

```
robm@homebox ~$ sudo su
Password:
robm is not in the sudoers file.
This incident will be reported.
robm@homebox ~$ █
```



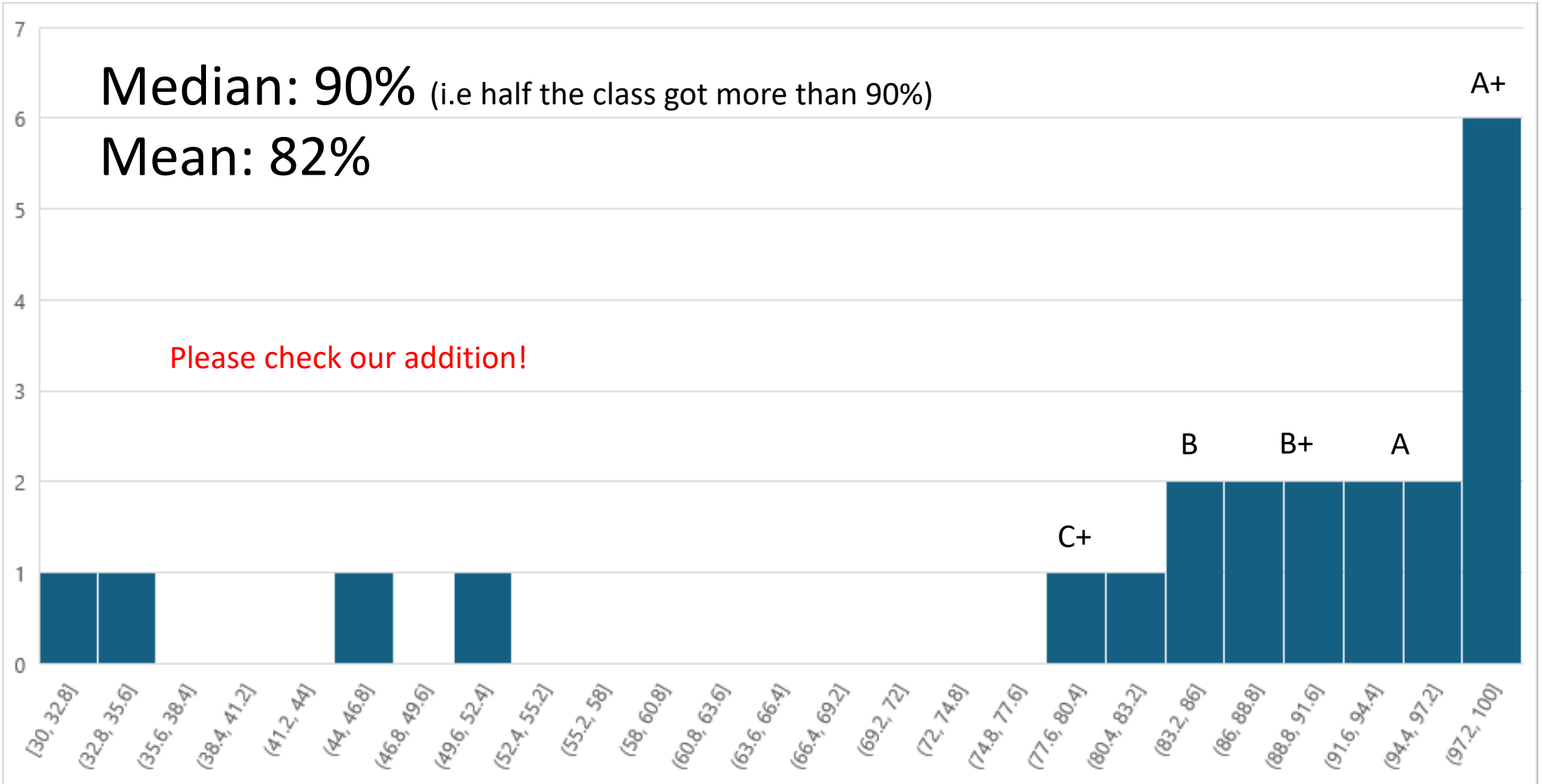
HEY — WHO DOES
SUDO REPORT THESE
"INCIDENTS" TO?

YOU KNOW, I'VE
NEVER CHECKED.



Assignments – HW2 Point Distribution

Number of Students



Points (Roughly aligned with fractional letter grade)

Assignments – HW2 Point Distribution

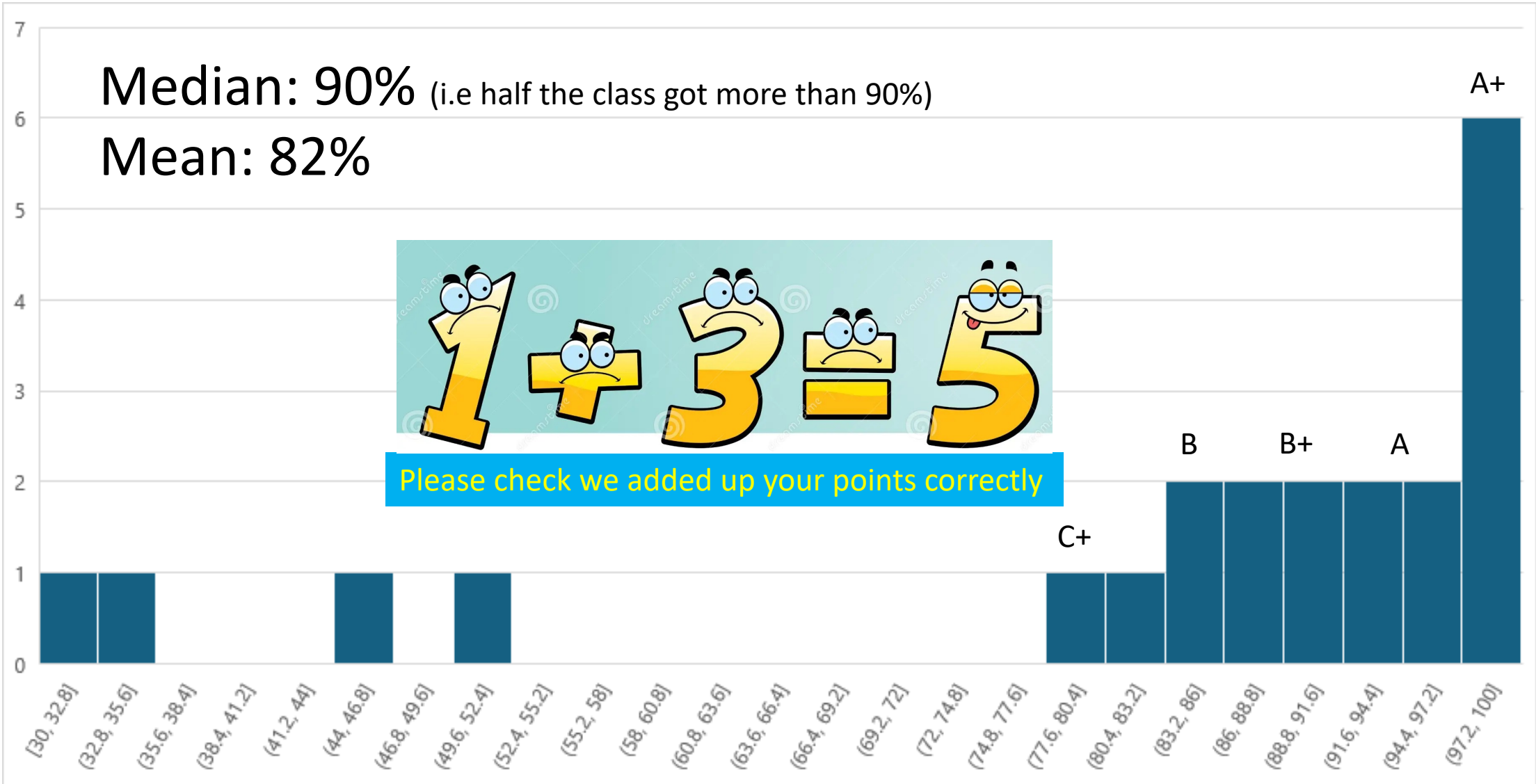
Median: 90% (i.e half the class got more than 90%)

Mean: 82%

Number of Students



Please check we added up your points correctly



Points (Roughly aligned with fractional letter grade)

Booting and the Kernel

Git Clone A little Kernel

- `git clone https://github.com/gmfricke/bootkernel.git`

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." –Linus Torvalds



Git Clone A little Kernel

```
git clone https://github.com/gmfricke/bootkernel.git
```

```
[matthew@moonshine bootkernel]$ ls -lh
```

```
total 24K
```

```
-rw-r--r--. 1 matthew matthew 261 Feb 14 12:00 boothello.asm  
-rw-r--r--. 1 matthew matthew 1.4K Feb 14 12:00 bootsect.asm  
-rw-r--r--. 1 matthew matthew 445 Feb 14 12:00 kernel.c  
-rw-r--r--. 1 matthew matthew 186 Feb 14 12:00 kernel_entry.asm  
-rw-r--r--. 1 matthew matthew 742 Feb 14 12:00 Makefile  
-rw-r--r--. 1 matthew matthew 43 Feb 14 12:00 README.md  
drwxr-xr-x. 2 matthew matthew 158 Feb 14 12:00 utils
```

Main Memory (RAM)

- RAM is a large collection of addressable bits.
- The CPU(s) fetch bits from a particular address in RAM and treat them as an instruction.
- The CPU(s) fetch bits from RAM and treat them as data.
- The CPU decodes the instruction bits into an operation to execute on the data.

Nearly everything the Kernel does revolves around main memory.



The Kernel – Memory Management

- One of the Kernel's jobs is to divide the bits in RAM into regions.
- It has to allocate a region of RAM to each process on the system and remember who owns that memory.
- The Kernel also has to make sure no process accesses RAM that doesn't belong to it.

The Kernel

The Kernel has four main jobs

1. Decide when each program gets to use the CPU
2. Keep track of all memory allocations and which programs own them
3. Interface with hardware through device drivers
4. Support System Calls. System calls are functions programs can use to interact with the Kernel.

Process Management

- Processes vs Programs: A program is a set of instructions. A process is a program plus its current data laid out in memory.
- The Kernel is responsible for starting, pausing, resuming, scheduling, and terminating processes.
- On a typical computer many processes are running simultaneously
- Vi and Chrome might be running at the same time for example (but they are not really running at exactly the same time [on a one core system]).
- These processes run using a time-slice approach. Each processes uses the CPU to execute instructions for a fraction of a second before the next process uses the CPU.
- The Kernel is responsible changing which processes is executing.

Process Management – A Single Time Slice.

- The CPU interrupts the current process execution based on a timer, switches to “Kernel Mode” and begins executing the Kernel process.
- The Kernel records the current state of the CPU and its registers (including the program counter which keeps track of which instruction is to be executed next).
- The process’ memory region is made read-only so it can’t change before the next time it gets a time-slice.
- The Kernel processes and events that were queued up since the last time it had control of the CPU. (maybe a TCP/IP network packet arrived and is queued up in a buffer ready for reading or maybe a keystroke was detected)
- The Kernel analyzes the list of processes waiting to run on the CPU and chooses one.
- The Kernel restores the registers for the process it chose (it recorded them in step 2 above last time it saw this processes) – this is called a “Context Switch”
- The Kernel tells the CPU how long to execute this process.
- The Kernel switches the CPU into User Mode.
- The CPU begins executing the next instruction in the process (using the program counter to keep track.)

The Kernel – Memory Management

- One of the Kernel's jobs is to divide the bits in RAM into regions.
- It has to allocate a region of RAM to each process on the system and remember who owns that memory.
- The Kernel also has to make sure no process accesses RAM that doesn't belong to it.

The Kernel – Memory Management

- The Kernel has its own region of RAM that other processes can't access.
- This is accomplished by the CPU not allowing any process executing while the CPU is in User Mode from accessing the Kernel memory.
- This is enforced at the hardware level. Only when the CPU is in Kernel Mode can it load data from Kernel Memory.
- The Kernel controls access to memory owned by User Processes. Some Processes can share their memory (shared vs private memory)
- Some processes want their memory space to be read only.
- The Kernel also has to handle paging memory to the swap partition if it runs out of physical RAM.

Memory Management Unit (MMU)

- To help the Kernel do its job modern computers have hardware called the Memory Management Unit.
- Each User process on the computer sees a “virtual memory” space. On 32 bit systems virtual memory is 3 GB per process, on 64-bit systems it is 256 TB. (This is just the address space).
- The MMU maps addresses in virtual memory to real physical addresses in actual RAM.
- The Kernel’s job is to keep track of the mapping between virtual memory addresses and real memory addresses*.
- That’s how the Kernel controls what regions of physical memory user processes can actually use.

*This mapping from virtual to physical memory is called a page table. It is also how the Kernel can send some memory pages to swap on disk if needed.

The Kernel – Device Management

- Only the Kernel can tell hardware what to do. This is to prevent user processes from doing things like telling the system to turn off.
- Devices, even of the same type, rarely have the same API (application programming interface).
- Device drivers provide a common interface for similar devices, and the kernel exposes an even more standard interface to user programs. Most devices are reduced to a device file that can read and/or write data.
- This is all the usual layers of abstraction you will have seen all through programming to make managing the complexity of using a computer manageable.

The Kernel - System Calls

- User processes interact with the Kernel through System Calls (syscalls).
- You already saw the `read()` and `write()` system calls. You ran them directly when reading and writing to the pseudo-terminal.

Install Additional Manual Pages

```
[matthew@localhost ~]$ sudo yum install man-pages man-db man
```

```
Last metadata expiration check: 0:43:29 ago on Wed 14 Feb 2024 12:33:11 AM CST.
```

```
Package man-db-2.9.3-7.el9.x86_64 is already installed.
```

```
Package man-db-2.9.3-7.el9.x86_64 is already installed.
```

```
Dependencies resolved.
```

```
=====
=====
Package                Size                Architecture        Version                Repository
```

```
Installing:
```

```
man-pag                noarch                6.04-1.el9                baseos                5.7 M
```

```
Installing weak dependencies:
```

```
man-pages-overrides    noarch                9.0.0.0-1.el9                appstream                16 k
```

Fork and Exec Syscalls

- The Kernel starts executing processes through two main system calls:

Fork – makes a copy of the current process.

The Fork Syscall (Manual Section 2)

```
[matthew@moonshine ~]$ man 2 fork
fork(2)                               System Calls Manual          fork(2)
NAME
    fork - create a child process
LIBRARY
    Standard C library (libc, -lc)
SYNOPSIS
    #include <unistd.h>
    pid_t fork(void);
DESCRIPTION
    fork() creates a new process by duplicating the calling process. The
    new process is referred to as the child process. The calling process
    is referred to as the parent process.

    The child process and the parent process run in separate memory spaces.
    At the time of fork() both memory spaces have the same content. Memory
    writes, file mappings (mmap(2)), and unmappings (munmap(2)) performed
    by one of the processes do not affect the other.
```

Fork and Exec Syscalls

- The Kernel starts executing processes through two main system calls:

Fork – makes a copy of the current process.

Exec(program) – replaces the current process with the one containing the binary “program”.

The Exec library functions (Manual Section 3)

```
[matthew@moonshine ~]$ man 3 exec
```

```
exec(3)
```

```
Library Functions Manual
```

```
exec(3)
```

NAME

execl, execlp, execl, execv, execvp, execvpe - execute a file

DESCRIPTION

The `exec()` family of functions replaces the current process image with a new process image. The functions described in this manual page are layered on top of `execve(2)`. (See the manual page for `execve(2)` for further details about the replacement of the current process image.)

The initial argument for these functions is the name of a file that is to be executed.

The functions can be grouped based on the letters following the "exec" prefix.

Process Tree

- These fork and exec commands create a tree structure of processes
- The root of this tree is the “init” process (we will talk about init at the end of this lecture)

For example,

When you run “ls” this is what happens:

Shell Process



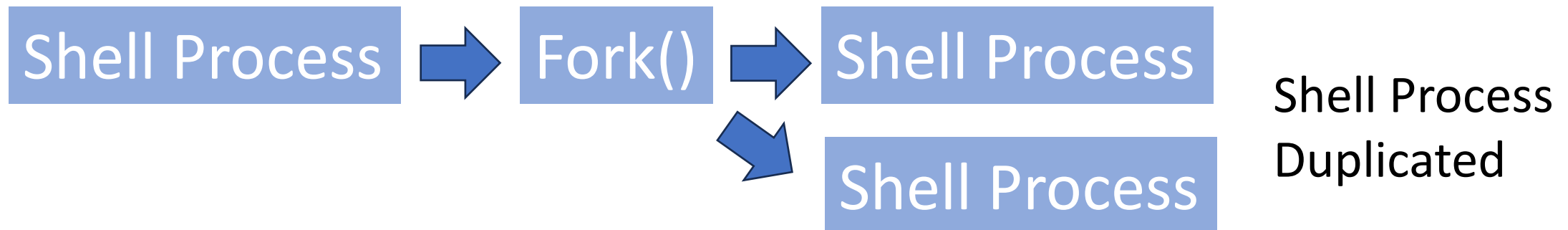
Fork()

Process Tree

- These fork and exec commands create a tree structure of processes
- The root of this tree is the “init” process (we will talk about init at the end of this lecture)

For example,

When you run “ls” this is what happens:



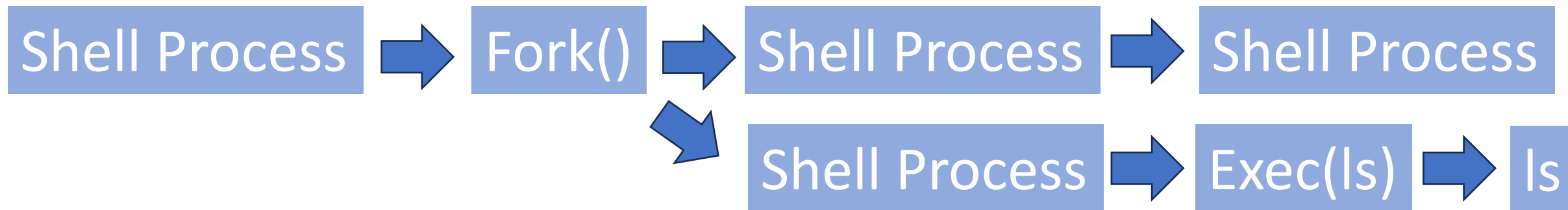
Process Tree

- These fork and exec commands create a tree structure of processes
- The root of this tree is the “init” process (we will talk about init at the end of this lecture)

For example,

When you run “ls” this is what happens:

One shell process replaced with “ls”



Let's print the process tree (systemd is the most popular Linux init program at the moment)

```
[matthew@moonshine ~]$ pstree -a
systemd --switched-root --system --deserialize 31
├─NetworkManager --no-daemon
│   └─2*[{NetworkManager}]
├─rdma-ndd --systemd
├─rsyslogd -n
│   └─2*[{rsyslogd}]
├─sshd
│   └─sshd
│       └─sshd
│           └─bash
│               └─pstree -a
├─dbus-broker --log 4 --controller 9 --machine-id...
├─firewalld -s /usr/sbin/firewalld --nofork --nopid
│   └─{firewalld}
└─systemd-udev
```

Let's print the process tree (we can see our own pstree process running under sshd and bash)

```
[matthew@moonshine ~]$ pstree -a
systemd --switched-root --system --deserialize 31
├─NetworkManager --no-daemon
│   └─2*[{NetworkManager}]
├─rdma-ndd --systemd
├─rsyslogd -n
│   └─2*[{rsyslogd}]
├─sshd
│   └─sshd
│       └─sshd
│           └─bash
│               └─pstree -a
├─dbus-broker --log 4 --controller 9 --machine-id...
├─firewalld -s /usr/sbin/firewalld --nofork --nopid
│   └─{firewalld}
└─systemd-udev
```

Linux is written in C and used the GNU C Library to access many Kernel Syscalls

```
[matthew@moonshine ~]$ ldd --version
```

```
ldd (GNU libc) 2.34
```

```
Copyright (C) 2021 Free Software Foundation, Inc.
```

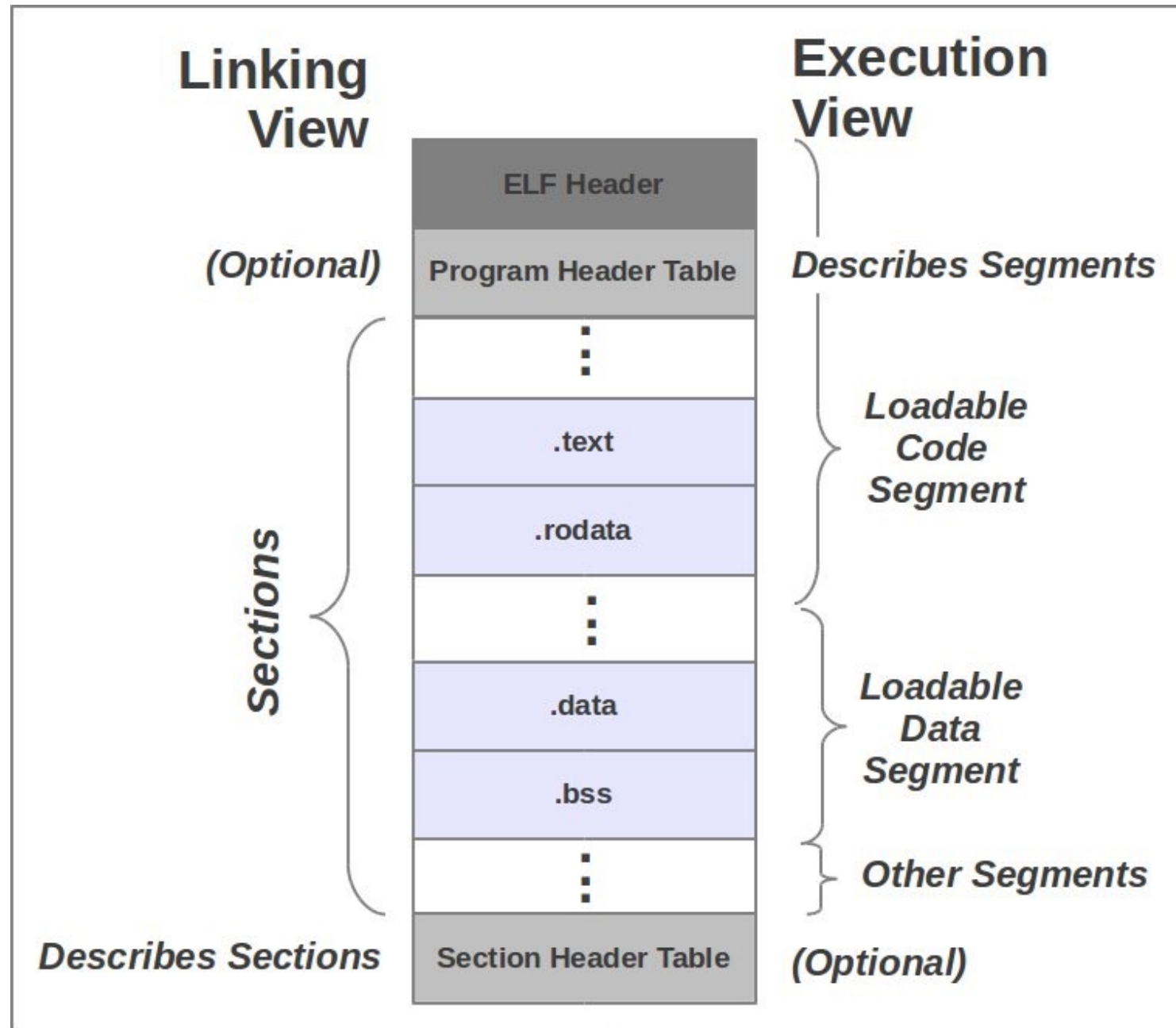
```
This is free software; see the source for copying  
conditions.  There is NO
```

```
warranty; not even for MERCHANTABILITY or FITNESS FOR A  
PARTICULAR PURPOSE.
```

```
Written by Roland McGrath and Ulrich Drepper.
```

Programs on Disk (Images*)

- Executables and Object files (parts of a compiled program that can be linked together) are stored on disk as ELF (Executable and Linkable Format).
- This binary file is called a “program image”.
- Program Images can be loaded into memory and executed with the `exec()` syscall.



*nothing to do with pictures

The GNU C Compiler writes executables in ELF

```
[matthew@moonshine ~]$ readelf --file-header helloworld
```

```
ELF Header:
```

```
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                   EXEC (Executable file)
  Machine:                                Advanced Micro Devices X86-64
  Version:                                0x1
  Entry point address:                   0x401090
  Start of program headers:              64 (bytes into file)
  Start of section headers:              24480 (bytes into file)
  Flags:                                  0x0
  Size of this header:                    64 (bytes)
  Size of program headers:                56 (bytes)
  Number of program headers:              13
  Size of section headers:                64 (bytes)
  Number of section headers:              31
  Section header string table index:     30
```

```
[matthew@moonshine ~]$ cat helloworld.cpp
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    cout << "Hello World" << endl;
```

```
    return 0;
```

```
}
```

```
[matthew@moonshine ~]$ cat helloworld.f90
```

```
program helloworld  
  print *, "Hello World"  
end program helloworld
```


The GNU C Compiler writes executables in ELF format

```
[matthew@moonshine ~]$ readelf --file-header helloworldf
```

```
ELF Header:
```

```
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                   EXEC (Executable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                                0x1
  Entry point address:                   0x401090
  Start of program headers:              64 (bytes into file)
  Start of section headers:              24480 (bytes into file)
  Flags:                                  0x0
  Size of this header:                    64 (bytes)
  Size of program headers:                56 (bytes)
  Number of program headers:              13
  Size of section headers:                64 (bytes)
  Number of section headers:              31
  Section header string table index:     30
```

The GNU Fortran Compiler writes executables in ELF too

```
[matthew@moonshine ~]$ readelf --file-header helloworldf
```

```
ELF Header:
```

```
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF64
  Data:                                   2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                   EXEC (Executable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x401090
  Start of program headers:              64 (bytes into file)
  Start of section headers:              24480 (bytes into file)
  Flags:                                  0x0
  Size of this header:                    64 (bytes)
  Size of program headers:                56 (bytes)
  Number of program headers:              13
  Size of section headers:                64 (bytes)
  Number of section headers:              31
  Section header string table index:      30
```

C ELF Helloworld – Text Section (Instructions)

```
[matthew@moonshine ~]$ readelf --hex-dump .text helloworld
```

```
Hex dump of section '.text':
```

```
0x00401090 f30f1efa 31ed4989 d15e4889 e24883e4 .....1.I..^H..H..
0x004010a0 f0505445 31c031c9 48c7c776 114000ff .PTE1.1.H..v.@..
0x004010b0 152b2f00 00f4662e 0f1f8400 00000000 .+/. . . .f . . . . .
0x004010c0 f30f1efa c3662e0f 1f840000 00000090 .....f . . . . .
0x004010d0 488d3d79 2f000048 8d05722f 00004839 H.=y/. .H..r/. .H9
0x004010e0 f8741548 8b05fe2e 00004885 c07409ff .t.H . . . . .H..t..
0x004010f0 e00f1f80 00000000 c30f1f80 00000000 .....
0x00401100 488d3d49 2f000048 8d35422f 00004829 H.=I/. .H.5B/. .H)
0x00401110 fe4889f0 48c1ee3f 48c1f803 4801c648 .H..H..?H...H..H
```

These are virtual addresses

Fortran ELF Helloworld – Text Section (Instructions)

```
[matthew@moonshine ~]$ readelf --hex-dump .text helloworldf
```

```
Hex dump of section '.text':
```

```
0x00401080 f30f1efa 31ed4989 d15e4889 e24883e4 .....1.I..^H..H..
0x00401090 f0505445 31c031c9 48c7c7d4 114000ff .PTE1.1.H....@..
0x004010a0 15532f00 00f4662e 0f1f8400 00000000 .S/...f.....
0x004010b0 f30f1efa c3662e0f 1f840000 00000090 .....f.....
0x004010c0 488d3d81 2f000048 8d057a2f 00004839 H.=./..H..z/..H9
0x004010d0 f8741548 8b050e2f 00004885 c07409ff .t.H.../..H..t..
0x004010e0 e00f1f80 00000000 c30f1f80 00000000 .....
0x004010f0 488d3d51 2f000048 8d354a2f 00004829 H.=Q/..H.5J/..H)
0x00401100 fe4889f0 48c1ee3f 48c1f803 4801c648 .H..H..?H...H..H
```

These are virtual addresses

Program Image Display Read-Only Data

```
[matthew@moonshine ~]$ readelf --hex-dump .rodata helloworldf
```

```
Hex dump of section '.rodata':
```

```
0x00402000 01000200 00000000 00000000 00000000 .....
```

```
0x00402010 68656c6c 6f776f72 6c642e66 39300048 helloworld.f90.H
```

```
0x00402020 656c6c6f 20576f72 6c640000 00000000 ello World.....
```

```
0x00402030 44080000 ff0f0000 00000000 01000000 D.....
```

```
0x00402040 01000000 00000000 1f000000
```

```
.....
```

```
[matthew@moonshine ~]$ readelf --hex-dump .rodata helloworld
```

```
Hex dump of section '.rodata':
```

```
0x00402000 01000200 00000000 00000000 00000000 .....
```

```
0x00402010 48656c6c 6f20576f 726c6400 Hello World.
```

Program Image Display Read-Only Data

```
[matthew@moonshine ~]$ readelf --hex-dump .rodata helloworldf
```

```
Hex dump of section '.rodata':
```

```
0x00402000 01000200 00000000 00000000 00000000 .....
```

In both programs we defined the output as a literal string, i.e. a constant: "Hello World"

```
39300048 helloworld.f90.H
```

```
00000000 ello World.....
```

```
01000000 D.....
```

```
0x00402040 01000000 00000000 1f000000
```

```
.....
```

```
[matthew@moonshine ~]$ readelf --hex-dump .rodata helloworld
```

```
Hex dump of section '.rodata':
```

```
0x00402000 01000200 00000000 00000000 00000000 .....
```

```
0x00402010 48656c6c 6f20576f 726c6400 Hello World.
```

You can see the calls to glibc in the “ls” binary

```
[matthew@moonshine ~]$ readelf --all /usr/bin/ls
```

```
Symbol table '.dynsym' contains 125 entries:
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	000000000000000000	0	NOTYPE	LOCAL	DEFAULT	UND	
1:	000000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	__[...]@GLIBC_2.3 (2)
2:	000000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	[...]@GLIBC_2.2.5 (3)
3:	000000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	cap_to_text
4:	000000000000000000	0	OBJECT	GLOBAL	DEFAULT	UND	[...]@GLIBC_2.2.5 (3)
5:	000000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	[...]@GLIBC_2.2.5 (3)
6:	000000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	[...]@GLIBC_2.3.4 (4)
7:	000000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	raise@GLIBC_2.2.5 (3)
8:	000000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	free@GLIBC_2.2.5 (3)
9:	000000000000000000	0	FUNC	GLOBAL	DEFAULT	UND	_[...]@GLIBC_2.34 (5)

The Kernel is also an ELF file names vmlinux. It is normally compressed into a vmlinuz file. You can find it in the /boot directory
Let's install a decompression tool.

```
[matthew@moonshine ~]$ sudo yum install kernel-devel
Last metadata expiration check: 0:01:43 ago on Wed 14 Feb 2024 10:40:13 AM CST.
Dependencies resolved.
=====
Package                                Architecture Size          Version
=====
Installing:
kernel-devel                            x86_64        5.14.0-
362.18.1.el9_3                          appstream    20 M
Upgrading:
openssl                                 x86_64        1:3.0.7-
25.el9_3                                baseos       1.2 M
openssl-libs                            x86_64
```


Create a directory under temp to store the decompressed kernel program image.

```
[matthew@moonshine ~]$ mkdir /tmp/kernel-extract
```

Uname -r returns the version of the running kernel

```
[matthew@moonshine ~]$ uname -r  
5.14.0-362.8.1.el9_3.x86_64
```

Copy the compressed kernel image into your temp directory

```
[matthew@moonshine ~]$ sudo cp /boot/vmlinuz-$(uname -r) /tmp/kernel-extract/
```

Copy the compressed kernel image into your temp directory

```
[matthew@moonshine ~]$ sudo cp /boot/vmlinuz-$(uname -r) /tmp/kernel-extract/
```

We captured output of the `uname -r` command and used it in a command with `$(uname -r)`.

`$(uname -r)` is substituted for `5.14.0-362.8.1.el9_3.x86_64`

Capturing output like this to use in a command is very common and useful.

Let's make sure the extract-vmlinux script was installed. We use the "file" command to see what type of file extract-vmlinux is.

```
[matthew@moonshine]$ file /usr/src/kernels/5.14.0-362.18.1.el9_3.x86_64/scripts/extract-vmlinux
```

```
This kernel source version might be later than the installed kernel - just tab  
complete the version you see in
```

```
When I tested this the path had version x.18 instead of version x.8
```

Let's make sure the `extract-vmlinux` script was installed. We use the “`file`” command to see what type of file `extract-vmlinux` is.

```
[matthew@moonshine]$ file /usr/src/kernels/5.14.0-362.18.1.el9_3.x86_64/scripts/extract-vmlinux
/usr/src/kernels/5.14.0-362.18.1.el9_3.x86_64/scripts/extract-vmlinux: a
/usr/bin/sh script, ASCII text executable
```

Let's make sure the extract-vmlinux script was installed. We use the "file" command to see what type of extract-vmlinux.

```
[matthew@moonshine ~]$ cd /tmp/kernel-extract/  
[matthew@moonshine kernel-extract]$ ls -lah  
total 13M  
drwxr-xr-x.  2 matthew matthew   64 Feb 14 10:50 .  
drwxrwxrwt. 11 root      root    4.0K Feb 14 10:43 ..  
-rwxr-xr-x.  1 root      root   13M Feb 14 10:45 vmlinux-5.14.0-362.8.1.el9_3.x86_64
```

Run the extract script on vmlinuz

```
[matthew@moonshine kernel-extract]$ /usr/src/kernels/5.14.0-362.18.1.el9_3.x86_64/scripts/extract-vmlinux vmlinux-5.14.0-362.8.1.el9_3.x86_64 > vmlinux
```



Send the output to a file

Run the extract script on vmlinuz and check it is there.

```
[matthew@moonshine kernel-extract]$ /usr/src/kernels/5.14.0-362.18.1.el9_3.x86_64/scripts/extract-vmlinux vmlinuz-5.14.0-362.8.1.el9_3.x86_64 > vmlinux
[matthew@moonshine kernel-extract]$ ls -lah
total 84M
drwxr-xr-x.  2 matthew matthew   64 Feb 14 10:50 .
drwxrwxrwt. 11 root     root    4.0K Feb 14 11:14 ..
-rw-r--r--.  1 matthew matthew  72M Feb 14 11:14 vmlinux
-rwxr-xr-x.  1 root     root   13M Feb 14 10:45 vmlinuz-5.14.0-362.8.1.el9_3.x86_64
```

Inspect the kernel image with readelf.

```
[matthew@moonshine kernel-extract]$ readelf --headers vmlinux
```

```
[matthew@moonshine kernel-extract]$ readelf -hex-dump .text vmlinux
```

```
[matthew@moonshine kernel-extract]$ readelf -hex-dump .rodata vmlinux
```

Inspect the kernel image with readelf.

```
[matthew@moonshine kernel-extract]$ readelf --headers vmlinux
```

```
[matthew@moonshine kernel-extract]$ readelf -hex-dump .text vmlinux
```

```
[matthew@moonshine kernel-extract]$ readelf -hex-dump .rodata vmlinux
```

The point of all this is to show you that the Kernel Image is just an ELF that can be loaded into memory like any other program you might write.

Pseudodevices

- Lastly the Kernel provides some convenient, but fake, devices.
- You have already seen `/dev/random`.

Overview of the Boot Process

1. BIOS (Basic-Input-Output System) loads with settings stored on a CMOS (Complementary Metal-Oxide Semiconductor) chip

and/or

UEFI (Unified Extended Firmware Interface) loads from NVRAM (Non-Volatile Random-Access Memory) chip.

Overview of the Boot Process

1. BIOS (Basic-Input-Output System) loads with settings stored on a CMOS (Complementary Metal-Oxide Semiconductor) chip

and/or

UEFI (Unified Extended Firmware Interface) loads from NVRAM (Non-Volatile Random-Access Memory) chip.

2. The BIOS tell the CPU to execute a program starting at the first byte of the MBR.

or

The UEFI tell the CPU to begin executing a boot loader program from the EFI partition.

Overview of the Boot Process

1. BIOS (Basic-Input-Output System) loads with settings stored on a CMOS (Complementary Metal-Oxide Semiconductor) chip

and/or

UEFI (Unified Extended Firmware Interface) loads from NVRAM (Non-Volatile Random-Access Memory) chip.

2. The BIOS tell the CPU to execute a program starting at the first byte of the MBR.

or

The UEFI tell the CPU to begin executing a boot loader program from the EFI partition.

3. The bootloader finds the kernel program, loads it into “Kernel Space” RAM and starts executing it. The bootloader finishes.

Overview of the Boot Process

1. BIOS (Basic-Input-Output System) loads with settings stored on a CMOS (Complementary Metal-Oxide Semiconductor) chip

and/or

UEFI (Unified Extended Firmware Interface) loads from NVRAM (Non-Volatile Random-Access Memory) chip.

2. The BIOS tell the CPU to execute a program starting at the first byte of the MBR.

or

The UEFI tell the CPU to begin executing a boot loader program from the EFI partition.

3. The bootloader finds the kernel program loads it into “Kernel Space” RAM and starts executing it. The bootloader finishes.

4. The Kernel discovers devices and loads the appropriate drivers.

Overview of the Boot Process

1. BIOS (Basic-Input-Output System) loads with settings stored on a CMOS (Complementary Metal-Oxide Semiconductor) chip

and/or

UEFI (Unified Extended Firmware Interface) loads from NVRAM (Non-Volatile Random-Access Memory) chip.

2. The BIOS tell the CPU to execute a program starting at the first byte of the MBR.

or

The UEFI tell the CPU to begin executing a boot loader program from the EFI partition.

3. The bootloader finds the kernel program loads it into “Kernel Space” RAM and starts executing it. The bootloader finishes.

4. The Kernel discovers devices in and attached to the computer and loads the appropriate drivers.

5. The Kernel mounts the root filesystem.

Overview of the Boot Process

1. BIOS (Basic-Input-Output System) loads with settings stored on a CMOS (Complementary Metal-Oxide Semiconductor) chip

and/or

UEFI (Unified Extended Firmware Interface) loads from NVRAM (Non-Volatile Random-Access Memory) chip.

2. The BIOS tell the CPU to execute a program starting at the first byte of the MBR.

or

The UEFI tell the CPU to begin executing a boot loader program from the EFI partition.

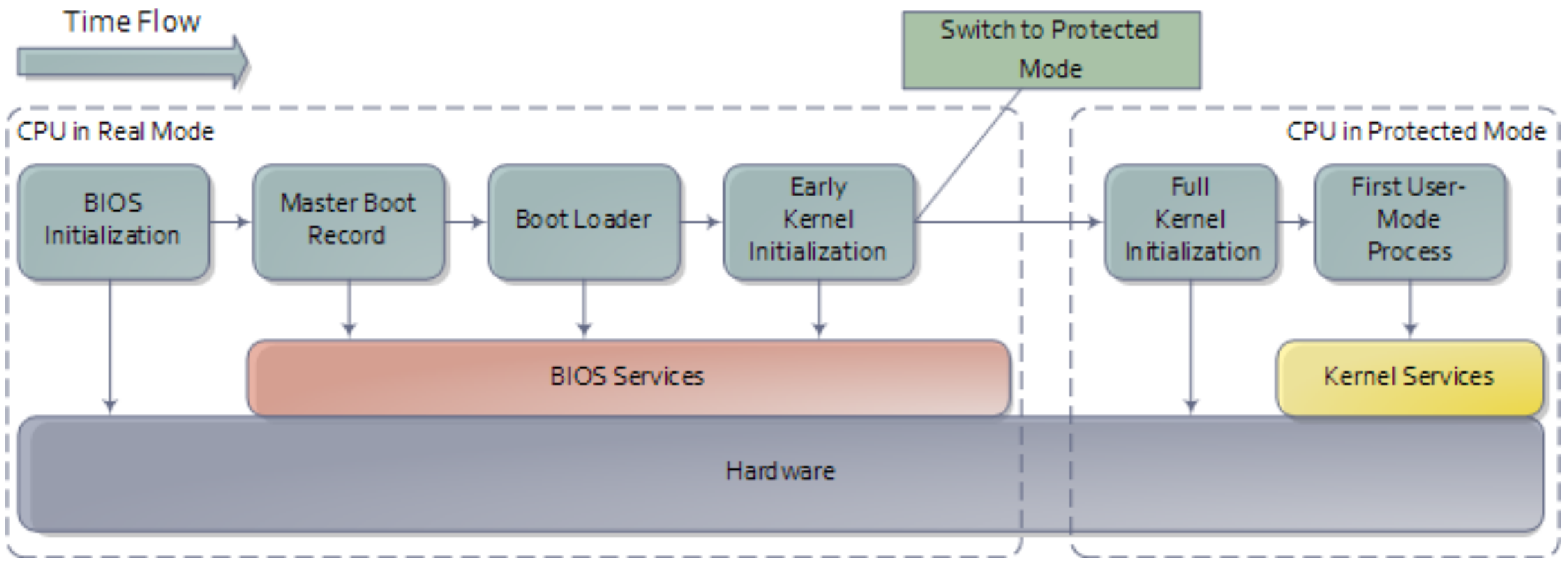
3. The bootloader finds the kernel program loads it into “Kernel Space” RAM and starts executing it. The bootloader finishes.

4. The Kernel discovers devices in and attached to the computer and loads the appropriate drivers.

5. The Kernel mounts the root filesystem.

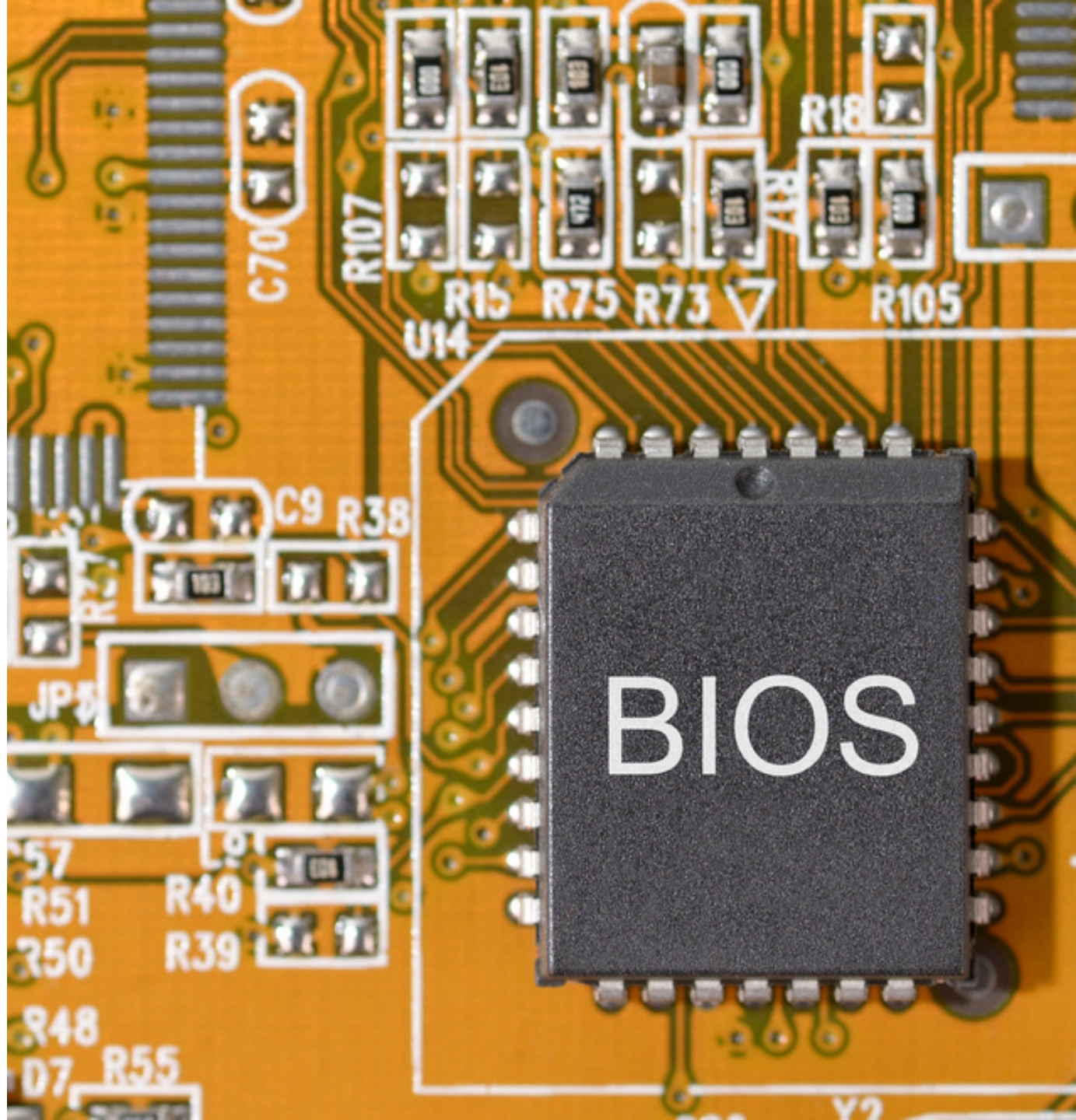
6. The Kernel loads “Init” (usually the systemd program) program into “User space” RAM.

Booting and the Kernel - Timeline

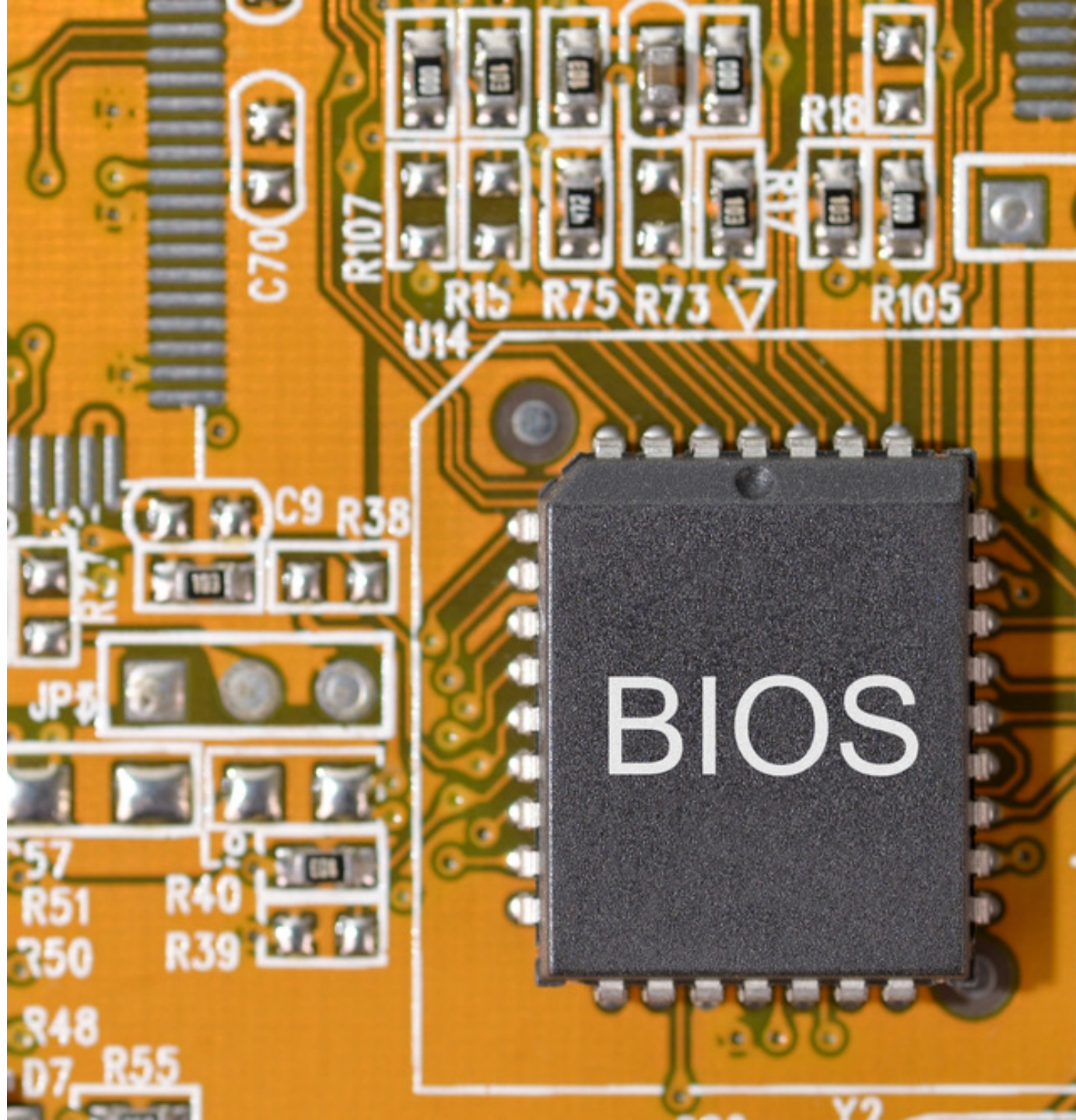


BIOS

- Basic I/O System
- First program that runs when you turn-on/reset the computer
- Initial interface between the hardware and the operating system
- Responsible for allowing you to control your computer's hardware settings for booting up
- In a multi-processor or multi-core system one CPU is dynamically chosen to be the bootstrap processor (BSP) that runs all of the BIOS and kernel initialization code, others are called application processors(AP)
 - So when these processors come into play?? Wait, we will get there!!



- BIOS ROM
 - Stored on EEPROM (programmable)
 - Called flash BIOS
- BIOS CMOS Memory
 - Non-volatile storage for boot-up settings
 - Need very little power to operate
 - Powered by lithium battery



x86 Compatible CPUs have 16 bit (real mode) and 32 bit (protected mode)

The x86 dominated the CPU market since 1978 when Intel introduces the 16-bit 8086 microprocessor. It will become an industry standard.



16bit: 64KB RAM
32bit: 4GB RAM



x64 was invented by Advanced Micro Devices (AMD)
64bits: 16 EB (exabyte=billion GB)
RAM



x86 Compatible CPUs have 16 bit and 32 bit modes

- All the x86 compatible chips maintain compatibility and go through their 16 bit and 32 bit modes before 64 bit.
- ARM CPUs (like your phone or a new Mac) don't have 16 bit mode and are not x86 compatible.



Basic BIOS Startup Routine

- Check CMOS setup for custom settings
- Load the interrupt handlers and device drivers
- Initialize registers and power management settings (ACPI)
- Initializes RAM
- POST (Power on Self-test)
- Display BIOS settings
- Determine which devices are bootable
- Initiate bootstrap sequence



IBM BIOS Assembly Language Source Code

```

;-----
; WRITE_TTY
; THIS INTERFACE PROVIDES A TELETYPE LIKE INTERFACE TO THE VIDEO :
; CARD. THE INPUT CHARACTER IS WRITTEN TO THE CURRENT CURSOR :
; POSITION, AND THE CURSOR IS MOVED TO THE NEXT POSITION. IF THE :
; CURSOR LEAVES THE LAST COLUMN OF THE FIELD, THE COLUMN IS SET :
; TO ZERO, AND THE ROW VALUE IS INCREMENTED. IF THE ROW VALUE :
; LEAVES THE FIELD, THE CURSOR IS PLACED ON THE LAST ROW, FIRST :
; COLUMN, AND THE ENTIRE SCREEN IS SCROLLED UP ONE LINE. WHEN :
; THE SCREEN IS SCROLLED UP, THE ATTRIBUTE FOR FILLING THE NEWLY :
; BLANKED LINE IS READ FROM THE CURSOR POSITION ON THE PREVIOUS :
; LINE BEFORE THE SCROLL, IN CHARACTER MODE. IN GRAPHICS MODE, :
; THE 0 COLOR IS USED. :
;-----
;----- WRITE THE CHAR TO THE SCREEN

        MOV     AH,10          ; WRITE CHAR ONLY
        MOV     CX,1          ; ONLY ONE CHAR
        INT     10H          ; WRITE THE CHAR

;----- POSITION THE CURSOR FOR NEXT CHAR

        INC     DL
        CMP     DL,BYTE PTR CRT_COLS      ; TEST FOR COLUMN OVERFLOW
        JNZ     U7          ; SET CURSOR
        MOV     DL,0          ; COLUMN FOR CURSOR
        CMP     DH,24
        JNZ     U6          ; SET_CURSOR_INC

;----- SCROLL REQUIRED

U1:
        MOV     AH,2
        INT     10H          ; SET THE CURSOR

; ENTRY
;
; (AH) = CURRENT CRT MODE
;
; (AL) = CHARACTER TO BE WRITTEN
;
; NOTE THAT BACK SPACE, CAR RET, BELL AND LINE FEED ARE HANDLED :
; AS COMMANDS RATHER THAN AS DISPLAYABLE GRAPHICS :
; (BL) = FOREGROUND COLOR FOR CHAR WRITE IF CURRENTLY IN A :
; GRAPHICS MODE :
;
; EXIT
;
; ALL REGISTERS SAVED
;-----
WRITE_TTY  ASSUME      : CS:CODE,DS:DATA
           PROC       NEAR
           PUSH      AX          ; SAVE REGISTERS
           PUSH      AX          ; SAVE CHAR TO WRITE
           MOV       AH,3
           MOV       BH,ACTIVE_PAGE      ; GET THE CURRENT ACTIVE PAGE
           INT      10H          ; READ THE CURRENT
           POP       AX          ; RECOVER CHAR
           CURSOR POSITION
           POP       AX
;----- DX NOW HAS THE CURRENT CURSOR POSITION

           CMP      AL,8          ; IS IT A BACKSPACE
           JE       U8          ; BACK_SPACE
           CMP     AL,0DH        ; IS IT CARRIAGE RETURN
           JE       U9          ; CAR_RET
           CMP     AL,0AH        ; IS IT A LINE FEED
           JE       U10         ; LINE_FEED
           CMP     AL,07H        ; IS IT A BELL
           JE       U11         ; BELL

```

Startup Messages (-k for this boot, -b for previous kernel boot logs)

```
[matthew@moonshine ~]$ sudo journalctl -k
Feb 06 20:21:02 localhost kernel: microcode: microcode updated early to revision 0x71a, date = 2020-03-24
Feb 06 20:21:02 localhost kernel: Linux version 5.14.0-362.8.1.el9_3.x86_64 (mockbuild@iad1-prod-build001.bld.equ.rockylinux.org) (gcc (GCC) 11.4.1 20230
Feb 06 20:21:02 localhost kernel: The list of certified hardware and cloud instances for Enterprise Linux 9 can be viewed at the Red Hat Ecosystem Catalo
Feb 06 20:21:02 localhost kernel: Command line: BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64 root=/dev/mapper/rl_dhcp52-root ro crashkernel=
Feb 06 20:21:02 localhost kernel: x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point registers'
Feb 06 20:21:02 localhost kernel: x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
Feb 06 20:21:02 localhost kernel: x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
Feb 06 20:21:02 localhost kernel: x86/fpu: xstate_offset[2]: 576, xstate_sizes[2]: 256
Feb 06 20:21:02 localhost kernel: x86/fpu: Enabled xstate features 0x7, context size is 832 bytes, using 'standard' format.
Feb 06 20:21:02 localhost kernel: signal: max sigframe size: 1776
Feb 06 20:21:02 localhost kernel: BIOS-provided physical RAM map:
Feb 06 20:21:02 localhost kernel: BIOS-e820: [mem 0x00000000000010000-0x0000000000009ffff] usable
Feb 06 20:21:02 localhost kernel: BIOS-e820: [mem 0x00000000000100000-0x000000000c42d4fff] usable
```

Kernel Init & Boot Options

```
Feb 06 20:21:02 localhost kernel: Built 2 zonelists, mobility grouping on. Total pages:
16491006
Feb 06 20:21:02 localhost kernel: Policy zone: Normal
Feb 06 20:21:02 localhost kernel: Kernel command line:
BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64
root=/dev/mapper/rl_dhcp52-root ro crash>
Feb 06 20:21:02 localhost kernel: Unknown kernel command line parameters
"BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64", will be passed to u
Feb 06 20:21:02 localhost kernel: mem auto-init: stack:off, heap alloc:off, heap free:off
Feb 06 20:21:02 localhost kernel: software IO TLB: area num 32.
Feb 06 20:21:02 localhost kernel: Memory: 2739448K/67011080K available (16384K kernel
code, 5596K rwdata, 11444K rodata, 3824K init, 18424K bss, 1954084K
Feb 06 20:21:02 localhost kernel: random: get_random_u64 called from
kmem_cache_open+0x1e/0x310 with crng_init=0
Feb 06 20:21:02 localhost kernel: SLUB: HWalig=64, Order=0-3, MinObjects=0, CPUs=32,
Nodes=2
Feb 06 20:21:02 localhost kernel: Kernel/User page tables isolation: enabled
```

The Kernel needs to know which partition contains the root filesystem

```
Feb 06 20:21:02 localhost kernel: Built 2 zonelists, mobility grouping on. Total pages: 16491006
Feb 06 20:21:02 localhost kernel: Policy zone: Normal
Feb 06 20:21:02 localhost kernel: Kernel command line: BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64 root=/dev/mapper/rl_dhcp52-root ro crash>
Feb 06 20:21:02 localhost kernel: Unknown kernel command line parameters "BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64", will be passed to u
Feb 06 20:21:02 localhost kernel: mem auto-init: stack:off, heap alloc:off, heap free:off
Feb 06 20:21:02 localhost kernel: software IO TLB: area num 32.
Feb 06 20:21:02 localhost kernel: Memory: 2739448K/67011080K available (16384K kernel code, 5596K rwd data, 11444K ro data, 3824K init, 18424K bss, 1954084K
Feb 06 20:21:02 localhost kernel: random: get_random_u64 called from kmem_cache_open+0x1e/0x310 with crng_init=0
Feb 06 20:21:02 localhost kernel: SLUB: Hwalign=64, Order=0-3, MinObjects=0, CPUs=32, Nodes=2
Feb 06 20:21:02 localhost kernel: Kernel/User page tables isolation: enabled
```

Any arguments the Kernel doesn't understand gets passed to init later. For example, `BOOT_IMAGE` is not something the Kernel understands.

```
Feb 06 20:21:02 localhost kernel: Built 2 zonelists, mobility grouping on. Total pages: 16491006
Feb 06 20:21:02 localhost kernel: Policy zone: Normal
Feb 06 20:21:02 localhost kernel: Kernel command line:
BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64
root=7dev/mapper/r1_dhcp52-root ro crash>
Feb 06 20:21:02 localhost kernel: Unknown kernel command line parameters
"BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64", will be passed to u
Feb 06 20:21:02 localhost kernel: mem auto-init: stack:off, heap alloc:off, heap free:off
Feb 06 20:21:02 localhost kernel: software IO TLB: area num 32.
Feb 06 20:21:02 localhost kernel: Memory: 2739448K/67011080K available (16384K kernel
code, 5596K rwddata, 11444K rodata, 3824K init, 18424K bss, 1954084K
Feb 06 20:21:02 localhost kernel: random: get_random_u64 called from
kmem_cache_open+0x1e/0x310 with crng_init=0
Feb 06 20:21:02 localhost kernel: SLUB: HWalig=64, Order=0-3, MinObjects=0, CPUs=32,
Nodes=2
Feb 06 20:21:02 localhost kernel: Kernel/User page tables isolation: enabled
```

The kernel eventually starts the init process (systemd)

```
[matthew@moonshine ~]$ sudo journalctl -k | grep -A15 "init process"
```




Get the Kernel boot log

The kernel eventually starts the init process (systemd)

```
[matthew@moonshine ~]$ sudo journalctl -k | grep -A15 "init process"
```



Get the Kernel boot log




"Pipe" the output
to grep

The kernel eventually starts the init process (systemd)


```
[matthew@moonshine ~]$ sudo journalctl -k | grep -A15 "init process"
```



Get the Kernel boot log



"Pipe" the output to grep



Filter the output on "init process"

The kernel eventually starts the init process (systemd)

```
[matthew@moonshine ~]$ sudo journalctl -k | grep -A15 "init process"
```



Print 15 lines
After the line that matches
"init process"

The kernel eventually starts the init process (systemd)

```
[matthew@moonshine ~]$ sudo journalctl -k | grep -A15 "init process"
Feb 06 20:21:02 localhost kernel: Run /init as init process
Feb 06 20:21:02 localhost kernel:   with arguments:
Feb 06 20:21:02 localhost kernel:     /init
Feb 06 20:21:02 localhost kernel:   with environment:
Feb 06 20:21:02 localhost kernel:     HOME=/
Feb 06 20:21:02 localhost kernel:     TERM=linux
Feb 06 20:21:02 localhost kernel:     BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64
Feb 06 20:21:02 localhost kernel: ERST: NVRAM ERST Log Address Range not implemented yet.
Feb 06 20:21:02 localhost kernel: usb 1-1.5: new low-speed USB device number 4 using ehci-pci
Feb 06 20:21:02 localhost systemd[1]: systemd 252-18.el9 running in system mode (+PAM +AUDIT +SELINUX -
APPARMOR +IMA +SMACK +SECCOMP +GCRYPT +GNUTLS +OPENSSL +ACL +BLKID +CURL +ELFUTILS -FIDO2 +IDN2 -IDN -
IPTC +KMOD +LIBCRYPTSETUP +LIBFDISK +PCRE2 -PWQUALITY +P11KIT -QRENCODE +TPM2 +BZIP2 +LZ4 +XZ +ZLIB
+ZSTD -BPF_FRAMEWORK +XKBCOMMON +UTMP +SYSVINIT default-hierarchy=unified)
Feb 06 20:21:02 localhost systemd[1]: Detected architecture x86-64.
Feb 06 20:21:02 localhost systemd[1]: Running in initrd.
Feb 06 20:21:02 localhost systemd[1]: No hostname configured, using default hostname.
Feb 06 20:21:02 localhost systemd[1]: Hostname set to <localhost>.
Feb 06 20:21:02 localhost kernel: usb 1-1.5: New USB device found, idVendor=413c, idProduct=2105,
bcdDevice= 3.52
Feb 06 20:21:02 localhost kernel: usb 1-1.5: New USB device strings: Mfr=1, Product=2, SerialNumber=0
```

The kernel eventually starts the init process (systemd)

```
[matthew@moonshine ~]$ sudo journalctl -k | grep -A15 "init process"
Feb 06 20:21:02 localhost kernel: Run /init as init process
Feb 06 20:21:02 localhost kernel:   with arguments:
Feb 06 20:21:02 localhost kernel:     /init
Feb 06 20:21:02 localhost kernel:   with environment:
Feb 06 20:21:02 localhost kernel:     HOME=/
Feb 06 20:21:02 localhost kernel:     TERM=linux
Feb 06 20:21:02 localhost kernel:     BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-362.8.1.el9_3.x86_64
Feb 06 20:21:02 localhost kernel: ERST: NVRAM ERST Log Address Range not implemented yet.
Feb 06 20:21:02 localhost kernel: usb 1-1.5: new low-speed USB device number 4 using ehci-pci
Feb 06 20:21:02 localhost systemd[1]: systemd 252-18.el9 running in system mode (+PAM +AUDIT +SELINUX
-APPARMOR +IMA +SMACK +SECCOMP +GCRYPT +GNUTLS +OPENSSL +ACL +BLKID +CURL +ELFUTILS -FIDO2 +IDN2 -IDN
-IPTC +KMOD +LIBCRYPTSETUP +LIBFDISK +PCRE2 -PWQUALITY +P11KIT -QRENCODE +TPM2 +BZIP2 +LZ4 +XZ +ZLIB
+ZSTD -BPF_FRAMEWORK +XKBCOMMON +UTMP +SYSVINIT default-hierarchy=unified)
Feb 06 20:21:02 localhost systemd[1]: Detected architecture x86-64.
Feb 06 20:21:02 localhost systemd[1]: Running in initrd.
Feb 06 20:21:02 localhost systemd[1]: No hostname configured, using default hostname.
Feb 06 20:21:02 localhost systemd[1]: Hostname set to <localhost>.
Feb 06 20:21:02 localhost kernel: usb 1-1.5: New USB device found, idVendor=413c, idProduct=2105,
bcdDevice= 3.52
Feb 06 20:21:02 localhost kernel: usb 1-1.5: New USB device strings: Mfr=1, Product=2, SerialNumber=0
```

You can see the kernel's boot arguments with

```
[matthew@moonshine ~]$ cat /proc/cmdline
BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-
362.8.1.el9_3.x86_64 root=/dev/mapper/r1_dhcp52-root ro
crashkernel=1G-4G:192M,4G-64G:256M,64G-:512M
resume=/dev/mapper/r1_dhcp52-swap
rd.lvm.lv=r1_dhcp52/root rd.lvm.lv=r1_dhcp52/swap
```

Kernel Space vs User Space RAM

- This separation of RAM spaces can be enforced at the hardware level (e.g. on x86 chips SMEP/SMAP)

Boot Loaders

- The boot loader (remember it's in the MBR or EFI) starts the kernel.
- To do this it needs two things:
 1. Where is the Kernel image located
 2. What arguments should be passed to the Kernel
 3. The Kernel is usually in the root filesystem, but it's the Kernel's job to know about filesystems and devices like harddrives and its not loaded yet.

To get around this BIOS and UEFI systems provide an LBA (Logical Block Addressing) interface to the hard disk. This works but is very slow. After the Kernel is loaded it can use high-speed drivers.

The bootloader used LBA to open and read the Kernel image into RAM and the Kernel starts executing.



You can check the boot settings (and if it uses EFI)

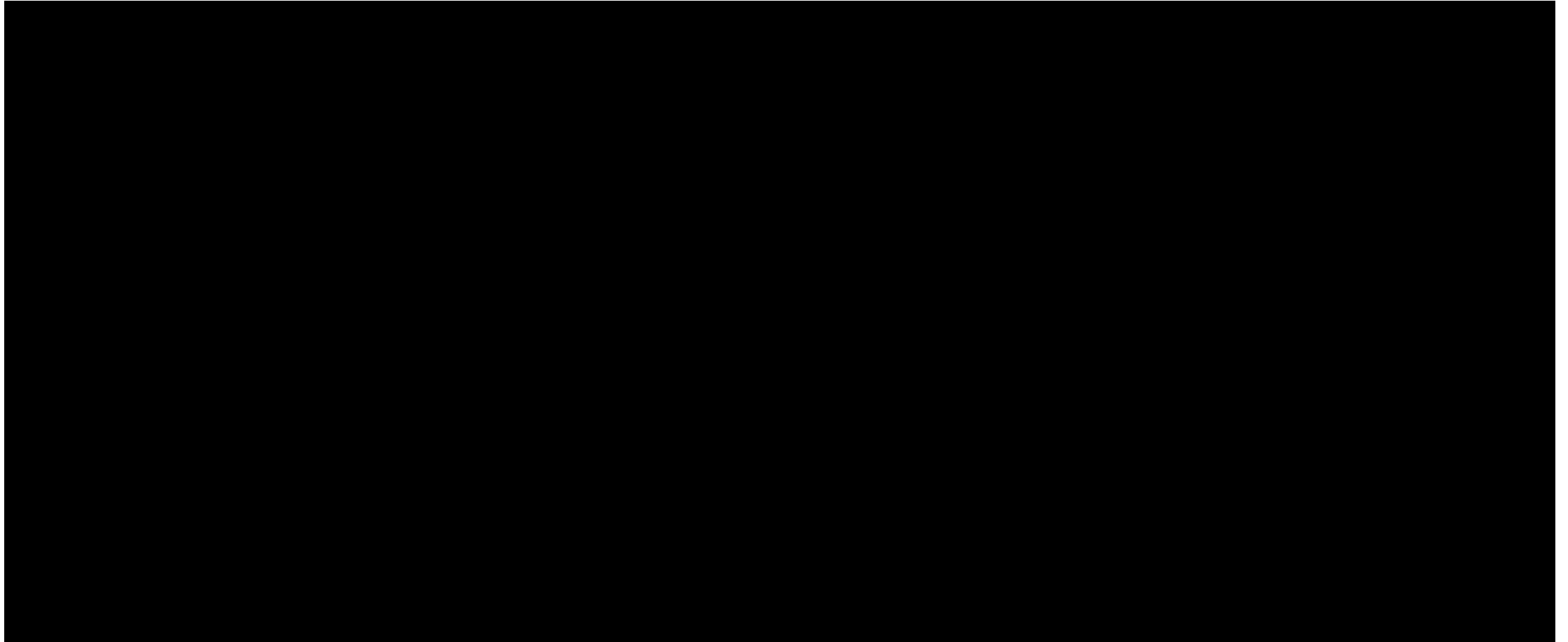
```
[matthew@moonshine ~]$ efibootmgr
BootCurrent: 0007
Timeout: 0 seconds
BootOrder: 0007,0006,0000,0008,0001,0002,0003,0004,0005
Boot0000* HL-DT-ST DVD-ROM DU30N
Boot0001* Broadcom NetXtreme Gigabit Ethernet (BCM5720)
Boot0002* Broadcom NetXtreme Gigabit Ethernet (BCM5720)
Boot0003* Broadcom NetXtreme Gigabit Ethernet (BCM5720)
Boot0004* Broadcom NetXtreme Gigabit Ethernet (BCM5720)
Boot0005* USB DISK 3.0
Boot0006* Windows Boot Manager
Boot0007* Rocky Linux
Boot0008* EFI Fixed Disk Boot Device 1
```

The Grand Unified Bootloader

- Grub is the most common boot loader program.
- It is just powerful enough to understand partitions tables and how to use LBA to read files from disk.
- It is able to display a menu to the user in case they want to select from multiple Linux Kernels, or give Kernels different options, or even boot from Windows.

When troubleshooting a Linux boot problem, the first program you have to interact with will be the bootloader.

Grub2 Showing Rocky 9 Boot Options



GRUB2

- Grub is only able to read disks with LBA, but that's enough to read long config files stored outside the MBR if needed.

```
[matthew@moonshine ~]$ sudo ls /boot/grub2/  
[sudo] password for matthew:  
fonts  grub.cfg  grubenv
```

GRUB2

```
[matthew@moonshine ~]$ sudo less /boot/grub2/grub.cfg
```

```
#  
# DO NOT EDIT THIS FILE  
#  
# It is automatically generated by grub2-mkconfig using templates  
# from /etc/grub.d and settings from /etc/default/grub  
#  
  
### BEGIN /etc/grub.d/00_header ###  
set pager=1  
  
if [ -f ${config_directory}/grubenv ]; then  
    load_env -f ${config_directory}/grubenv  
elif [ -s $prefix/grubenv ]; then  
    load_env  
fi  
if [ "${next_entry}" ] ; then  
    set default="${next_entry}"  
    set next_entry=  
    save_env next_entry  
    set boot_once=true  
else  
    set default="${saved_entry}"  
fi
```

GRUB2 – Config File

```
[matthew@moonshine ~]$ sudo cat /etc/default/grub
GRUB_TIMEOUT=5
GRUB_DISTRIBUTOR="$(sed 's, release .*$, ,g' /etc/system-release)"
GRUB_DEFAULT=saved
GRUB_DISABLE_SUBMENU=true
GRUB_TERMINAL_OUTPUT="console"
GRUB_CMDLINE_LINUX="crashkernel=1G-4G:192M,4G-64G:256M,64G- :512M
resume=/dev/mapper/r1_dhcp52-swap rd.lvm.lv=r1_dhcp52/root
rd.lvm.lv=r1_dhcp52/swap"
GRUB_DISABLE_RECOVERY="true"
GRUB_ENABLE_BLSCFG=true
```

GRUB2 – Config Directory

```
[matthew@moonshine ~]$ sudo ls -l /etc/grub.d/
total 104
-rwxr-xr-x. 1 root root  9346 Nov  8 08:28 00_header
-rwxr-xr-x. 1 root root   236 Nov  8 08:28 01_users
-rwxr-xr-x. 1 root root   835 Nov  8 08:28 08_fallback_counting
-rwxr-xr-x. 1 root root 19665 Nov  8 08:28 10_linux
-rwxr-xr-x. 1 root root   833 Nov  8 08:28 10_reset_boot_success
-rwxr-xr-x. 1 root root   892 Nov  8 08:28 12_menu_auto_hide
-rwxr-xr-x. 1 root root   410 Nov  8 08:28 14_menu_show_once
-rwxr-xr-x. 1 root root 13613 Nov  8 08:28 20_linux_xen
-rwxr-xr-x. 1 root root   2562 Nov  8 08:28 20_ppc_terminfo
-rwxr-xr-x. 1 root root 10869 Nov  8 08:28 30_os-prober
-rwxr-xr-x. 1 root root   1122 Nov  8 08:28 30_uefi-firmware
-rwxr-xr-x. 1 root root   725 Nov  8 07:29 35_fwupd
-rwxr-xr-x. 1 root root   218 Nov  8 08:28 40_custom
-rwxr-xr-x. 1 root root   219 Nov  8 08:28 41_custom
-rw-r--r--. 1 root root   483 Nov  8 08:28 README
```

GRUB2 – For UEFI Systems with an GPT Partition Scheme and EFI Partition

```
[matthew@moonshine ~]$ sudo file  
/boot/efi/EFI/rocky/grubx64.efi  
/boot/efi/EFI/rocky/grubx64.efi: PE32+ executable (EFI application) x86-  
64 (stripped to external PDB), for MS Windows
```

Windows 64-bit Program (PE is Portable Executable) not ELF format.

This is because the EFI Bootloader has to be Windows compatible to boot Windows.

GRUB2 – For UEFI Systems with an GPT Partition Scheme and EFI Partition

```
[matthew@moonshine ~]$ sudo objdump -x -D /boot/efi/EFI/rocky/grubx64.efi | head -n 20
```

```
/boot/efi/EFI/rocky/grubx64.efi:      file format pei-x86-64  
/boot/efi/EFI/rocky/grubx64.efi  
architecture: i386:x86-64, flags 0x00000103:  
HAS_RELOC, EXEC_P, D_PAGED  
start address 0x0000000000001000
```

```
Characteristics 0x20e  
executable  
line numbers stripped  
symbols stripped  
debugging information removed
```

```
Time/Date Wed Dec 31 18:00:00 2014  
Magic 020b (PE32+)  
MajorLinkerVersion 0  
MinorLinkerVersion 0  
SizeOfCode 000000000019000  
SizeOfInitializedData 000000000024e000  
SizeOfUninitializedData 0000000000000000
```

GRUB2 – Here is some of the bootloader assembly code.

```
[matthew@moonshine ~]$ sudo objdump -x -D /boot/efi/EFI/rocky/grubx64.efi
```

```
180be8: 83 e0 01          and     $0x1,%eax
180beb: 89 c2            mov     %eax,%edx
180bed: 0f b6 45 e0      movzbl -0x20(%rbp),%eax
180bf1: 83 e0 fe          and     $0xfffffffffe,%eax
180bf4: 09 d0            or      %edx,%eax
180bf6: 88 45 e0          mov     %al,-0x20(%rbp)
180bf9: 48 8b 45 c8      mov     -0x38(%rbp),%rax
180bfd: 48 83 c0 2e      add     $0x2e,%rax
180c01: 48 8d 55 e0      lea    -0x20(%rbp),%rdx
180c05: 48 83 c2 08      add     $0x8,%rdx
180c09: 48 89 d6          mov     %rdx,%rsi
180c0c: 48 89 c7          mov     %rax,%rdi
180c0f: 48 b8 00 00 00 00 movabs $0x0,%rax
```


A tiny bootloader and kernel

Git Clone A little Kernel

```
mfricke@wheeler$ git clone https://github.com/gmfricke/bootkernel.git
mfricke@wheeler$ cd bootkernel
[mfricke@wheeler bootkernel]$ ls -lh
total 24K
-rw-r--r--. 1 matthew matthew 261 Feb 14 12:00 boothello.asm
-rw-r--r--. 1 matthew matthew 1.4K Feb 14 12:00 bootsect.asm
-rw-r--r--. 1 matthew matthew 445 Feb 14 12:00 kernel.c
-rw-r--r--. 1 matthew matthew 186 Feb 14 12:00 kernel_entry.asm
-rw-r--r--. 1 matthew matthew 742 Feb 14 12:00 Makefile
-rw-r--r--. 1 matthew matthew 43 Feb 14 12:00 README.md
drwxr-xr-x. 2 matthew matthew 158 Feb 14 12:00 utils
```

```
mfricke@wheeler:~/bootkernel [main ?]$ module load nasm qemu
```

```
mfricke@wheeler:~/bootkernel [main ?]$ module load nasm qemu
```

Loading the Network Assembler (most popular assembly program) and qemu. Qemu is an emulator that will let us run our little bootloader and kernel without having to burn a disk.

```
mfricke@wheeler:~/bootkernel [main ?]$ cat Makefile
# Some useful Makefile abbreviations
# $@ = target file
# $< = first dependency
# $^ = all dependencies
\

# First rule is the one executed when no parameters are fed to the Makefile
all: run

# Notice how dependencies are built as needed
kernel.bin: kernel_entry.o kernel.o
    ld -m elf_i386 -s -o $@ -Ttext 0x1000 $^ --oformat binary

kernel_entry.o: kernel_entry.asm
    nasm $< -f elf -o $@

kernel.o: kernel.c
    gcc -fno-pic -m32 -ffreestanding -c $< -o $@

# Rule to disassemble the kernel - may be useful to debug
kernel.dis: kernel.bin
    ndisasm -b 32 $< > $@

bootsect.bin: bootsect.asm
    nasm $< -f bin -o $@

boothello.bin: bootsect.asm
    nasm $< -f bin -o $@

os-image.bin: bootsect.bin kernel.bin
    cat $^ > $@

run: os-image.bin
    qemu-system-i386 -curses -fda $<

clean:
    rm *.bin *.o *.dis
```

```
mfricke@wheeler:~/bootkernel [main]$ cat boothello.asm
```

```
mov ah, 0x0e ; tty  
mode
```

```
mov al, 'H'  
int 0x10  
mov al, 'e'  
int 0x10  
mov al, 'l'  
int 0x10  
int 0x10 ; 'l' is still on al,  
remember?
```

```
mov al, 'o'  
int 0x10
```

```
jmp $ ; jump to current address = infinite  
loop
```

```
; padding and magic  
number
```

```
times 510 - ($-$$) db 0
```

```
dw 0xaa55
```

```
mfricke@wheeler:~/bootkernel [main]$ make boothello.bin  
nasm bootsect.asm -f bin -o boothello.bin
```

This is gets compiled into binary. It's the MBR bootloader.

Launch an interactive shell on a Wheeler compute node

```
mfricke@wheeler:~ $ qgrok
```

queues	free	busy	offline	jobs	nodes	CPUs	GPUs	CPUs/node	GPUs/node	Memory/node	time_limit	CPU_limit	GPU_limit	RAM_limit
normal	71	224	5	45	300	2400	0	8	0	48G	2-00:00:00	400	0	2415G
debug	4	0	0	0	4	32	0	8	0	48G	4:00:00	16	0	96580M
totals:	75	224	5	45	304	2432	0							

First, we check to make sure there are CPUs available for us to use with qgrok.

Launch an interactive shell on a Wheeler compute node

```
mfricke@wheeler:~ $ qgrok
```

queues	free	busy	offline	jobs	RAM_limit
-----	-----	-----	-----	-----	-----
normal	71	224	5	45	2415G
debug	4	0	0	0	96580M
totals:	75	224	5	45	

First, we check to make sure there are CPUs available for us to use with qgrok.

Launch an interactive shell on a Wheeler compute node

```
mfricke@wheeler:~/bootkernel $ srun --pty bash  
mfricke@wheeler005:~/bootkernel $
```

`srun` is a SLURM command that allocates compute resources. `--pty` creates a pseudoterminal. `Bash` is the process we start on the compute node.

```
mfricke@wheeler005:~/bootkernel $ qemu-system-i386 -curses -hda boothello.bin
```

Qemu emulates a standard BIOS.

We set our boot disk to be the code we compiled.

Boothello meets the requirements for an MBR bootloader

```
mfricke@wheeler005:~/bootkernel $ qemu-system-i386 -curses -hda boothello.bin
```

After running this command. You can kill the process by closing your terminal.

You will have to login to wheeler again to run the next part.

```

[org 0x7c00]
KERNEL_OFFSET equ 0x1000 ; The same one we used when linking
the kernel

    mov [BOOT_DRIVE], dl ; The BIOS sets the boot drive in
'dl' on boot
    mov bp, 0x9000
    mov sp, bp

    mov bx, MSG_REAL_MODE
    call print
    call print_nl

    call load_kernel ; read the kernel from
disk

    call switch_to_pm ; disable interrupts, load GDT, etc.
Finally jumps to 'BEGIN_PM'
    jmp $ ; Never
executed

[bits 16]
load_kernel:
    mov bx, MSG_LOAD_KERNEL
    call print
    call print_nl

    mov bx, KERNEL_OFFSET ; Read from disk and store in
0x1000
    mov dh, 2
    mov dl, [BOOT_DRIVE]
    call disk_load
    ret

```

```

[bits 32]
BEGIN_PM:
    mov ebx, MSG_PROT_MODE
    call print_string_pm
    call KERNEL_OFFSET ; Give control to the
kernel

    jmp $ ; Stay here when the kernel returns control
to us (if
ever)

BOOT_DRIVE db 0 ; It is a good idea to store it in
memory because 'dl' may get
overwritten
MSG_REAL_MODE db "Started in 16-bit Real Mode", 0
MSG_PROT_MODE db "Landed in 32-bit Protected Mode", 0
MSG_LOAD_KERNEL db "Loading kernel into memory", 0

;
padding

times 510 - ($-$$) db 0
dw 0xaa55

```

This bootloader loads the kernel and jumps to address 0x1000 where the kernel is in RAM.

```
/* This will force us to create a kernel entry function instead of jumping
to kernel.c:0x00 */
void dummy_test_entrypoint() {
}

void main() {
    char text[] = "My CS499 Kernel";
    unsigned int text_length = sizeof(text)/sizeof(char);

    char* video_memory = (char*) 0xb8000;
    unsigned int i;
    for (i = 0; i < text_length; i++)
    {
        *video_memory = text[i];
        video_memory=video_memory+2; // 2 because to skip colour
bytes
    }
}
```

This kernel was compiled into machine code and added to os-image.bin.
When our bootloader runs this kernel it prints some text.

```
mfricke@wheeler:~/bootkernel $ module load nasm qemu
mfricke@wheeler:~/bootkernel $ srun --pty bash
mfricke@wheeler005:~/bootkernel $ make
```

```
My CS499 Kernel Protected Mode(a5cab58e9a3f-prebuilt.qemu.org)
```

```
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F91510+07EF1510 C980
```

```
Booting from Hard Disk...
```

```
Boot failed: could not read the boot disk
```

```
Booting from Floppy...
```

```
Started in 16-bit Real Mode
```

```
Loading kernel into memory
```



This kernel was compiled into machine code and added to os-image.bin.
When our bootloader runs this kernel it prints some text.

```
mfricke@wheeler:~/bootkernel [main]$ make
```

```
My CS499 Kernel Protected Modea5cab58e9a3f-prebuilt.qemu.org)
```

```
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F91510+07EF1510 C980
```

```
Booting from Hard Disk...
```

```
Boot failed: could not read the boot disk
```

```
Booting from Floppy...
```

```
Started in 16-bit Real Mode
```

```
Loading kernel into memory
```



After running this command. You can kill the process by closing your terminal.

This kernel was compiled into machine code and added to os-image.bin.
When our bootloader runs this kernel it prints some text.

Conclusions

- Now you know a lot about how Linux starts up and the role of the Kernel.
- Next you will learn about user space and how to manage system services.