

Lecture 9: April 6

*Lecturer: Alessandro Pellegrini**Scribe: Anzhelo Xhebraj*

9.1 Interprocessor Interrupts

Interrupts generated at hardware level but processed at software level. The sending is a synchronous operation while the receiving is asynchronous. A portion of the kernel code is sending a message to another core.

There are 2 priorities: High and low. High will be processed immediately (but still asynchronous). Low are serialized, in a sort of FIFO order. IPIs are sent through the ICR register and there is a Destination field that lets you choose to which core to send the interrupt. The Linux kernel uses 251-255 entries in the IDT associated to IPIs.

What is related to interrupt requests it is because some hardware component is asking for the attention of the cpu. Traps is about dealing with something generated by software. `init_IRQ` setups the interrupts related to the devices on board on the machine. In order to do it it uses the ACPI to discover the devices in the system. With this function it reads from memory the acpi table and establish the irq and finalizes the initialization of the IDT. ACPI is a std used for describing the hardware. ACPI is composed of a set of vectors and tables and from the flags and codes it can understand what irqs needs to install in the idt.

Three main functions that manipulate some bits to setup an entry in the idt. `set_trap_gate` will initialize one idt entry setting 0 the privilege level. `system_gate` the privilege level is 3, the function that setups the entry for interrupt 80. `Set_intr` ensures interrupts are cleared before entering the handler.

9.2 Interrupt Management

Entries 0 to 19 are about hardware interrupts such as division by 0. These low level interrupts are managed by a dispatcher. This dispatch depending by the parameters will call the right handler and will pass the parameters on the stack. Similar to the way system calls are handled. When receiving the interrupt a handler is called (the same), will call a single dispatcher. Why the handler is needed? The dispatcher expects to find something in the stack which is related to the actual handler management. Depending on the type of interrupts some parameters may be missing. The handler identifies the handler that should be activated and pushes the pointer to the handler in the stack. The dispatcher saves the context of execution, it is only a way to keep the code smaller.

page fault handler doesn't have push 0 because the cpu already pushes some parameters in the stack. Some interrupts already push some parameters on the stack, some don't therefore the early handler pushes for them.

`error_code` takes a snapshot of the cpu similarly to software traps. The difference is that the info is organized on stack in the same way of the struct `pt_regs`. And then calls the actual handler with the pointer to the struct.

`do_page_fault` handler has the first param the pointer to `ptregs` and then an error code which is pushed by the firmware (is a bitmask in which only the three last bits are meaning-

ful) and tells what is the meaning of the fault. Description at pg 167. In the cpu context you can also inspect the instruction pointer which will tell which instruction was trying to access memory but not which page it was trying to access. There is one specific control register which is cr2 that is written by the firmware with the address of the page that triggered the page fault. This is related to User Space applications.

9.3 Kernel Exception Handling

What happens when kernel mode code tries to access a pointer passed by user space code. It uses verify area to check that are but there are some performance issues. Much of the user space code uses system calls and these functions are expensive. But we don't want to enter directly that memory region triggering a page fault. Kernel crashes. After crash it activates fixup that tries to restore the kernel to a working point. The kernel code has a label called `bad_area` that tries to find an instruction to restore the kernel. Bad area reads the eip and checks the error code to find if the exception was triggered in kernel mode. It uses this address to find an instruction to fix this state. How is the fixup address found?

Example: get user takes some data from userspace to kernel space.

Two non standard sections are defined in the executable: fixup and ex table. the former executable while the latter only readable (exception table). The compiler will put that code to the end of the executable. Once the exception is generated from the ip we look into the exception table finding that address and finds the next address and puts it into the struct ip returning to the dispatcher. The fixup address is just moving some negative value to eax and xor dl which tells the amount of data loaded by user spece and finally jumps to the next instruction.

What we saw was for 32bit but in 64 bit either use offset from the table or enlarge the table to 64bit.

9.4 IPIs

There is no way to describe a payload telling what has to be done by the CPU delivering the interrupt.

Kernel panics use IPIs to freeze all the processors. It doesn't have any payload. INVALIDATE TLB VECTOR is a vector in the IDT used to sync all cores to flush tlb entries. Is mapped to all except self. Call function vector is used to run some specific function that is passed by the sender. SMP call function. How are IPI messages generated? Through the IPI apis. The kernel uses fixed memory locations in which the sender can put some data and receivers look in order to understand what they have to do. The access to the location is managed through a spinlock.

`smp_call_function()`: takes three parameters: one function pointer one buffer and one flag. The flag tells whether it should be blocking or not. High or low priority. The function is the specific routine which the receiver of the interrupt should execute when receiving the ipi. Two global variables are used to pass the function and its parameters. First thing we do is check whether the interrupts are disabled. Then we take the spinlock that protects the data structure. Set the global variables. Set to atomic variables that tell how many cores have started to execute the routine and how many have finished.

`synchronize_all` used to synchronize all cores such that only one core is executing some stuff. still atomic counters used. It sets synch enter to the number of cpus that should execute the ipi and synch leave to the number of cores that received the message. Why preemption is

disabled? Linux is a preemptable kernel. And preemption is done through a timer interrupt. Preemption can be disabled through masking interrupts (but too heavyweight) or through preempt disable that just increments a counter. The scheduler checks whether the counter is greater than 0 and if so it just returns instead of preempting the kernel.

References

- [a20()] A20 line. URL https://wiki.osdev.org/A20_Line.
- [car()] Writing a bootloader from scratch. URL <https://www.cs.cmu.edu/~410-s07/p4/p4-boot.pdf>.
- [con()] Context switching. URL https://wiki.osdev.org/Context_Switching.
- [des(a)] Descriptor cache, a. URL https://wiki.osdev.org/Descriptor_Cache.
- [des(b)] Interrupt descriptor table, b. URL https://wiki.osdev.org/Interrupt_Descriptor_Table.
- [osd()] "8042" ps/2 controller. URL https://wiki.osdev.org/8042_PS2_Controller.
- [rma()] The workings of: x86-16/32 realmode addressing. URL https://web.archive.org/web/20130609073242/http://www.osdever.net/tutorials/rm_addressing.php?the_id=50.
- [ser()] Interrupt service routines. URL https://wiki.osdev.org/Interrupt_Service_Routines.
- [tas()] Task state segment. URL https://wiki.osdev.org/Task_State_Segment.
- [vec()] Interrupt vector table. URL https://wiki.osdev.org/Interrupt_Vector_Table.
- [wra()] Who needs the address wraparound, anyway? URL <http://www.os2museum.com/wp/who-needs-the-address-wraparound-anyway/>.
- [x86()] Why doesn't linux use the hardware context switch via the tss? URL <https://stackoverflow.com/questions/2711044/why-doesnt-linux-use-the-hardware-context-switch-via-the-tss>.
- [wik(2017)] Task state segment, Jun 2017. URL https://en.wikipedia.org/wiki/Task_state_segment.
- [wik(2018a)] x86 memory segmentation, Mar 2018a. URL https://en.wikipedia.org/wiki/X86_memory_segmentation#cite_note-Arch-1.
- [wik(2018b)] A20 line, Feb 2018b. URL https://en.wikipedia.org/wiki/A20_line.
- [Bovet and Cesati(2006)] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, 2006.
- [Brouwer()] Andries E. Brouwer. A20 - a pain from the past. URL <https://www.win.tue.nl/~aeb/linux/kbd/A20.html>.
- [Collins(a)] Robert Collins. A20 - reset anomalies, a. URL <http://www.rcollins.org/Productivity/A20Reset.html>.
- [Collins(b)] Robert Collins. The segment descriptor cache, b. URL <http://www.rcollins.org/ddj/Aug98/Aug98.html>.
- [Duarte(2008a)] Gustavo Duarte. Memory translation and segmentation, Aug 2008a. URL <https://manybutfinite.com/post/memory-translation-and-segmentation/>.
- [Duarte(2008b)] Gustavo Duarte. Cpu rings, privilege, and protection, Nov 2008b. URL <https://manybutfinite.com/post/cpu-rings-privilege-and-protection/>.

[Intel()] Intel. *Intel 64 and IA-32 Architecture Software Developer's Manual*, volume 3A.

[Oostenrijk(2016)] Alexander van Oostenrijk. Writing your own toy operating system: Jumping to protected mode, Sep 2016. URL <http://www.independent-software.com/writing-your-own-toy-operating-system-jumping-to-protected-mode/>.