Today’s lecture

- An overview of HW functionality
  - read the cs323 textbook

- How to bootstrap?

- An overview of OS structures
  - OS components and services
  - how OS interacts with IO devices? interrupts
  - how OS interacts with application program? system calls
What makes a “computer system” ?

- **Hardware**
  - motherboard (cpu, buses, I/O controllers, memory controller, timer); memory; hard disk & flash drives, CD&DVDROM; keyboard, mouse; monitor & graphics card; printer, scanner, sound board & speakers; modem, networking card; case, power supply.
  - all connected through buses, cables, and wires

- **Software**
  - *a bunch of 0/1s; stored on a hard disk or a usb drive or a DVD*
    - * operating system (e.g., Linux, Windows, Mac OS)*
    - * application programs (e.g., gcc, vi)*

- **User** (it is “you”)

How a “computer” becomes alive?

**Step 0:** connect all HWs together, build the computer

**Step 1:** power-on and bootstrap
  - assuming that OS is stored on the boot drive (e.g., USB drive, hard disk, or CDROM)

**Step 2:** OS takes over and set up all of its services

**Step 3:** start the window manager and the login prompt

**Step 4:** user logs in; start the shell; run applications
Computer-system architecture (1980)

```
---
```

Computer-system architecture (Intel Skylake 2015)

```
---
```

**Computer-system architecture (Intel Skylake 2015)**


**Computer-system architecture (Raspberry Pi3)**

![Raspberry Pi 3 Model B](http://www.rleman.ru/i/Image/2016/02/29/RaspberryPi_3_B.jpg)
An overview of HW functionality

- **Executing the machine code** (cpu, cache, memory)
  - instructions for ALU-, branch-, and memory-operations
  - instructions for communicating with I/O devices

- **Performing I/Os**
  - I/O devices and the CPU can execute concurrently
  - Each device controller in charge of one device type
  - Each device controller has a local buffer
  - CPU moves data btw. main memory and local buffers
  - I/O is from the device to local buffer of controller
  - Device controller uses **interrupt** to inform CPU that it is done

- **Protection hardware**
  - timer, paging HW (e.g. TLB), mode bit (e.g., kernel/user)

Today's lecture

- **An overview of HW functionality**
  - read the cs323 textbook

- **How to bootstrap?**

- **An overview of OS structures**
  - OS components and services
  - how OS interacts with IO devices? **interrupts**
  - how OS interacts with application program? **system calls**
How to bootstrap?

- Power up a computer
- Processor reset
  - Set to known state
  - Jump to ROM code (for x86 PC, this is BIOS)
- Load in the boot loader from stable storage
- Jump to the boot loader
- Load the rest of the operating system
- Initialize and run

System boot

- Power on (processor waits until Power Good Signal)

- On an Intel PC, processor jumps to address FFFF0h (maps to FFFFFF0h = \(2^{32}-16\))
  - \(1M = 1,048,576 = 2^{20} = \text{FFFF}h+1\)
  - \(\text{FFFF}h=\text{FFFF}0h+15\) is the end of the (first 1MB of) system memory
  - The original PC using Intel 8088 (in 1970's) had 20-bit address lines :-)

- (FFFFFFFFFF0h) is a JMP instruction to the BIOS startup program
BIOS startup (1)

- POST (Power-On Self-Test)
  - If pass then AX:=0; DH:=5 (Pentium);
  - Stop booting if fatal errors, and report
- Look for video card and execute built-in BIOS code (normally at C000h)
- Look for other devices ROM BIOS code
  - IDE/ATA disk ROM BIOS at C8000h (=819,200d)
  - SCSI disks may provide their own BIOS
- Display startup screen
  - BIOS information
- Execute more tests
  - memory
  - system inventory

BIOS startup (2)

- Look for logical devices
  - Label them
    * Serial ports: COM 1, 2, 3, 4
    * Parallel ports: LPT 1, 2, 3
  - Assign each an I/O address and IRQ
- Detect and configure PnP devices
- Display configuration information on screen
BIOS startup (3)

- Search for a drive to BOOT from
  - Hard disk or USB drive or CD/DVD
  - Boot at cylinder 0, head 0, sector 1
- Load code in boot sector
- Execute boot loader
- Boot loader loads program to be booted
  - If no OS: "Non-system disk or disk error - Replace and press any key when ready"
- Transfer control to loaded program
  - Which maybe another feature-rich bootloader (e.g., GRUB), which then loads the actual OS

Today’s lecture

- An overview of HW functionality
  - read the cs323 textbook

- How to bootstrap?

- An overview of OS structures
  - OS components and services
    - how OS interacts with IO devices? interrupts
    - how OS interacts with application program? system calls
Typical Unix OS structure

Application
Libraries
Portable OS Layer
Machine-dependent layer

User level
Kernel level

User function calls written by programmers and compiled by programmers.
**Typical Unix OS structure**

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

  - Objects pre-compiled
  - Defined in headers
  - Input to linker
  - Invoked like functions
  - May be "resolved" when program is loaded

**Pipeline of creating an executable file**

```
gcc foo.c -> foo.s -> as foo.o
bar.c -> gcc bar.s -> as bar.o
ld foo.o bar.o libc.a -> a.out
```

- gcc can compile, assemble, and link together
- Compiler part of gcc compiles a program into assembly
- Assembler compiles assembly code into relocatable object file
- Linker links object files into an executable
- For more information:
  - Read man page of a.out, ld, and nm
  - Read the document of ELF
Execution (run an application)

- On Unix, “loader” does the job
  - Read an executable file
  - Layout the code, data, heap and stack
  - Dynamically link to shared libraries
  - Prepare for the OS kernel to run the application

![Application execution diagram](image)

What’s an application?

- Four segments
  - Code/Text - instructions
  - Data - initialized global variables
  - Stack
  - Heap

- Why?
  - Separate code and data
  - Stack and heap go towards each other

![Application segments diagram](image)
Responsibilities

- **Stack**
  - Layout by compiler
  - Allocate/deallocate by process creation (fork) and termination
  - Local variables are relative to stack pointer
- **Heap**
  - Linker and loader say the starting address
  - Allocate/deallocate by library calls such as malloc() and free()
  - Application program use the library calls to manage
- **Global data/code**
  - Compiler allocates statically
  - Compiler emits names and symbolic references
  - Linker translates references and relocates addresses
  - Loader finally lays them out in memory

Typical Unix OS structure
OS service examples

- **Examples that are not provided at user level**
  - System calls: file open, close, read and write
  - Control the CPU so that users won’t stuck by running
    * while (1);
  - Protection:
    * Keep user programs from crashing OS
    * Keep user programs from crashing each other

- **Examples that can be provided at user level**
  - Read time of the day
  - Protected user level stuff

Typical Unix OS structure

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

- System initialization
- Interrupt and exception
- I/O device driver
- Memory management
- Mode switching
- Processor management
OS components

- Resource manager for each HW resource
  - processor management (CPU)
  - memory management
  - file system and secondary-storage management
  - I/O device management (keyboards, mouse, ...)
- Additional services:
  - networking
  - window manager (GUI)
  - command-line interpreters (e.g., shell)
  - resource allocation and accounting
  - protection
    * Keep user programs from crashing OS
    * Keep user programs from crashing each other

Processor management

- Goals
  - Overlap between I/O and computation
  - Time sharing
  - Multiple CPU allocations
- Issues
  - Do not waste CPU resources
  - Synchronization and mutual exclusion
  - Fairness and deadlock free
Memory management

- **Goals**
  - Support programs to run
  - Allocation and management
  - Transfers from and to secondary storage

- **Issues**
  - Efficiency & convenience
  - Fairness
  - Protection

| Register: 1x | L1 cache: 2-4x | L2 cache: ~10x | L3 cache: ~50x | DRAM: ~200-500x | Disks: ~30M x | Disks: >1000M x |

I/O device management

- **Goals**
  - Interactions between devices and applications
  - Ability to plug in new devices

- **Issues**
  - Efficiency
  - Fairness
  - Protection and sharing

<table>
<thead>
<tr>
<th>User 1</th>
<th>...</th>
<th>User n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library support</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Driver</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device</td>
<td>...</td>
</tr>
</tbody>
</table>
File system

- A typical file system
  - open a file with authentication
  - read/write data in files
  - close a file

- Efficiency and security

- Can the services be moved to user level?

Today's lecture

- An overview of HW functionality
  - read the cs323 textbook

- How to bootstrap?

- An overview of OS structures
  - OS components and services
  - how OS interacts with IO devices? interrupts
  - how OS interacts with application program? system calls
Device interrupts

How does an OS kernel communicate with physical devices?

- Devices operate *asynchronously* from the CPU
  - Polling: Kernel waits until I/O is done
  - Interrupts: Kernel can do other work in the meantime

- Device access to memory
  - Programmed I/O: CPU reads and writes to device
  - Direct memory access (DMA) by device

- How do device interrupts work?
  - Where does the CPU run after an interrupt?
  - What is the interrupt handler written in?
  - What stack does it use?
  - Is the work the CPU had been doing before the interrupt lost?
  - If not, how does the CPU know how to resume that work

Challenge: protection

- How do we execute code with restricted privileges?
  - Either because the code is buggy or if it might be malicious

- Some examples:
  - A user program running on top of an OS
  - A third party device driver running within an OS
  - A script running in a web browser
  - A program you just downloaded off the Internet
  - A program you just wrote that you haven’t tested yet
Main points

- **Process concept**
  - A process is the OS abstraction for executing a program with limited privileges

- **Dual-mode operation: user vs. kernel**
  - Kernel-mode: execute with complete privileges
  - User-mode: execute with fewer privileges

- **Safe control transfer**
  - How do we switch from one mode to the other?
Process abstraction

- **Process**: an *instance* of a program, running with limited rights
  - **Thread**: a sequence of instructions within a process
    - Potentially many threads per process
  - **Address space**: set of rights of a process
    - Memory that the process can access
    - Other permissions the process has (e.g., which system calls it can make, what files it can access)

Thought experiment

- **How can we implement execution with limited privilege?**
  - Execute each program instruction in a simulator
  - If the instruction is permitted, do the instruction
  - Otherwise, stop the process
  - Basic model in Javascript and other interpreted languages

- **How do we go faster?**
  - Run the unprivileged code directly on the CPU!
Hardware support: dual-mode operation

- **Kernel mode**
  - Execution with the full privileges of the hardware
  - Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet

- **User mode**
  - Limited privileges
  - Only those granted by the operating system kernel

- **On the x86, mode stored in EFLAGS register**
- **On the MIPS, mode in the status register**

A model of a CPU
A CPU with dual-mode operation

Hardware support: dual-mode operation

- **Privileged instructions**
  - Available to kernel
  - Not available to user code

- **Limits on memory accesses**
  - To prevent user code from overwriting the kernel

- **Timer**
  - To regain control from a user program in a loop

- **Safe way to switch from user mode to kernel mode, and vice versa**
Privileged instruction examples

- Memory address mapping
- Cache flush or invalidation
- Invalidating TLB entries
- Loading and reading system registers
- Changing processor modes from kernel to user
- Changing the voltage and frequency of processor
- Halting a processor
- I/O operations

What should happen if a user program attempts to execute a privileged instruction?

Virtual addresses

- Translation done in hardware, using a table
- Table set up by operating system kernel

<table>
<thead>
<tr>
<th>Virtual Addresses (Process Layout)</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Code</td>
</tr>
<tr>
<td>Data</td>
<td>Data</td>
</tr>
<tr>
<td>Heap</td>
<td>Heap</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack</td>
</tr>
</tbody>
</table>
Hardware timer

- Hardware device that periodically interrupts the processor
  - Returns control to the kernel handler
  - Interrupt frequency set by the kernel
    * Not by user code!
  - Interrupts can be temporarily deferred
    * Not by user code!
    * Interrupt deferral crucial for implementing mutual exclusion

“User ⇔ Kernel” model switch

An interrupt or exception or system call (INT)

User mode
- Regular instructions
- Access user-mode memory

Kernel (privileged) mode
- All instructions
- Access all memory

A special instruction (IRET)
Mode switch

- From user mode to kernel mode
  - System calls (aka protected procedure call)
    * Request by program for kernel to do some operation on its behalf
    * Only limited # of very carefully coded entry points
  - Interrupts
    * Triggered by timer and I/O devices
  - Exceptions
    * Triggered by unexpected program behavior
    * Or malicious behavior!

System calls

- User code can be arbitrary
- User code cannot modify kernel memory
- Makes a system call with parameters
- The call mechanism switches code to kernel mode
- Execute system call
- Return with results

They are like "local" remote procedure calls (RPCs)
Interrupts and exceptions

- **Interrupt sources**
  - Hardware (by external devices)
  - Software: INT n

- **Exceptions**
  - Program error: faults, traps, and aborts
  - Software generated: INT 3
  - Machine-check exceptions

- See Intel document volume 3 for details

### Interrupt 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>NMI 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>
### 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></td>
</tr>
</tbody>
</table>

### How to take interrupt & syscall safely?

- **Interrupt & trap & syscall vector**
  - Limited number of entry points into kernel

- **Atomic transfer of control**
  - Single instruction to change:
    * Program counter
    * Stack pointer
    * Memory protection
    * Kernel/user mode

- **Transparent restartable execution**
  - For HW interrupts: user program does not know interrupt occurred
  - For system calls: it is just like return from a function call
Interrupt & trap & syscall vector

- Table set up by OS kernel; pointers to code to run on different events

```
Processor Register Interrupt Vector Table

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

.............. handleTimerInterrupt() {

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

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

.............. handleDivideByZero() {

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

.............. handleSystemCall() {

.............. ...
```

Also see: [http://shell-storm.org/shellcode/files/syscalls.html](http://shell-storm.org/shellcode/files/syscalls.html)
Interrupt stack

Per-processor, located in kernel memory. Why can't the interrupt handler run on the stack of the interrupted user process?

User Stack
- Running: ...
- Ready to Run: ...
- Waiting for I/O: Syscall

Kernel Stack
- User CPU State

Interrupt handler & interrupt masking

- Interrupt handler often non-blocking (with interrupts off), run to completion (then re-enable interrupts)
  - Minimum necessary to allow device to take next interrupt
  - Any waiting must be limited duration
  - Wake up other threads to do any real work
    * Linux: semaphore

- Rest of device driver runs as a kernel thread

- Interrupt masking: OS kernel can also turn interrupts off
  - Eg., when determining the next process/thread to run
  - On x86
    * CLI: disable interrupts
    * STI: enable interrupts
    * Only applies to the current CPU (on a multicore)
Case study: x86 interrupt & syscall

- Save current stack pointer
- Save current program counter
- Save current processor status word (condition codes)
- Switch to kernel stack; put SP, PC, PSW on stack
- Switch to kernel mode
- Vector through interrupt table
- Interrupt handler saves registers it might clobber

Before interrupt

User-level Process

```plaintext
foo () {
  while(...) {
    x = x+1;
    y = y-2;
  }
}
```

User Stack

```plaintext
User Stack
```

Registers

```
| SS: ESP |
| CS: EIP |
| EFLAGS |
| Other Registers: EAX, EBX, ... |
```

Kernel

```plaintext
handler () {
  pushad ...
}
```

Interrupt Stack

```
Interrupt Stack
```
During interrupt

User-level Process

foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

User Stack

other registers:
EAX, EBX,
...

EFLAGS
CS: EIP
SS: ESP

Kernel

handler() {
pushad
...
}

Interrupt Stack

Error
EIP
CS
EFLAGS
SS
ESP
...

After interrupt

User-level Process

foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

Stack

other registers:
EAX, EBX,
...

EFLAGS
CS: EIP
SS: ESP

Kernel

handler() {
pushad
...
}

Interrupt Stack

All Registers
EBX
EAX
ESP
SS
Error
EIP
CS
EFLAGS
SS
ESP
...

Error
At end of handler

- Handler restores saved registers
- Atomically return to interrupted process/thread
  - Restore program counter
  - Restore program stack
  - Restore processor status word/condition codes
  - Switch to user mode

Kernel system call handler

- Locate arguments
  - In registers or on user stack
  - Translate user addresses into kernel addresses
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back into user memory
  - Translate kernel addresses into user addresses
System call stubs

User Program

```c
main () {
    file_open(arg1, arg2);
}
```

Kernel

```c
file_open(arg1, arg2) {
    // do operation
}
```

User Stub

```c
file_open(arg1, arg2) {
    push #SYSCALL_OPEN
    trap
    return
}
```

Kernel Stub

```c
file_open_handler() {
    // copy arguments
    // from user memory
    // check arguments
    file_open(arg1, arg2);
    // copy return value
    // into user memory
    return;
}
```

User-level system call stub

```
// We assume that the caller put the filename onto the stack,
// using the standard calling convention for the x86.

open:
// Put the code for the system call we want into %eax.
    movl #SysCallOpen, %eax

// Trap into the kernel.
    int #TrapCode

// Return to the caller; the kernel puts the return value in %eax.
    ret
```
Kernel-level system call stub

```c
int KernelStub_Open() {
    char *localCopy[MaxFileNameSize + 1];
    // Check that the stack pointer is valid and that the arguments are stored at
    // valid addresses.
    if (!validUserAddressRange(userStackPointer, userStackPointer + size of arguments))
        return error_code;
    // Fetch pointer to file name from user stack and convert it to a kernel pointer.
    filename = VirtualToKernel(userStackPointer);
    // Make a local copy of the filename. This prevents the application
    // from changing the name surreptitiously.
    // The string copy needs to check each address in the string before use to make sure
    // it is valid.
    // The string copy terminates after it copies MaxFileNameSize to ensure we
    // do not overwrite our internal buffer.
    if (!VirtualToKernelStringCopy(filename, localCopy, MaxFileNameSize))
        return error_code;
    // Make sure the local copy of the file name is null terminated.
    localCopy[MaxFileNameSize] = 0;
    // Check if the user is permitted to access this file.
    if (!UserFileAccessPermitted(localCopy, current_process)
        return error_code;
    // Finally, call the actual routine to open the file. This returns a file
    // handle on success, or an error code on failure.
    return Kernel_Open(localCopy);
}
```