Lecture 4: Interrupt

An operating system is a collection of service routines. The service routine can be executed by a request from an application (**system call interrupt**). Or it runs automatically when there is a serious error while running an application (**exception interrupt**) or when there is an external hardware event that the operating system has to handle (**hardware interrupt**). The service routines are called ISRs (Interrupt Service Routines).

|  |  |  |
| --- | --- | --- |
| external event | interrupt number | ISR (ISR1 => ISR2) |
| An application callsread(...) | int 128(syscall num 3) | system\_call() => sys\_read() |
| An application callswrite(...) | int 128(syscall num 4) | system\_call() => sys\_write() |
| timer ticks | int 32 | interrupt[0] => timer\_interrupt() |
| key stroke | int 33 | interrupt[1] => atkbd\_interrupt() |
| An application runsx=x/0; | int 0 | divide\_error() => do\_divide\_error() |
| page fault while an application run | int 14 | page\_fault() => do\_page\_fault() |

ISR1s are all located in arch/x86/kernel/entry\_32.S. ISR2s are located in various locations of the kernel.

When an interrupt, INT x, happens, the cpu stores the current cs, eip, flag register into stack and jumps to ISR1 for INT x. The ISR1 locations are written in IDT (Interrupt Descriptor Table), and the cpu jumpts to the location written in IDT[x]. ISR1 knows the location of ISR2. It knows the location of ISR2 because it is hard-coded (exception interrupt case), or is written in irq\_desc table (hardware interrupt case) or is written in syscall\_table (system call interrupt case).

1. Interrupt classification and Interrupt number

There are two kinds of interrupt: **software interrupt and hardware interrupt**. Software interrupt is generated by a program when it makes a serious error (e.g. dividing by zero) or runs a special instruction (e.g. INT, SYSENTER, SYSCALL). The former is called **exception** and the latter **system call**. **Hardware interrupt** is generated by peripheral devices connected to CPU. Key press is one example of hardware interrupt. All interrupts have a unique number defined by the operating system. Operating system is a collection of routines that handle interrupts.

**Hardware interrupts** have been assigned following interrupt numbers in Linux.

|  |  |  |
| --- | --- | --- |
| device | interrupt number | irq number |
| timer | 32 | 0 |
| keyboard | 33 | 1 |
| PIC cascading | 34 | 2 |
| second serial port | 35 | 3 |
| first serial port | 36 | 4 |
| floppy disk | 38 | 6 |
| system clock | 40 | 8 |
| network interface | 42 | 10 |
| usb port, sound card | 43 | 11 |
| ps/2 mouse | 44 | 12 |
| math coprocessor | 45 | 13 |
| eide disk, first chain | 46 | 14 |
| eide disk, second chain | 47 | 15 |

**Exceptions** have been assigned following interrupt numbers.

|  |  |
| --- | --- |
| exception | interrupt number |
| divide-by-zero error | 0 |
| debug | 1 |
| NMI | 2 |
| breakpoint | 3 |
| overflow | 4 |
| bounds check | 5 |
| invalid opcode | 6 |
| device not available | 7 |
| double fault | 8 |
| coprocessor segment overrun | 9 |
| invalid TSS | 10 |
| segment not present | 11 |
| stack segment fault | 12 |
| general protection | 13 |
| page fault | 14 |
| intel-reserved | 15 |
| floating point error | 16 |
| alignment check | 17 |
| machine check | 18 |
| simd floating point | 19 |

Finally, **system calls** in Linux are all assigned the same interrupt number, **128 (0x80).** To differentiate between different system calls, a unique system call number has been given to each system call. For the full table, look at **arch/x86/kernel/syscall\_table\_32.S**.

|  |  |  |
| --- | --- | --- |
| system calls | interrupt number | system call number |
| exit | 128 | 1 |
| fork | 128 | 2 |
| read | 128 | 3 |
| write | 128 | 4 |
| open | 128 | 5 |
| close | 128 | 6 |
| .............. | ............... | ..................... |

2. How interrupts are detected?

Interrupts are detected by CPU. Exceptions are detected when the corresponding error happens. System calls are detected when the program executes INT 128 instruction. Hardware interrupts are detected when the corresponding devices are affected. Hardware interrupts need more detailed explanation.



The above picture shows how hardware interrupts are detected by the CPU. All hardware devices are connected to 8259A interrupt controller through IRQ lines. Timer is connected through IRQ0 line, keyboard is connected through IRQ1 line, and so on. When an event happens in one of these devices, the corresponding IRQ line is activated, and 8259A signals CPU about this event along with the corresponding interrupt number for this IRQ line. The interrupt number is computed as (IRQ line number + 32) in Linux.

3. How interrupts are handled

Interrupts are first handled by the CPU, and then the operating system takes care of the rest of things.

3.1 cpu part

When an interrupt happens, the CPU executes the corresponding INT instruction. For example, if the user presses a key (which corresponds to INT 33), the CPU executes INT 33. Executing “INT x” instruction is two steps:

 - save current EFLAG, CS,EIP registers in the stack

 - jump to the location specified in IDT[x]

IDT(Interrupt Descriptor Table) is a table containing the address of ISR’s(Interrupt Service Routines). More general name for IDT is Interrupt Vector Table. If IDT[32] indicates address 0x10200, the ISR for timer interrupt is located at address 0x10200, which means whenever the timer ticks, the cpu jumps to address 0x10200. If IDT[33] indicates address 0x10300, the ISR for keyboard is located at 0x10300. Whenever the user hits some key, the cpu will jump to 0x10300 and start to execute whatever program stored there.

 It is the responsibility of the operating system to provide the IDT and fill in proper address for each interrupt. Linux writes IDT in **arch/x86/kernel/traps\_32.c/trap\_init()** (for exception interrupts and system call interrupt) and in **arch/x86/kernel/i8259\_32.c/init\_IRQ()** (for hardware interrupts). Also the cpu knows the location of IDT by its IDTR register. Therefore, it is again the responsibility of operating system to write the location of the IDT in IDTR register. Each entry in IDT is 8 byte. The variable name of IDT table in Linux is idt\_table.

3.2 OS part

Once the cpu jumps to the corresponding ISR, OS takes the control since ISR belongs to the operating system. All ISR’s (I call ISR1) consist of three steps:

 - save the rest of registers (eflag, cs, eip are already saved by cpu)

 - call actual interrupt handler (I call ISR2)

 - recover the saved registers and go back to the interrupted location

Linux writes ISR1's in IDT by calling set\_intr\_gate() for hardware interrupts in arch/x86/kernel/i8259\_32.c/native\_init\_IRQ(), and by calling set\_trap\_gate() for most of the exceptions and set\_system\_gate() for system call interrupts in arch/x86/kernel/traps\_32.c/trap\_init(). Device drivers write their ISR2's for hardware interrupts in irq\_desc[] table by calling request\_irq(). ISR2's for exceptions are directly called in the corresponding ISR1's, the name always being do\_(ISR1 name). ISR2's for the system calls are hard-coded in sys\_call\_table[] in arch/x86/kernel/syscall\_table\_32.S. ISR1s are defined in arch/x86/kernel/entry\_32.S, and ISR2s are defined in various places.



Interrupt numbers and their ISR1 and ISR2 list.

|  |  |  |
| --- | --- | --- |
| interrupt number | ISR1 | ISR2 |
| 0 | divide\_error | do\_divide\_error |
| 1 | debug | do\_debug |
| ............... |  |  |
| 32 | interrupt[0] | timer\_interrupt |
| 33 | interrupt[1] | atkbd\_interrupt |
| ................ |  |  |
| 128 (syscall num 1) | system\_call | sys\_exit |
| 128 (syscall num 2) | system\_call | sys\_fork |
| ....................... |  |  |

4. Homework

1) Following events will cause interrupts in the system. What interrupt number will be assigned to each event? For system call interrupt, also give the system call number.

 - A packet has arrived

 - An application program calls scanf()

 - A key is pressed

 - An application causes a divide-by-zero error

 - An application program calls printf()

 - An application causes a page-fault error

2) Change drivers/input/keyboard/atkbd.c as follows.

static irqreturn\_t atkbd\_interrupt(....){

 return IRQ\_HANDLED; // Add this at the first line

 .............

}

Recompile the kernel and reboot with it. What happens and why does this happen? Show the sequence of events that happen when you hit a key in a normal Linux kernel (as detail as possible): hit a key => keyboard controller sends a signal through IRQ line 1 => ......etc. Now with the changed Linux kernel show which step in this sequence has been modified and prevents the kernel to display the pressed key in the monitor.

3) Change the kernel such that it prints "x pressed" for each key pressing, where x is the scan code of the key. After you change the kernel and reboot it, do followings to see the effect of your changing.

 # echo 8 > /proc/sys/kernel/printk

/proc/sys/kernel/printk shows the console log level, default log level, min and max log level.

 # cat /proc/sys/kernel/printk

 1 4 1 7

The above means the console log level is 1, default printk log level is 4, and min conole log level is 1 and default console log level is 7. Lower log level means higher priority. Since default log level has lower priority than console log level, using printk() will not show the message on the console. We change the console log level to lowest level so that printk() will be able to display message on the console.

# echo 8 > /proc/sys/kernel/printk

Above will set console log level to 8 which means all printk() message will appear on the console from now on. (Note the files in /proc file system are not real files. They are generated dynamically when needed.)

4) Change the kernel such that it displays the next character in the keyboard scancode table. For example, when you type "root", the monitor would display "tppy". How can you log in as root with this kernel?

5) Define a function "mydelay" in init/main.c which whenever called will stop the booting process until you hit 's'. Call this function after do\_basic\_setup() function call in kernel\_init() in order to make the kernel stop and wait for 's' during the booting process. You need to modify atkbd.c such that it changes exit\_mydelay to 1 when the user presses 's'.

init/main.c

........

int exit\_mydelay; // define a global variable

void mydelay(char \*str){

 printk(str);

 printk("enter s to continue\n");

 exit\_mydelay=0; // init to zero

 for(;;){ // and wait here until the user press 's'

 msleep(1); // sleep 1 micro-second so that keyboard interrupt ISR

 // can do its job

 if (exit\_mydelay==1) break; // if the user press 's', break

 }

}

void kernel\_init(){

 ...............

 do\_basic\_setup();

 mydelay("after do basic setup in kernel\_init\n"); // wait here

 .........

}

drivers/input/keyboard/atkbd.c

.........

extern int exit\_mydelay; // declare as extern since it is defined in main.c

static irqreturn\_t atkbd\_interrupt(....){

 .............

 // detect 's' key pressed and change exit\_mydelay

 .............

}

5-1) Add mydelay before do\_basic\_setup(). What happens and why?

void kernel\_init(){

 ...............

 mydelay("before do basic setup in kernel\_init\n"); // wait here

 do\_basic\_setup();

 mydelay("after do basic setup in kernel\_init\n"); // wait here

 .........

}

6) Which function call in atkbd\_interrupt() actually displays the pressed key in the monitor?

6-1) What are the interrupt numbers for divide-by-zero exception, keyboard interrupt, and "read" system call? Where is ISR1 and ISR2 for each of them (write the exact code location)? Show their code, too.

7) sys\_call\_table[] is in arch/x86/kernel/syscall\_table\_32.S. How many system calls does Linux 2.6 support? What are the system call numbers for exit, fork, execve, wait4, read, write, and mkdir? Find system call numbers for sys\_ni\_syscall, which is defined at kernel/sys\_ni.c. What is the role of sys\_ni\_syscall?

8) Change the kernel such that it prints "length 17 string found" for each printf(s) when the length of s is 17. Run a program that contains a printf() statement to see the effect. printf(s) calls write(1, s, strlen(s)) system call which in turn runs

 mov eax, 4 ; eax<--4. 4 is system call number for “write”

 int 128

INT 128 will make the cpu stop running current process and jump to the location written in IDT[128]. IDT[128] contains the address of system\_call (located in arch/x86/kernel/entry\_32.S). Finally, system\_call will execute

 call \*sys\_call\_table(,%eax,4)

which eventually calls sys\_write() since eax=4 for write() system call (the target function location is sys\_call\_table+eax\*4).

\* Sometimes the the system call runs "sysenter" instead of "int 128". In this case the cpu jumps to ia32\_sysenter\_target (also in entry\_32.S) instead of system\_call.

9) You can call a system call indirectly with “syscall()”.

 write(1, “hi”, 2);

can be written as

 syscall(4, 1, “hi”, 2); // 4 is the system call number for “write” system call

Write a program that prints “hello” in the screen using syscall.

10) Create a new system call, my\_sys\_call with system call number 17 (system call number 17 is one that is not being used currently). Define my\_sys\_call() just before sys\_write() in read\_write.c. Write a program that uses this system call:

 void main(){

 syscall(17); // calls a system call with syscall number 17

 }

When the above program runs, the kernel should display

 hello from my\_sys\_call

To define a new system call with syscall number x

 - insert the new system call name in arch/x86/kernel/syscall\_table\_32.S

 at index x

 - define the function in appropriate file (such as "read\_write.c")

 asmlinkage void my\_sys\_call(){

 printk("hello from my\_sys\_call\n");

 }

 - recompile and reboot

To use this system call in a user program

 - void main(){

 syscall(x);

 }

10-1) Create another system call that will add two numbers given by the user.

ex0.c:

 void main(){

 int sum;

 sum = syscall(31, 4, 9); // suppose 31 is an empty entry in sys\_call\_table

 printf("sum is %d\n", sum);

 }

#./ex0

sum is 13

11) Modify the kernel such that it displays the system call number for all system calls. Run a simple program that displays "hello" in the screen and find out what system calls have been called. Also explain for each system call why that system call has been used.

12) What system calls are being called when you remove a file? Use "system()" function to run a Linux command as below. Explain what each system call is doing. You need to make f1 file before you run it. Also explain for each system call why that system call has been used.

 ...........

 system("rm f1");

 ...........

13) Find rm.c in busybox-1.31.1 and show the code that actually removes "f1". Note all linux commands are actually a program, and running "rm" command means running rm.c program. "rm" needs a system call defined in uClibc-0.9.33.2 to remove a file. You may want to continue the code tracing all the way up to "INT 0x80" in uClibc for this system call.