CS 471 - Operating Systems

CPU Mode

Qiang Zeng, Ph.D.
Layered design: User program-> libs -> core kernel -> Kernel I/O subsystem -> driver -> controller -> device
Modern CPUs

• Two common modes
  – Kernel mode
    • To execute the kernel code
  – User mode
    • To execute the user code
Important questions

• How are CPU modes implemented?
• Why are CPU modes needed?
• Difference between Kernel mode and User mode
• How are system calls implemented?
• Advanced topic: Virtualization
How CPU Modes are implemented

- Implemented through protection rings
  - A modern CPU typical provides different protection rings, which represent different privilege levels
    - A ring with a lower number has higher privileges
  - Introduced by Multics in 60’s
  - E.g., an X86 CPU usually provides four rings, and a Linux/Unix/Windows OS uses Ring 0 for the kernel mode and Ring 3 for the user mode
Why are Protection Rings needed?

- **Fault isolation**: a fault (e.g., divided by 0) in the code running in a less-privileged ring can be captured and handled by code in a more-privileged ring.

- **Privileged instructions**: certain instructions can only be issued in a privileged ring; thus, an OS can implement resource management and isolation here.

- **Privileged memory space**: certain memory can only be accessed in a privileged ring.

All these are demonstrated in the difference between the kernel mode and the user mode.
Kernel Mode vs. User Mode?

• A **fault** in the user space (e.g., *divided by zero, invalid access, null pointer dereference*) can be captured by the Kernel (without crashing the whole system)
  – Details of fault handling will be covered in later lectures

• **Privileged instructions** can only be issued in the kernel mode
  – E.g., disk I/O
  – In X86, an attempt to execute them from ring 3 leads to GP (General Protection) exceptions

• The **kernel memory space** can only be accessed in the kernel mode
  – E.g., the list of processes for scheduling
Examples of Privileged Instructions

• I/O operations
• Switch page tables of processes: `load cr3`
• Enable/disble interrupts: `sti/cli`
• Change processor modes from kernel to user: `iret`
• Halt a processor to enter low-power stage: `hlt`
• Load the *Global Descriptor Table* register in x86: `lgdt`

• Ref: [http://www.brokenthorn.com/Resources/OSDev23.html](http://www.brokenthorn.com/Resources/OSDev23.html)

• Examples of non-privileged ones:
  – add, sub, or, etc.
Questions

• If I/O operations rely on privileged instructions, how does a user program read/write?
  – System calls
  – When a system call is issued, the process goes from user mode (Ring 3) to kernel mode (Ring 0)
  – `printf libc call (Ring 3) => write system call => Kernel code (Ring 0)`
A CPU enters user mode and kernel mode in an interleaved way
How to interpret the output of the `time` command in Linux/Unix

$ time any-command

real  0m1.734s
user  0m0.017s
sys   0m0.040s

• Real: wall clock time
• User: CPU time spent in user-mode
• Sys: CPU time spent in kernel-mode
• Actual CPU time: user + sys
• Why “real != user + sys”? 
Myth: “root” refers to the kernel mode?

• Short answer: no!
• Long answer: the root user and non-root user refer to the user account types; in Linux/Unix, the root user can access any files, while a non-root user only has access to some files.
• Kernel Mode and User Mode refer to the processor mode
• No matter the user is a root or non-root, a CPU still enter Kernel mode and User mode in an interleaved way
• Regardless of the current CPU mode, a root user is always a root user
• That is, they are orthogonal concepts
Background

- **The Program Counter (PC) register** in a processor stores the address of the instruction to be executed.
  - PC is incremented after fetching an instruction.
  - But “jump”, “call” and “ret” instruction can set the PC value.

- If the user code can set the PC register *arbitrarily* before changing from Ring 3 to Ring 0, how will you exploit the kernel code?
  - This is very dangerous, as the user code can exploit the power of Ring 0 to harm the whole system.
How to implement systems calls if you are a kernel designer?
Linux system call overview

```
{ printf("hello world!\n"); }
```

 libc

User mode

kernel mode

```
%eax = sys_write;
int 0x80
```

IDT

0x80

```
system_call() {
    fn = syscalls[0eax]
}
```

sys_write(...) {
    // do real work
}

syscalls table

Graph by Dr. Junfeng Yang
## System Call Table

| Offset | Symbol               | sys_call_table       | System call location                                      |
|--------|----------------------|----------------------|----------------------------------------------------------|
| 0      | __NR_restart_syscall | .long sys_restart_syscall | -- ➔ /linux/kernel/signal.c                               |
| 4      | __NR-exit            | .long sys_exit       | -- ➔ /linux/kernel/exit.c                                 |
| 8      | __NR_exit            | .long sys_fork       | -- ➔ /linux/arch/386/kernel/process.c                     |
| 1272   | __NR_getcpu          | .long sys_getcpu     | -- ➔ /linux/kernel/sys.c                                  |
| 1276   | __NR_epoll_pwait     | .long sys_epoll_pwait| -- ➔ /linux/kernel/sys_ni.c                               |

__NR_syscalls

- .linux/include/asm/unistd.h
- .linux/arch/386/kernel/syscall_table.S
“INT 0x80” is used to issue system calls
“SYSENTER” has replaced “INT 0x80” nowadays
How to trace system calls in Linux/Unix

- “strace” command can trace system calls
- “ltrace” command can trace library calls
Rings and Virtualization

- **A Hypervisor is a Virtual Machine Monitor (VMM) that runs and manages virtual machines**
- **A straightforward virtualization scheme**
  - Hypervisor: Ring 0; VM Kernel: Ring 1; VM User: Ring 3
  - But there are instructions in X86 (sensitive but non-privileged) that cause problems when running in Ring 1; e.g., SGDT returns the host GDT info
  - The hypervisor is supposed to handle them
    - E.g., the hypervisor can maintain a virtual GDT for each VM and returns the VM’s GDT info when SGDT is invoked from a VM
Ring -1 used by the Hypervisor

- In 2005 and 2006, Intel and AMD introduced **Ring -1**, respectively; it is used by the **Hypervisor**
  - The VM kernel uses Ring 0, and the Hypervisor -1
  - The Hypervisor can configure with the CPU which instructions are of interest, so whenever they are executed, the execution traps from Ring 0 to -1
  - Hardwar-assisted full virtualization
Take-away

- CPU provides protection rings, while an OS uses them for the kernel mode and the user mode
  - Root user vs. kernel mode
- User code cannot do I/O directly, but do it through system calls
- The design of system calls is beautiful, because...
  - they allow your program to do something powerful; in the meanwhile you cannot abuse them (easily)
- Ring -1 is used by hypervisors
What else?

Three very useful Linux/Unix commands:

- time
- strace
- ltrace