Lecture 2: Interaction between Hardware, OS, and applications

Yiying Zhang
Administrivia

• Project 0 and Homework 1
  ♦ Both out, both due 10/7 11:59pm and done individually

• Lab hours start tomorrow
  • Schedule posted as Google calendar on course website

• Midterm
  ♦ Reminder to fill poll on Piazza!

• Project groups
  ♦ We will use a Google form to collect group members
  ♦ Just need one submission per group
  ♦ Fill out even if you are working alone

• Zoom meetings: we need to prevent Zoom bombin
  ♦ Do not share meeting links outside the class!
A Typical Computer from a Hardware Point of View

CPU

Chipset

Memory

I/O bus

Network

CPU
A Typical Computer System: adding software

- CPU
- Memory
  - Programs and data
  - Operating System Software
- OS
- Apps
- Data
- Network
Typical OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer
Typical Unix OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Written by programmer
Compiled by programmer
Uses library calls
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

Written by gurus
Provided pre-compiled
interface defined in
headers
Invoked like functions
Input to linker (compiler)
May be “resolved” when
program is loaded
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

“Guts” of system calls
All “high-level” code
Typical Unix OS Structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Bootstrap
System initialization
I/O device driver
Kernel/user mode switching
Interrupt and exception
Processor management

(OS ~= kernel)
Another Look: UNIX “Onion”

- Applications
- OS Service
- Device
- Hardware
- Driver

User and Kernel boundary
Roadmap

• System calls

• Interrupt
[lec1] What is an OS?

- Resource manager
  - Manage shared resources (CPU, mem, I/O, networking)

- Extended (abstract) machine
Dual-Mode Operation

- OS manages shared resources
- OS protects programs from other programs
- OS needs to be “privileged”

Every CPU (a CPU core actually) can run in one of the two modes:
- Kernel mode – can run all instructions
- User mode – can only run non-privileged instructions
- Mode is indicated by a status bit in a protected CPU control register
Privileged Instructions

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

• Privileged instructions include
  ♦ Directly access I/O devices (disks, printers, etc.)
    » For security, fairness
  ♦ Manipulate memory management state
    » Page table pointers, page protection, TLB management, etc.
  ♦ Manipulate protected control registers
    » E.g., mode bit, interrupt level
  ♦ Halt instruction
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

User level
(run in user mode)

Kernel level
(run in kernel mode)
Dual-Mode Operation

- OS manages shared resources
- OS protects programs from other programs
  ➔ OS needs to be privileged

- If OS manages shared resources, how does a user program request for accessing shared resources (e.g. hard drive)?
System calls

- Interface between a *process* and the operating system kernel
  - Kernel manages shared resources & exports interface
  - Process requests for access to shared resources

- Generally available as assembly-language instructions

- Directly invoked in many languages (C, C++, Perl)
  - Who is helping out here?
Typical Unix OS Structure

- **Application**
- **Libraries**
  - Written by gurus
  - Provided pre-compiled
  - Interface defined in headers
  - Invoked like functions
  - Input to linker (compiler)
  - May be “resolved” when program is loaded
- **Portable OS Layer**
- **Machine-dependent layer**
Example

• Given the I/O instructions are privileged, how does the user program perform I/O?
  ♦ open()
  ♦ read()
  ♦ write()
  ♦ close()
System calls

• Categories
  ♦ Process management
  ♦ Memory management
  ♦ File management
  ♦ Device management
  ♦ Networking
System calls in Linux (man syscalls)

- SYSCALLS(2) Linux Programmer’s Manual SYSCALLS(2)

- NAME
  - none - list of all system calls

- SYNOPSIS
  - Linux 2.4 system calls.

- DESCRIPTION
  - The system call is the fundamental interface between an application and the Linux kernel. As of Linux 2.4.17, there are 1100 system calls listed in /usr/src/linux/include/asm-*/unistd.h. This man page lists those that are common to most platforms (providing hyperlinks if you read this with a browser).

  _llseek(2), _newselect(2), _sysctl(2), accept(2), access(2), acct(2), adjtimex(2), afs_syscall, alarm(2), bdflush(2), bind(2), break, brk(2), cacheflush(2), capget(2), capset(2), chdir(2), chmod(2), chown(2), chown32, chroot(2), clone(2), close(2), connect(2), creat(2), create_module(2), delete_module(2), dup(2), dup2(2), execve(2), exit(2), fchdir(2), fchmod(2), fchown(2), fchown32, fcntl(2), fcntl64, fdata-
DESCRIPTION

syscall() performs the system call whose assembly language interface has the specified number
with the specified arguments. Symbolic constants for system calls can be found in the header file
<sys/syscall.h>.

RETURN VALUE

The return value is defined by the system call being invoked. In general, a 0 return value
indicates success. A -1 return value indicates an error, and an error code is stored in errno.

EXAMPLE

#define _GNU_SOURCE /* or _BSD_SOURCE or _SVID_SOURCE */
#include <unistd.h>
#include <sys/syscall.h> /* For SYS_xxx definitions */
#include <sys/types.h>

int main(int argc, char *argv[])
{
    pid_t tid;
    tid = syscall(SYS_gettid);
}
Transition from user to kernel mode (simplified)
<table>
<thead>
<tr>
<th>OS @ run (kernel mode)</th>
<th>Hardware</th>
<th>Program (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create entry for process list</td>
<td>restore regs (from kernel stack)</td>
<td>Run main()</td>
</tr>
<tr>
<td>Allocate memory for program</td>
<td>move to user mode</td>
<td>...</td>
</tr>
<tr>
<td>Load program into memory</td>
<td>jump to main</td>
<td>Call system call</td>
</tr>
<tr>
<td>Setup user stack with argv</td>
<td>trap into OS</td>
<td>trap</td>
</tr>
<tr>
<td>Fill kernel stack with reg/PC</td>
<td></td>
<td>(via exit())</td>
</tr>
<tr>
<td>return-from-trap</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Handle trap
Do work of syscall
return-from-trap

Free memory of process
Remove from process list

---

Figure 6.2: Limited Direct Execution Protocol
System Calls

- CPU ISA provides a system call instruction that:
  - Causes a trap to kernel
  - Passes a syscall # to determine which syscall handler to invoke
  - Saves caller state (PC, regs, mode) so it can be restored
  - Returning from system call restores this state

- Requires architectural support to:
  - Restore saved state, reset mode, resume execution

What would happen if the kernel did not save state?
System Call

Firefox: read()

Kernel mode

Trap to kernel mode, save state

Trap handler

Find the handler for read() in syscall table

read() kernel routine

User mode

Restore state, return to user level, resume execution
Roadmap

• System calls

• Interrupt
A Typical Computer from a Hardware Point of View

CPU

Chipset

Memory

I/O bus

Network
Concurrency & Unexpected Events

• How do human handle unexpected events (e.g., has a mail)?
  ♦ Go check the mailbox myself
  ♦ I have a secretary who receives mails in person for me and inform me when there is one
  ♦ Which one is more efficient?
    » If I have one mail per day?
    » If I have lots of mail per delivery?

• Poll vs. interrupt
  ♦ Usually one interrupt is more costly than one poll
  ♦ Which is better?
Interrupt

- A mechanism for
  - coordination between concurrently operating units of a computer system (e.g. CPU and I/O devices)
  - for responding to specific conditions within a computer

- Results in transfer of flow of control (to interrupt handler in the OS), forced by hardware
Two types of Interrupts

- **Hardware interrupts**
  - Timer expires
  - I/O device events: keyboard strokes, receiving a network packet, etc.

- **Software interrupts** (aka. *trap, exception*)
  - an error (floating point exception)
  - System calls can also be viewed as software interrupts, in a way

- The kernel defines a handler for each interrupt type
  - Interrupt handlers always execute in kernel mode
  - The specific types of interrupts are defined by the hardware
Handling interrupts

- Incoming interrupts are disabled (at this and lower priority levels) while the interrupt is being processed to prevent a lost interrupt.
- Interrupt architecture must save the address of the interrupted instruction.
- Interrupt transfers control to the interrupt service routine:
  - Generally, through the interrupt vector, which contains the addresses of all the service routines.
- If interrupt routine modifies process state (register values):
  - Save the current state of the CPU (registers and the program counter) on the system stack.
  - Restore the state before returning.
- Interrupts are re-enabled after servicing current interrupt.
- Resume the interrupted instruction.
## X86 Interrupt and Exceptions

<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 Interrupt 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></td>
</tr>
<tr>
<td>32-255</td>
<td>User defined</td>
<td></td>
<td>Interrupt</td>
</tr>
</tbody>
</table>

Vector 128 is for system calls
Interrupt time line for a single process doing I/O

Diagram showing the timeline of CPU and I/O device activities during an I/O operation. The timeline includes:
- CPU: user process executing
- I/O interrupt processing
- I/O device: idle, transferring

Key events:
- I/O request
- Transfer done
- I/O request
- Transfer done
I/O Completion

• Interrupts are the basis for asynchronous I/O
  ♦ OS initiates I/O
  ♦ Device operates independently of rest of machine
  ♦ Device sends an interrupt signal to CPU when done
  ♦ OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  ♦ CPU looks up kernel address indexed by interrupt number, context switches to routine
I/O Example

1. NIC receives packet, writes packet into memory
2. NIC signals a hardware interrupt
3. CPU stops current operation, switches to the kernel mode, saves machine state on the kernel stack
4. CPU reads address from interrupt table indexed by interrupt number, jumps to the address of the interrupt handle (in the NIC driver)
5. NIC device driver processes the packet
6. Upon completion, CPU restores saved state from stack and returns to user mode

Are there any other ways to perform I/O?
Timer

- The timer is critical for an operating system
- It is the fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
  - When timer expires, generates a hardware interrupt
  - Handled by kernel, which controls what runs next
    - Basis for OS scheduler (more later…)
- Prevents infinite loops
  - OS can always regain control from erroneous or malicious programs that try to hog CPU
- Also used for time-based functions (e.g., \texttt{sleep})
<table>
<thead>
<tr>
<th>OS @ boot</th>
<th>Hardware</th>
<th>Program (user mode)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(kernel mode)</td>
<td>initialize trap table</td>
<td>remember addresses of...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>start interrupt timer</td>
<td>syscall handler</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>timer handler</td>
<td></td>
</tr>
<tr>
<td>OS @ run</td>
<td></td>
<td>start timer</td>
<td></td>
</tr>
<tr>
<td>(kernel mode)</td>
<td></td>
<td>interrupt CPU in X ms</td>
<td></td>
</tr>
</tbody>
</table>

**Handle the trap**

Call `switch()` routine
- save `regs(A) → proc.t(A)`
- restore `regs(B) ← proc.t(B)`
- switch to `k-stack(B)`

**Return-from-trap (into B)**
- restore `regs(B) ← k-stack(B)`
- move to `user mode`
- jump to `B’s PC`

**Process A**

...  

**Process B**

...  

Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)
What is an OS?

- Resource manager
- Extended (abstract) machine
- (will have a 3rd def based on pragmatics next time)
Modern OSes are interrupt driven

• “An OS is a giant interrupt handler!” (Def 3)
• Once the system is booted, all entry to the kernel occurs as the result of an interrupt
  ♦ Timer interrupt → Context switches in multiprogramming
  ♦ (unexpected) I/O interrupts
  ♦ System calls to switch from user to kernel mode

• At the lowest level an OS is just a bunch of interrupt service routines
  ♦ Each routine simply returns to whatever was executing before it was interrupted
    » A user process, an OS process, another interrupt routine
  ♦ Else infinite wait loop
  ♦ There are, however, some exception: OS background threads
**Interrupt Questions**

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the interrupt handler itself be interrupted? (Consider why there might be different IRQ levels)
  - Can we and shall we disable interrupts?
- Interrupts are used by devices to have the OS do stuff
  - What is an alternative approach to using interrupts?
  - What are the drawbacks of that approach?
Questions?

• We will dive in process management next week

• Read Chapters 4-5