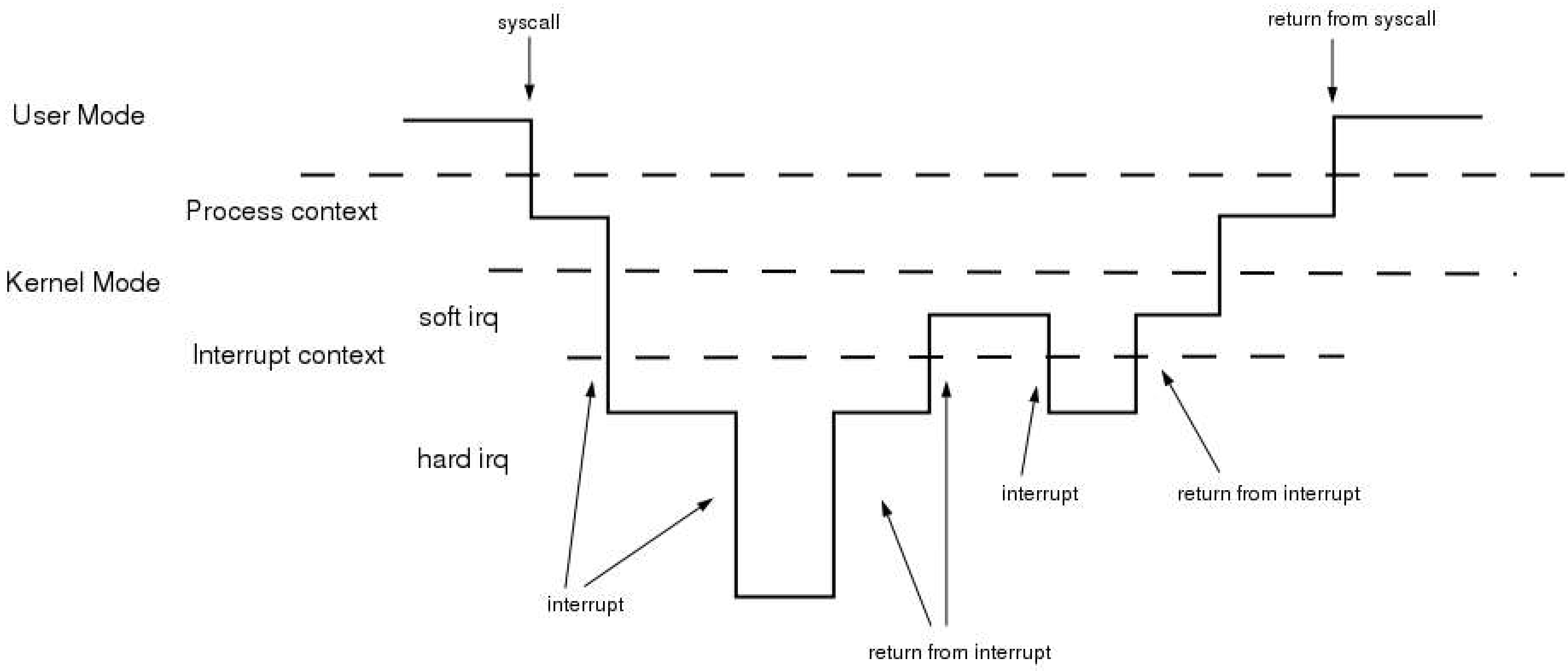
IRQ flags

- `SA_INTERRUPT`
  - Handler is not itself interruptible
  - If not given, interrupts are enabled before executing handler
- `SA_SHIRQ`
  - Interrupt line might be shared
- `SA_SAMPLE_RANDOM`
  - Suitable as entropy source for random number generation

IRQ nesting

- Interrupts can be nested unless `SA_INTERRUPT` prohibits this
- Exceptions cannot be nested
  - Kernel should never trigger exceptions
    - Page faults are sometimes legitimate
  - Exceptions (with syscalls as special case) can be interrupted

IRQ nesting (2)



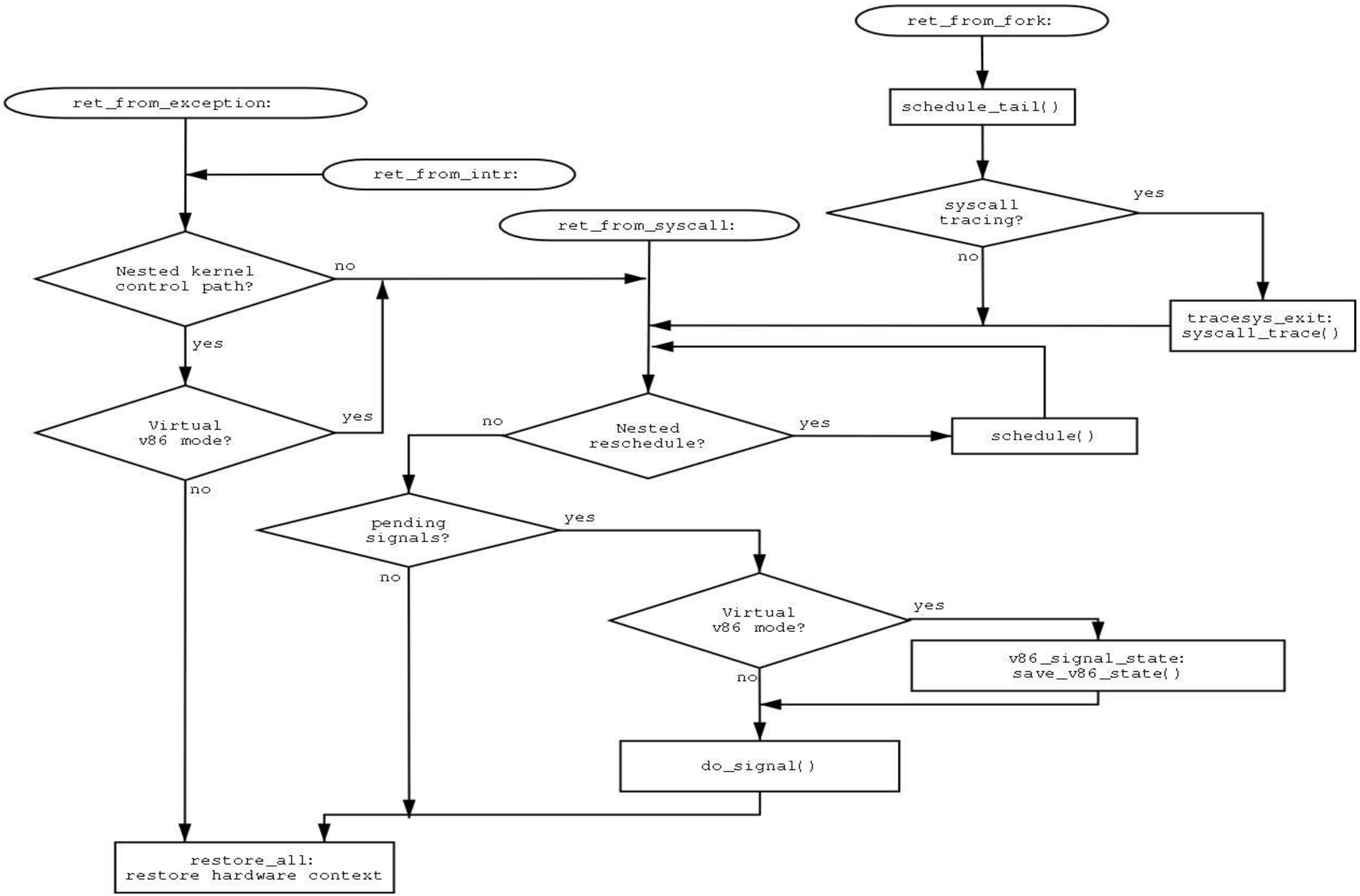
Interrupt context

- Execution environment of interrupt handler
  - Stack of arbitrary process
- Restrictions
  - Cannot sleep, could block the underlying activity
  - Cannot access user space, arbitrary address space mapped in
  - Memory allocation only with `GFP_ATOMIC`
- Interrupt handlers should finish promptly
  - Device activities often consist of fast device access and not time critical data processing (network stack)

Return from interrupt/exception

- Things to consider
  - Reschedule
  - Signals
- Kernel control path
  - No function since control is not returned to caller
  - Four similar cases
    - ret_from_syscall
    - ret_from_exception
    - ret_from_syscall
    - ret_from_fork

Return flow


ret_from_fork:
schedule_tail()
ret_from_exception:
ret_from_intr:
syscall
tracing?
yes
no
ret_from_syscall:
Nested kernel
control path?
no
yes
tracesys_exit:
syscall_trace()
Virtual
v86 mode?
yes
no
Nested
reschedule?
no
yes
schedule()
pending
signals?
yes
no
Virtual
v86 mode?
yes
no
v86_signal_state:
save_v86_state()
do_signal()
restore_all:
restore hardware context

Deferred invocation

- Non-time critical longer lasting activities are marked for execution and executed later
- Fast Interrupt handler
  - Services the device
  - Acknowledges IRQ
  - Markes appropriate activity for later execution
- Evolving concepts
  - Bottom halves, task queues, tasklets, soft irq, work queues
  - Differ on supported parallelism and need for synchronisation

Supported parallelism

	Same activity	Different activity
HW IRQ	-	+
Soft IRQ	+	+
Tasklet	-	+
Bottom Half	-	-

Parallel execution on multiple processors

Interruptibility

	HW-IRQ	Soft-IRQ	Tasklet	Bottom Half
HW-IRQ	+/-	-	-	-
Soft-IRQ	+	-	-	-
Tasklet	+	-	-	-
Bottom Half	+	-	-	-
Syscall	+	+	+	+
User mode	+	+	+	+

- HW IRQ can specify if nesting is possible
- Sequential execution eases synchronisation requirements

Soft IRQ

- Software handled IRQ
  - Mechanism to executed functionality upon request
    - Triggered by other software (most likely interrupt handler)
  - Some soft IRQs used to run tasklets
  - Since triggered in SW does not kick in automatically
- Six kinds of distinguishable softirqs (2.5.69)

```
enum
{
    HI_SOFTIRQ=0,
    TIMER_SOFTIRQ,
    NET_TX_SOFTIRQ,
    NET_RX_SOFTIRQ,
    SCSI_SOFTIRQ,
    TASKLET_SOFTIRQ
};
```

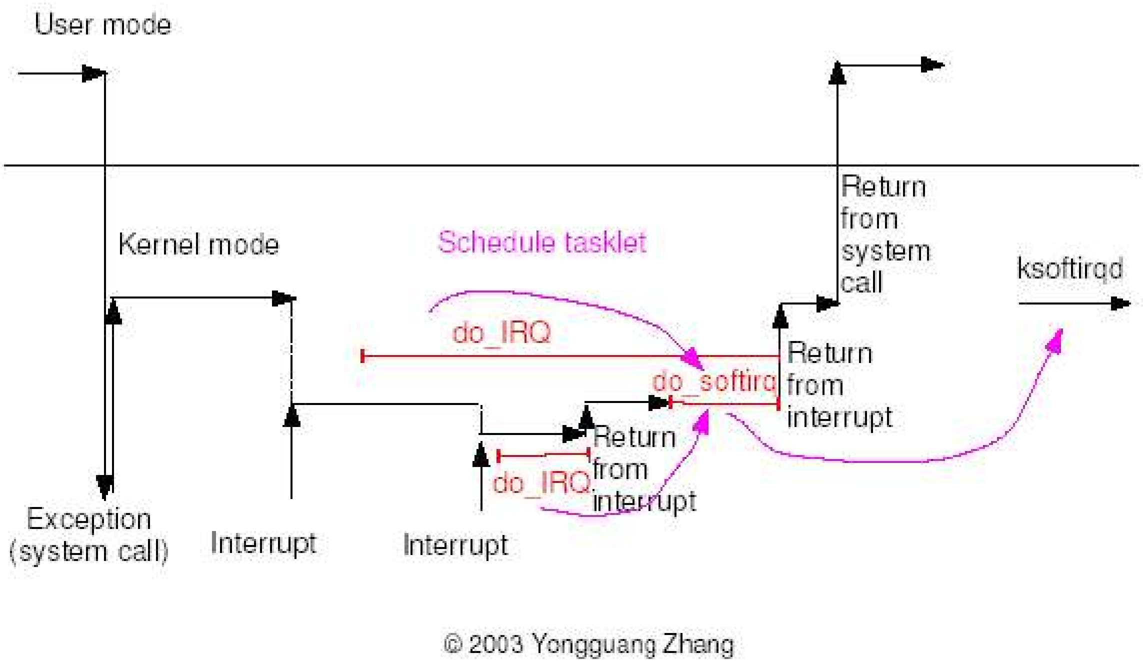
Soft IRQ (2)

- `do_softirq()` (kernel/softirq.c)
  - Upon return from `do_IRQ()`
    - Interrupt handlers might have raised soft IRQs
    - No association with current process context
  - ksoftirqd kernel thread
    - Defined process context, particularly no user process context
    - One thread per CPU
      - Scales with number of cpus
    -
```
while(softirq_pending(cpu)){
    do_softirq();
    if (current->need_resched)
        schedule();
}
```

do_softirq

- Return if in nested IRQ or softirq already executed on that processor
  - Checks `in_interrupt()`
    - Incremented from both hard and soft irq
  - A better suited time will come timely
- Run all pending soft irq
- If new soft irq were raised while current activation was executed, wake up `ksoftirqd`

do_softirq (2)


User mode
Kernel mode
Schedule tasklet
do_IRQ
do_softirq
Return from system call
ksoftirqd
Return from interrupt
Return from interrupt
do_IRQ
Exception (system call)
Interrupt
Interrupt
© 2003 Yongguang Zhang

Tasklet

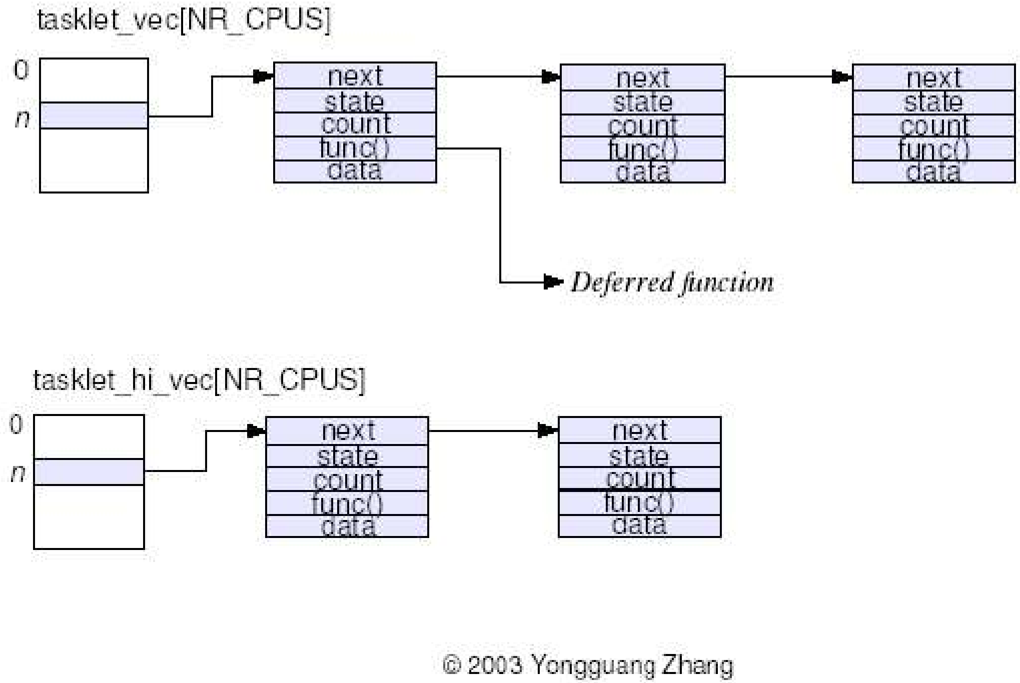
- Guaranteed to run exactly once.
  - Multiple activations are fused.
  - Can schedule itself while running for another execution.
- The execution of a particular tasklet does not nest.
- Different tasklet can run in parallel on different processors.

Tasklet (2)

- Function with supplied argument
- Defined in `include/linux/interrupt.h`

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

Tasklet (3)



tasklet_vec[NR_CPUS]
0
n
next
state
count
func()
data
next
state
count
func()
data
next
state
count
func()
data
Deferred function
tasklet_hi_vec[NR_CPUS]
0
n
next
state
count
func()
data
next
state
count
func()
data
© 2003 Yongguang Zhang

Tasklet (4)

- Define a function with one argument
  - `void tasklet_func(unsigned long data)`
- `DECLARE_TASKLET(name,function, data)`

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

- `static inline void tasklet_schedule(struct tasklet_struct *t)`
  - Add tasklet to the corresponding tasklet list and raise softirq
- ```
  static inline void tasklet_enable(struct tasklet_struct *t)
  static inline void tasklet_disable(struct tasklet_struct *t)
  ```
  - disable/enable tasklet, irrespective pending activation
  ```

Mechanics

```
#define __cpu_raise_softirq(cpu, nr) do { softirq_pending(cpu) |= 1UL << (nr); } while (0)
```

```
inline void cpu_raise_softirq(unsigned int cpu, unsigned int nr)
{
        __cpu_raise_softirq(cpu, nr);

        /*
         * If we're in an interrupt or bh, we're done
         * (this also catches bh-disabled code). We will
         * actually run the softirq once we return from
         * the irq or bh.
         *
         * Otherwise we wake up ksoftirqd to make sure we
         * schedule the softirq soon.
         */
        if (!(local_irq_count(cpu) | local_bh_count(cpu)))
                wakeup_softirqd(cpu);
}
```

```
static inline void tasklet_schedule(struct tasklet_struct *t)
{
        if (!test_and_set_bit(TASKLET_STATE_SCHED, &t->state))
                __tasklet_schedule(t);
}
```

```
void __tasklet_schedule(struct tasklet_struct *t)
{
        int cpu = smp_processor_id();
        unsigned long flags;

        local_irq_save(flags);
        t->next = tasklet_vec[cpu].list;
        tasklet_vec[cpu].list = t;
        cpu_raise_softirq(cpu, TASKLET_SOFTIRQ);
        local_irq_restore(flags);
}
```

ksoftirqd

- do_softirq is invoked upon each return to non-interrupt context
  - Soft IRQ might raise itself
  - Device IRQ might arrive at a higher rate than soft IRQ completion rate
- Not handling reraised soft irqs might result in high latency
- Immediate handling might result in effectively blocking user code execution
- Dedicated kernel thread per CPU executes reraised soft IRQs
  - Low priority so user code gets executed

do_softirq

```
asmlinkage void do_softirq(void)
{
        int max_restart = MAX_SOFTIRQ_RESTART;
        __u32 pending;
        unsigned long flags;

        if (in_interrupt())
                return;

        local_irq_save(flags);

        pending = local_softirq_pending();

        if (pending) {
                struct softirq_action *h;

                local_bh_disable();
restart:
                /* Reset the pending bitmask before enabling irqs */
                local_softirq_pending() = 0;

                local_irq_enable();
```

```
                h = softirq_vec;

                do {
                        if (pending & 1)
                                h->action(h);
                        h++;
                        pending >>= 1;
                } while (pending);

                local_irq_disable();
                pending = local_softirq_pending();
                if (pending && --max_restart)
                       goto restart;
                if (pending)
                  wakeup_softirqd();
                __local_bh_enable();
        }

        local_irq_restore(flags);
}
```

ksoftirqd implementation

```
static int ksoftirqd(void * __bind_cpu)
{
        int bind_cpu = (int) (long) __bind_cpu;
        int cpu = cpu_logical_map(bind_cpu);

        daemonize();
        current->nice = 19;
        sigfillset(&current->blocked);

        /* Migrate to the right CPU */
        current->cpus_allowed = 1UL << cpu;
        while (smp_processor_id() != cpu)
                schedule();
         sprintf(current->comm, "ksoftirqd_CPU%d", bind_cpu);

        __set_current_state(TASK_INTERRUPTIBLE);
        mb();
         ksoftirqd_task(cpu) = current;

        for (;;) {
                if (!softirq_pending(cpu))
                        schedule();

                __set_current_state(TASK_RUNNING);

                while (softirq_pending(cpu)) {
                        do_softirq();
                        if (current->need_resched)
                                schedule();
                }
                 __set_current_state(TASK_INTERRUPTIBLE);
        }
}
```

Tasklet summary

- Preferred meachanism for deferred activity
- Utilizes to softirq
- Limitations
  - Must not block
  - No user space access
  - Limited memory allocation (GPF_ATOMIC)

Bottom halves

- Backward compatibility
  - Sequential execution model
  - Mindcraft study showed negativ impact of network handling bottom half on networking
  - Emulated in 2.4 with tasklets
  - Removed in late 2.5
- Unsufficiencies
  - Not dynamically allocatable (fixed number of 32)
    - Cannot be used by dynamically loaded drivers
  - No parallelism among independent BH

BH implementation

- Backward compatibility
  - List of BH functions, each as a seperate tasklet
  - mark_bh(int nr) to mark BH ready for execution
  - include/linux/interrupt.h (2.4.26)

```
extern struct tasklet_struct bh_task_vec[];
static inline void mark_bh(int nr)
{
        tasklet_hi_schedule(bh_task_vec+nr);
}
```

Task queues

- First extension to bottom halves
- Mechanism to group functionality and execute at an appropriate time
  - Execution environment not specified
    - Interrupt and non-interrupt execution possible
- Interface
  - `DECLARE_TASK_QUEUE`
  - `run_task_queue`
  - `queue_task`
  - `schedule_task` (for `tq_context` only)

Task queues (2)

- Predefined task queues
  - `tq_immediate`
    - de facto bottom half semantics
      - Special bottom half (`BH_IMMEDIATE`) executes all accumulated tasks
      - Queueing must be followed by `mark_bh(BH_IMMEDIATE)`
  - `tq_timer`
      - Executed on each timer tick
  - `tq_disk`
  - `tq_scheduler`

keventd

- Dedicated kernel thread for task queue execution
- Does not run in interrupt context
  - Blocking is allowed

```
int schedule_task(struct tq_struct *task)
{
        int ret;
        need_keventd(__FUNCTION__);
        ret = queue_task(task, &tq_context);
        wake_up(&context_task_wq);
        return ret;
}
```

Work queues

- Replaced task queues in 2.5.41
- Each work queue has its own kernel thread
  - Work is subject to regular scheduling
  - Scales with number of instanciated work queues
  - Can handle queues with different execution behavior
  - Non-interrupt context, blocking and relaxed memory allocation are possible
- More functionality
  - Delayed execution
  - Flushing
  - Easy to create
- See also lwn.net/Articles/23634

Default workqueue

- Each cpu provides a freely usable work queue
  - Successor to former keventd

```
vm-guest:~# uname -a; ps aux
Linux vm-guest 2.6.5 #6 Wed Apr 21 17:41:22 CEST 2004 i686 GNU/Linux
USER       PID %CPU %MEM   VSZ  RSS TTY      STAT START   TIME COMMAND
root         1  0.4  0.5  1524  520 ?        S    18:23   1:05 init [2]
root         2  0.0  0.0     0    0 ?        SN   18:23   0:00 [ksoftirqd/0]
root         3  0.0  0.0     0    0 ?        S<   18:23   0:00 [events/0]
root         4  0.0  0.0     0    0 ?        S<   18:23   0:00 [kblockd/0]
root         5  0.0  0.0     0    0 ?        S    18:23   0:00 [pdflush]
root         6  0.0  0.0     0    0 ?        S    18:23   0:02 [pdflush]
```

```
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
void flush_scheduled_work(void);
```

workqueue API

```
struct workqueue_struct *create_workqueue(const char *name);
DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work,
            void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work,
               void (*function)(void *), void *data);

int queue_work(struct workqueue_struct *queue,
             struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue,
                    struct work_struct *work,
                     unsigned long delay);

int cancel_delayed_work(struct work_struct *work);
void flush_workqueue(struct workqueue_struct *queue);
void destroy_workqueue(struct workqueue_struct *queue);
```

Wait queues

- Rendevouz endpoint
  - Location to wait
  - Place where to find people for wakeup
- Base for higher level constructions
  - Notifications
  - Semaphores
- Concepts involved
  - Scheduling
    - Indicate willingness to release cpu

Wait queues (2)

- Wait for an event
  - Process cannot proceed due to unsatisfied data dependencies
  - Release cpu
- Wake up after event happened
  - Often asynchronous
    - timer
    - irq handler, tasklet
  - Make waiting process running again

Wait queues (3)

- DECLARE_WAITQUEUE
- DECLARE_WAIT_QUEUE_HEAD
- init_waitqueue_entry
- init_waitqueue_head
- sleep_on
- interruptible_sleep_on
- wake_up
- wake_up_interruptible

sleep intrinsics

```
#define SLEEP_ON_VAR                                    \
        unsigned long flags;                            \
        wait_queue_t wait;                              \
        init_waitqueue_entry(&wait, current);

#define SLEEP_ON_HEAD                                   \
        spin_lock_irqsave(&q->lock,flags);              \
        __add_wait_queue(q, &wait);                     \
        spin_unlock(&q->lock);

#define SLEEP_ON_TAIL                                   \
        spin_lock_irq(&q->lock);                        \
        __remove_wait_queue(q, &wait);                  \
        spin_unlock_irqrestore(&q->lock, flags);
```

```
void fastcall sleep_on(wait_queue_head_t *q)
{
        SLEEP_ON_VAR

        current->state = TASK_UNINTERRUPTIBLE;

        SLEEP_ON_HEAD
        schedule();
        SLEEP_ON_TAIL
}
```

The race

```
DECLARE_WAIT_QUEUE_HEAD(q);
```

```
if (!condition)
  {
    /* if preempted here */
    /* wakeups might get lost */
    sleep_on(&q);
  }
```

```
wakeup_all(&q);
```

Doing it properly

```
DECLARE_WAIT_QUEUE_HEAD(queue);
DECLARE_WAITQUEUE(wait, current);
for (;;) {
    add_wait_queue(&queue, &wait);
    set_current_state(TASK_INTERRUPTIBLE);
    if (condition)
        break;
    schedule();
    remove_wait_queue(&queue, &wait);
    if (signal_pending(current))
        return -ERESTARTSYS;
}
set_current_state(TASK_RUNNING);
```

The Candidates

- wait_event
  - Simple model
- prepare_to_wait, finish_wait
  - Provides finer control over the various steps
  - lwn.net/Archives/22913
- completions
  - Include counters
  - lwn.net/Archives/23993

wait_event

```
#define __wait_event(wq, condition)                                     \
do {                                                                    \
        wait_queue_t __wait;                                            \
        init_waitqueue_entry(&__wait, current);                         \
                                                                        \
        add_wait_queue(&wq, &__wait);                                   \
        for (;;) {                                                      \
                set_current_state(TASK_UNINTERRUPTIBLE);                \
                if (condition)                                          \
                        break;                                          \
                schedule();                                             \
        }                                                               \
        current->state = TASK_RUNNING;                                  \
        remove_wait_queue(&wq, &__wait);                                \
} while (0)

#define wait_event(wq, condition)                                       \
do {                                                                    \
        if (condition)                                                  \
                break;                                                  \
        __wait_event(wq, condition);                                    \
} while (0)
```

DECLARE_WAIT_QUEUE_HEAD(queue);
wait_event(queue, condition);
int wait_event_interruptible (queue, condition);
int wait_event_interruptible_timeout(queue, condition, timeout);

prepare_to_wait

```
void fastcall prepare_to_wait(wait_queue_head_t *q,
                              wait_queue_t *wait, int state)
{
        unsigned long flags;

        wait->flags &= ~WQ_FLAG_EXCLUSIVE;
        spin_lock_irqsave(&q->lock, flags);
        if (list_empty(&wait->task_list))
                __add_wait_queue(q, wait);
        set_current_state(state);
        spin_unlock_irqrestore(&q->lock, flags);
}
```

```
void fastcall finish_wait(wait_queue_head_t *q,
                          wait_queue_t *wait)
{
        unsigned long flags;

        __set_current_state(TASK_RUNNING);
    /*
     * We can check for list emptiness outside the lock
     * IFF:
     *  - we use the "careful" check that verifies both
     *    the next and prev pointers, so that there cannot
     *    be any half-pending updates in progress on other
     *    CPU's that we haven't seen yet (and that might
     *    still change the stack area.
     * and
     *  - all other users take the lock (ie we can only
     *    have _one_ other CPU that looks at or modifies
     *    the list).
     */
        if (!list_empty_careful(&wait->task_list)) {
                spin_lock_irqsave(&q->lock, flags);
                list_del_init(&wait->task_list);
                spin_unlock_irqrestore(&q->lock, flags);
        }
}
```

```
DECLARE_WAIT_QUEUE_HEAD(queue);
DEFINE_WAIT(wait);
while (! condition) {
     prepare_to_wait(&queue, &wait, TASK_INTERRUPTIBLE);
     if (! condition)
          schedule();
     finish_wait(&queue, &wait)
}
```

Completions

```
/* static declaration */
DECLARE_COMPLETION(my_comp);

/* dynamic delcaration */
struct completion my_comp;
init_completion(&my_comp);

/* obvious meaning, resistent against lost wakeups*/
void wait_for_completion(struct completion *comp);

void complete(struct completion *comp);
void complete_all(struct completion *comp);
```

Completions (2)

- `DECLARE_COMPLETION`
- `wait_for_completion`
- `complete`
- `complete_all`