

# COS318 Precept 2 Bootup Mechanism

Wei Dong  
Sept. 27, 2006

1



## Outline

- More about X86 assembly language
- bootblock.s
- createimage.c

2

## DF and String instructions

- **lodsb**:  $\%al \leftarrow \%ds:(\%si)$ , update  $\%si$
- **stosb**:  $\%es:(\%di) \leftarrow \%al$ , update  $\%di$
- **movsb**:  $\%es:(\%di) \leftarrow \%ds:(\%si)$ , update  $\%si$  and  $\%di$
- All string instructions update indices by:
  - If  $DF == 1$  then  $index \leftarrow index - 1$
  - If  $DF == 0$  then  $index \leftarrow index + 1$

3

## Repeat String Operations

- **rep** only work with string instructions
  - Repeat while  $\%cx \neq 0$
  - Decrease  $\%cx$  by 1 each time
- Usage
  - Setup  $\%ds:\%si$  and/or  $\%es:\%di$
  - `cld/std`
  - $\%cx \leftarrow$  number of bytes
  - `rep lodsb/stosb/movsb`

4


## Calling convention

- How to pass parameters & receive result
- **cdecl**, pascal, fastcall, stdcall, ...
- Use same convention for both calling & called functions
- **cdecl**:
  - Parameters pushed to stack from right to left
  - Stack cleanup performed by the caller
  - Return value in `%eax`
  - `%eax, %ecx, %edx` are available for function

5

## Example (32-bit code)

```
int function (int a, int b);  
  
int a, b, x;  
...  
x = function(a,b);
```



```
    pushl  b  
    pushl  a  
    call   function  
    add    $12, %esp  
    movl   %eax, x
```

6

## Example (cont.)

```
int function (int a, int b)
{
    int c = a + b;
    return c;
}
```

```
.text
.globl _function
.def    _function
_function:
    pushl   %ebp
    movl   %esp, %ebp
    subl   $4, %esp
    movl   12(%ebp), %eax
    addl   8(%ebp), %eax
    movl   %eax, -4(%ebp)
    movl   -4(%ebp), %eax
    leave
    retl
```

%ebp

0x9FFE8	c
0x9FFEC	old %ebp
0x9FFF0	return addr
0x8FFF4	a
0x8FFF8	b
0x8FFFC	...

7

## Stack layout in 16-bit code

Short call

-2(%bp)	local var
(%bp)	old %bp
2(%bp)	ret %ip
4(%bp)	param 1
6(%bp)	param 2

Long call

-2(%bp)	local var
(%bp)	old %bp
2(%bp)	ret %ip
4(%bp)	ret %cs
6(%bp)	param 1

8

## bootblock.s: 16-bit or 32-bit?

- Concepts
  - 16/32-bit code, real/protected mode
- X86: start at real mode, later switch to protected mode
- The clean approach:
  - Bootloader does the switch
  - Kernel is pure 32-bit protected mode code
  - See bootblock.s in future projects if interested
- This project:
  - Only work with real mode

9

## bootblock.s: Common Errors

- Not setting up %ds, %ss, %sp
- No "\$" for constants
- Wrong offset from %bp for parameters
- Insert instructions before os\_size

10

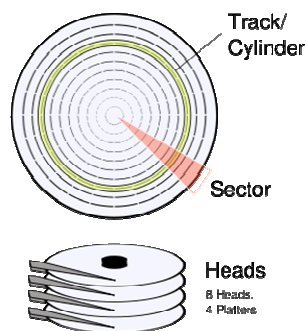
## Moving code!

- Code can be moved as data
- Using label to figure out where to jump to

start:	Jump to where?
jmp over	Assume code moved to %es:%di, then
...	%cs ← %es
rep movsb	%ip ← %di + \$(next - start)
ljmp     ?	How to do that?
next:	

11

## Disc geometry



cylinder 0, head 0, sector 1  
...  
cylinder 0, head 0, sector MAX\_SEC  
cylinder 0, head 1, sector 1  
...  
cylinder 0, head MAX\_HEAD, sector MAX\_HEAD  
cylinder 1, head 0, sector 1  
...  
...  
Use INT 0x13 Function 8 for MAX\_SEC and  
MAX\_HEAD

12

## Offset address overflow

- INT 0x13 use `%es:%bx` as buffer
- `%bx` is 16-bit, range from 0x0000-0xFFFF
- Example
  - Assume: `%bx = 0xFEA0`, then after reading a sector  
`%bx + 0x200 = 0x00A0`, lose 0x10000
  - Solution: update `%es` to reflect the overflow
  - Problem: how to detect overflow, how to adjust `%es`?

13

## Get ELF segment information

- How to avoid dealing with the file format?
  - Write the output of `readelf` to a text file
  - Read from that text file
  - Or use `popen` to avoid a temporary file
- But that's not interesting

14



## createimage.c

- Segments in program header table is not ordered!
- Actually no need to pad after internal segments
  - fseek beyond the end of the file will automatically cause the next file writing operation to fill the gap with 0s.
- Pad at the end so the whole file is divisible by 0x200