Linux Interrupts: The Basic Concepts

Mika J. Järvenpää
University of Helsinki
Department of Computer Science
mjjarven@cs.helsinki.fi

ABSTRACT

The idea of this document is to shed some light to the concepts and main ideas of implementation of interrupts in Linux kernel. Main focus is in Linux kernel version 2.4.18-10.

1. LINUX INTERRUPTS

At any time one CPU in a Linux system can be:
- serving a hardware interrupt in kernel mode
- serving a softirq, tasklet or bottom half in kernel mode
- running a process in user mode.

There is a strict order between these. Other than the last category (user mode) can only be preempted by those above. For example, while a softirq is running on a CPU, no other softirqs will preempt it, but a hardware interrupt can.
In the next chapters different interrupt and exception types, initialization, hardware handling and software handling of interrupts, interrupt data structures and terms like IDT, bottom half, softirq and tasklet are explained in more detail.

2. INTERRUPTS AND EXCEPTIONS

An interrupt is an asynchronous event that is typically generated by an I/O device not synchronized to processor instruction execution.
An exception is a synchronous event that is generated when the processor detects one or more predefined conditions while executing an instruction.
Interrupts can be divided to maskable interrupts and non-maskable interrupts.
Also exceptions can be divided to processor-detected exceptions ie. faults, traps, aborts and programmed exceptions. Example of fault is Page fault. Traps are used mainly for debugging purposes. Aborts inform about hardware failures and invalid system tables and and abort handler has no choice but to force affected process to terminate. [1], [3], p. 4-9, 4-10.

Table 1: Signals sent by the exception handlers

#	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 avail.	device_not_avail.()	SIGSEGV
8	Double fault	double_fault()	SIGSEGV
9	Coproc.seg.ovr.	coproc._seg._ovr.()	SIGFPE
10	Invalid TSS	invalid_tss()	SIGSEGV
11	Seg. not present	seg._not_present()	SIGBUS
12	Stack seg. fault	stack_segment()	SIGBUS
13	General protect.	general_prot.()	SIGSEGV
14	Page fault	page_fault()	SIGSEGV
15	Intel reserved	None	None
16	FP error	coprocessor_error()	SIGFPE
17	Alignment check	alignment_check()	SIGSEGV
18	Machine check	machine_check()	None
19	SIMD coproc.err	simd_coproc._error	Depends

2.1 Exception and Interrupt Vectors

Intel architecture identifies different interrupts and exceptions by a number ranging from 0 to 255 ([5], p. 4-11). This number is called a vector. Linux uses vectors 0 to 31, which are for exceptions and non-maskable interrupts, vectors 32 to 47, which are for maskable interrupts ie. interrupts caused by interrupt requests (IRQs) and only one vector (128) from the remaining vectors ranging from 48 to 255, which are meant for software interrupts.
Linux uses this vector 128 to implement a system call ie. when an `int` 0x80 opcode ([6]) is executed by a process running in user mode the CPU switches into kernel mode and starts executing kernel function `system_call()`.

2.2 Hardware Interrupts

Hardware devices capable of issuing IRQs are connected to *Interrupt Controller*. The Intel 8259A *Programmable Interrupt Controller* (PIC) handles up to eight vectored priority interrupts for the CPU and PICs can be cascaded ([2]). Typical configuration for 15 IRQs is cascade of two PICs. PIC can remember one IRQ while IRQ is masked. Vector of masked IRQ is sent to CPU after unmasking. Processing of previous interrupt is finished by writing *End Of Interrupt*

(EOI) to PIC or both PICs in cascade, if source was the slave PIC. Since the number of available IRQ lines is limited the same IRQ line can be shared between several I/O devices. This is possible, if interrupt handlers poll every I/O device connected to same IRQ line to determine which I/O device should be serviced. Intel processors having local APIC (more about APICs in section: *APIC System*) could receive external interrupts through pins on the processor or through the local APIC serial bus. All maskable hardware interrupts ie. interrupts delivered to CPU by means of `INTR` signal (0-255) or through local APIC (16-255) can be masked as a group by using `IF` flag in the `EFLAGS` register.

2.3 Software Generated Interrupts

The `INT` n instruction permits interrupts to be generated by software. Interrupts generated in software with `INT` n cannot be masked by the `IF` flag in the `EFLAGS` register.

2.4 Exceptions

Intel 80x86 processors generate roughly 20 different exceptions. Some types of exceptions may provide error code, which report additional information about the error. In Linux each exception has specific handler, which usually sends a UNIX signal to the process that caused the exception (see Table 1)[1].

Faults are type of exceptions, in which `eip` contains the address of instruction that caused the exception, so the handler can re-execute the instruction for example after loading the needed page to memory. In case of trap the saved value of `eip` is the address of the next instruction since traps are used mainly for debugging purposes to implement for example breakpoints. Aborts are caused by serious errors and there might not possible to get any address in `eip`. Programmed exceptions are triggered by `int`, `int3` and conditionally by `into` (check for overflow) and `bound` (check on address bound).

Proper list containing conditions which generate exceptions and interrupts on Intel architecture can be found from *Intel Architecture Software Developer's Manual, Volume* 3*: System Programming, Chapter* 5.12*. Exception and Interrupt Reference*([7]).

2.5 Interrupt Descriptor Table

Interrupt Descriptor Table (IDT) associates each exception or interrupt vector with a gate descriptor for the handler used to service the associated exception or interrupt ([7] p. 5-11). IDT need not contain more than 256 descriptors since there are only that amount of interrupt or exception vectors. The IDT must be properly initialized before the kernel enables interrupts. The `idtr` register allows the IDT to be located anywhere in the memory. Figure 1 shows the format of the three different gate descriptors. Linux uses interrupt gates to handle interrupts and trap gates to handle exceptions. Linux does not use task gates.

2.6 Hardware Handling of Interrupts and Exceptions

After executing an instruction, the `cs` and `eip` contain the logical address of the next instruction to be executed. Before executing that instruction the control unit checks, if an interrupt or an exception has occurred while executing the previous instruction. If one occurred, the control unit:

1. Determines the vector i associated with the interrupt or exception.
2. Reads the ith gate descriptor of the IDT referred by the `idtr` register (an interrupt or a trap gate assumed)
3. Gets the base address of the GDT from the `gdtr` register and checks GDT to find the Segment Descriptor for the selector of IDT entry in order to find base address of the segment having interrupt or exception handler.
4. Compares Current Privilege Level (CPL) with the Descriptor Privilege Level (DPL) and issues General Protection (GP) exception, if the DPL is lower than CPL. Interrupt handler cannot have lower privilege than program that caused the interrupt.
5. Check if the CPL is different from the selected Segment Descrictor's DPL. If so, the control unit must start using stack that is associated with the new prilege level.
   (a) Read `tr` to access the TSS of the current process.
   (b) Load new `ss` and `esp` found from TSS.
   (c) Save `ss` and `esp` to new stack.
6. If a fault occurred load `cs` and `eip` with the logical address caused the exception so that it can be executed again.
7. Save `eflags`, `cs` and `eip` to the stack.
8. If the exception generated a hardware error code save it to the stack.
9. Load `cs` and `eip` with values got from ith entry of the IDT ie. the address of the first instruction of the interrupt or exception handler.

Next step caused by control unit is the execution of the first instruction of the handler. After the interrupt or exception

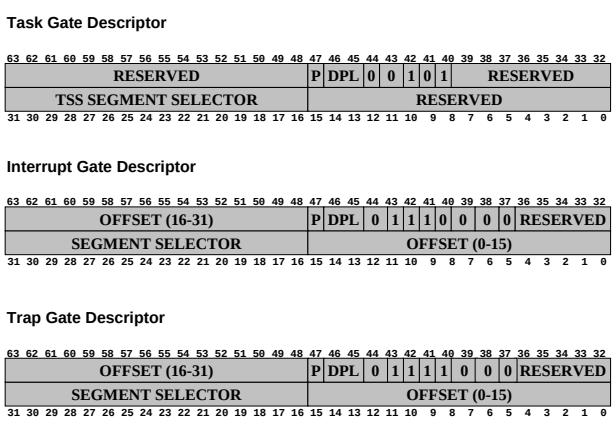

Figure 1: Gate descriptor's format

[1]Exception handler names truncated by typographical reasons

has been processed the handler must issue the `iret` instruction which causes the control unit to:

1. Load `cs`, `eip` and `eflags` from the stack. If hardware error code was pushed to the stack on top of `eip`, it must be popped before executing `iret`.
2. Check if the CPL of the handler is equal as value of two least significant bits of `cs`. If so `iret` finishes execution, otherwise next step is entered.
3. Load the `ss` and `esp` from the stack.
4. Check `ds`, `es`, `fs` and `gs`. If any of them contains a selector that refers to Segment Descriptor whose DPL value is lower than CPL, clear the corresponding segment register in order to forbid the user mode program that run with a CPL 3 from making use of segment registers previously used by the kernel routines.

2.7 Kernel Control Paths

A kernel control path consists of the sequence of instructions executed in kernel mode to handle an interrupt or exception. At most two kernel control paths can be stacked in case of exceptions since page fault exception never gives rise to further exceptions. Linux does not allow scheduling while the CPU is executing a kernel control path associated with an interrupt. But an interrupt handler may be interrupted by another interrupt handler and so on. An interrupt handler may defer an exception handler, but an exception handler never defers an interrupt handler. The only exception possible in kernel mode is the page fault. Interrupt handlers never perform operations that could cause page fault and thus potentially scheduling.
The Linux interleaves kernel control paths for two major reasons: to implement an interrupt model without priority levels and to improve the throughput of PICs by making possible to acknowledgement of IRQ while servicing another IRQ.

3. INITIALIZATION

Interrupt Descriptor Table should be initialized so that illegal interrupts and exceptions simulated by user space applications will be blocked. This is achieved by clearing the DPL field of the Interrupt and Trap Gate Descriptors. If the process attempts to issue one of such interrupt or exception signals, the control unit will check the CPL value against the DPL field and issue a GP exception. In those cases where user space process must be able to issue a programmed exception the DPL field of the corresponding interrupt or trap gate descriptor is set to 3.
Also the base addresses of the IDT should be aligned on an 8-byte boundary to maximize performance of cache line fills. Linux uses three descriptor formats in its IDT:

Interrupt gate An Intel interrupt gate that cannot be accessed by user mode process (gate's DPL field cleared). All Linux interrupt handlers are activated by using interrupt gates.

System gate An intel trap gate that can be accessed by user mode process (gate's DPL equal to 3). The four Linux exception handlers 3, 4, 5, 128 are activated by using system gates The `int3`, `into`, `bound` and `int` 0x80 can be accessed by user mode process.

Trap gates An Intel trap gate that cannot be accessed by a user mode process (gate's DPL field cleared). All Linux exception handlers except those four are activated using of trap gates.

4. EXCEPTION HANDLING

Linux uses exceptions to handle demand paging and to signal process an anomalous condition. Exception handlers have a structure consisting of three parts:

1. Save registers to kernel mode stack.
2. Handle exception using C function.
3. Exit from the handler using `ret_from_exception()`.

After registers are saved and some house-keeping done, C function is called. C function will find on the top of stack:

- The return address (instruction to be executed after C function terminates).
- The address of the stack where user mode registers are saved.
- The hardware error code.

After returning from the C function the code pops the stack address of the saved user mode registers and the hardware error code and then jumps to the `ret_from_exception()` explained later in chapter: *Returning from Interrupts and Exceptions*.
Most of the exception handlers written in C store the hardware error code and the exception vector to the process descriptor of `current` and send a suitable signal to a process which caused an exception. This is done by following code fragment:

```
struct task_struct *tsk = current;
tsk->thread.error_code = error_code;
tsk->thread.trap_no = trapnr;
force_sig(signr, tsk);
```

In the `ret_from_exception()` it is checked if the process has received a signal, but if there is no signal handler, then the kernel will usually kill the process and handle it by itself.
Finally the `die_if_kernel()` or `die_if_no_fixup()` is executed. The `die_if_kernel()` function checks if the exception occurred in kernel space. If it is true then `die()` is invoked, which prints registers to console and terminates the `current` process by invoking `do_exit()`. The `die_if_no_fixup()` function is similar, but before invoking `die()` it checks if the exception was due to an invalid argument of a system call. If it was, then it uses `fixup` approach (mechanism to recover from known exceptional situations) to recover.

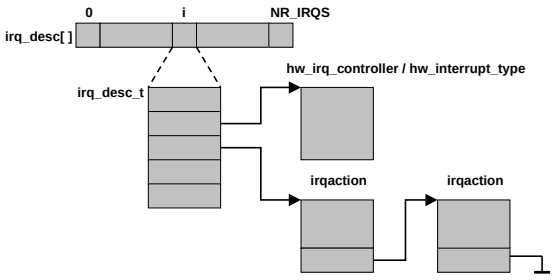

Figure 2: IRQ descriptors

5. INTERRUPT HANDLING

All interrupt handlers preform the same five basic actions:

1. Save IRQ value and the registers to the kernel mode stack.
2. Send ack to the PIC thus allowing it to issue next interrupt.
3. Execute ISRs associated with all the devices that share same IRQ.
4. Execute active softirqs
5. Terminate by jumping to the `ret_from_intr()`.

5.1 Data Structures

An `irq_desc` array includes `NR_IRQS irq_desc_t` type descriptors.

```
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;
```

The `status` field may contain following values:

IRQ_INPROGRESS IRQ handler active - do not enter

IRQ_DISABLED IRQ disabled - do not enter

IRQ_PENDING IRQ pending - replay on enable

IRQ_REPLAY IRQ has been replayed but not acked yet

IRQ_AUTODETECT IRQ is being autodetected

IRQ_WAITING IRQ not yet seen - for autodetection

IRQ_LEVEL IRQ level triggered

IRQ_MASKED IRQ masked - shouldn't be seen again

IRQ_PER_CPU IRQ is per CPU

The `hw_interrupt_type` descriptor includes group of pointers to the low-level I/O routines that interact with a specific interrupt controller.

```
struct hw_interrupt_type {
  const char * typename;
  unsigned int (*startup)(unsigned int irq);
  void (*shutdown)(unsigned int irq);
  void (*enable)(unsigned int irq);
  void (*disable)(unsigned int irq);
  void (*ack)(unsigned int irq);
  void (*end)(unsigned int irq);
  void (*set_affinity)(unsigned int irq,
                       unsigned long mask);
};
typedef struct
  hw_interrupt_type hw_irq_controller;
```

Sample structure initialized for 8259A PIC:

```
static struct hw_interrupt_type i8259A_irq_type = {
  "XT-PIC",
  startup_8259A_irq,
  shutdown_8259A_irq,
  enable_8259A_irq,
  disable_8259A_irq,
  mask_and_ack_8259A,
  end_8259A_irq,
  NULL
};
```

The `irqaction` descriptors can be chained in case of shared IRQs used as shown in figure 2.

```
struct irqaction {
  void (*handler)(int, void *, struct pt_regs *);
  unsigned long flags;
  unsigned long mask;
  const char *name;
  void *dev_id;
  struct irqaction *next;
};
```

The `flags` field can contain following values:

SA_SHIRQ Interrupt is shared.

SA_INTERRUPT Disable local interrupts while processing.

SA_SAMPLE_RANDOM The interrupt can be used for source of random number.

5.2 Saving Registers

In `include/asm-i386/hw_irq.h` is implemented `BUILD_IRQ` macro which is used to implement interrupt handler entry points (not IPI or SMP):

```
IRQn_interrupt:
  pushl $n-256
  jmp common_interrupt
```

There is also `BUILD_COMMON_IRQ` macro which builds common interrupt handler:

```
common_interrupt:
  SAVE_ALL
  call do_IRQ
  jmp ret_from_intr
```

The SAVE_ALL macro expands to:

```
  cld
  pushl %es
  pushl %ds
  pushl %eax
  pushl %ebp
  pushl %edi
  pushl %esi
  pushl %edx
  pushl %ecx
  pushl %ebx
  movl $" STR(__KERNEL_DS) ",%edx
  movl %edx,%ds
  movl %edx,%es
```

5.3 The do_IRQ() Function

The `do_IRQ()` function handles all normal device IRQs (IPIs have their own specific handlers). It first gets lock to the specific IRQ descriptor, so that the first CPU getting the lock takes care of that specific IRQ. Next thing is to acknowledge the IRQ to PIC as fast as possible:

```
desc->handler->ack(irq);
```

After that the status of the IRQ descriptor is updated ("we want to handle this specific IRQ") and after that IRQ descriptor is unlocked. Next thing is to call the IRQ handler:

```
handle_IRQ_event(irq, &regs, action);
```

following IRQ descriptor locking, IRQ descriptor status updating, IRQ descriptor unlocking and calling:

```
desc->handler->end(irq);
```

to deal with interrupts which got disabled while the handler was running and finally check and execute possible softirqs:

```
if (softirq_pending(cpu))
  do_softirq();
return 1;
```

5.4 ISRs

Interrupt service routines (ISRs) implementing device-specific function are all called with same parameters, which are: `irq`, `dev_id` and `regs`. The first parameter allows service of multiple IRQs inside one ISR, the second parameter allows service of several devices of same type and the last parameter allows the access to the execution context of the interrupted kernel control path. In practice these parameters are seldom used inside ISRs.

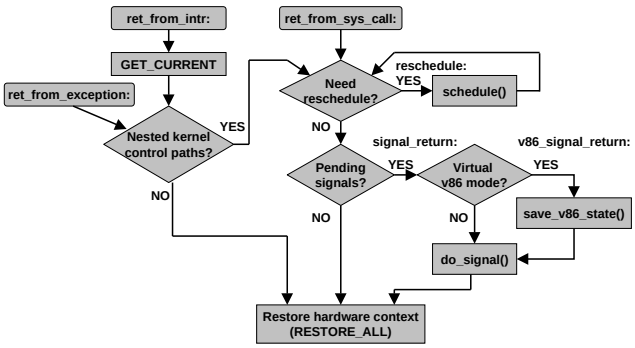

Figure 3: Returning from interrupts and exceptions

5.5 Bottom Halves/Softirqs/Tasklets

Bottom halves (BHs) is the oldest concept in the era of BHs, softirqs and tasklets. The idea of BH is to divide the handling of interrupt into two pieces and make the first piece ie. interrupt handler (top half) to serve hardware as fast as possible and suspend the rest of interrupt handling (bottom half) to the later point of the time. BHs didn't take an advantage of multiple CPUs. Existence of 32 available BHs is defined by a pointer table.
Softirqs are versions of BHs which can run on as many CPUs at once as required (they also need to deal mutual exclusion of shared data using their own locks). Activity of 32 available softirqs is defined by a bit mask.
After developing softirqs BHs were built on top of softirqs.
Tasklets are like softirqs, except they are registered dynamically (count of tasklets is unlimited), and they also guarantee that any tasklet will run on only one CPU at any time (no need for re-entrant code), although different tasklets can run simultaneously on different CPUs (unlike different BHs can't).
BHs are built on top of tasklets and tasklets are built on top of softirqs.
Task queues are dynamic extension of old BHs. Task queue is basically linked list containing function pointers.

5.6 Dynamic Handling of IRQ Lines

There is a way to share the IRQ even if the I/O device do not allow the IRQ sharing. The concept is to serialize the activation of the devices so that one at a time owns the IRQ line. This is implemented by three functions: `request_irq()`, `setup_x86_irq()` and `free_irq()`. The `setup_x86_irq()` function called inside the `request_irq()` function returns the error code, if the IRQ line is already in use. In this case device driver aborts the operation and could try again later.

6. RETURNING FROM INTERRUPTS AND EXCEPTIONS

There are three entry points `ret_from_intr()`, `ret_from_sys_call()` and `ret_from_exception()` which are used when returning from interrupts, exceptions and system calls. Those are covered by following code fragments and a flow diagram in figure 3.

```
#define GET_CURRENT(reg) \
  movl $-8192, reg; \
  andl %esp, reg

ENTRY(ret_from_intr)
  GET_CURRENT(%ebx)
ret_from_exception:
  movl EFLAGS(%esp),%eax
  movb CS(%esp),%al
  testl $(VM_MASK | 3),%eax
  jne ret_from_sys_call
  jmp restore_all
```

The address of the `current` process descriptor is stored to `ebx`. The values of the `eflags` and `cs` (pushed to the stack when the interrupt occurred) are used to determine if the interrupted program was running in kernel mode. If the interrupted program was running in user mode or if the `VM` flag of `eflags` was set a jump is made to the `ret_from_sys_call` entry point:

```
ENTRY(ret_from_sys_call)
  cli
  cmpl $0,need_resched(%ebx)
  jne reschedule
  cmpl $0,sigpending(%ebx)
  jne signal_return
restore_all:
  RESTORE_ALL
```

The `need_resched` and `sigpending` are offsets into the process descriptor. If the `need_resched` field is 1 the `schedule()` function is called:

```
reschedule:
  call SYMBOL_NAME(schedule)    # test
  jmp ret_from_sys_call
```

If the `sigpending` field is not `null` the `signal_return` branch is taken to handle pending signals of `current`:

```
signal_return:
  sti
  testl $(VM_MASK),EFLAGS(%esp)
  movl %esp,%eax
  jne v86_signal_return
  xorl %edx,%edx
  call SYMBOL_NAME(do_signal)
  jmp restore_all
v86_signal_return:
  call SYMBOL_NAME(save_v86_state)
  movl %eax,%esp
  xorl %edx,%edx
  call SYMBOL_NAME(do_signal)
  jmp restore_all
```

After this the process `current` can resume execution in user mode. The `RESTORE_ALL` macro (loads the values saved by `SAVE_ALL` macro) expands to:

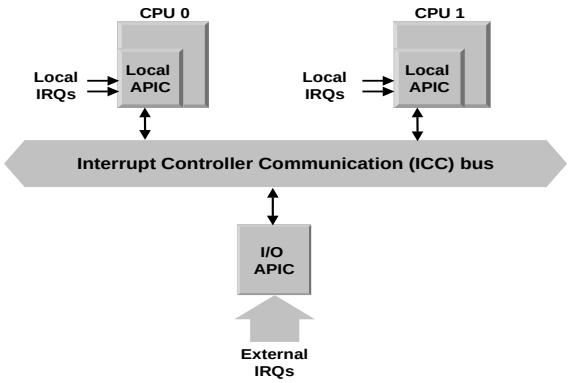

Figure 4: APIC system

```
  popl %ebx;
  popl %ecx;
  popl %edx;
  popl %esi;
  popl %edi;
  popl %ebp;
  popl %eax;
  popl %ds;
  popl %es;
  addl $4,%esp;
  iret;
```

As shown in figure 3 and the code fragments above the exceptions and system calls terminate same way as interrupts.

7. SMP SYSTEM

Symmetrical multiprocessing (SMP) system sets slightly different requirements to interrupt handling by hardware and software than ordinary uniprocessing (UP) system. Taking advantage of parallelism requires distributed handling of hardware interrupts, which then requires synchronization ie. only one CPU should handle certain hardware interrupt. Secondly some kind of efficient mechanism is needed to pass messages between CPUs. Latter is needed for scheduling tasks between CPUs and for different synchronization purposes.

7.1 APIC

To be able to fully distribute interrupt handling among CPUs in SMP system Intel has developed I/O APIC (*Advanced Programmable Interrupt Controller*) which replaces the old 8259A Programmable Interrupt Controller ([4]).
The sample SMP system with local APICs and I/O APIC is shown in figure 4. Local APIC has 32-bit registers, an internal clock, a timer device and two additional IRQ lines reserved for local interrupts. Local interrupts are typically used to reset the system.
The I/O APIC consists of a set of IRQ lines, a 24-entry Interrupt Redirection Table, programmable registers and a message unit for sending and receiving APIC messages over the ICC bus. Each entry in the Redirection Table can be individually programmed to indicate the interrupt vector and priority, the destination CPU, and how the CPU is selected. Table is used to translate any external IRQ to signal into a

message to one or more local APIC units via the ICC bus. Interrupt requests can be distributed to CPUs in two different ways: Fixed mode and Lowest-priority mode. [8]
Inportant feature of the APIC is that it allows CPUs to generate interprocessor interrupts. CPU can store the interrupt vector and the identifier of the target's local APIC in the Interrupt Command Register (ICR) of its own local APIC. A message is then sent via the ICC bus to the target's local APIC, which then issues a corresponding interrupt to its own CPU.
There is support for multiple external I/O APICs in kernel 2.4.18-10.

7.2 IPIs

Interprocessor interrupts (IPIs) are used to exchange messages between CPUs in SMP system. SMP kernel provides following functions to handle them: `send_IPI_all()`, `send_IPI_allbutself()`, `send_IPI_self()` and `send_IPI_single()`([9]). SMP kernel recognizes four different type of messages identified by interrupt vectors:

RESCHEDULE_VECTOR(0x30) Used at least when the `best_cpu` for the woken up task is not `this_cpu`. Handler: `smp_reschedule_interrupt()`.

INVALIDATE_TLB_VECTOR(0x31) Used when the TLBs of the other CPU need to be invalidated. Handler: `smp_invalidate_interrupt()`.

LOCAL_TIMER_VECTOR(0x41) Used for finer grained (better than traditional 100 Hz timer) kernel profiling and process statistics and rescheduling. Handler: `smp_apic_timer_interrupt()`, which handles pending softirqs also.

CALL_FUNCTION_VECTOR(0x50) Used to call functions with a given argument on other CPUs like `flush_tlb_all_ipi()` and `stop_this_cpu()`. Handler: `smp_call_function_interrupt()`.

8. CONCLUSIONS

Linux provide efficient mechanisms for interrupt and exception handling such as possibility to defer part of interrupt handling, optimized control paths and CPU cache utilization (alignments) and SMP support using local APICs and I/O APIC(s). It is also possible to share IRQs (capable to handle more I/O devices than there are IRQ lines) in Linux.

9. REFERENCES

[1] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel: From I/O ports to process management.* O'Reilly and Associates, Sebastopol, 2001.

[2] Intel. *8259A Programmable Interrupt Controller.* Intel Corporation, Dec 1988.

[3] Intel. *Pentium Pro Family Developers Manual, Volume 2: Programmer's Reference Manual.* Intel Corporation, 1995.

[4] Intel. *82093AA I/O Advanced Programmable Interrupt Controller (IOAPIC).* Intel Corporation, May 1996.

[5] Intel. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture.* Intel Corporation, 1997.

[6] Intel. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference.* Intel Corporation, 1997.

[7] Intel. *Intel Architecture Software Developer's Manual, Volume 3: System Programming.* Intel Corporation, 1997.

[8] Intel. *Multiprocessor Specification, Version 1.4.* Intel Corporation, May 1997.

[9] S. A. Maxwell. *Linux Core Kernel Commentary: In-Depth Code Annotation.* Coriolis, Arizona, 2001.