Basic Information

- Lab is in 010 Friend Center
- Use your OIT username/password
  - If you get the error “unable to mount /u/username” contact me, (most students should have it setup)
- Use scp/sftp to move files from arizona to lab machines
  - Eventually will be able to smbmount your cs home directory, wait for mailing list notification.

OS Bootup Process

- When a PC is booted:
  - Startup code in ROM (BIOS*) loads boot sector (Floppy, hard disk or USB flash disk) and jumps to it.
    - You might need to change your home machine’s BIOS setup to let it boot from USB flash disk first. (Older machines might not support boot from USB flash disk.)
  - Boot sector code loads OS kernel (start at sector: 2) and jumps to it
    - BIOS supplies minimal but sufficient hardware support (screen, disk, keyboard etc.)
Bootstrapping Layout

Disk layout

- BIOS
  - Bootblock
  - Kernel
- Memory layout (segment:offset)
  - BIOS data
    - 0x00000
    - 0x1000
  - Bootblock
    - 0x07c00
    - 0x90000
  - Stack
    - 0x9ffe
    - 0xb8000
  - Video RAM
    - 0x07c00
  - BIOS
    - 0xfffff

What You Must Do

- Design review
  - Have a bootblock that can print a string
  - Have print_char and print_string assembly functions
- bootblock.s
  - Load the kernel
  - Setup stack, data segments
  - Transfer control to kernel
- createimage.c
  - Extract code and data from executables
  - Assemble into boot disk (bootblock image + kernel image)

Too Hard? Too easy?

- bootblock.s
  - About 80 lines of assembly
  - Mostly mov instructions and BIOS calls
- createimage.c
  - About 200 lines of C
  - Use ELF headers and fopen, fseek, fread
- Little debugging ability
  - No printf or gdb to debug with
  - Can use BIOS print \texttt{print (int $0x10)} or just write directly to screen buffer in memory

x86 Structures

- Real Mode
  - Memory limited to 1Mbyte (bytes)
  - Originally 16-bit registers (Can only address $2^{16} = 64K$ bytes)
  - Segment : offset
  - Segment << 4 + offset (Can address)
- Protected Mode
  - Still segment : offset
  - Virtual address instead of physical address
**x86 structures** - Register Set

### General purpose registers

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>8</th>
<th>0</th>
<th>16 Bit</th>
<th>32 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>AH</td>
<td>AL</td>
<td>AX</td>
<td>EAX</td>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>BH</td>
<td>BL</td>
<td>BX</td>
<td>EBX</td>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>CH</td>
<td>CL</td>
<td>CX</td>
<td>ECX</td>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>DH</td>
<td>DL</td>
<td>DX</td>
<td>EDX</td>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>EBP</td>
<td></td>
<td></td>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>ESI</td>
<td></td>
<td></td>
<td>GS</td>
<td></td>
</tr>
<tr>
<td>DI</td>
<td>EDI</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>ESP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Segment registers

<table>
<thead>
<tr>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
</tbody>
</table>

### Status & control registers

<table>
<thead>
<tr>
<th>31</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EFLAGS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**GNU Assembly (AT&T Syntax)**

- **Data representations**
  - Registers: `%eax, %ax, %ah, %al`
  - Definitions (.equ): `BOOT_SEGMENT, 0x07c0`
  - Constants: `$0x0100, $1000`
  - Memory contents: `(0x40), %es:(0x40), (label)`

- **Labels**
  - Terminated by colon
  - Represent instruction pointer location

- **Comments**
  - `/* enclosed like this */`
  - `# or to the end of a line`

---

**GNU Assembly (AT&T Syntax)**

- **Data operations**
  - `mov{b,w,l}, lods{b,w,l}, ...`

- **Logic and arithmetic**
  - `cmp{b,w,l}, xor{b,w,l}, ...`

- **Process control**
  - `jmp, ljmp, call, ret, int, jne, ...`

- **Directives**
  - `.equ, .byte, .word, .ascii, .asciz`

---

**A Bit on Memory Access**

- `segment:[base+index*scale+disp] (Intel syntax)`
- `segment:disp(base, index, scale) (AT&T syntax)`
- `(segment)Default: Override:
  
  - `movw $0xb800,%bx`
  - `movw %bx,%ds`
  - `movw $0x074b,(0x40)`

  - Result = `(0xb800<<4) + 0x40 = 0xb8040`
  - Bootblock loaded at `0x07c0:0x0000`
  - Kernel need to be loaded at `0x0000:0x1000`
Common mistakes

- Don’t use `movw 4, %ax`, when you mean to use: `movw $4, %ax`
- Pair up with `pushw` and `popw`
- Setup `ds`, `ss` before using memory reference and stack
- Use `int $0x10` BIOS call, rather than `int $10`

Bootstrapping Layout

Setup stack and segment registers

- bootblock and kernel use same stack
- Set up (ss:sp)
- Stack pointer at the bottom
- Set bootblock data segment (ds=0x7c0)
- bootblock code segment (cs=0x0, offset = 0x7c0) set by BIOS before executing bootblock code.

Read the kernel into memory

- Kernel starts at 0x0:0x1000
- Use hardcoded kernel size
  - (os_size: number of sectors)
**bootblock.s (cont’d)**

- Set the kernel data segment
  - Set data segment (ds) to 0x0
- Long jump to kernel
  - ljmp 0x0,0x1000
  - This automatically sets code segment (cs) to 0x0

**ELF**

- What’s ELF?
  - Executable & Linkable Format
- ELF header, Program header table & segments
- Utilities: objdump, readelf.

**createimage.c**

- Read a list of executable files (ELF)
- Write segments (real code) into bootblock + kernel image file
- Note: Segments expand when loaded into memory (need padding)

```
<table>
<thead>
<tr>
<th>Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF Header (Elf32_Ehdr)</td>
</tr>
<tr>
<td>Program Header Table (Elf32_Phdr list)</td>
</tr>
<tr>
<td>Segment 1</td>
</tr>
<tr>
<td>Segment 2</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
```

**createimage.c (cont’d)**

- Read ELF header to find offset of program header **table**
- Read program header to find start address, size and location of segment
- Pad and copy segment into image file
- Write kernel size to hardcoded location in image file (in bootblock and be used when loading kernel)
ELF to Image Example

Bootblock ELF

Kernel ELF

ELF Header (Elf32_Ehdr)

Program Header 0 (Elf32_Phdr)

Segment 0

Segment 1

Other ELFs

Segment image file Location = p_vaddr-0x1000+512 (4096)

*In decimal for clarity

elf.h (/usr/include/elf.h)

- Utilize the Elf32_Ehdr and Elf32_Phdr structures
- Use fseek() and fread() to get them
- Example:
  /* ... */
  Elf32_Ehdr elfHdr;
  /* ... */
  ret=fread(&elfHdr,1,sizeof(elfHdr),fd);

FAQ

- Cylinders, Heads, Tracks?
  - Use 0x13 BIOS call to get parameters. (webpage)
- Use 32bit registers in real mode?
  - You can, but it’s not necessary.
- Won’t a big kernel overwrite the bootblock?
  - Yes. For extra credit, you can move the bootblock elsewhere first.
  - Int13 can only load 36 sectors at once. For large kernels, you might load one sector at a time...
- How many files should createimage handle?
  - As many as are in the command line (bootblock, kernel, and any number of others)