

Delivering science and technology
to protect our nation
and promote world stability

Overview of Linux

Theory of Linux with a focus on HPC

Presented by CSCNSI



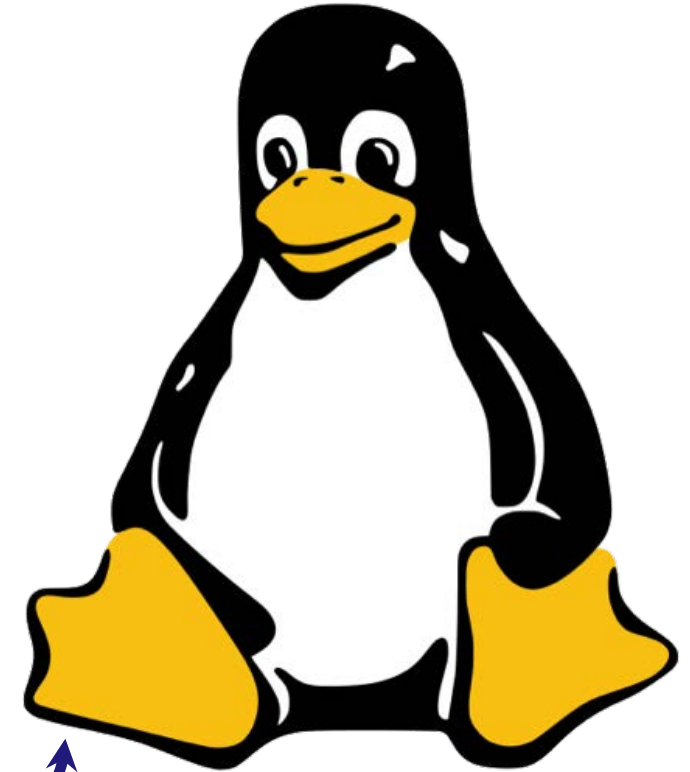
Outline

1. What is Linux?
2. The Linux Kernel
3. Linux distributions

What is Linux?

What's in a name?

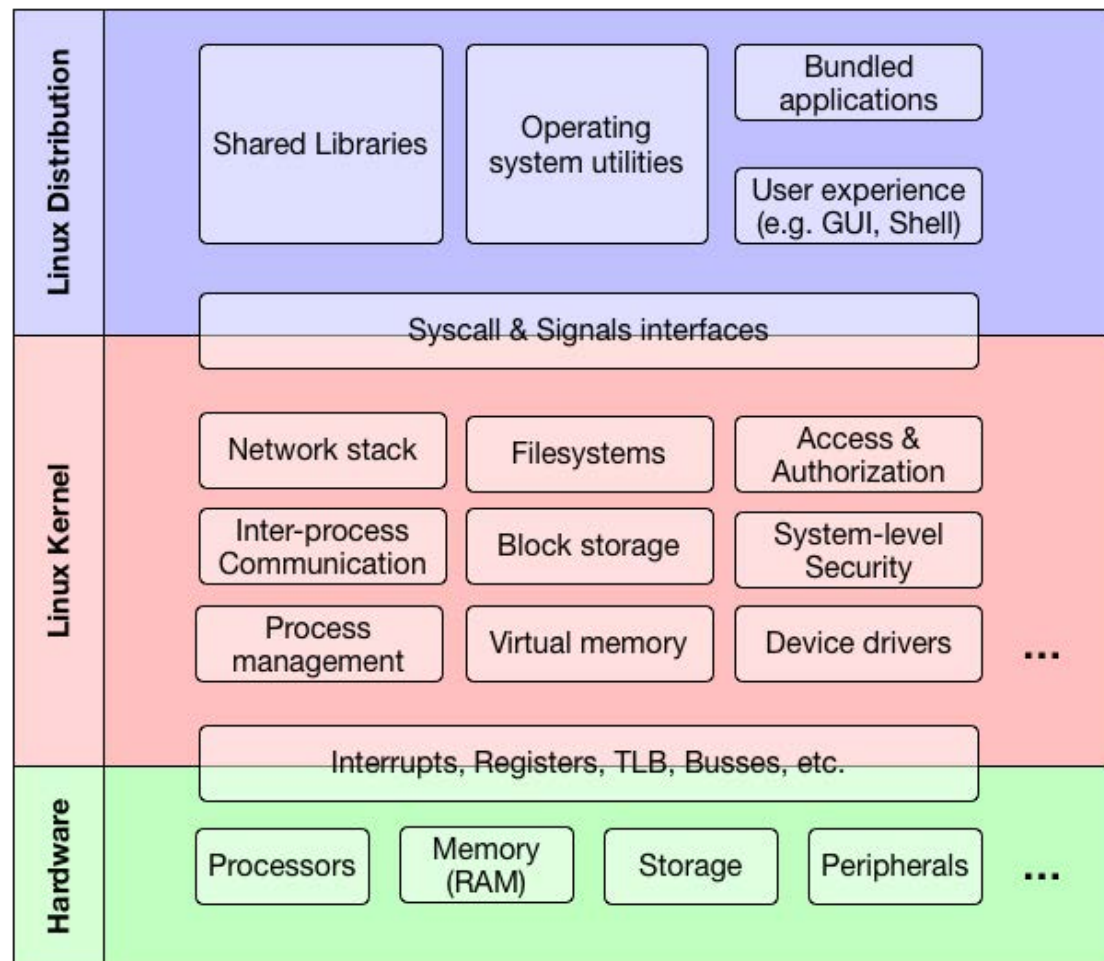
- The term “**Linux**” is overloaded. People often mean one of three things:
 - The Linux **kernel**—the core of the operating system
 - A Linux **distribution**—a bundle of software centered around the Linux kernel
 - The Linux software **community**—a vast and vibrant grouping of (mostly) open source development groups
 - *...and probably other things too.*



This is “Tux” the penguin. Tux is the official Linux mascot.

Linux vs. Linux distribution

- The Linux Kernel is the core of Linux
 - Provides interface between software and hardware
 - Provides process scheduling, memory access, hardware drivers, filesystems, and much more
- A Linux distribution is a bundle of software centered around the Linux kernel
 - To have a usable system we need more than the kernel, so various Linux Distributions exist that bundle needed tools
 - Red Hat, CentOS, Scientific, Debian, Ubuntu, Slackware, Gentoo, Arch, Mint, Raspbian, and so many more...



The Linux community



The Open Source community drives the development of the Linux

Special mention: **GNU** (pronounced *g'noo*)

- Much of the system software typically used with the Linux kernel is part of the GNU project. All of the GNU project is free, open source software. This combination is often called **GNU/Linux**.

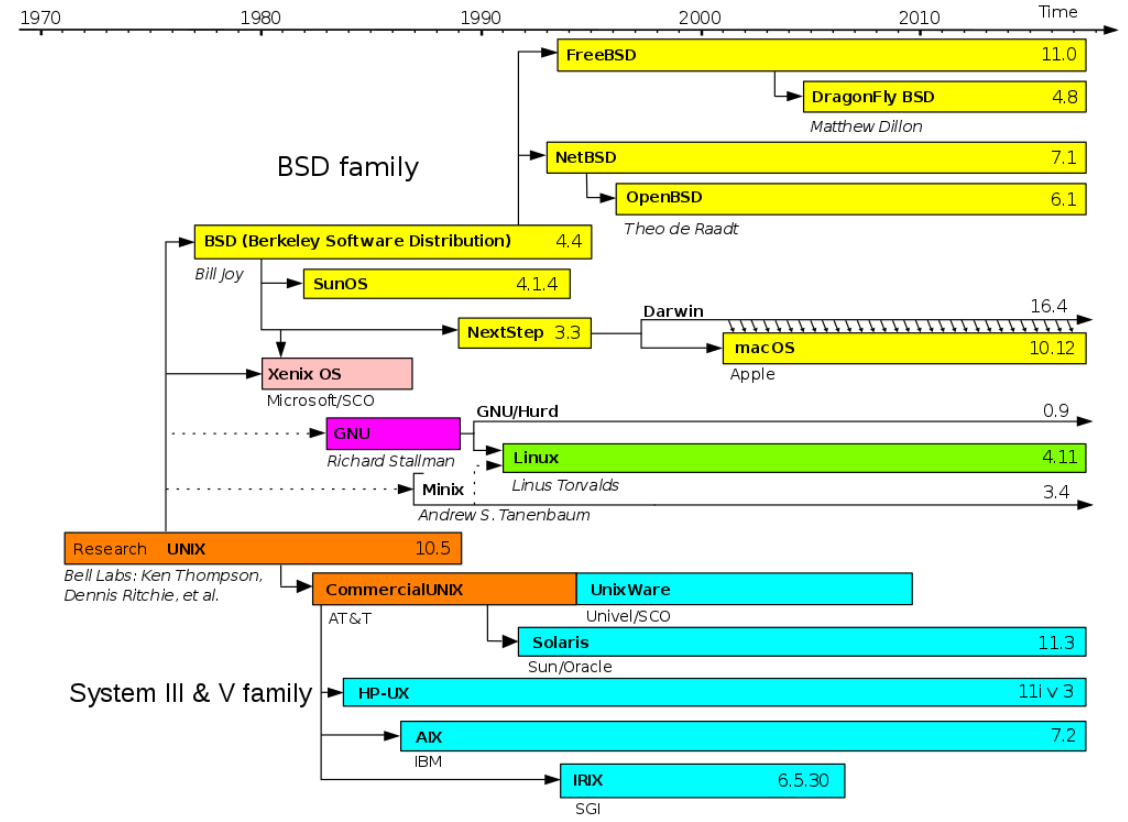
- Many other communities:

- Linux Foundation
- Free Software Foundation (FSF)
- Apache Software Foundation
- GNOME Foundation
- KDE e.V.
- Mozilla Foundation
- Open Source for America (OSFA)
- OpenStack Foundation
- ...many more

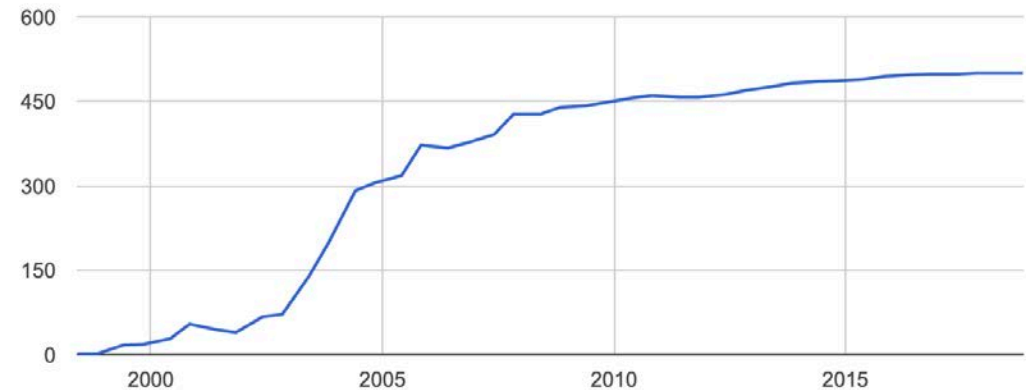
Plus lots of communities and individual developers at places like github, SourceForge,...

History of Linux in HPC

- 1991—Linux kernel is written by Linus Torvaldes (a 21yr old student).
- 1994—First compute cluster, “Beowulf,” created at NASA.
- 1996—Linux kernel v2 released, supports SMP (symmetric multiprocessing).
- 1998—First Linux cluster on the top500 list.
- 2004—Cray (the supercomputer company) releases its first Linux-based systems.
- 2012—Linux runs on all of the top10 of the top500.



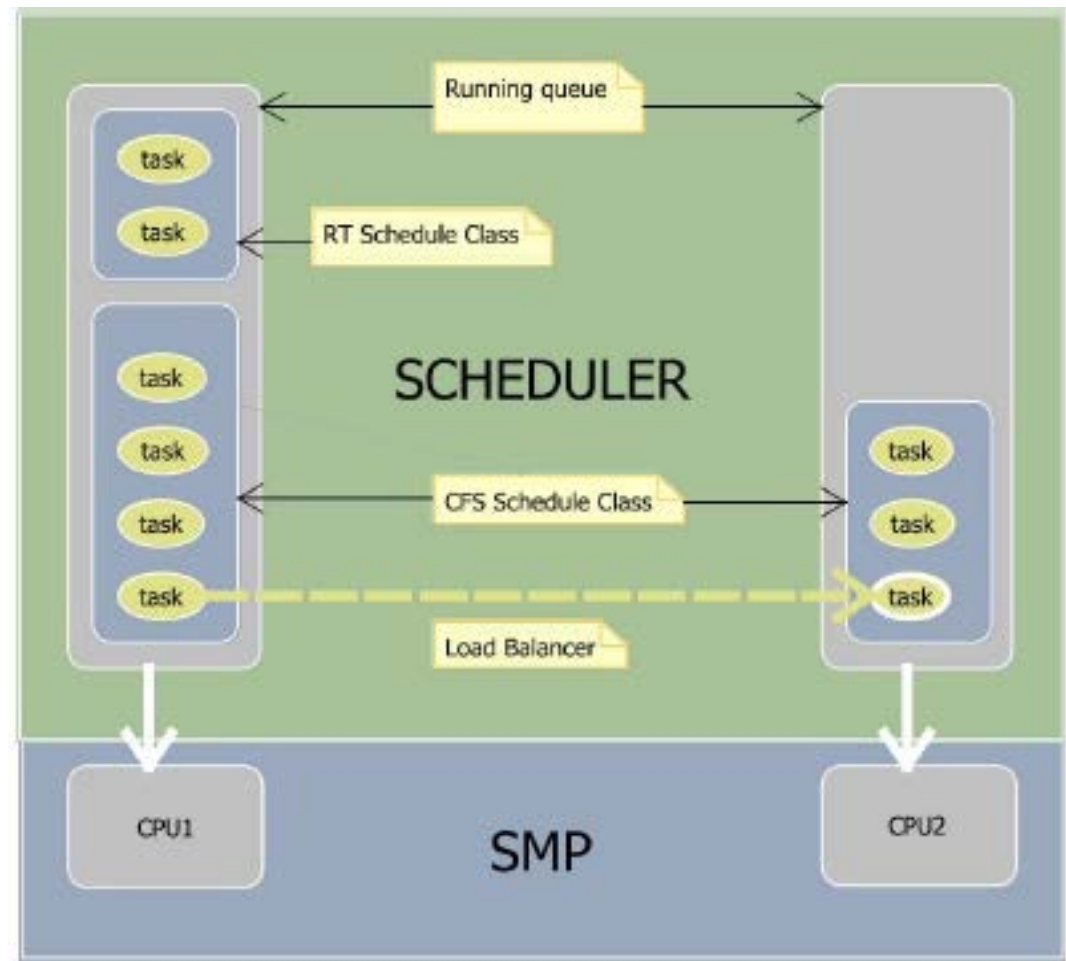
(above) History of Linux & UNIX. (below) Linux systems on the top500.



What the Linux Kernel Does

Processes: Process Management & Scheduling

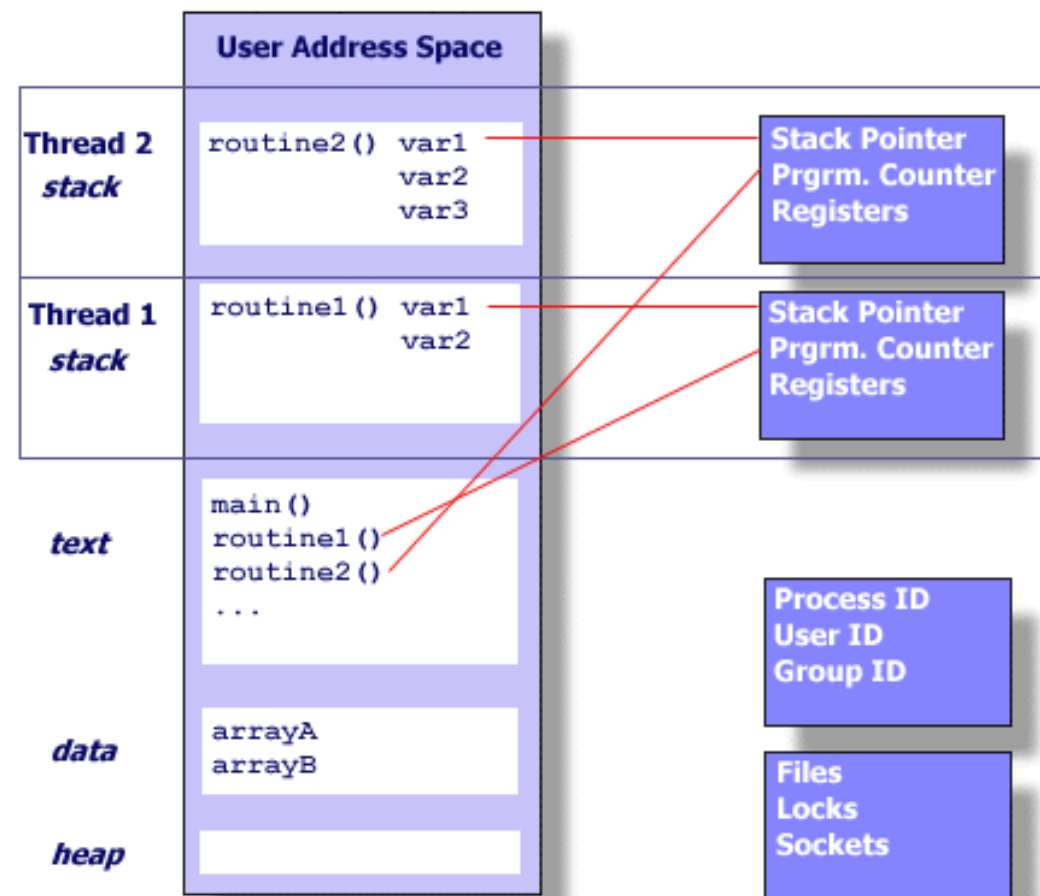
- The Linux kernel is responsible for process scheduling: what, when and how executable code runs on the CPU(s)
- It does this by allocating a process to a program. Processes have:
 - Their allocated memory (stack + heap)
 - Various process metadata (process ID, cmd name, usage statistics, ...)



<https://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS>

Processes: Processes & Threads

- A process is further divided into threads. A process may have one or many threads active at any time.
- Threads can run concurrently with each other
- Threads can see each others' memory areas
- Threads are lighter weight than a full process
- Threads can be used to implement parallel algorithms



<https://computing.llnl.gov/tutorials/pthreads/>

API: What is a syscall?

- A system call (syscall) is a special software-triggered system interrupt
- It allows a program to make an API request to the kernel
- It transitions the system from user-space to kernel-space
- Lots of things are syscalls, e.g. file operations (open, close, read, write)

```
1  ;example inspired by:
2  ; https://taufanlubis.wordpress.com
3  section .text
4  gobal _start
5  _start:
6
7      ;display a message
8      mov eax, 4      ;syscall for write is 4
9      mov ebx, 1      ;set ebx to a file descriptor
10     ;   stdin=0, stdout=1, stderr=2
11     mov ecx,msg      ;set ecx to a pointer to our string
12     mov edx,len      ;set edx to the length of the string
13     int 0x80          ;call interrupt 0x80 (syscall)
14     ;note: modern systems use "syscall"
15     ;       instead of int 0x80
16     mov eax, 1      ;syscall for exit prgram is 1
17     int 0x80        ;call interrupt 0x80 (syscall)
18
19 section .data
20     msg db 'This is a test.',0xA,0xD
21     len equ $ - msg
```

API: Signals

- Signals are in some ways like syscalls from the kernel to the program
- The kernel can send signals to applications
 - but some signals just, e.g. kill the process
- These can be done by the kernel itself (e.g. for program faults)
- ...or can be requested through a syscall (like the kill command)
- Signals can be used to do a lot of things:
 - Abruptly terminate a program (SIGKILL)
 - Communicate between threads (“real-time” signals)
 - Or other program defined functions...

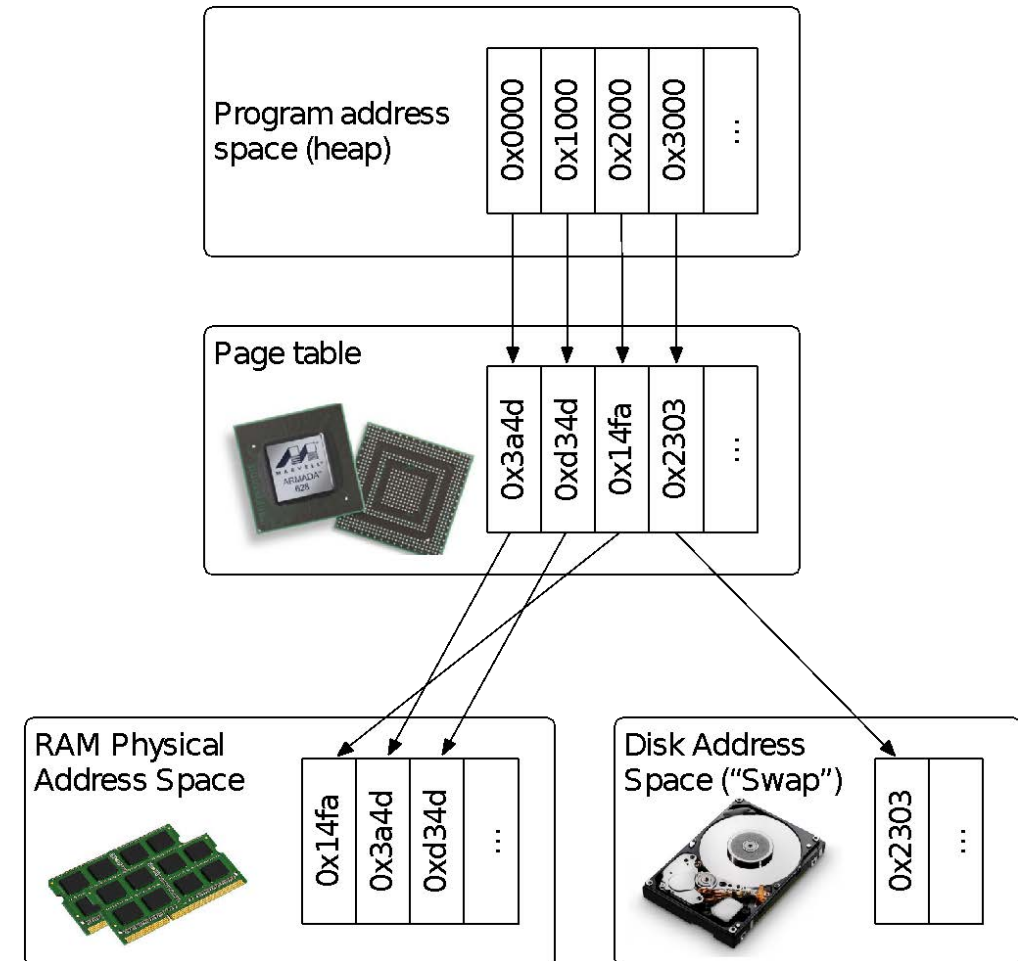
Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Taken from `man 7 signal`, showing POSIX signals.

Memory: Virtual Memory Architecture

- Memory (RAM) divided into fixed-size “pages”, and given numeric addresses
- The kernel maintains a mapping between virtual memory addresses and real, physical memory addresses
- The CPU helps with this process by using a Memory Management Unit (MMU), and maintaining a hardware-accelerated mapping (TLB)

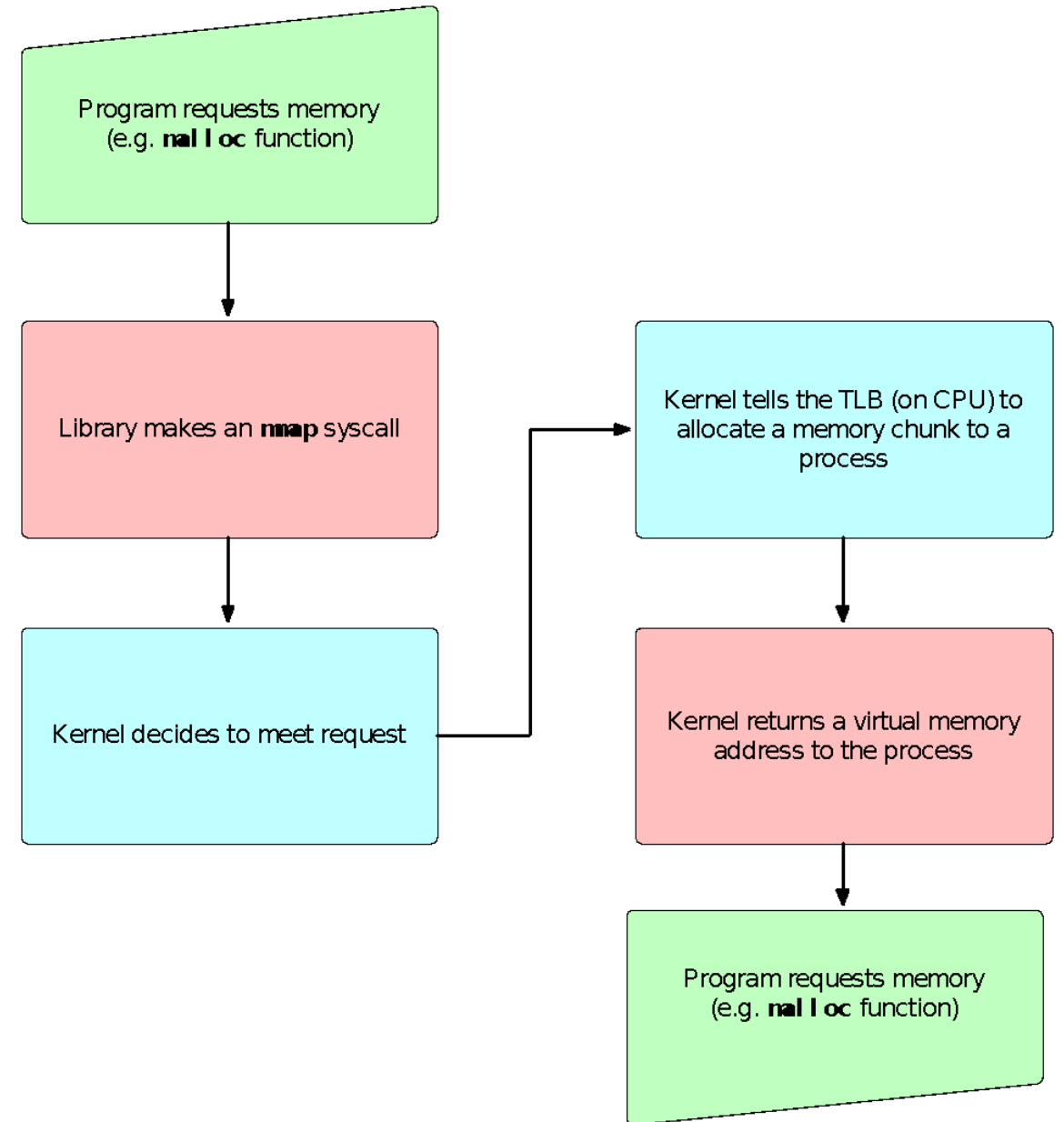


Memory: Why use virtual memory?

- The kernel controls which programs see which memory regions
- One program can't alter another program's memory (very important for security)
- Can create non-RAM memory areas: "swap" space on disk, memory areas on other devices
- Control the amount of memory a process can use
- User programs don't need to know about memory hardware

Memory: Allocation process

- Program tells the kernel it needs new memory through a syscall (typically mmap)
- Kernel decides if it will allocate memory
- Kernel tells the CPU's MMU it wants to allocate a chunk of memory
- MMU creates a TLB entry mapping physical to virtual memory
- Kernel gives the program a virtual address it can use



Authorization: Users & Groups

- The Linux kernel provides an *authorization* layer for processes and system objects (e.g. files)
- The basic mechanism is provided through users/groups
 - A **user** is an authorization domain that can be tied to a person or processes.
 - A **group** is an authorization domain consisting of a collection of users.
 - A special user, **root**, is (mostly) unconstrained.
- A processes runs in a user/group domain
- System objects have access permissions based on one user/group that decided if a process can perform an action.

Character: -wrswrX-rX

Octal: 4775

	Special			User			Group			Other		
	su	sg	sv	w	r	x	w	r	x	w	r	x
Bitmask:	1	0	0	1	1	1	1	1	1	0	1	1

These are representations of object ACLs

- **su** – setuid (run as specific user)
- **sg** – setgid (run as specific group)
- **sv** – sticky (means different things)
- **w = Write, r = Read, x = eXecute**
 - For directories: execute means “can enter”

Authorization: SELinux

- SELinux (“Security-Enhanced Linux”) was developed by the NSA
 - Open source release: 12/22/2000
- Extends the user/group model:
 - All processes and system objects have a *context*
 - E.g. files have SELinux context
 - A (possibly very complicated) *policy* determines what can happen in what contexts
 - Compiles to binary
 - Can be extended with modules
 - Enforces that both context and policy are specified, and policy is obeyed
- While SELinux is great, we’ll be disabling it since it makes development harder.*

(right) Example of an selinux policy module.

(below) Inspecting the status of current selinux contexts.

```
module mysemanage 1.0;

require {
    class fd use;
    type init_t;
    type semanage_t;
    role system_r;
};

allow semanage_t init_t:fd use;
```

```
~]# sestatus -v
SELinux status:                enabled
SELinuxfs mount:              /selinux
Current mode:                  enforcing
Mode from config file:         enforcing
Policy version:                21
Policy from config file:       targeted

Process contexts:
Current context:               user_u:system_r:unconfined_t
Init context:                  system_u:system_r:init_t
/sbin/mingetty                 system_u:system_r:getty_t
/usr/sbin/sshd                  system_u:system_r:unconfined_t:s0-s0:c0.c1023

File contexts:
Controlling term:              user_u:object_r:devpts_t
/etc/passwd                     system_u:object_r:etc_t
/etc/shadow                     system_u:object_r:shadow_t
/bin/bash                       system_u:object_r:shell_exec_t
/bin/login                      system_u:object_r:login_exec_t
/bin/sh                         system_u:object_r:bin_t ->
system_u:object_r:shell_exec_t
/sbin/agetty                    system_u:object_r:getty_exec_t
/sbin/init                      system_u:object_r:init_exec_t
/sbin/mingetty                  system_u:object_r:getty_exec_t
/usr/sbin/sshd                  system_u:object_r:sshd_exec_t
/lib/libc.so.6                  system_u:object_r:lib_t ->
system_u:object_r:lib_t
/lib/ld-linux.so.2              system_u:object_r:lib_t ->
system_u:object_r:ld_so_t
```

Monolithic vs. Modular

- Since version 1.2 (1995), Linux supports “modules”
- Modules can add functionality to a running Linux kernel
 - Can implement new kinds of syscalls
 - Can support new hardware
 - Can add new kinds of functionality (e.g. firewall features)
- Modern Linux Distributions tend to heavily use modules, moving even essential functionality into modules. These modules must be loaded before the system is fully functional. Common examples:
 - Network card drivers (including high-speed networks like InfiniBand and OPA)
 - Filesystem, storage and software RAID drivers
 - Extra functionality for the IP Firewall system (iptables/nftables), cryptography
 - Video, sound, USB devices, ...

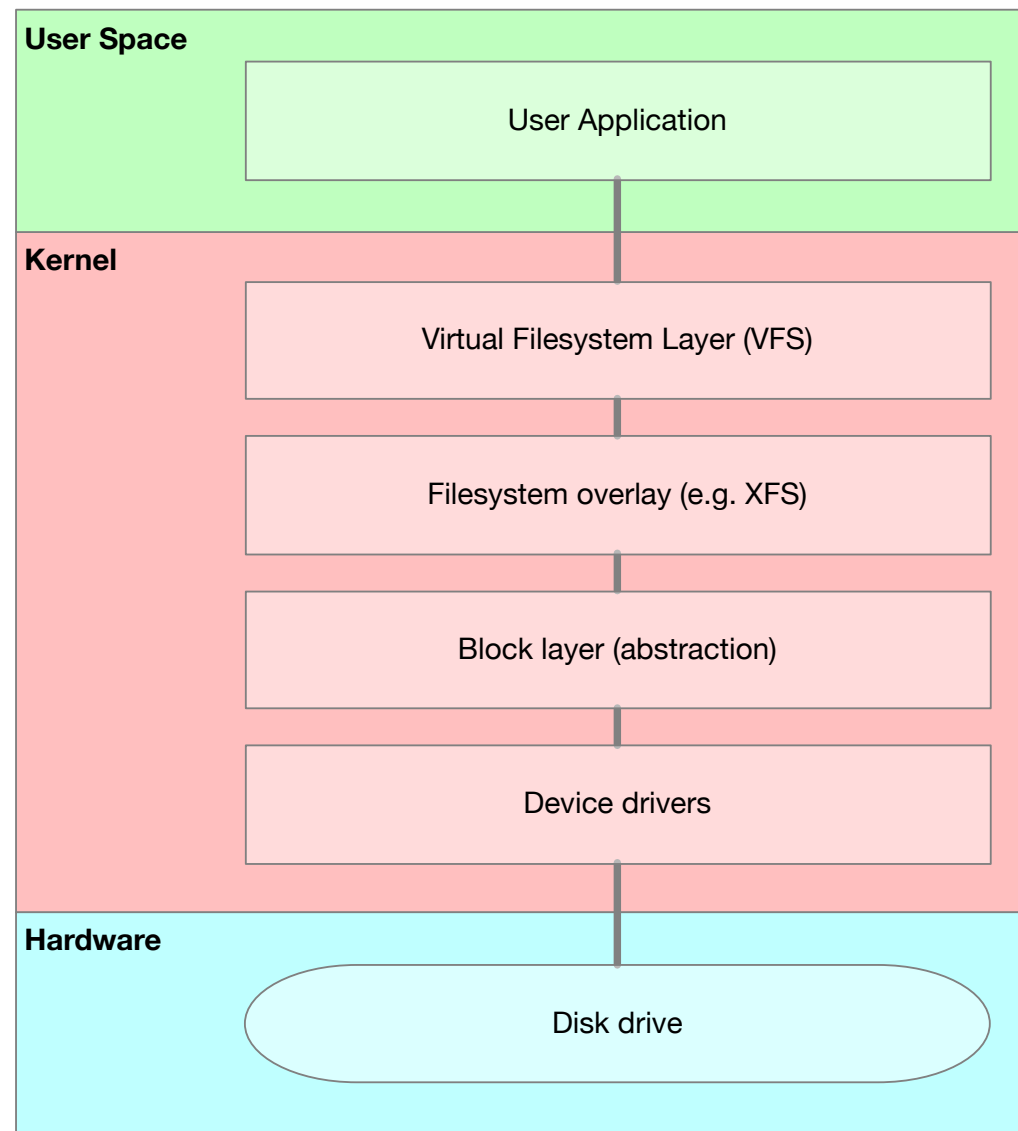
Device Drivers

- The kernel can provide interfaces for working with hardware in the system (drivers)
- Generally, this will provide mechanisms for talking to the device through the kernel APIs
 - Through syscalls
 - Through ioctl calls on a file (e.g. in /dev)
- Will provide handlers for dealing with hardware events
- Often, the device will be presented to the user through a common abstraction layer
 - E.g. storage block layer, network socket layer, ...
- In modern systems, device drivers are usually loaded as kernel modules

A couple of examples

Example: Accessing storage

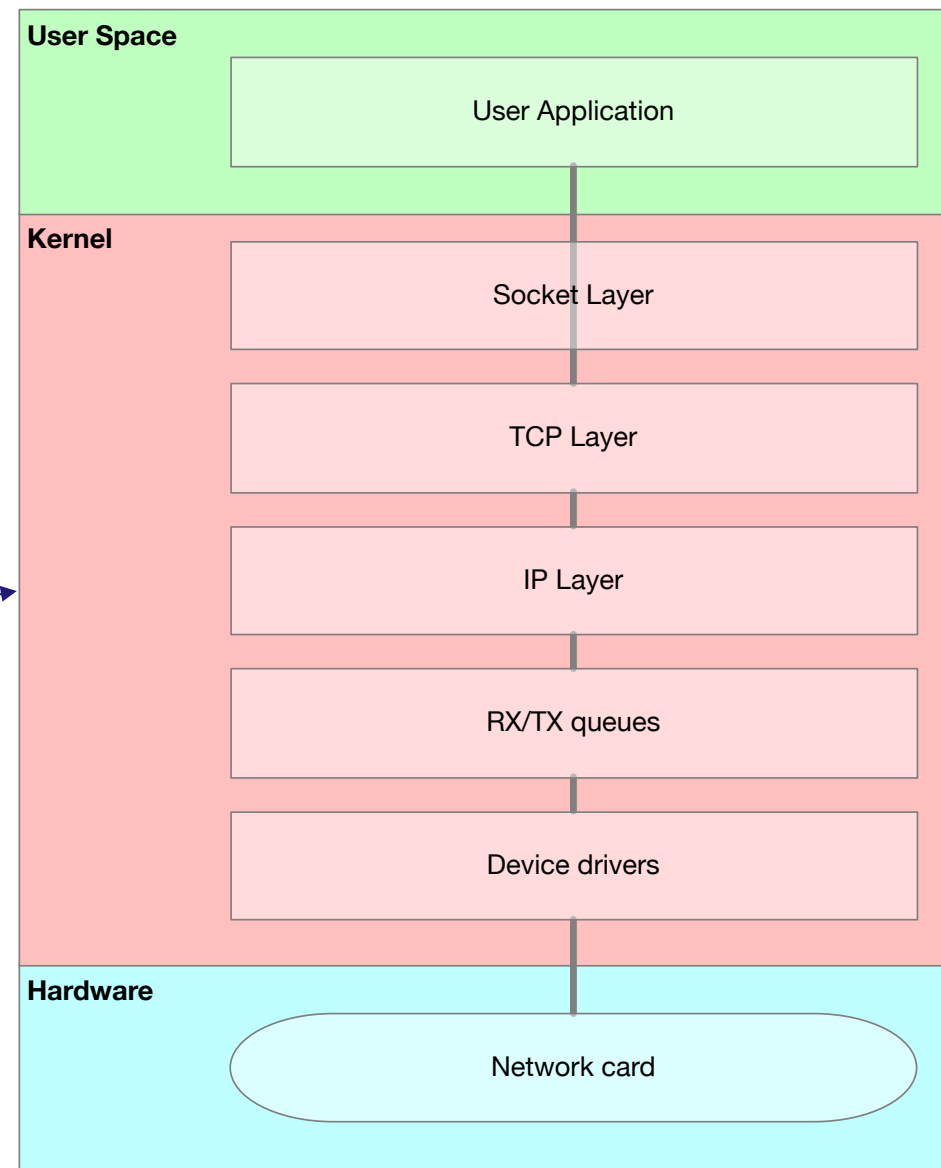
1. The user application accesses a file through a syscall (say *open* or *write*).
2. This syscall is caught by the VFS layer that abstracts filesystem access.
3. The VFS layer passes to the Filesystem layer that maps the operation to storage blocks in the specific filesystem.
4. The block layer maps these blocks to particular hardware addresses.
5. The device drivers determine how to make these changes happen on a physical disk.



Example: reading an HTTP GET request

1. The network card receives a string of high/low voltages it interprets as data
2. The network card sends a hardware interrupt (IRQ)
3. The IRQ is caught by a kernel handler for IRQs that maps to a handler in the network card device driver
4. The device driver reads in the data
5. The driver hands the data to the rx/tx queue
6. The IP layer reads the IP header; the TCP layer reads the TCP header
7. The TCP layer identifies (by port and address) and identifies a listening socket
8. The socket layer provides the data to the socket
9. The listening web server reads (syscall) the data
10. It reads "GET" and sends some data back down the channel to the client

Lots of filtering (e.g. iptables/nft) happens here!



Linux Distributions

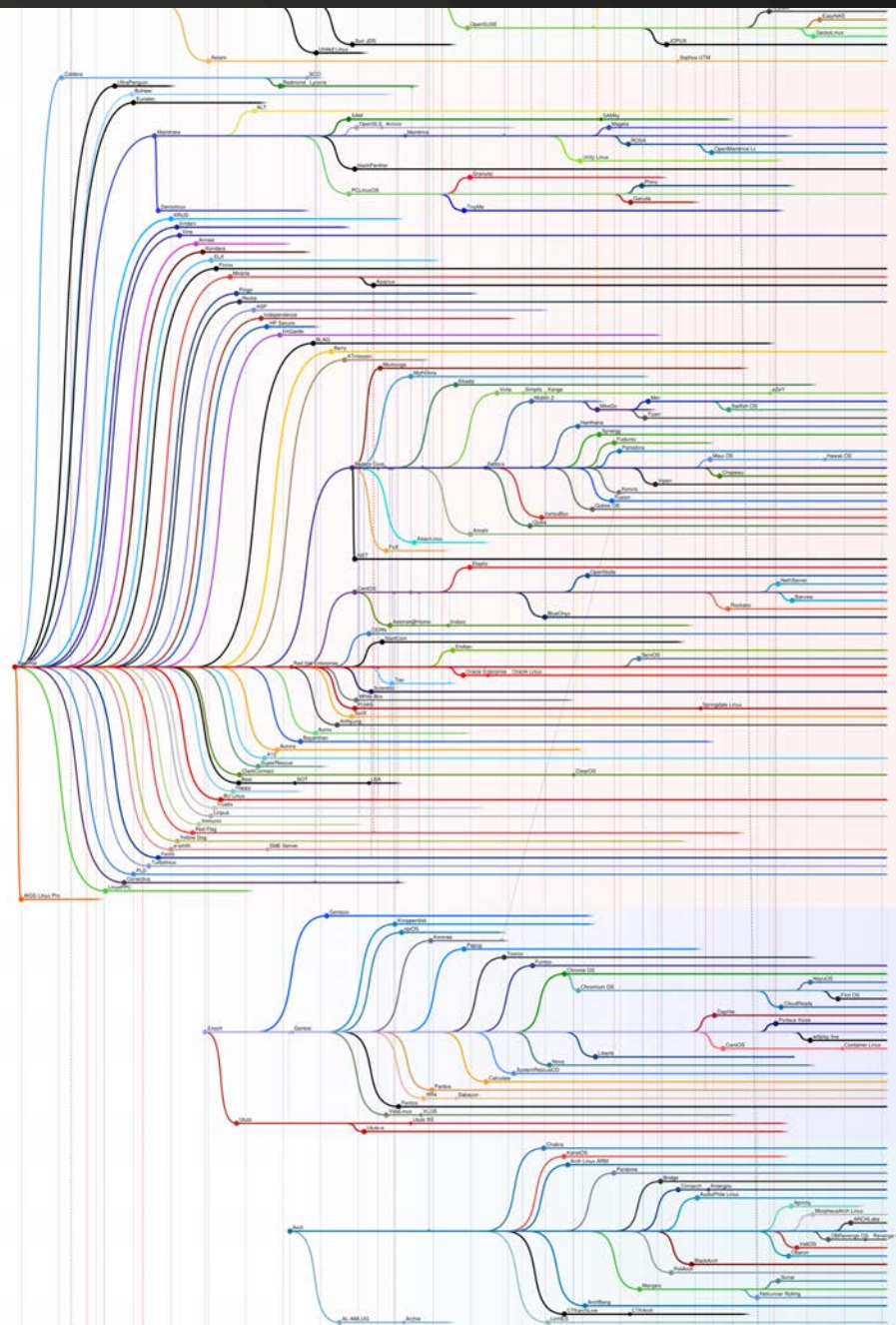
What makes a Linux Distribution?

*There are many combinations of software that can build a usable Linux system.
A Linux distribution is a particular usable arrangement of these tools.*

Must Have	Typically Has
<ul style="list-style-type: none"> • One (or more) provided Linux Kernel(s) • One (or more) system initialization system (systemd, sysV, supervisord, ...) • A collection of userspace software for using/managing the system 	<ul style="list-style-type: none"> • Easy installation system • Some form of software installation management (e.g. rpm, yum, apt, dpkg, yast, portage) • Collections of useful software for users (e.g. firefox, libreoffice, GNOME, ...) • Software development tools

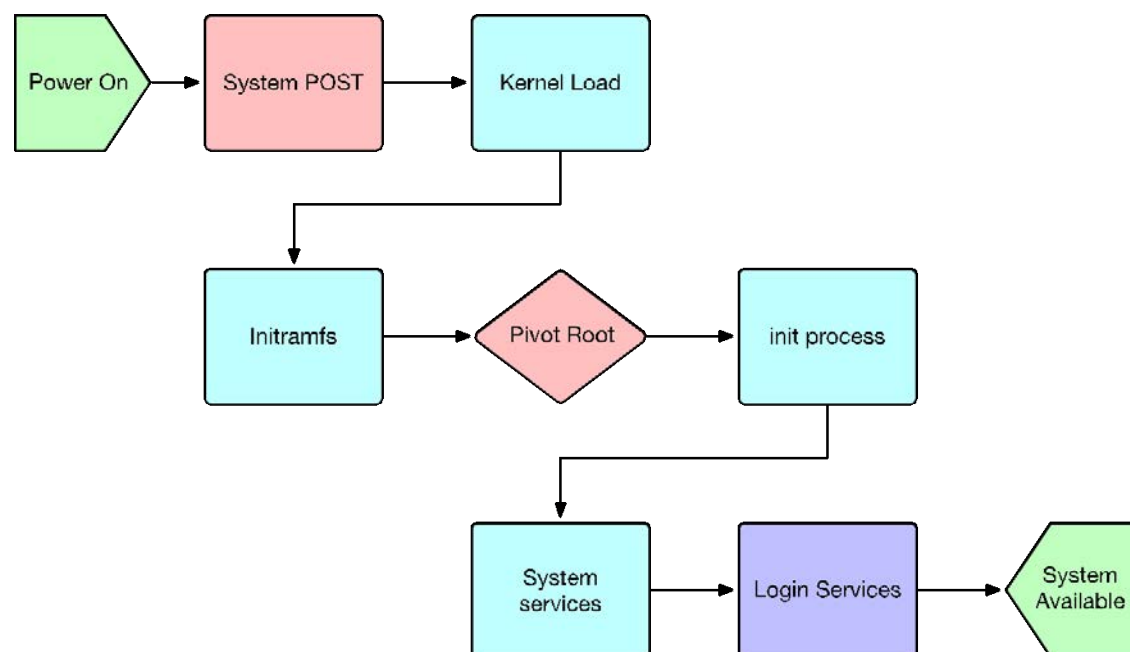
There are many...

- A site that tracks distributions claims there have been as many as 323 distributions at a given time.
- Many distributions are branches from one of the original major distributions:
 - Red Hat (CentOS, Fedora,...)
 - Debian (Ubuntu, Mint,...)
 - SUSE
 - SLS/Slackware
- And some major software like Android (non-GNU) and Chromium (based on Gentoo)



The Linux system boot process

What happens when you hit the power button?



- System Power-on Self Test (POST)
- Kernel entry point
- Kernel loads initramfs (optional)
- Initramfs sets up sytem (e.g. loads modules)
- Initramfs calls `pivot_root` (or `switch_root`) to switch to real root filesystem
- `pivot_root` calls the new init process (e.g. `systemd`)
- RC system (e.g. `systemd`) starts services
- Login services (TUI and/or GUI) loaded

System POST

- POST = Power-on Self Test
- With UEFI (Unified Extensible Firmware Interface), this stage does a lot more than basic tests, and can take a while
 - Loads pluggable DXE (“dixie”) modules to do various system init tasks
- Does some really important stuff, like figure out timings needed to efficiently use RAM
- Loads the requested operating system (in our case, the Linux Kernel)



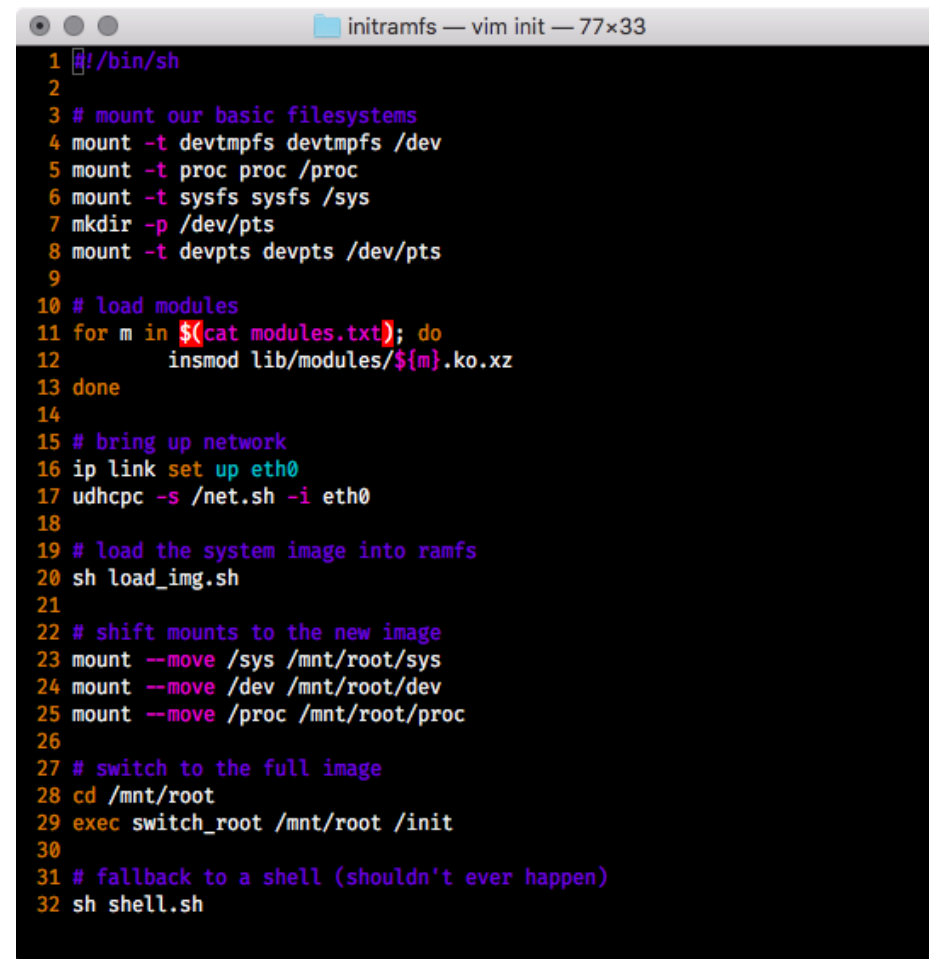
Kernel init

- The kernel provides an executable entry point for the system
- It takes control of things like:
 - Switching to “protected mode” (memory protection)
 - Registering memory
 - Registering interrupt handlers
 - ...and lots of other things
- When it's done getting everything ready, it extracts the initramfs filesystem and treats it as the filesystem “root” (/)
- Then, it attempts to execute the “/sbin/init” command (or other if specified)

```
centos — vagrant@localhost:~ — ssh • vagrant ssh — 112x36
[ 0.000000] Zone ranges:
[ 0.000000]   DMA      [mem 0x00001000-0x00ffffff]
[ 0.000000]   DMA32    [mem 0x01000000-0xffffffff]
[ 0.000000]   Normal    empty
[ 0.000000] Movable zone start for each node
[ 0.000000] Early memory node ranges
[ 0.000000]   node 0: [mem 0x00001000-0x0009efff]
[ 0.000000]   node 0: [mem 0x00100000-0x1ffeffff]
[ 0.000000] Initmem setup node 0 [mem 0x00001000-0x1ffeffff]
[ 0.000000] On node 0 totalpages: 130958
[ 0.000000]   DMA zone: 64 pages used for memmap
[ 0.000000]   DMA zone: 21 pages reserved
[ 0.000000]   DMA zone: 3998 pages, LIFO batch:0
[ 0.000000]   DMA32 zone: 1984 pages used for memmap
[ 0.000000]   DMA32 zone: 126960 pages, LIFO batch:31
[ 0.000000] ACPI: PM-Timer IO Port: 0x4008
[ 0.000000] ACPI: Local APIC address 0xfee00000
[ 0.000000] ACPI: LAPIC (acpi_id[0x00] lapic_id[0x00] enabled)
[ 0.000000] ACPI: IOAPIC (id[0x01] address[0xfec00000] gsi_base[0])
[ 0.000000] IOAPIC[0]: apic_id 1, version 32, address 0xfec00000, GSI 0-23
[ 0.000000] ACPI: INT_SRC_OVR (bus 0 bus_irq 0 global_irq 2 dfl dfl)
[ 0.000000] ACPI: INT_SRC_OVR (bus 0 bus_irq 9 global_irq 9 low level)
[ 0.000000] ACPI: IRQ0 used by override.
[ 0.000000] ACPI: IRQ9 used by override.
[ 0.000000] Using ACPI (MADT) for SMP configuration information
[ 0.000000] smpboot: Allowing 1 CPUs, 0 hotplug CPUs
[ 0.000000] PM: Registered nosave memory: [mem 0x0009f000-0x0009ffff]
[ 0.000000] PM: Registered nosave memory: [mem 0x000a0000-0x000effff]
[ 0.000000] PM: Registered nosave memory: [mem 0x000f0000-0x000fffff]
[ 0.000000] e820: [mem 0x20000000-0xfebfffff] available for PCI devices
[ 0.000000] Booting paravirtualized kernel on KVM
[ 0.000000] setup_percpu: NR_CPUS:5120 nr_cpumask_bits:1 nr_cpu_ids:1 nr_node_ids:1
[ 0.000000] PERCPU: Embedded 38 pages/cpu @ffff890adfc00000 s118784 r8192 d28672 u2097152
[ 0.000000] pcpu-alloc: s118784 r8192 d28672 u2097152 alloc=1*2097152
[ 0.000000] pcpu-alloc: [0] 0
[ 0.000000]
```


Initramfs

- The initramfs is a specially file archive that gets to temporarily act as the root filesystem.
- It's in the "cpio" archive format, possibly compressed.
- It can come from many places, like the "/boot" directory, or over the network.
- It gives us a chance to do any pre-setup we need (like getting the real root filesystem).
- When it's ready, it (typically) will "switch_root" into the new root filesystem and execute the real /sbin/init process.



```
1 #!/bin/sh
2
3 # mount our basic filesystems
4 mount -t devtmpfs devtmpfs /dev
5 mount -t proc proc /proc
6 mount -t sysfs sysfs /sys
7 mkdir -p /dev/pts
8 mount -t devpts devpts /dev/pts
9
10 # load modules
11 for m in $(cat modules.txt); do
12     insmod lib/modules/${m}.ko.xz
13 done
14
15 # bring up network
16 ip link set up eth0
17 udhcpc -s /net.sh -i eth0
18
19 # load the system image into ramfs
20 sh load_img.sh
21
22 # shift mounts to the new image
23 mount --move /sys /mnt/root/sys
24 mount --move /dev /mnt/root/dev
25 mount --move /proc /mnt/root/proc
26
27 # switch to the full image
28 cd /mnt/root
29 exec switch_root /mnt/root /init
30
31 # fallback to a shell (shouldn't ever happen)
32 sh shell.sh
```


The init process

- The init process is responsible for doing any non-kernel level initialization, like starting system services
- The most common init process these days is “systemd”, a complete initialization and service management system (and more...)
 - We’ll be learning more about systemd and system services later on.
- One of the last services started is a login service that provides a login prompt to the user

```

Welcome to Fedora 17 (Beefy Miracle)!

    Expecting device dev-ttyS0.device...
[ OK ] Reached target Remote File Systems.
[ OK ] Listening on Delayed Shutdown Socket.
[ OK ] Listening on /dev/initctl Compatibility Named Pipe.
[ OK ] Reached target Encrypted Volumes.
[ OK ] Listening on udev Kernel Socket.
[ OK ] Listening on udev Control Socket.
[ OK ] Set up automount Arbitrary Executable File Formats F...utomount Point.
    Expecting device dev-disk-by\x2duuid-6038ea52\x2d80a...ce4c9.device...
[ OK ] Listening on Journal Socket.
    Starting File System Check on Root Device...
    Starting udev Kernel Device Manager...
    Mounting Debug File System...
    Starting Journal Service...
[ OK ] Started Journal Service.
    Starting Apply Kernel Variables...
    Starting udev Coldplug all Devices...
    Mounting Huge Pages File System...
    Mounting POSIX Message Queue File System...
    Starting Setup Virtual Console...
    Starting Set Up Additional Binary Formats...
    Mounting Configuration File System...
[ OK ] Started Apply Kernel Variables.
[ OK ] Started udev Kernel Device Manager.
    Mounting Arbitrary Executable File Formats File System...
[ OK ] Started udev Coldplug all Devices.
[ OK ] Mounted POSIX Message Queue File System.
[ OK ] Mounted Debug File System.
[ OK ] Mounted Configuration File System.
[ OK ] Mounted Huge Pages File System.
[ OK ] Mounted Arbitrary Executable File Formats File System.
[ OK ] Started Set Up Additional Binary Formats.
[ OK ] Started Setup Virtual Console.
[ OK ] Found device /dev/ttyS0.
systemd-fsck[53]: fedora: clean, 319575/983040 files, 2914206/3932160 blocks
[ OK ] Started File System Check on Root Device.
    Starting Remount Root and Kernel File Systems...
[ OK ] Started Remount Root and Kernel File Systems.
[ OK ] Reached target Local File Systems (Pre).
    Mounting Temporary Directory...
    Starting Load Random Seed...

```

Login & shell

- Once the user logs in, a new process is started (“the shell”) with that user’s running context
- The shell can technically be anything, but typically it’s an interactive tool for using the system (like BASH)
- A shell allows you to do things like:
 - Run more processes to do useful things
 - ...and see their output
 - Keep track of certain session variables (“environment variables”)
 - Redirect input and output in useful ways
 - Use a simple scripting language to string these tasks together (“shell script”)
 - *...and generally a lot more.*

```
CentOS Linux 7 (Core)
Kernel 3.10.0-862.2.3.el7.x86_64 on an x86_64

localhost login: vagrant
Password:
[vagrant@localhost ~]$ ls
[vagrant@localhost ~]$ hostname
localhost.localdomain
[vagrant@localhost ~]$ whoami
vagrant
[vagrant@localhost ~]$ ps
  PID TTY          TIME CMD
 1100 tty1        00:00:00 bash
 1128 tty1        00:00:00 ps
[vagrant@localhost ~]$ pwd
/home/vagrant
[vagrant@localhost ~]$ _
```

Questions?