Lecture 17

COP3402 FALL 2015 – DR. MATTHEW GERBER – 11/18/2015

FROM EURIPIDES MONTAGNE, FALL 2014

Tonight

- The Absolute Loader
- The Bootstrap Loader
- Loading Programs
- Relocation

Last Time, On COP3402...

	Label	opcode	address
01		; This is	
02		; a comment	
03	start	.begin	x200
04	here	LOAD	sum
05		ADD	a
06		STORE	sum
07		LOAD	b
08		SUB	one
09		STORE	b
0A		SKIPZ	
0B		JMP	here
0C		LOAD	sum
0D		HALT	
0E	sum	.data	x000
0F	a	.data	x005
10	b	.data	x003
11	one	.data	x001
12		.end	start

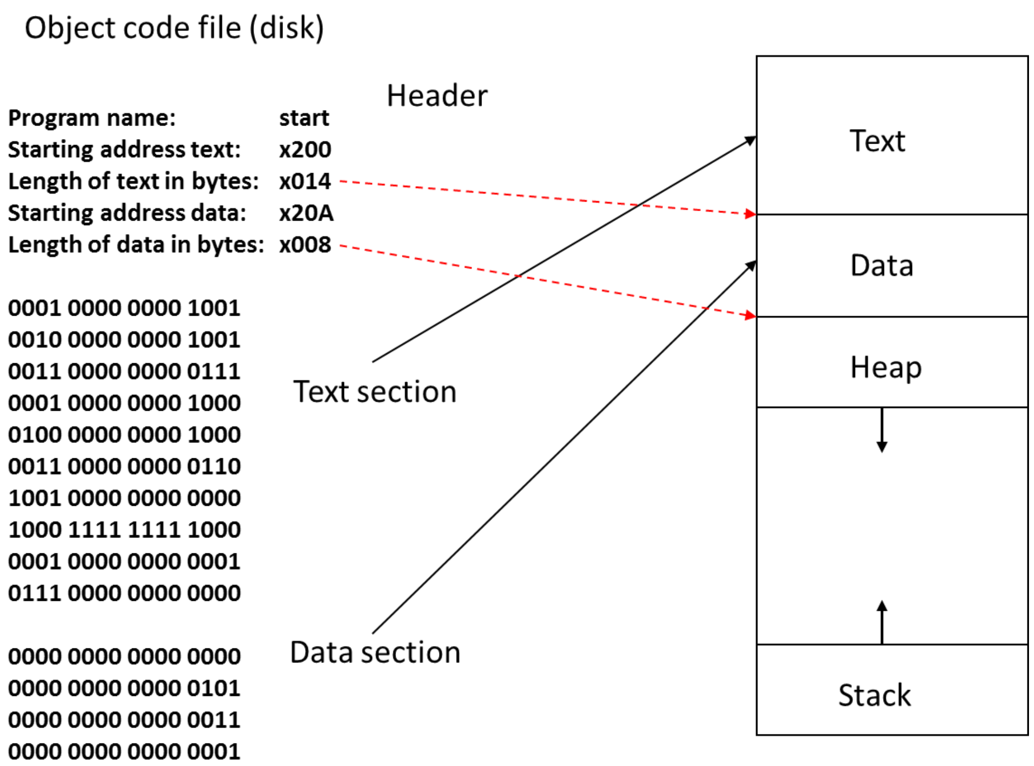
Assembler

Program name:	start
Starting address text:	x200
Length of text in bytes:	x014
Starting address data:	x20A
Length of data in bytes:	x008

```
0001 0000 0000 1001
0010 0000 0000 1001
0011 0000 0000 0111
0001 0000 0000 1000
0100 0000 0000 1000
0011 0000 0000 0110
1001 0000 0000 0000
1000 1111 1111 1000
0001 0000 0000 0001
0111 0000 0000 0000

0000 0000 0000 0000
0000 0000 0000 0101
0000 0000 0000 0011
0000 0000 0000 0001
```

Absolute Loading



An *absolute loader* will load the program at memory location x200.

- The header record is checked toverify that the correct program has been presented for loading.
- Each text record is read and moved to the indicate address in memory.
- When the “end” record is encountered, the loader jumps to the specified address to begin execution.

Bootstrapping

In a modern operating system, just about the only place we use an absolute loader is *bootstrapping*, or the *boot process*.

- You already have an intuitive sense of the boot process, but it is worth thinking about clearly

When the computer is turned on it does not have an operating system loaded in memory.

- The hardware alone cannot do the operations of an OS
  - Otherwise we wouldn't have an OS
- The bootstrap loader, or simply *boot loader*, is a program with the sole purpose of loading the operating system
- It *does not* have all the functionality of an operating system
- It *is* capable of loading the OS, and transferring control to it

Bootstrapping Then and Now

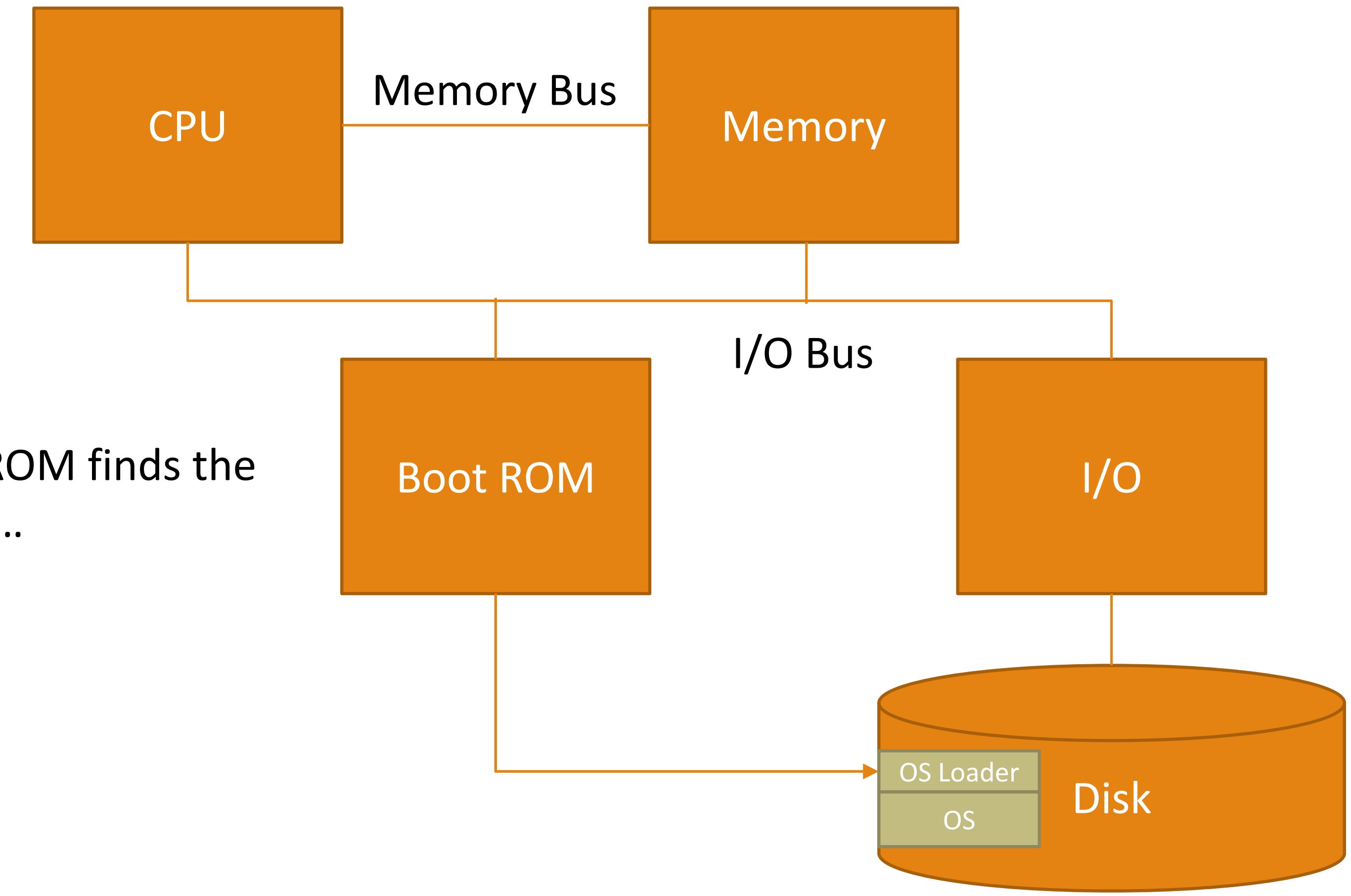
Early programmable computers had toggle switches on the front panel for the boot process.

- The operator would literally manually enter the bootloader into the program store!

In modern computers the bootstrapping process begins with the CPU executing software contained in ROM (usually rewritable flash ROM) at a predefined address. This software will:

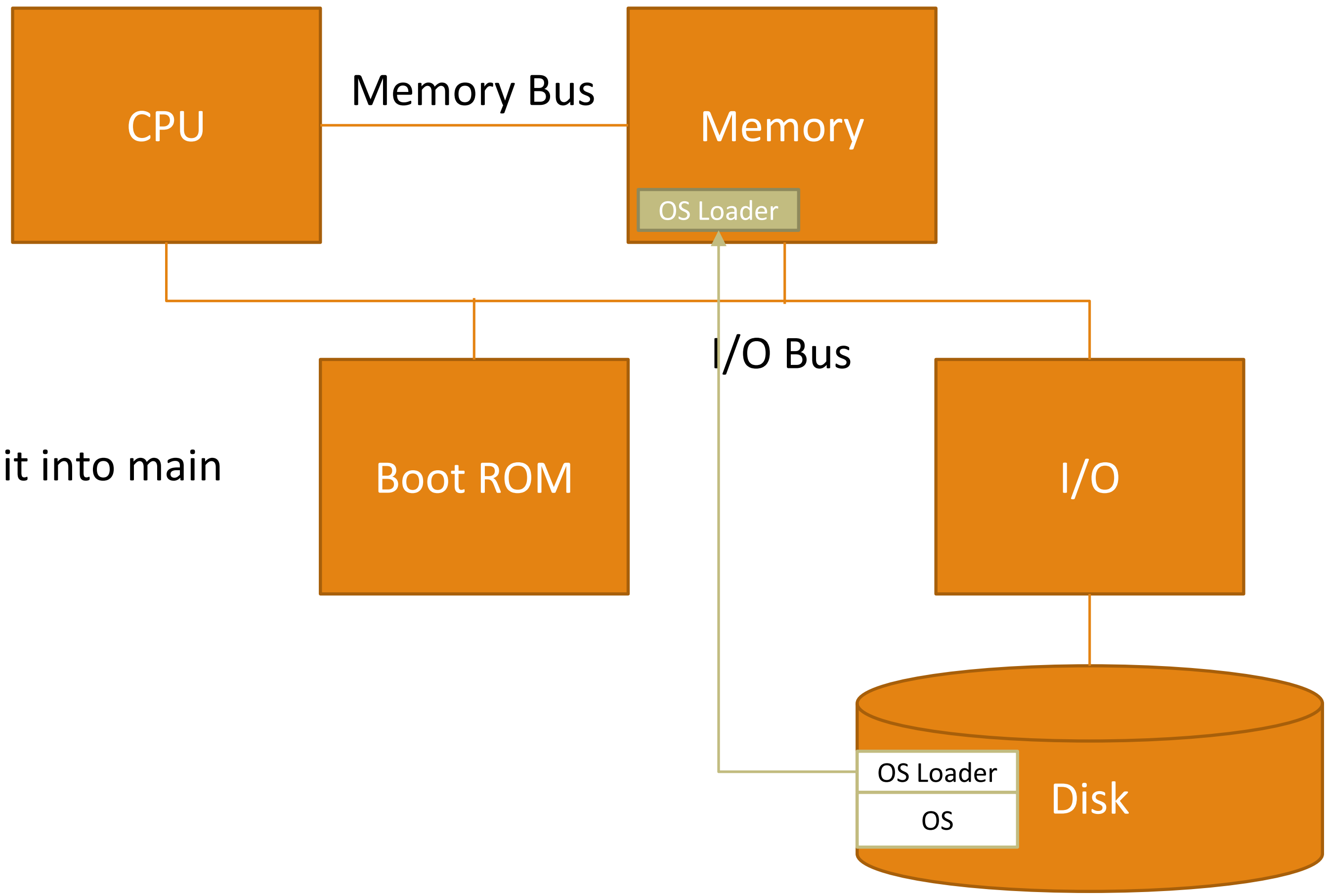
- Search for devices eligible to participate in booting the computer
- Select the highest-priority device that is ready to so participate
- Load a program from a specific location on that device
- This program is typically the *actual* OS loader
  - …though it can easily go more levels down than that!

Booting in Brief



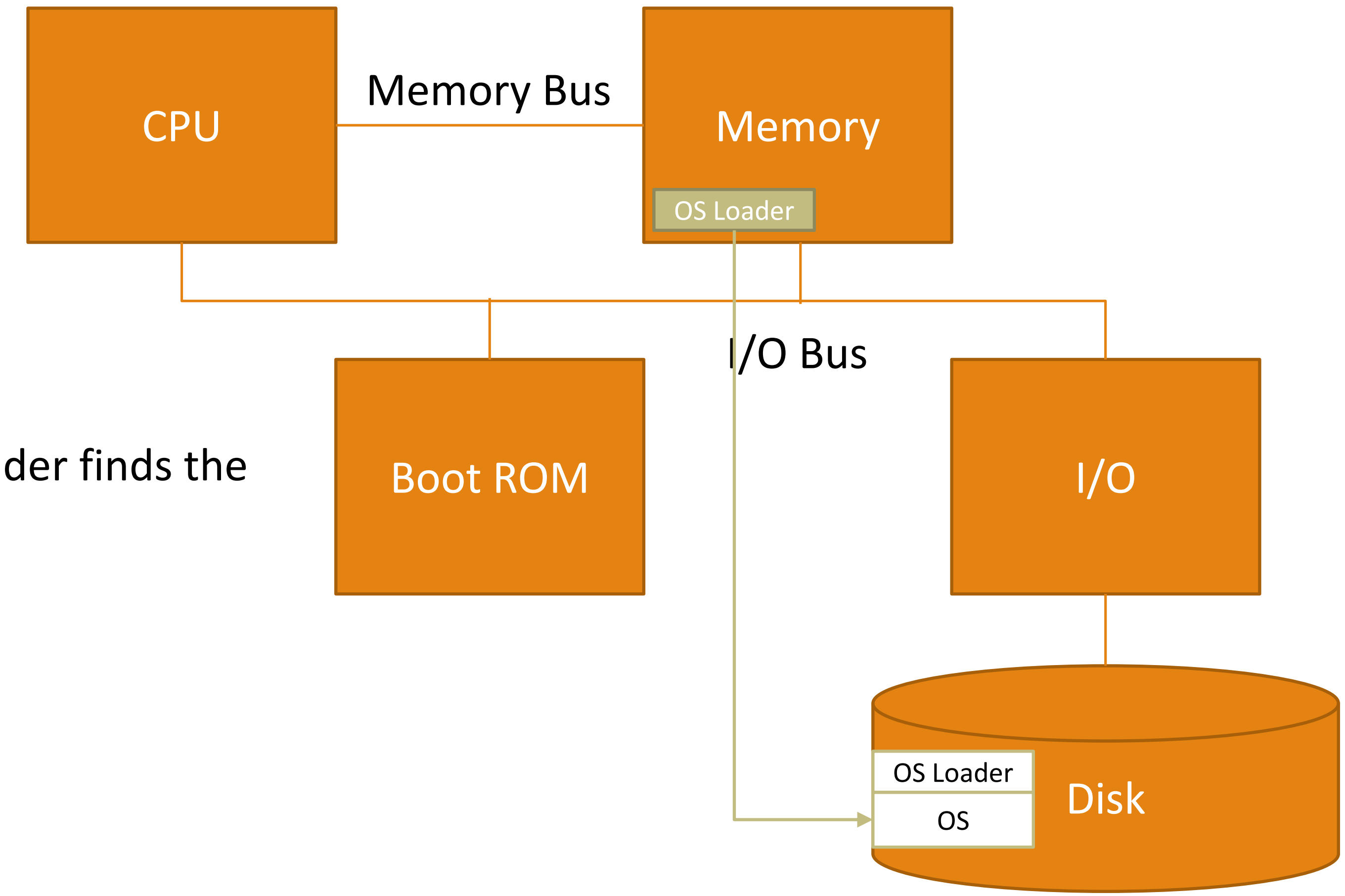
The Boot ROM finds the OS Loader…

Booting in Brief



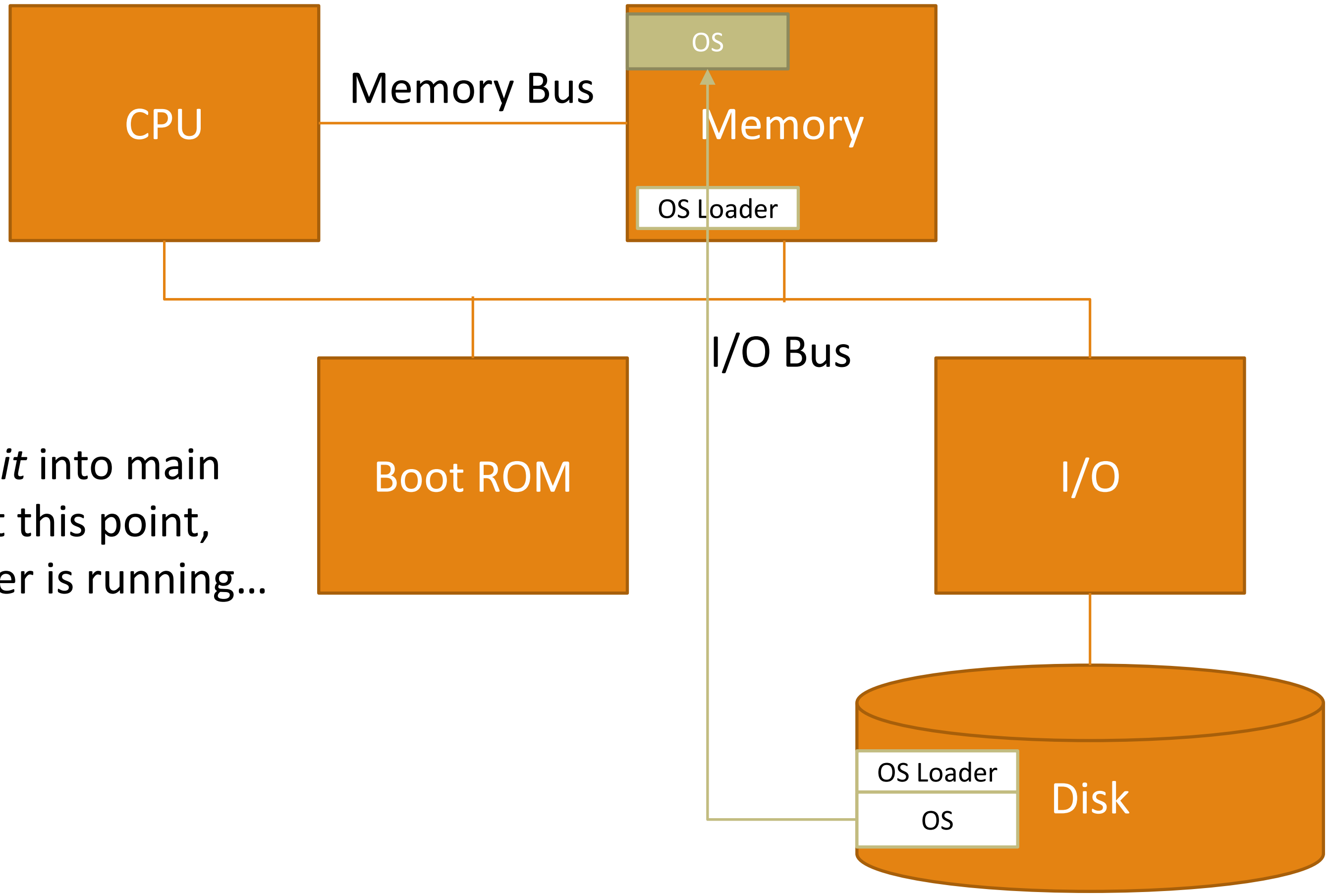
...and loads it into main memory.

Booting in Brief



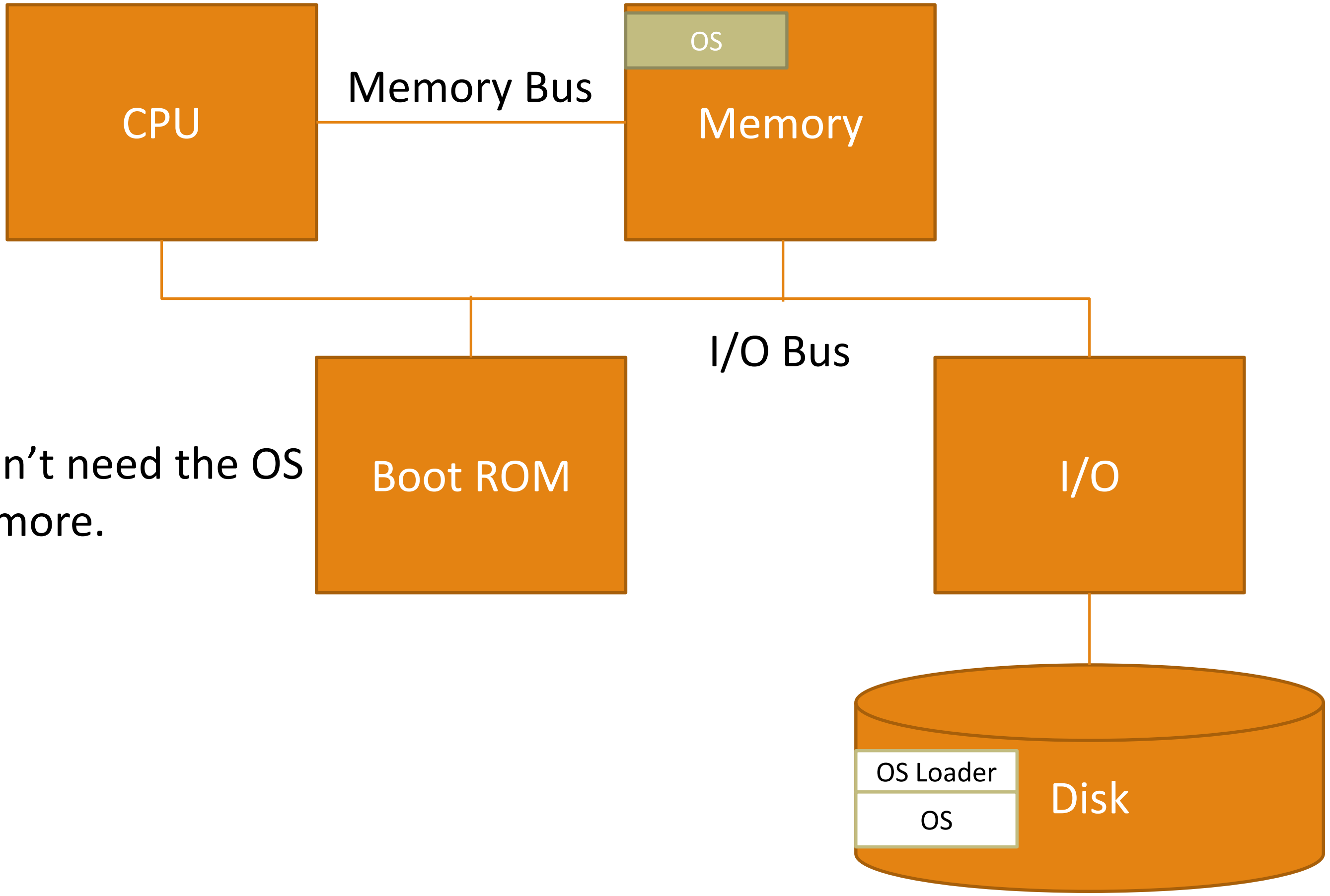
The OS Loader finds the OS...

Booting in Brief



…and loads *it* into main memory. At this point, the computer is running…

Booting in Brief



...and we don't need the OS Loader any more.

A Really Simple Bootloader

Location	Instruction/Data
0	(Blank area for OS)
99998	LC = 0
100000	Read from disk 0
	Store into location LC
	LC := LC + 1
	If (EOF) jump to 0
	Else jump to 100000

Here's an example of what a really, really simple bootloader might do.

- This isn't that far off from what some old bootloaders actually *did*
- It's written in pseudocode-assembly
- You should still get the idea

All bootloaders do a more complicated version of this.

- Check the hardware
- Load the OS loader into memory
- Pass control to the OS loader
- The OS loader will (hopefully) load the OS and pass control to *it*

After the bootloader

So you've loaded an operating system. Now what?

- We know the OS will load device drivers, a user interface, and so on
- How does it do that?
- That's mainly for an OS course, but let's give you the basics

The only purpose of an operating system is to run other programs that you want to run.

- Once you load and run a program, it's no longer called a program

Processes

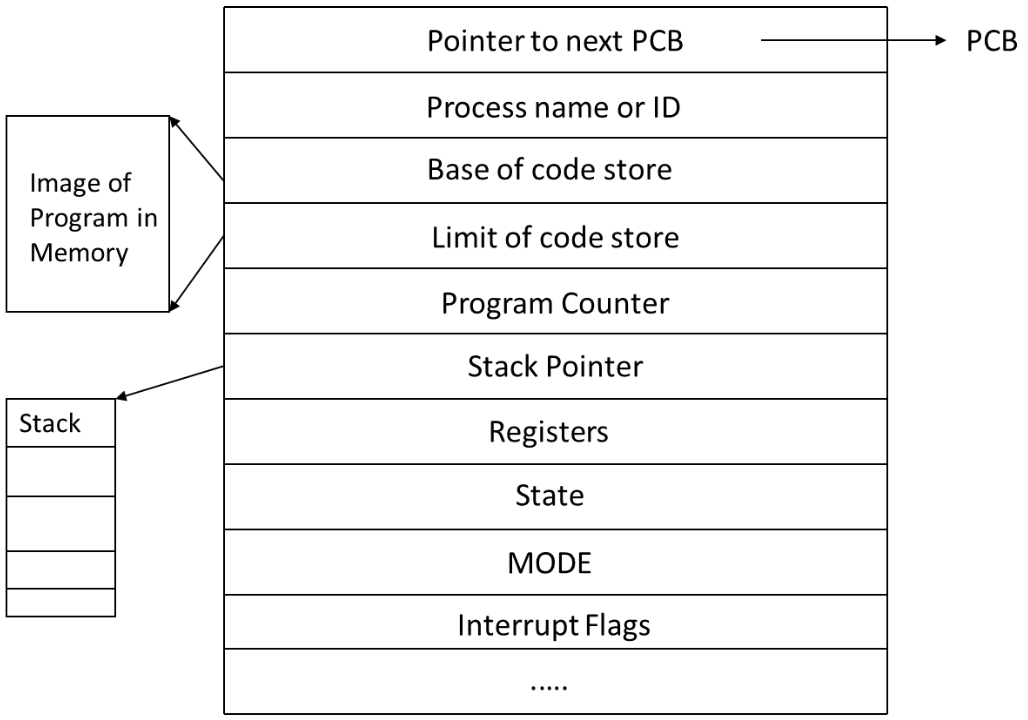
A *process* is *a program in execution.*

- Conceptually, it's an asynchronous activity that can run independent of the OS and other processes
  - In reality it's *heavily* dependent on the OS and other processes
  - But *basic execution* still proceeds independently

To system software, a process can be viewed as the *locus of control* of a program in execution

- An operating system will invariably maintain some sort of *process control block* (PCB) for every running process
- A process may be roughly thought of as the couple of:
  - The PCB
  - The image of the program in memory

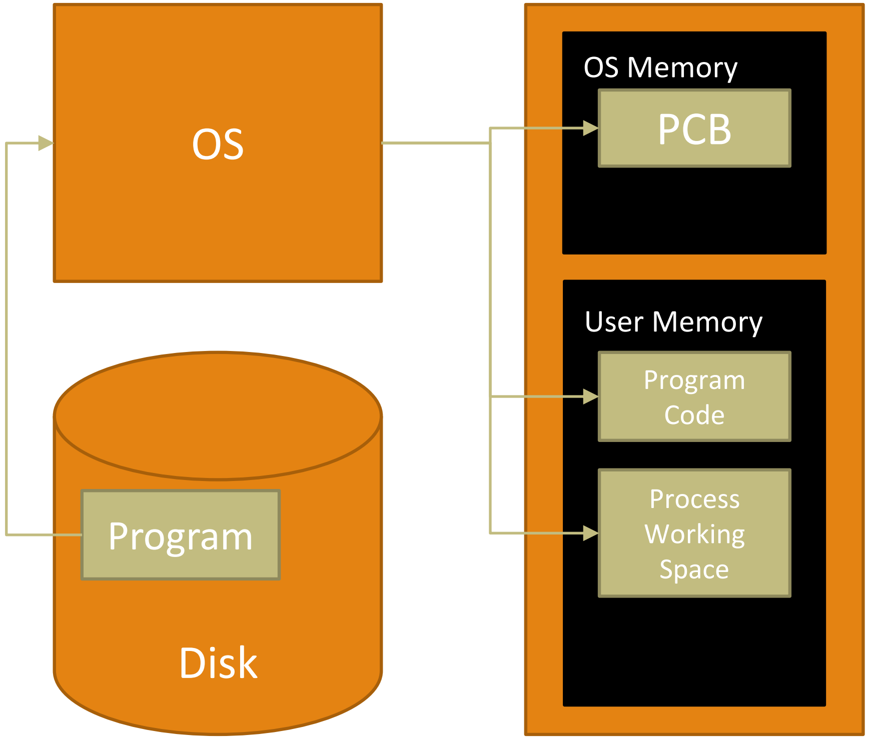
The PCB



Each PCB will contain, at very least, versions of this information.

- This is *not* a complete list, but there are a few things to note. In particular…
- The code store base and limit point to the image of the program in memory
- The stack points to a dedicated stack for this process
  - Each process *must* have a dedicated stack
  - Think about it
- Most of the other information is for the OS to use for context switching purposes

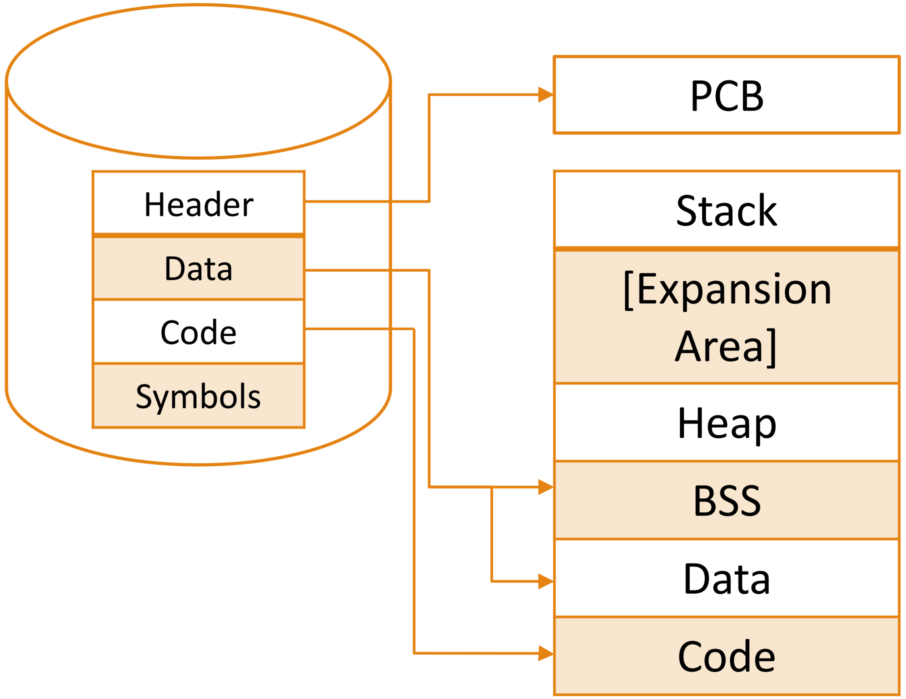
Process Creation



The OS reads the program from disk, and creates several things in memory:

- The process control block
- The program code image
- A working space for the process

The Process Working Space



The process working space contains at least the following:

- You already know about the **stack**
- The **heap** is where dynamically allocated memory lives
  - When you malloc() or construct something, it goes here
- The **BSS** and **data** areas are similar, except:
  - The BSS area is for **uninitialized** or **zero-initialized** global and static variables
    - It stands for “Block Started by Symbol”, a historical name that literally goes all the way back to FORTRAN
  - The data area is for **initialized** global and static variables
- The **code** area is exactly what you think it is

Relocation

Absolute loaders load a program into a specific memory location.

- Obviously, we want to run more than one program at a time on a computer
  - (This wasn't always obvious!)
- Equally obviously, there's no good way to know where all the programs will be before we start them
  - (This *also* wasn't always obvious!)

We want the loader to be able to load a program into memory wherever there's room for it.

- A loader that can do this is called a *relocating loader*

There are a few different ways to do this; one is for the assembler to help us out.

- For each instruction, a set of *relocation bits* says what parts of the instruction are to be modified
- Say the program is loaded at location *r*; then all modifiable parts of the instruction will have *r* added
- Next slide has an example, *intentionally* ***not*** *using either the Tiny or PM/0 instruction sets*

Relocation Using Relocation Bits ($r = 40$)

Source

Label		opcode	address	address
00		copy	zero	older
03		copy	one	old
06		read	limit	
08		write	old	
10	comp	load	older	
12		add	old	
14		store	new	
16		sub	limit	
18		brpos	finalL	
20		write	new	
22		copy	old	older
25		copy	new	old
28		br	comp	
30	final	write	limit	
32		stop		
33	zero	CONST	0	
34	one	CONST	1	
35	older	SPACE		
36	old	SPACE		
37	new	SPACE		
38	limit	SPACE		

Before

Loc#	Len	reloc	text
00	3	011	13 33 35
03	3	011	13 34 36
06	2	01	12 38
08	2	01	08 36
10	2	01	03 35
12	2	01	02 36
14	2	01	07 37
16	2	01	06 38
18	2	01	01 30
20	2	01	08 37
22	3	011	13 36 35
25	3	011	13 37 36
28	2	01	00 10
30	2	01	08 38
32	1	0	11
33	1	0	00
34	1	0	01
35			
36			
37			
38			

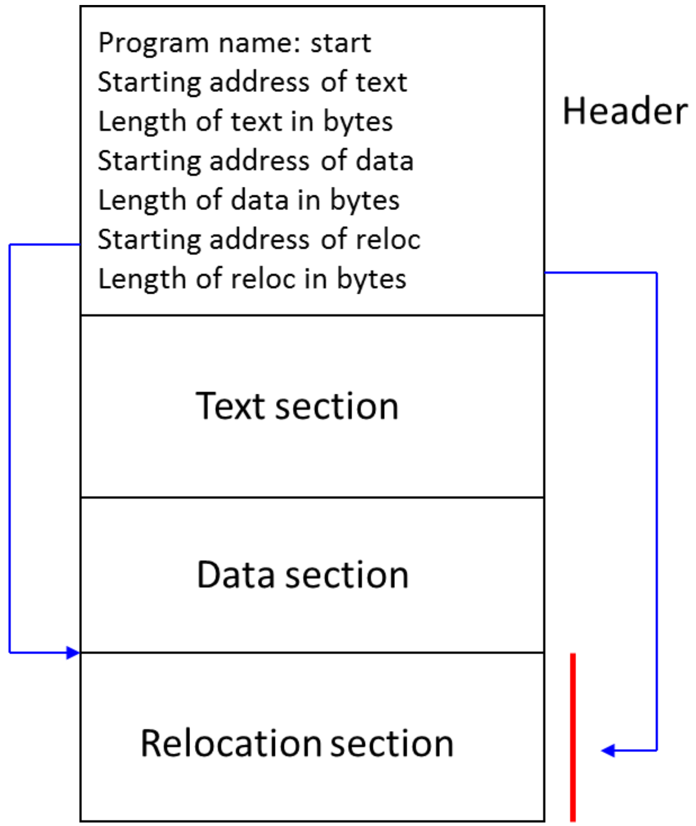
After

Loc#	text
40	13 73 75
43	13 74 76
46	12 78
48	08 76
50	03 75
52	02 76
54	07 77
56	06 78
58	01 70
60	08 77
62	13 76 75
65	13 77 76
68	00 50
70	08 78
72	11
73	00
74	01
75	
76	
77	
78	

Relocation Records

Relocation bits do the job, but they make loading the text directly into memory cumbersome and messy.

- What we *actually* do is put all the relocation information in one place in the program file
- This area of the file has all the same information the relocation bits do in one format or another
- The loader uses the segment to determine what addresses need to be changed in the relocation process
- Depending on how good our memory management is, we may be able to get rid of this area after the loading process, or may have to keep it around in case we move the process



Next Time: Interrupts and the Hardware/Software Bridge