Lecture 2: Interactions with Apps and Hardware

Amy Ousterhout
We will podcast the course

Project 0
- Due 4/11 11:59pm (next Tuesday), done individually

Homework #1
- Due 4/18

Project groups
- Can’t work across sections
- Using a Google form to collect group members (see Piazza)
- Just need one submission per group, fill out even if you are working alone

Lab hours
- Posted as a Google calendar
What is an Operating System?

- Code that sits between applications and hardware
- Provides abstractions to layers above
- Implements abstractions for and manages resources below
Hardware of a Typical Computer

- CPU
- CPU
- CPU
- Memory
- Memory
- System bus
- Keyboard
- Mouse
- Network
- Monitor
- Storage
Software of a Typical (Unix) System

- **Application**
  - Written by experts
  - Pre-compiled
  - Interfaces defined in headers
  - Invoked like functions
  - May be “resolved” when program is loaded

- **Libraries**
  - User code
  - Written/compiled by programmers
  - Uses library calls

- **Portable OS Layer**
  - “Guts” of system calls
  - All “high-level” code

- **Machine-dependent layer**
  - Bootstrap
  - System initialization
  - Interrupts and exceptions
  - I/O device driver
  - Memory management
  - Mode switching
  - Processor management

Note on terminology: “kernel” ≈ “OS”
Questions for Today

How do we separate the OS layer from apps (and libraries)?

How do we cross between these layers when necessary?

With support from the hardware!
Today’s Outline

• Protection: how can the OS perform special tasks and protect itself from applications?
  ♦ Privileged instructions
  ♦ Memory protection

• Interacting with the OS: when/how does the OS run?
  ♦ Faults
  ♦ System calls
  ♦ Interrupts
Dual-Mode Operation

- How can the OS perform special tasks (e.g., manage resources)?
- How can the OS protect itself from applications and protect applications from each other?
- OS needs to be “privileged”
- Every CPU core can run in one of two modes:
  - Kernel mode – can run all instructions
  - User mode – can only run non-privileged instructions
  - Mode is indicated by a mode bit in a protected CPU control register
Privileged Instructions

• Privileged instructions: a subset of instructions that can only run in kernel mode
  ♦ CPU checks mode bit when privileged instructions execute
  ♦ Attempts to execute in user mode are detected and prevented by the CPU

• Privileged instructions can:
  ♦ Directly access I/O devices (disk, network, etc.)
    » For security, fairness
  ♦ Manipulate memory-management state (page table pointers, etc.)
    » Prevent apps from accessing other apps’ memory (or the OS’s memory)
  ♦ Manipulate protected control registers (e.g., mode bit)
    » Prevent apps from giving themselves privileges!
Software of a Typical (Unix) System

- **User level**
  - Run in user mode
  - Cannot execute privileged instructions

- **Kernel level**
  - Run in kernel mode
  - Can execute privileged instructions

Diagram:
- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer
Example of a Privileged Instruction

- HLT: halts the CPU

### HLT—Halt

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compag/Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F4</td>
<td>HLT</td>
<td>Z0</td>
<td>Valid</td>
<td>Valid</td>
<td>Halt</td>
</tr>
</tbody>
</table>

### Instruction Operand Encoding

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z0</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

 Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SM1), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

**The HLT instruction is a privileged instruction.** When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.
(Live Demo of HLT)
Memory Protection

- OS must protect itself from user programs
- OS must be able to protect programs from each other
- May or may not protect user programs from the OS
  - Should programs trust the OS?
- Memory-management hardware provides memory protection
  - Page table pointers, page protection, segmentation, TLB
- Manipulating memory-management hardware uses privileged instructions
Today’s Outline

- Protection: how can the OS perform special tasks and protect itself from applications?
  - Privileged instructions
  - Memory protection

- Interacting with the OS: when/how does the OS run?
  - Faults
  - System calls
  - Interrupts
Events

- An event is an unnatural change in control flow
  - Immediately stop the current execution
  - Changes mode, context (machine state), or both
- The OS defines a handler for each event type
  - Event handlers execute in kernel mode
  - Specific types of events are defined by the machine
- After the system is booted, all entry to the kernel occurs as the result of an event
  - In effect, the operating system is one big event handler
  - OS only executes in reaction to events
Types of Events

- Two main types of events: exceptions and interrupts
- Interrupts – caused by an external event
  - Device finishes I/O, timer expires, etc.
  - Analogy: receiving a phone call or text message
- Exceptions – caused by program executing instructions
  - Executing a privileged instruction (fault)
  - Requesting services from the operating system (system calls)
- Events can be unexpected or deliberate
Hardware detects and reports exceptional conditions
  ♦ Divide by zero, page faults

Upon exception, hardware faults (verb)
  ♦ Must save state (PC, registers, mode, etc.) so that the faulting process can be restarted
  ♦ Each exception type has an associated number
  ♦ CPU finds the exception handler for that number
  ♦ Switch to kernel mode and start executing the exception handler

When done, operating system returns to program
  ♦ Reverses the steps above

Could we prevent faults with software?
Handling Faults (Recovery)

- Some faults are handled by “fixing” the exceptional condition
  - Page faults cause the OS to bring missing pages into memory
  - Fault handler returns to program and re-executes the instruction that cause the page fault
- Some faults are handled by notifying the process
  - Applications can register a fault handler with the OS
  - OS fault handler will return to the user-mode handler
  - Example: Unix signals such as SIGFPE, SIGTERM, SIGSEGV
Handling Faults (Termination)

• Kernel may handle unrecoverable faults by killing the user process
  ♦ E.g., program fault with no registered handler
  ♦ Halt process, write process state to a file, destroy process

• What about faults in the kernel?
  ♦ E.g., dereference NULL, divide by zero, undefined instruction
  ♦ These faults considered fatal, operating system crashes
  ♦ Unix panic, Windows “blue screen of death”
    » Kernel is halted, state dumped to a core file, machine locks up
For a user application to do something “privileged” it must call an OS procedure

System calls = operating system API
  - Interface between an application and the operating system kernel

System call categories
  - Process management
  - Memory management
  - File management
  - Device management
  - Communication
CPUs provide a system call instruction that:

- Causes an exception, which vectors to a kernel handler
- Passes a parameter determining the system routine to call (which system call)
- Saves caller state (PC, registers, mode) so it can be restored
- Returning from system call restores this state

Requires hardware support to:

- Restore saved state, reset mode, resume execution
Example of a System Call Instruction

- INT: executes a syscall

**INT n/INT0/INT3/INT1—Call to Interrupt Procedure**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Op/En</th>
<th>64-Bit Mode</th>
<th>Compat/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>INT3</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate breakpoint trap.</td>
</tr>
<tr>
<td>CD ib</td>
<td>INT imm8</td>
<td>I</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate software interrupt with vector specified by immediate byte.</td>
</tr>
<tr>
<td>CE</td>
<td>INTO</td>
<td>ZO</td>
<td>Invalid</td>
<td>Valid</td>
<td>Generate overflow trap if overflow flag is 1.</td>
</tr>
<tr>
<td>F1</td>
<td>INT1</td>
<td>ZO</td>
<td>Valid</td>
<td>Valid</td>
<td>Generate debug trap.</td>
</tr>
</tbody>
</table>

**Instruction Operand Encoding**

<table>
<thead>
<tr>
<th>Op/En</th>
<th>Operand 1</th>
<th>Operand 2</th>
<th>Operand 3</th>
<th>Operand 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZO</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>I</td>
<td>imm8</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

**Description**

The INT n instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled "Interrupts and Exceptions" in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT n instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The
System Call Example

User level

Kernel level

Application

Library

Operating System

read()

INT $0x03

trap to kernel

trap handler

read() kernel routine

Return to user level, resume execution

CSE 120 – Lecture 2 – Interactions with Apps and Hardware
Assigning Numbers to System Calls

---

**LINUX System Call Quick Reference**

**Jiaging He**

http://www.bugnet.com/~jiaging

### Introduction

System call is the services provided by Linux kernel. In C programming, it often uses functions defined in `lib` which provides a wrapper for many system calls. Manual page section 2 provides more information about system calls. To get an overview, use the `man 2 syscalls` command.

It is also possible to invoke `syscalls.h` function directly. Each system call has a function number defined in `<syscalls.h>`. Internally, system call is invoked by software interrupt 0x80 to transfer control to the kernel. System call table is defined in Linux kernel source file `arch/i386/kernel/entry.S`.

### System Call Example

```c
#include <syscall.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

int main(void) {
    long ID1, ID2;
    /******************************/
    /* direct system call */
    /* SVS_getpid (func no. 19) */
    ID1 = syscall(SVS_getpid);  // Get the process ID
    printf("The process ID is %ld
", ID1);
    /******************************/
    /* "lib" wrapped system call */
    /* SVS_getpid (func No. 15, 20) */
    ID2 = getpid();              // Get the process ID
    printf("The process ID is %ld
", ID2);
    return(0);
}
```

### System Call Quick Reference

<table>
<thead>
<tr>
<th>No.</th>
<th>Func Name</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td><code>creat</code></td>
<td>create a file or device (&quot;man 2 open&quot; for information)</td>
<td>files.c</td>
</tr>
<tr>
<td>9</td>
<td><code>link</code></td>
<td>make a new name for a file</td>
<td>files.c</td>
</tr>
<tr>
<td>10</td>
<td><code>umknlink</code></td>
<td>delete a name and possibly the file it refers to</td>
<td>files.c</td>
</tr>
<tr>
<td>11</td>
<td><code>execv</code></td>
<td>execute program</td>
<td>arch/i386/execute.c</td>
</tr>
<tr>
<td>12</td>
<td><code>chdir</code></td>
<td>change working directory</td>
<td>files.c</td>
</tr>
<tr>
<td>13</td>
<td><code>time</code></td>
<td>get time in seconds</td>
<td>files.c</td>
</tr>
<tr>
<td>14</td>
<td><code>mknod</code></td>
<td>create a special or ordinary file</td>
<td>files.c</td>
</tr>
<tr>
<td>15</td>
<td><code>chmod</code></td>
<td>change permissions of a file</td>
<td>files.c</td>
</tr>
<tr>
<td>16</td>
<td><code>lchown</code></td>
<td>change ownership of a file</td>
<td>files.c</td>
</tr>
<tr>
<td>17</td>
<td><code>stat</code></td>
<td>get file status</td>
<td>files.c</td>
</tr>
<tr>
<td>18</td>
<td><code>lseek</code></td>
<td>lseek file offset within file</td>
<td>files.c</td>
</tr>
<tr>
<td>19</td>
<td><code>fstat</code></td>
<td>get file status</td>
<td>files.c</td>
</tr>
<tr>
<td>20</td>
<td><code>mmap</code></td>
<td>map file to virtual memory</td>
<td>maps.c</td>
</tr>
<tr>
<td>21</td>
<td><code>munmap</code></td>
<td>unmap virtual memory</td>
<td>files.c</td>
</tr>
<tr>
<td>22</td>
<td><code>rename</code></td>
<td>rename file</td>
<td>files.c</td>
</tr>
<tr>
<td>23</td>
<td><code>setpgid</code></td>
<td>set process group ID</td>
<td>kernel.c</td>
</tr>
<tr>
<td>24</td>
<td><code>setuid</code></td>
<td>set file user ID</td>
<td>kernel.c</td>
</tr>
<tr>
<td>25</td>
<td><code>setgid</code></td>
<td>set file group ID</td>
<td>kernel.c</td>
</tr>
<tr>
<td>26</td>
<td><code>lstat</code></td>
<td>get file status</td>
<td>files.c</td>
</tr>
<tr>
<td>27</td>
<td><code>getopt</code></td>
<td>set system time and date</td>
<td>kernel.c</td>
</tr>
<tr>
<td>28</td>
<td><code>setuid</code></td>
<td>allow a parent process to control the execution of a child process</td>
<td>kernel.c</td>
</tr>
<tr>
<td>29</td>
<td><code>setgid</code></td>
<td>set an alarm clock for delivery of a signal</td>
<td>kernel.c</td>
</tr>
<tr>
<td>30</td>
<td><code>alarm</code></td>
<td>signal process</td>
<td>kernel.c</td>
</tr>
<tr>
<td>31</td>
<td><code>sleep</code></td>
<td>suspend process until signal</td>
<td>kernel.c</td>
</tr>
<tr>
<td>32</td>
<td><code>usleep</code></td>
<td>set file access and modification times</td>
<td>files.c</td>
</tr>
<tr>
<td>33</td>
<td><code>access</code></td>
<td>check user’s permissions for a file</td>
<td>files.c</td>
</tr>
<tr>
<td>34</td>
<td><code>nice</code></td>
<td>change process priority</td>
<td>kernel.c</td>
</tr>
<tr>
<td>35</td>
<td><code>kill</code></td>
<td>update the super block</td>
<td>kernel.c</td>
</tr>
<tr>
<td>36</td>
<td><code>wait</code></td>
<td>send signal to a process</td>
<td>kernel.c</td>
</tr>
<tr>
<td>37</td>
<td><code>rename</code></td>
<td>change the name or location of a file</td>
<td>files.c</td>
</tr>
<tr>
<td>38</td>
<td><code>mkdir</code></td>
<td>create a directory</td>
<td>files.c</td>
</tr>
<tr>
<td>39</td>
<td><code>rmdir</code></td>
<td>remove a directory</td>
<td>files.c</td>
</tr>
<tr>
<td>40</td>
<td><code>dup</code></td>
<td>create an open file descriptor</td>
<td>files.c</td>
</tr>
<tr>
<td>41</td>
<td><code>dup2</code></td>
<td>duplicate an open file descriptor</td>
<td>arch/i386/kernel/sigsys.c</td>
</tr>
<tr>
<td>42</td>
<td><code>pipe</code></td>
<td>create an interprocess channel</td>
<td>files.c</td>
</tr>
<tr>
<td>43</td>
<td><code>times</code></td>
<td>get process times</td>
<td>kernel.c</td>
</tr>
<tr>
<td>44</td>
<td><code>exit</code></td>
<td>exit from a process</td>
<td>kernel.c</td>
</tr>
<tr>
<td>45</td>
<td><code>exit</code></td>
<td>change the amount of space allocated for the calling process’s data segment</td>
<td>kernel.c</td>
</tr>
</tbody>
</table>
Referencing Data

• Processes and the OS are in different address spaces
  ♦ How can the OS return references to kernel data structures?

• Use names instead of pointers
  ♦ E.g., integer object handles or descriptors such as Unix file descriptors
Interrupts

- Interrupts signal external events
- Interrupts are generated by hardware
  - I/O hardware interrupts
  - Timers
- Interrupts on modern CPUs are precise
  - CPU transfers control only on instruction boundaries
Handling Interrupts

- Interrupt handler is in the kernel
- Steps:
  - Disable interrupts at lower priorities
  - Save state
  - Transfer control to the interrupt service routine (in the kernel)
  - When done, re-enable interrupts
  - Resume the user-level program at the next instruction
Example of an Interrupt: Timer

- The timer is critical for an operating system
- Fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
    - Setting the timer is a privileged instruction
  - Handled by the kernel, which decides what program to run next
    - Basis for the OS scheduler (more later…)
- Prevents infinite loops
  - OS can always regain control from erroneous or malicious programs trying to hog the CPU
- Also used for time-based functions (e.g., sleep)
Example of an Interrupt: I/O

- Asynchronous I/O
  - OS initiates I/O
  - Device (e.g., disk) operates independently of the rest of the machine
  - Device sends an interrupt signal to CPU when done
  - CPU context switches to the interrupt handler
  - Eventually resumes the original process
## x86 Interrupts and Exceptions (1)

<table>
<thead>
<tr>
<th>Vector #</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide error (by zero)</td>
<td>Fault</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Fault/trap</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Non-Maskable interrupt</td>
<td>Interrupt</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>Trap</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>Trap</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND range exceeded</td>
<td>Trap</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid opcode</td>
<td>Fault</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device not available</td>
<td>Fault</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double fault</td>
<td>Abort</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>Coprocessor segment overrun</td>
<td>Fault</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td></td>
</tr>
</tbody>
</table>
### x86 Interrupts and Exceptions (2)

<table>
<thead>
<tr>
<th>Vector #</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment not present</td>
<td>Fault</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack-segment fault</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General protection</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td></td>
<td>Fault</td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-point error (math fault)</td>
<td>Fault</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment check</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>19-31</td>
<td>Reserved</td>
<td></td>
<td>Abort</td>
</tr>
<tr>
<td>32-255</td>
<td>User defined</td>
<td></td>
<td>Interrupt</td>
</tr>
</tbody>
</table>
Once the system is booted, all entry to the kernel occurs due to traps and interrupts:

- Timer interrupts
- I/O interrupts
- Faults
- System calls
(Live Demo of Divide by Zero)
Summary

- **Protection:** how can the OS perform special tasks and protect itself from applications?
  - Privileged instructions
  - Memory protection

- **Interacting with the OS:** when/how does the OS run?
  - Faults
  - System calls
  - Interrupts
For next class…

- Read chapters 3-5
- Sign up for project teams (due 4/10)
- Work on PR 0 (due 4/11)
- Start looking at HW 1 (due 4/18)