LOADING APPLICATIONS & LOADING THE OS

Azza Abouzied

HOW DOES THE OS COME TO LIFE?

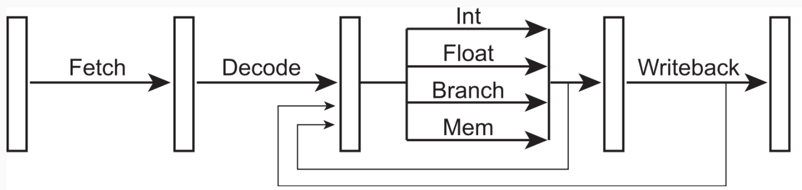
This is a natural question to start an Operating Systems course with.

But before we can answer it, we need to ask perhaps a simpler question: how does the kernel bring any application to life?

LOADING APPLICATIONS

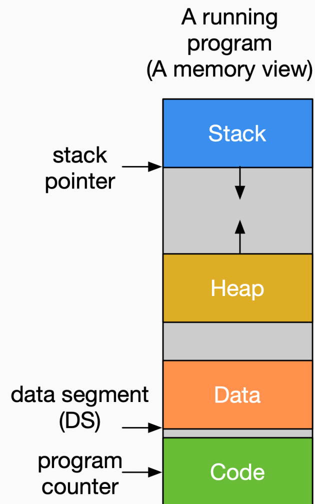
HOW THE PROCESSOR FUNCTIONS?

Recall the instruction cycle:



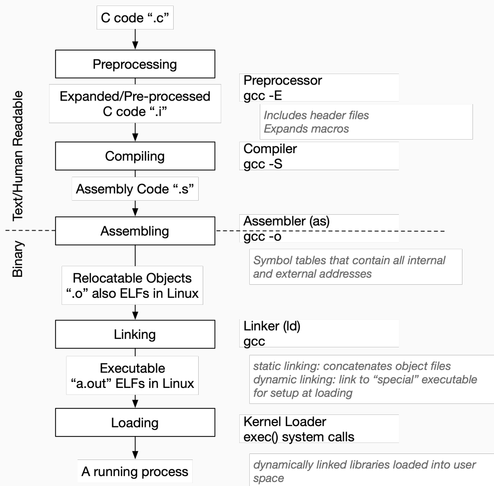
So our program needs to reside on memory!

1. Why divide an executable into sections or segments?
2. What features do modern hardware architectures provide to support this layout?
3. What is the purpose of the stack?
4. What is the heap?



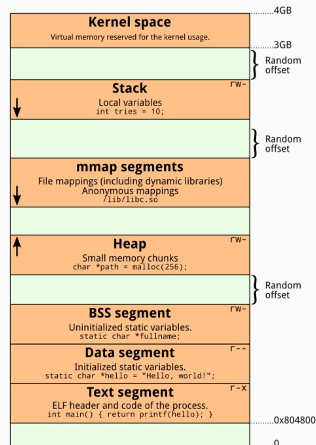
FROM CODE TO PROCESS

1. How does the kernel loader know how to lay out an executable and start its execution? "Magical ELFs"
2. How do we get Executable and Linkable Files (ELFs)?
3. Why are loaders part of the kernel? Why do they run with kernel privileges?
4. What about linkers, assemblers, and compilers?



A RUNNING C PROGRAM IN MORE DETAIL

1. Why separate initialized from uninitialized data segments?
2. What are these dynamically linked libraries?
3. What are statically linked libraries?



COMPILATION IN ACTION

The C preprocessor:

```
gcc -E file.c > file.i
```

The Compiler:

```
gcc -S file.c
```

Only the assembler (sidestepping the linker):

```
gcc -c file.c -o file.o
```

The complete process with linking:

```
gcc file.c
```

The readelf tool lets you see the binary file's header -h, section headers -S and symbol tables -s

Executable and linkable file format (ELF):

```
readelf -h -S -s a.out
```

Relocatable Objects (cannot run without linking!)

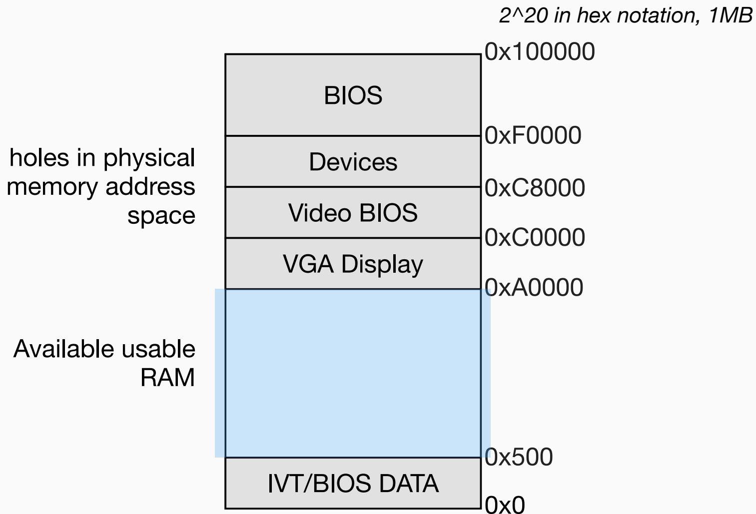
```
readelf -h -S -s file.o
```

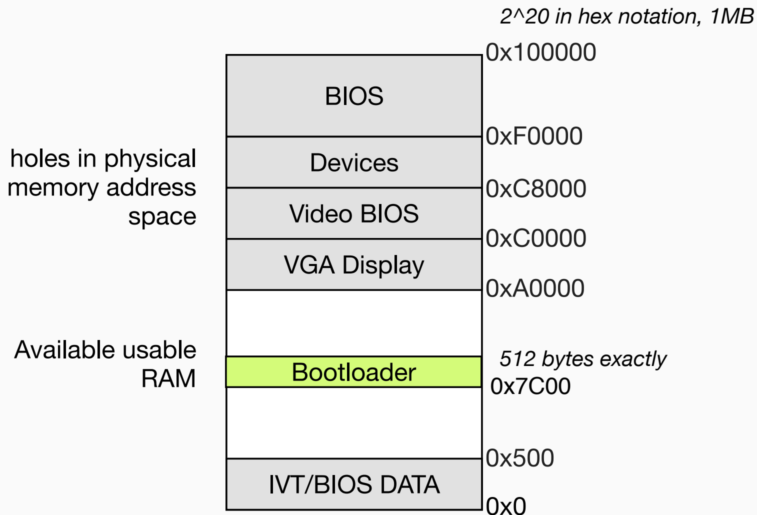
THE BOOT PROCESS

THE KERNEL IS AN ELF TOO!

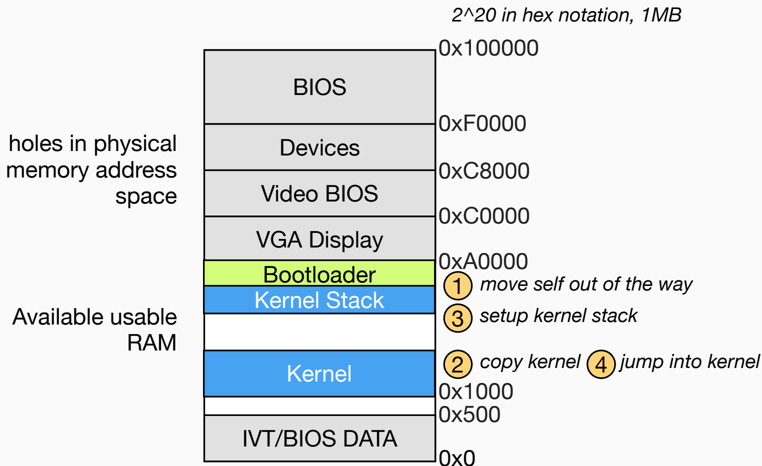
So we need a loader that reads the right sections and loads them from disk onto memory!

1. Hardware wakes up and loads bootstrap routine
   - X86: this routine is handled by BIOS, which lives on ROM
   - X86 wakes up in real mode: 20-bit addressing mode accessing 1MB of memory.
2. Bootstrap finds storage devices and initializes them
3. One of these devices has a boot sector — the boot device
4. Boot sector is 512 bytes with 0x55 0xAA at the end of the sector
5. Bootstrap reads bootloader from boot device
6. Bootloader loads kernel from boot device
7. Kernel is now alive: its loads its remaining parts and starts.

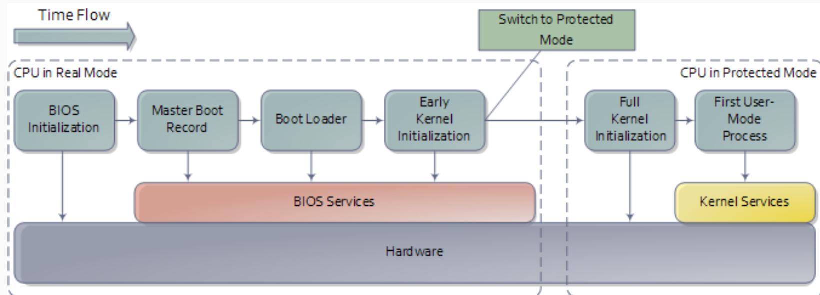

2^20 in hex notation, 1MB
0x100000
BIOS
0xF0000
holes in physical memory address space
Devices
0xC8000
Video BIOS
0xC0000
VGA Display
0xA0000
Available usable RAM
0x500
IVT/BIOS DATA
0x0


2^20 in hex notation, 1MB
0x100000
BIOS
0xF0000
holes in physical memory address space
Devices
0xC8000
Video BIOS
0xC0000
VGA Display
0xA0000
Available usable RAM
512 bytes exactly
Bootloader
0x7C00
0x500
IVT/BIOS DATA
0x0

BOOTING FROM A MEMORY PERSPECTIVE



BOOTING TIMELINE



Taken from this now defunct site: `https://rahulkumar4.wordpress.com/tag/system-boot/`

Can we have multiple partitions on disk each with its own boot sector? *Yes, the MBR, a special kind of bootsector usually found on partitioned devices, contains a partition table and code to load the bootloader from the right boot sector.*

A typical kernel today is 2GB, how do we handle the loading of the kernel? What about our limited address space? *You start in real bit-addressing mode, which only gives you 2^{20} bit addressing before you move to protected or 2^{32} bit addressing, which gives you access to 4GB of RAM and hardware memory protection support. What about 64-bit? There is long-mode!*

Bootkits are nasty malware. Why? *Thunderstrike is a Mac OS X bootkit... Don't leave your machine unattended*

YOU IN THE NEXT FEW DAYS!

Bear Grylls climbs a tree

Questions?