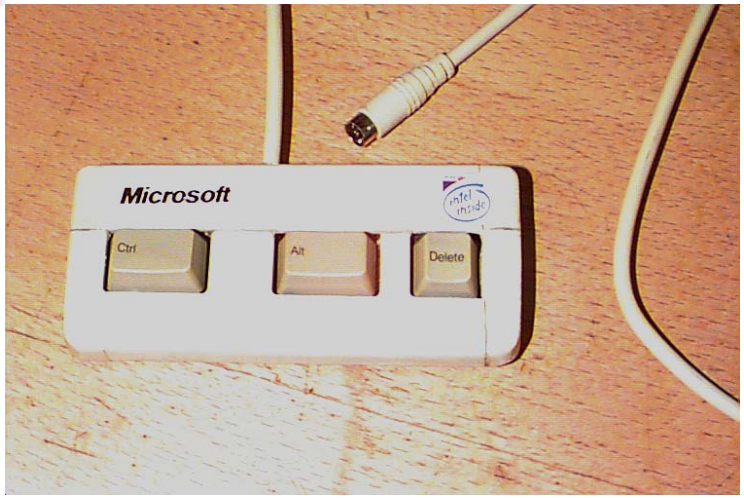
CS318 Project #1

Bootup Mechanism

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 student 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.

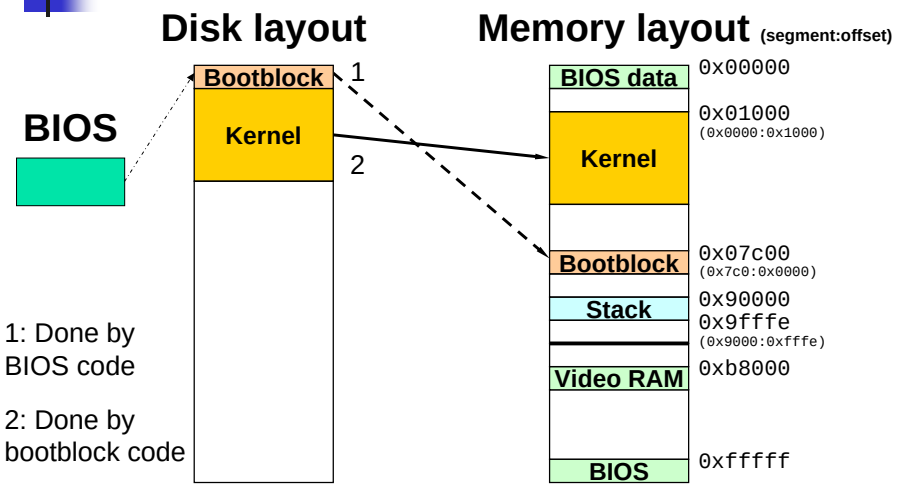
Reboot



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



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 *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

31 16	8	0	16 Bit	32 bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Segment registers

15 0
CS
DS
SS
ES
FS
GS

Status & control registers

31 16 8 0
EFLAGS
EIP

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:

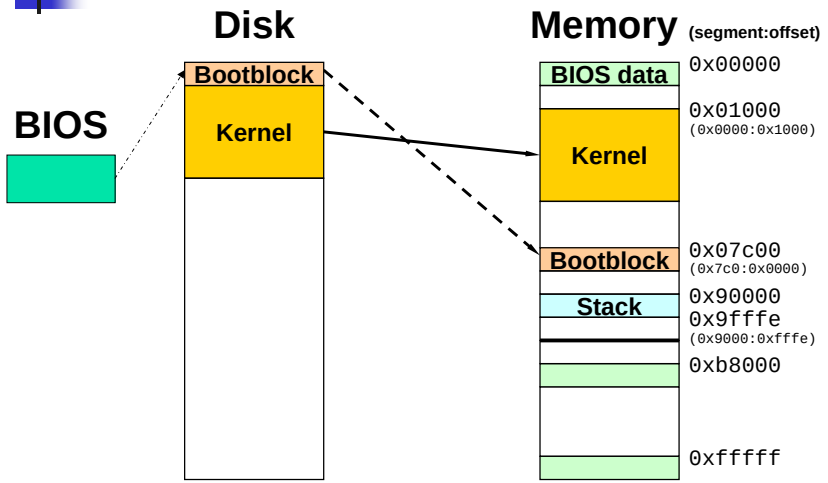
```
movw $0xb800,%bx          movw $0xb800,%bx
movw %bx,%ds              movw %bx,%es
movw $0x074b,(0x40)       movw $0x074b,%es:(0x40)
```

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

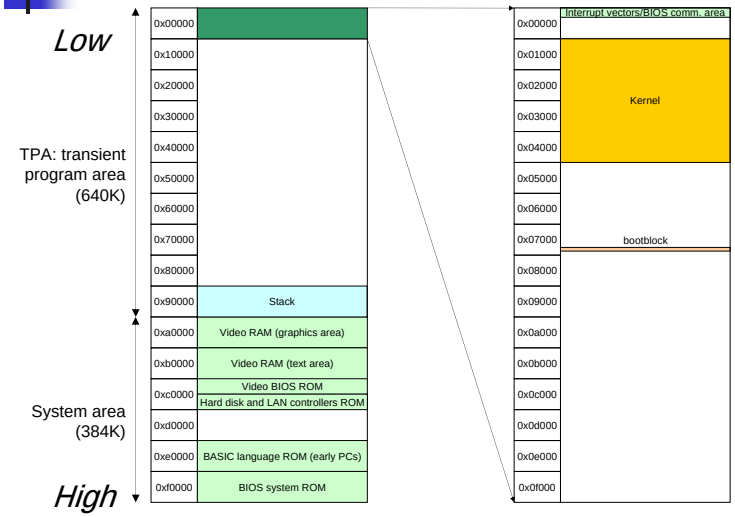
Common mistakes

- 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



Layout (to scale)



bootblock.s

- 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)

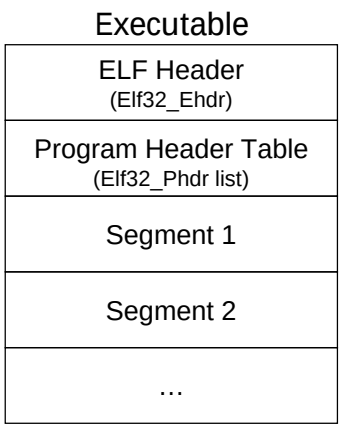
- Set the kernel data segment
  - Set data segment (ds) to 0x0
- Long jump to kernel
  - ljmp 0x0,0x1000
  - This automatically sets code segment (cs) to 0x0

ELF

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

createimage.c

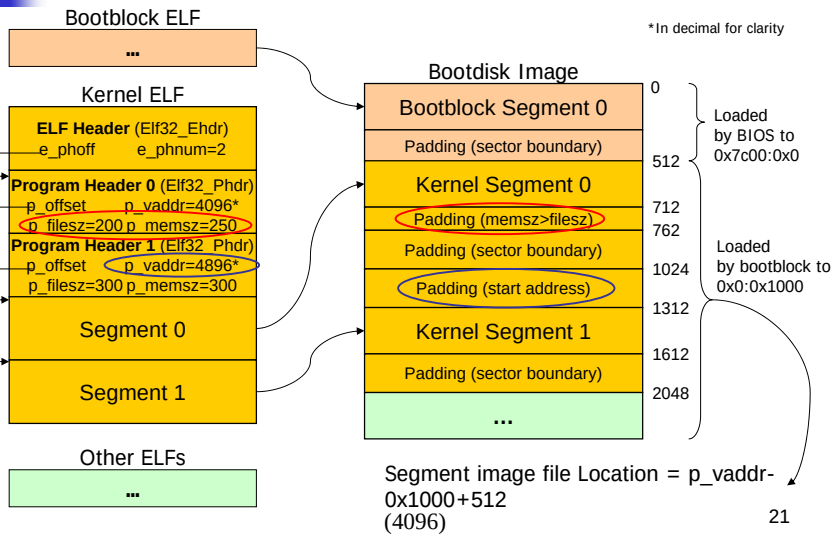
- Read a list of executable files (ELF)
- Write segments (real code) into bootblock + kernel image file
- Note: Segments expand when loaded into memory (need padding)



createimage.c (cont'd)

- Read ELF header to find offset of program header *table*
- Loop:
  - 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



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)