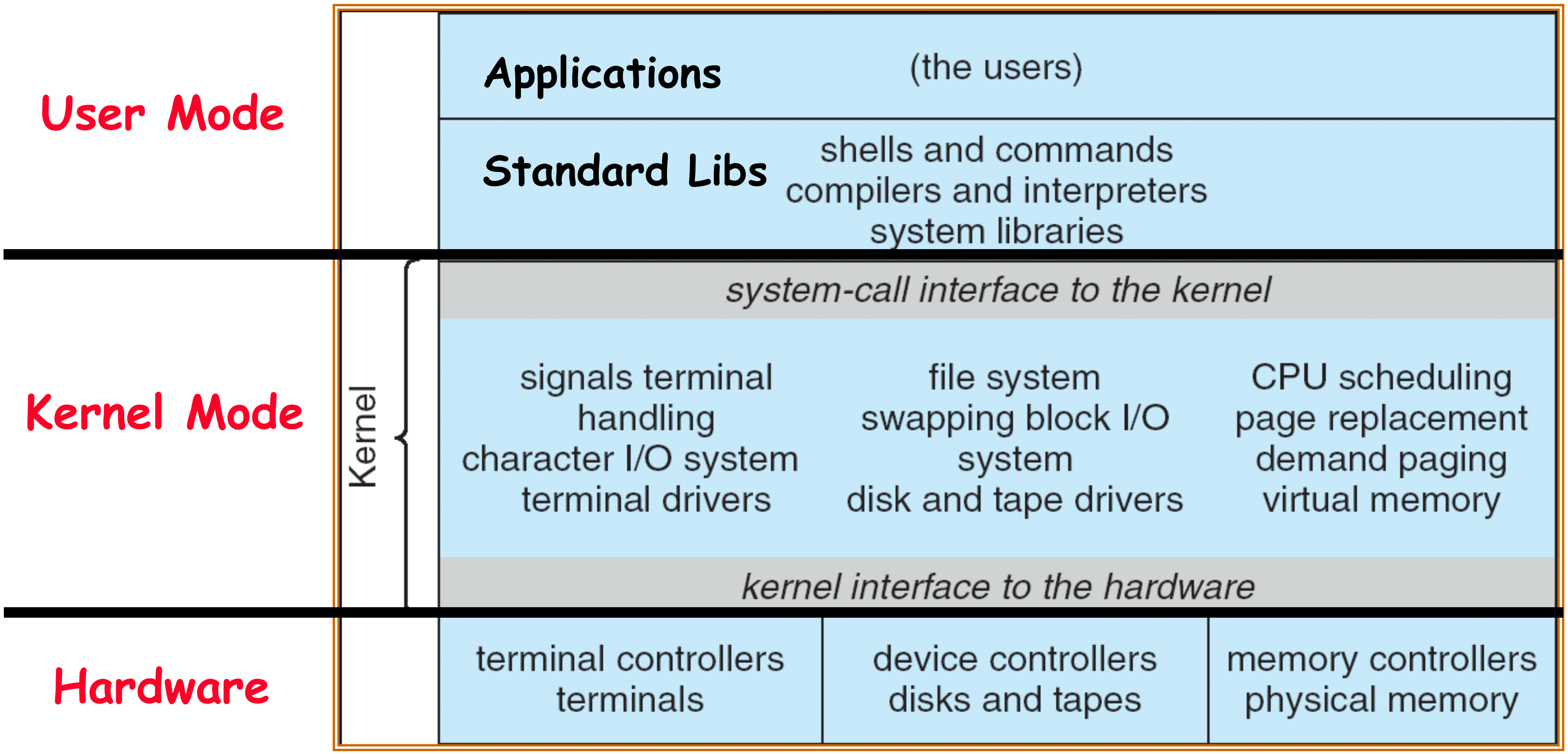
CS3510
Operating Systems

System Calls and Boot Process

Bheemarjuna Reddy
IIT HYD

UNIX System Structure



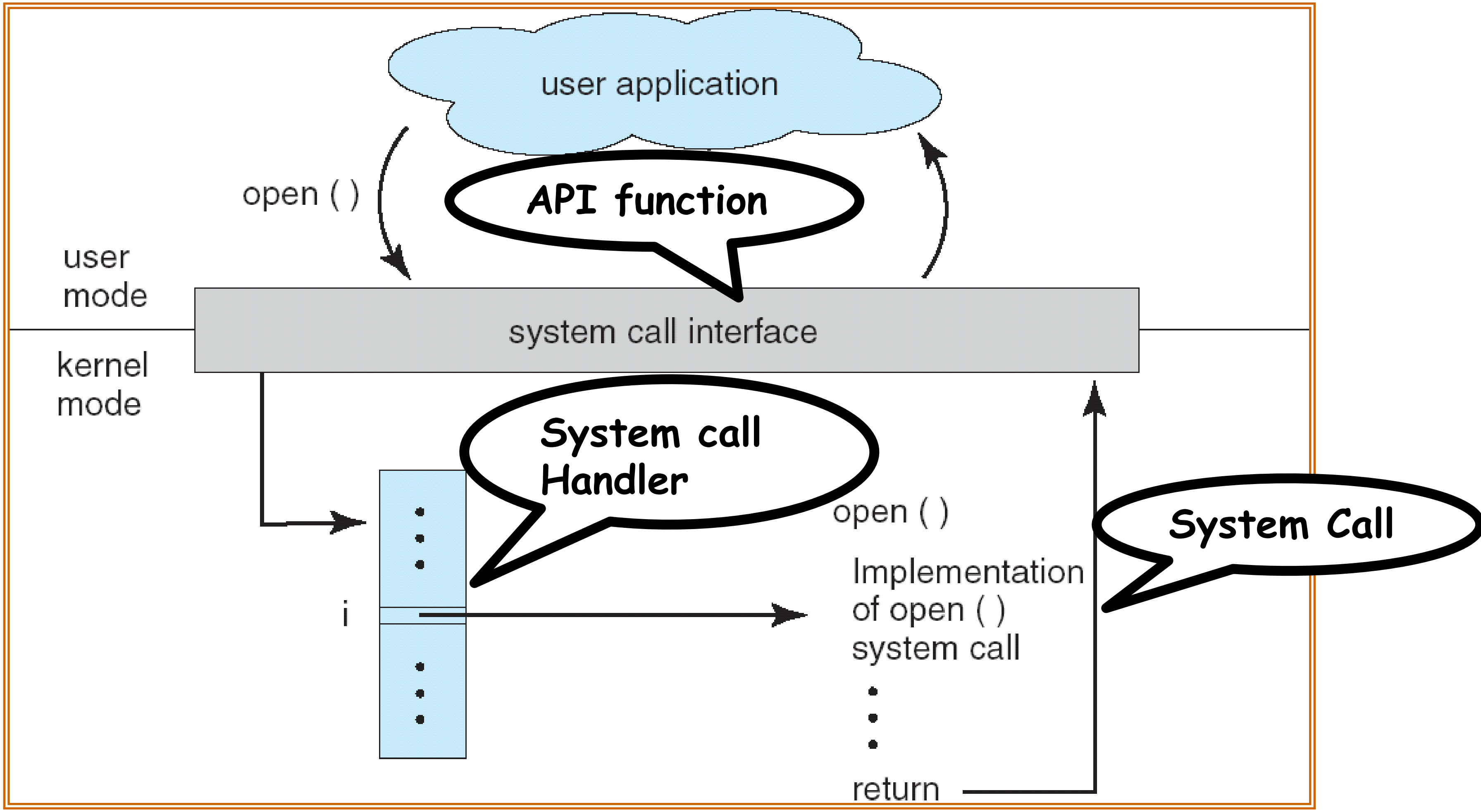
System Calls

- Programming interface to the services provided by the OS to system/app programs
- Typically written in a high-level language (C/C++)
- Mostly accessed by application/system programs using procedure calls in APIs
- Programmer/job → procedure in API → System call
  - API is a function definition that specifies how to obtain a given service
  - System call is an explicit request to kernel made via a trap i.e., software interrupt
- Three most common APIs:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (UNIX, Linux, Mac OS X)
  - Java API for the Java virtual machine (JVM)
- A programmer accesses an API via a library of code (eg., libc for Linux) provided by OS

System Calls

- **POSIX.1-2017** (IEEE 1003.1, ISO/IEC 9945)
  - Very widely used standard based on (and including) C-language
  - Defines both
    - *APIs* and
    - compulsory *system programs/common utilities* together with their functionality and command-line format
      - E.g. `ls -w dir` prints the list of files in a directory in a 'wide' format
  - Complete specification is at http://www.opengroup.org/onlinepubs/9699919799/nframe.html
- Strong correlation b/w a procedure/function in API and its associated system call within kernel
  - Typically One-to-one (e.g., read, exit)
  - many-to-one (e.g., exec,brk) and one-to-many

System Calls



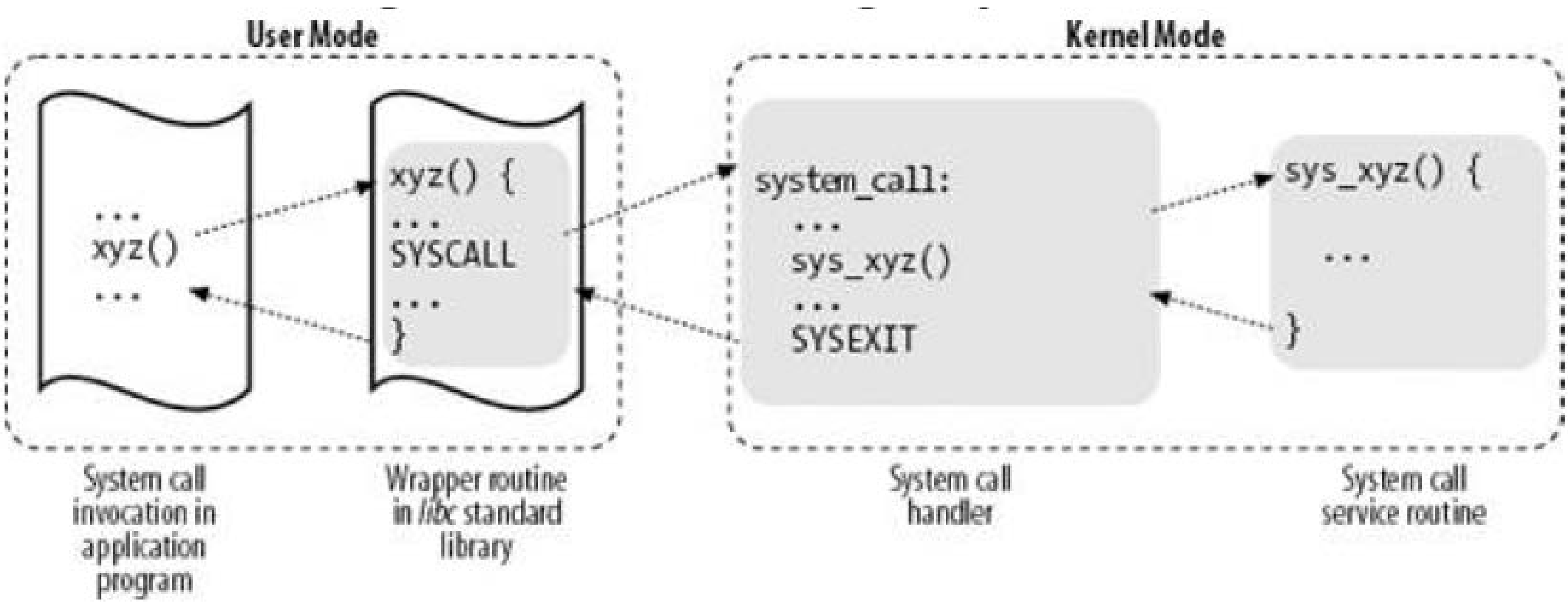
Instructions used to transition to kernel mode in diff archs

- i386 (int 0x80), eax register is used to indicate syscall number
- x86_64 (syscall), rax register is used similarly

API-System Call Implementation

- The interface to the services provided by the OS has two parts:
  1. Higher language interface – a part of system library
     - Executes in user mode
     - Implemented to accept a standard procedure call
     - Traps to the Part 2
  2. Kernel part
     - Executes in kernel mode
     - Implements the required system service
     - May cause blocking the caller (forcing it to wait)
     - After completion returns back to Part 1 (may report the success or failure of the call)

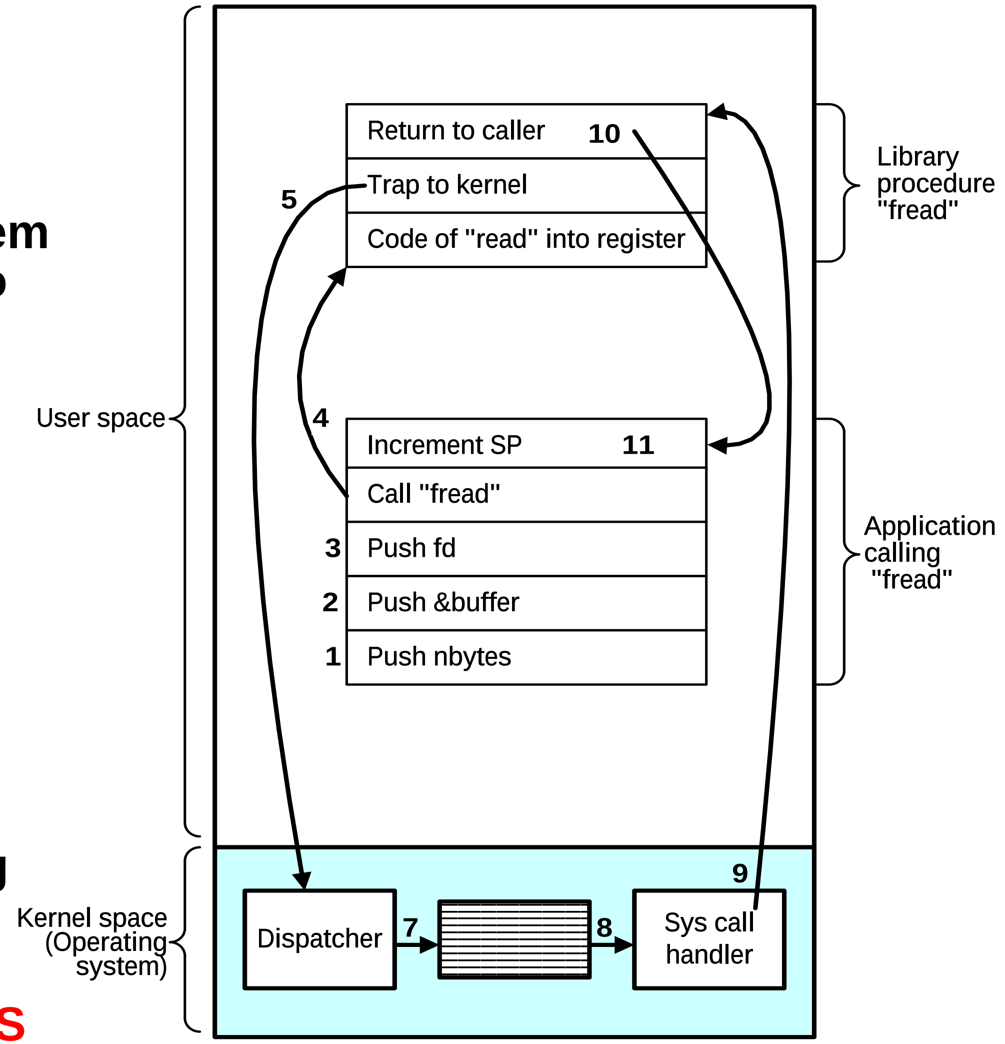
System Calls



- POSIX std refers to the API, not actual system calls provided by the kernel
- Why use APIs rather than system calls directly?
  - Program portability
  - Easier to use

System Call Interface: Implementation

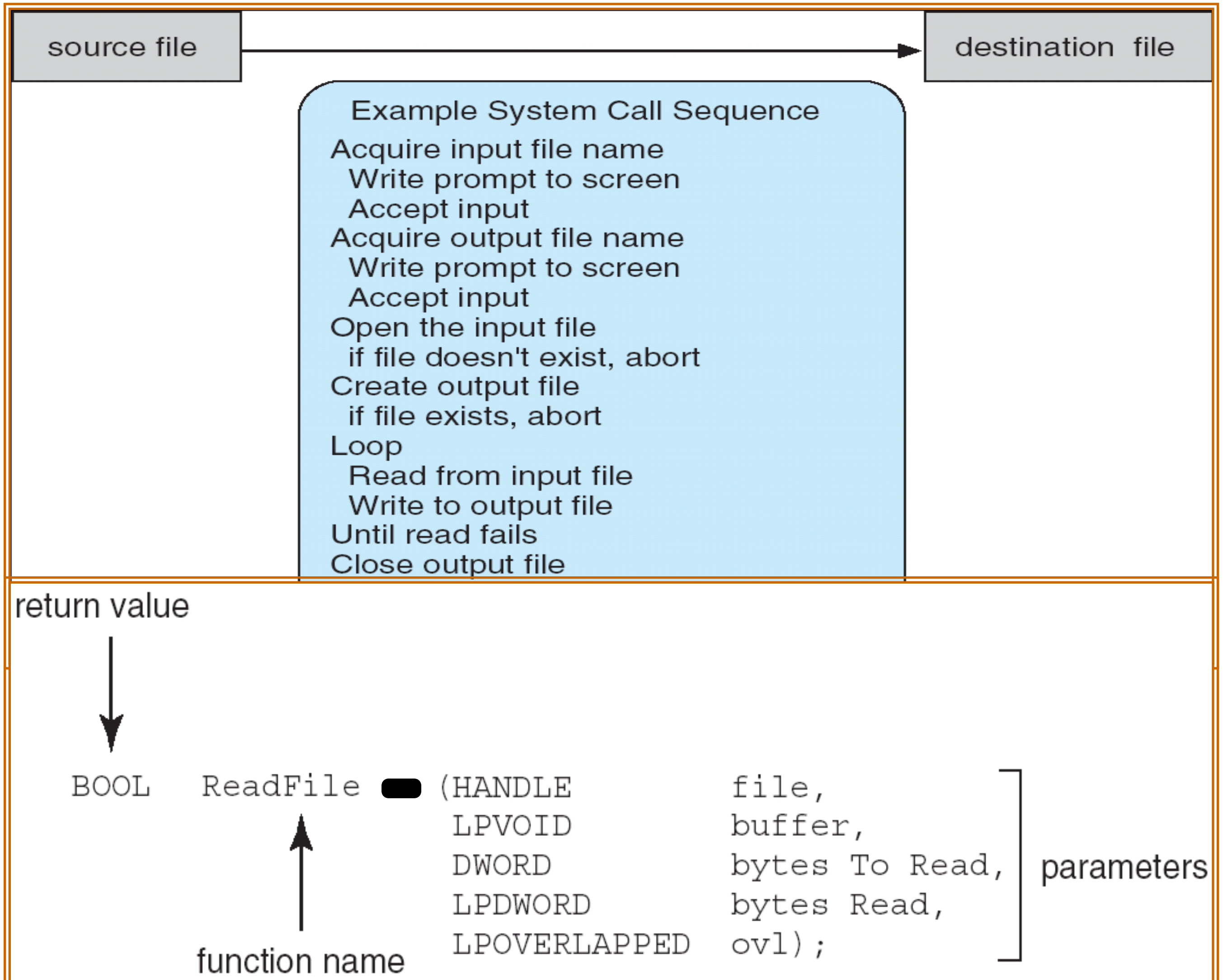
- An application program wants to make use of a System Call:
  - A system library routine is called first
  - It transforms the call to the system standard (*native API*) and traps to the kernel
  - Control is taken by the kernel running in the kernel mode
  - According to the service "code", the *Call dispatcher* invokes the responsible part of the Kernel
  - Depending on the nature of the required service, the kernel may block the calling process
- After the call is finished, the calling process execution resumes obtaining the result (success/failure) as if an ordinary function was called
- Three ways to pass parameters to OS
  - Registers
  - Stack
  - Memory block



11 steps to execute the service *fread (fd, buffer, nbytes)*

System Calls

- **System call sequence to copy contents of one file to another**



glibc: GNU C Library

- Any Unix-like OS needs a C library
- C lang has no built-in facilities for doing I/O, memory management, string manipulation, etc
- A std C library (ISO C std) provides these facilities
- The GNU C Library (glibc) implements all of the functions specified in
  - ISO C library (malloc, printf, fopen, exit, etc)
  - POSIX.1 (system calls)
  - And extensions specific to GNU systems
- glibc is used as *the* C library in GNU systems and most systems with Linux kernel
  - Current version 2.30 (link)

glibc: GNU C Library

- glibc has procedures (wrapper functions) which in turn call system calls
  - getpid(), getppid(), chmod() are defined in glibc
- glibc provides syscall which helps you to call system calls explicitly (directly) from user/app program
  - syscall is also a library function!, but very simple one
  - long syscall (*long sysno, ...*)
  - sysno is system call number, refer <sys/syscall.h> for Macros
  - Return val is the return value of syscall pointed to by sysno
    - » -1 when system call is failed
  - Employing syscall() is useful when invoking a system call that has no wrapper function defined in the C library

  - http://man7.org/linux/man-pages/man2/syscall.2.html
  - http://man7.org/linux/man-pages/man2/syscalls.2.html
  - http://man7.org/linux/man-pages/man7/vdso.7.html
  - http://man7.org/linux/man-pages/man7/libc.7.html

Example 1

```
#define _GNU_SOURCE
 #include <unistd.h> //wrapper for syscalls
 #include <sys/syscall.h> // loc: /usr/src/include/i386-linux-gnu/bits/syscall.h, defines syscall numbers/Macros
 #include <sys/types.h>
 #include <stdio.h>

int main(int argc, char *argv[]) {
  pid_t tid;

  tid = syscall(SYS_gettid); //SYS_gettid does not have glibc wrapper function, so calling syscall directly using "syscall" func; refer man
syscall, man gettid

  printf("TID=%d\n", tid);

  tid = getpid(); //getpid is wrapper function given in glibc

  printf("PID=%d\n", tid);

  tid = getppid(); //getppid is wrapper in glibc

  printf("PPID=%d\n", tid);

  tid = syscall(__NR_getpid); //calling SYSCALL directly

  printf("PID=%d\n", tid);

  tid = syscall(SYS_getpid); //calling SYSCALL directly

  printf("PID=%d\n", tid);

  tid = syscall(__NR_getppid); //calling SYSCALL directly

  printf("PPID=%d\n", tid);

  return 0; }
```

Example 2: (kind of) direct system call

```
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

        ...
        int rc;

        rc = syscall(SYS_chmod, "/etc/passwd", 0444);

    if (rc == -1)
    fprintf(stderr, "chmod failed, errno = %d\n", errno);
         …
```

Example 2': glibc wrapper call

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>

        ...

        int rc;

        rc = chmod("/etc/passwd", 0444);
        if (rc == -1)
fprintf(stderr, "chmod failed, errno = %d\n", errno);
         …
```

Some API Calls For Process Management

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Some API Calls For File Management

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Some API Calls For Directory Management

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Some API Calls For Other Tasks

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

POSIX and Win32 Calls Comparison

- **Only some important calls are shown**

POSIX	Win32	Description
fork	CreateProcess	Create a new process
wait	WaitForSingleObject	The parent process may wait for the child to finish
execve	--	CreateProcess = fork + execve
exit	ExitProcess	Terminate process
open	CreateFile	Create a new file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from an open file
write	WriteFile	Write data into an open file
lseek	SetFilePointer	Move read/write offset in a file (file pointer)
stat	GetFileAttributesExt	Get information on a file
mkdir	CreateDirectory	Create a file directory
rmdir	RemoveDirectory	Remove a file directory
link	--	Win32 does not support “links” in the file system
unlink	DeleteFile	Delete an existing file
chdir	SetCurrentDirectory	Change working directory
chmod	SeFileSecurity	Change file mode bits (rwx)

Example

- A stripped down shell:

```
while (TRUE) {                                  /* repeat forever */
    type_prompt( );                             /* display prompt */
    read_command (command, parameters)          /* input from terminal */

if (fork() != 0) {                              /* fork off child process */
    /* Parent code */
    waitpid( -1, &status, 0);                   /* wait for child to exit */
} else {
    /* Child code */
    execve (command, parameters, 0);            /* execute command */
 }
}
```

Reducing System Call Overhead

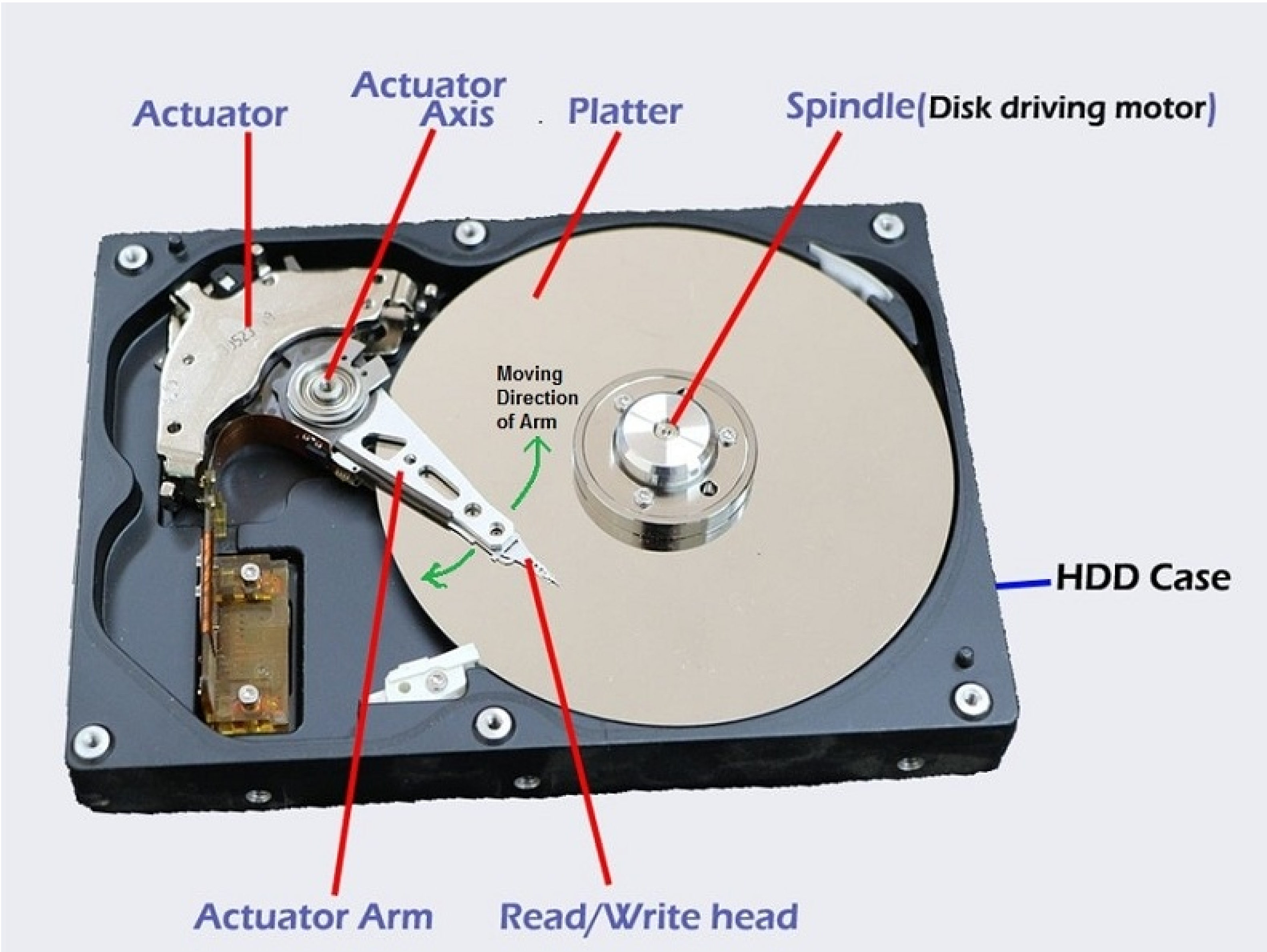
- Problem: The user-kernel mode distinction poses a performance barrier
  - » Crossing this hardware barrier is costly.
  - » System calls take 10x-1000x more time than a regular procedure call
- Solution: Perform some system functionality in user mode itself
  - » *Libraries (DLLs)* can reduce number of system calls
    - E.g.,"vDSO" (virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user-space applications
  - » by caching results (*getpid, gettimeofday*)
  - » buffering I/O operations to minimize no. of system calls made (open/read/write vs. *fopen/fread/ fwrite*).

https://lwn.net/Articles/771441/

http://arkanis.de/weblog/2017-01-05-measurements-of-system-call-performance-and-overhead

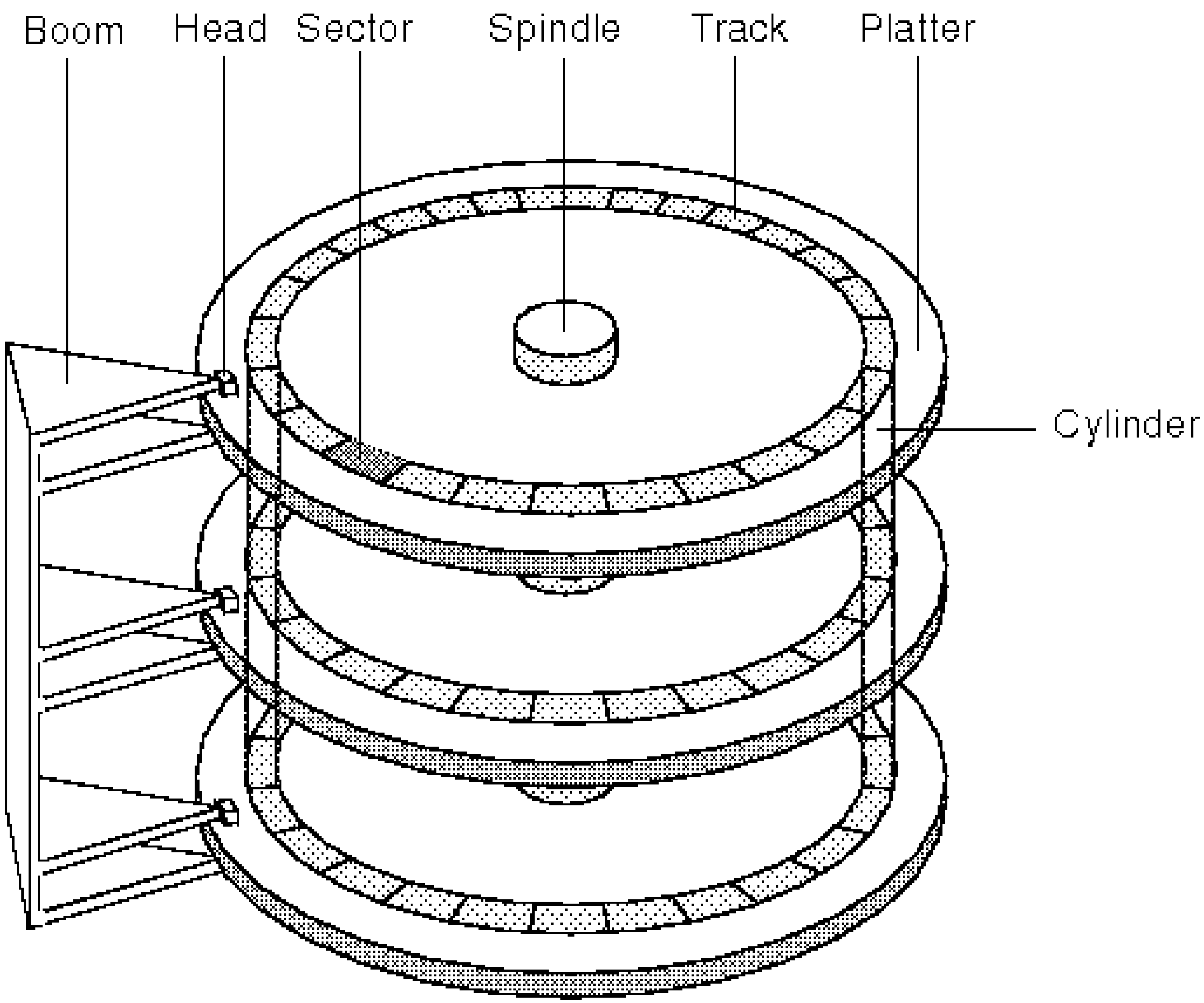
Boot process

Hard Disk



Actuator
Actuator Axis
Platter
Spindle(Disk driving motor)
Moving Direction of Arm
HDD Case
Actuator Arm
Read/Write head

HDD Components



On hard drives and floppies, each sector can hold 512 bytes of data.
Disk Block: a group of 1 or more sectors OS can refer at a time.

BIOS/MBR Boot Process

BIOS
- The PC is turned on & the BIOS initializes the hardware.

MBR
- The BIOS calls code stored in the MBR at the start of disk 0.

Active Partition
- The MBR loads code from the bootsector of the active partition.

Bootloader
- The bootsector loads & runs the bootloader from its filesystem.

Credits: NeoSmart Tech

How do you start the OS?

- Your computer has a very simple program pre-loaded in a special ROM (EEPROM) aka firmware:
  - The Basic Input/Output Subsystem (BIOS)
  - Other names: System BIOS, ROM BIOS, PC BIOS
- When the machine boots, CPU first runs the BIOS
  - The lowest level s/w that interfaces with hardware: read KB, write to display, disk I/O, etc
  - It checks which I/O devices (inc. disks) present and whether basic I/O devices working correctly by scanning PCIe/PCI buses (known as POST: Power-On Self Test phase)
  - Configures/initializes the hardware devices present
  - Then determines boot device (list of boot devices is stored in CMOS memory, in some order)

How do you start the OS?

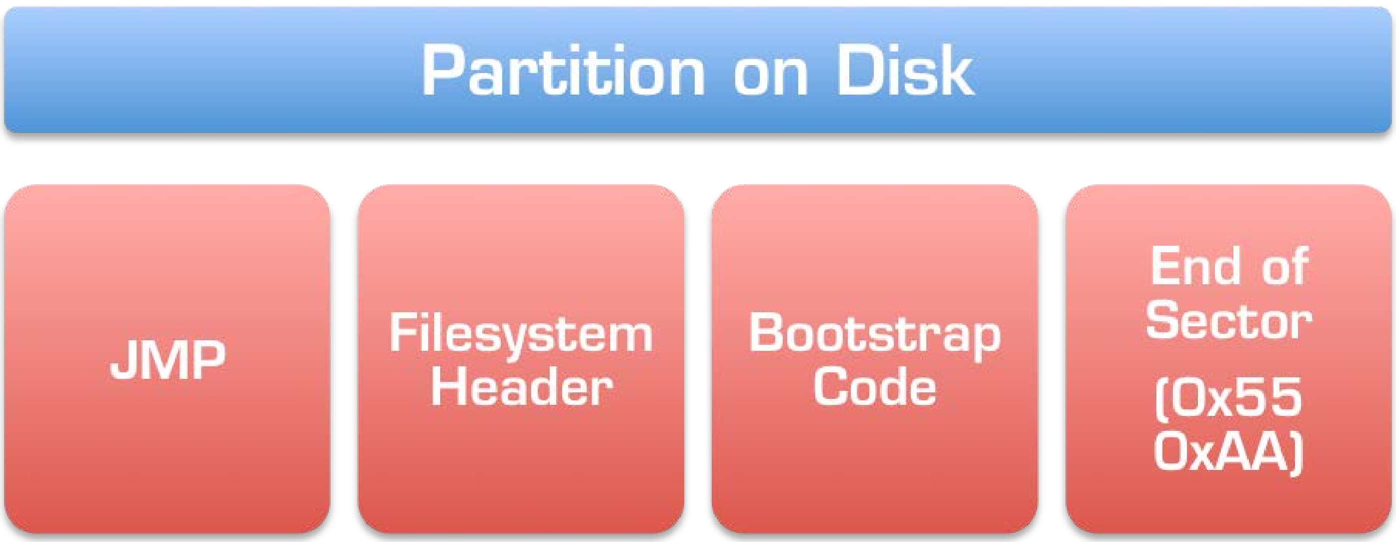
- The BIOS, in turn, loads a "small" OS executable (boot loader-1) aka MBR
  - From hard disk, CD-ROM, or Flash which is located in 1st sector of the bootable disk. Eg. /dev/hda, or /dev/sda
    - » MBR (boot loader-1) is written in a small, special-purpose file system that the BIOS does understand
  - Then transfers control to a standard start address in this MBR image of size 512 Bytes!
  - MBR loads and starts the "big" version of OS (real boot loader from active partition specified in partition table)
    - This multi-stage mechanism is used so that BIOS won't need to understand the file system implemented by the "big" OS kernel
    - File systems are complex data structures and different kernels implement them in different ways (FAT32/NTFS/ext2/ext3)

MBR

Only one partition can be marked as active at a time

Credits: NeoSmart Tech

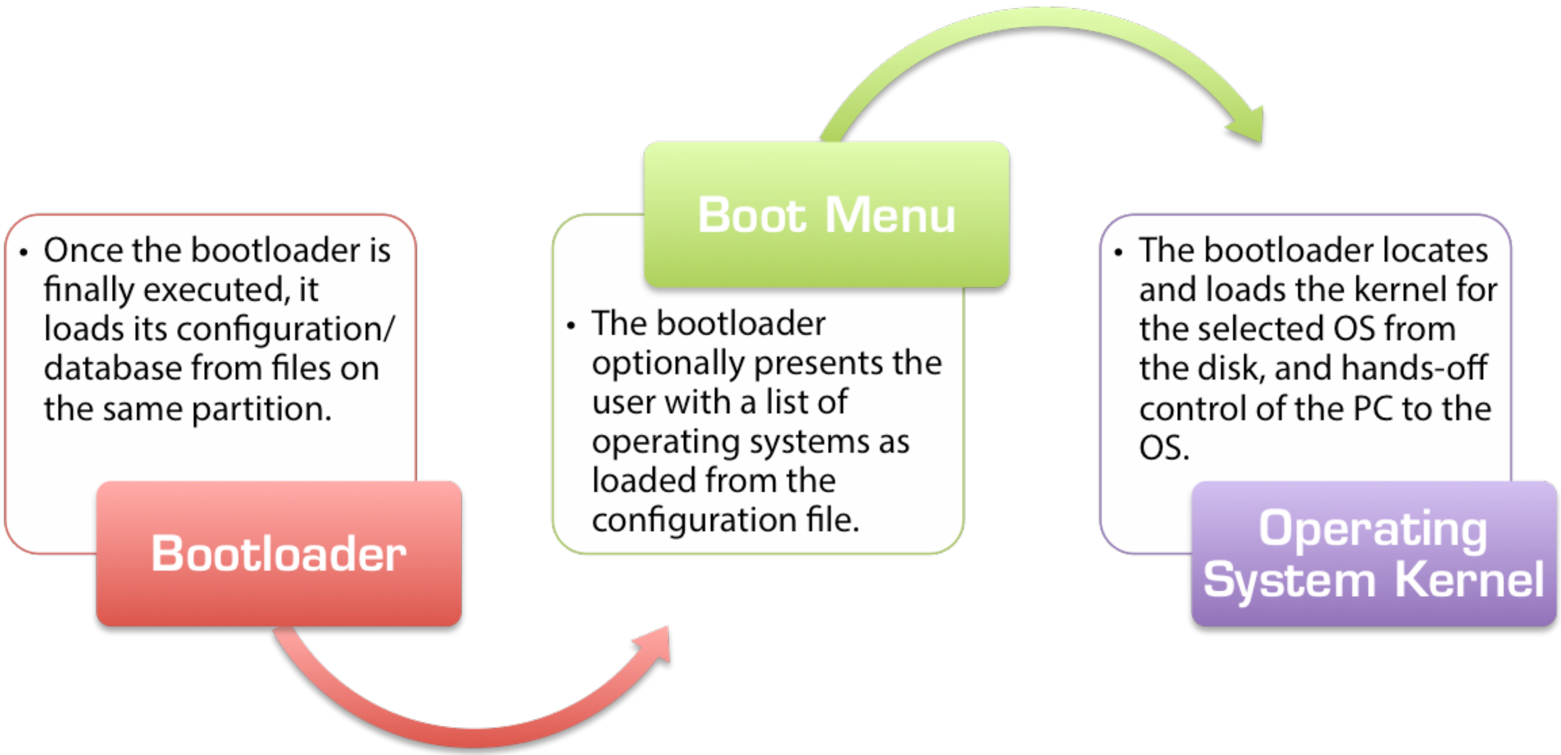
Active/Boot Partition on Disk



- This is all packed into the first sector (512 Bytes) of the partition
- CPU follows the JMP instruction and executes Bootstrap Code

Credits: NeoSmart Tech

Real Boot Loader



Credits: NeoSmart Tech

Typical Job of Boot Loader

Load basic filesystem drivers

The bootloader must load and run the primitive filesystem "drivers" that give it the ability to read, at the very least, the filesystem it is located on.

Load and read configuration file

With support for the filesystem loaded, the bootloader can now read the list of operating systems from the disk and prepare it for display.

Load and run supporting modules

If the configuration file specifies that additional modules are required, they're loaded and run accordingly.

Display operating system menu

The bootloader displays a list of operating systems for the user to choose from (if applicable), and optionally allow for specifying parameters and settings.

Load the selected OS

The bootloader can now load and execute the kernel, handing off control of the PC to the OS and ending its role in the boot process.

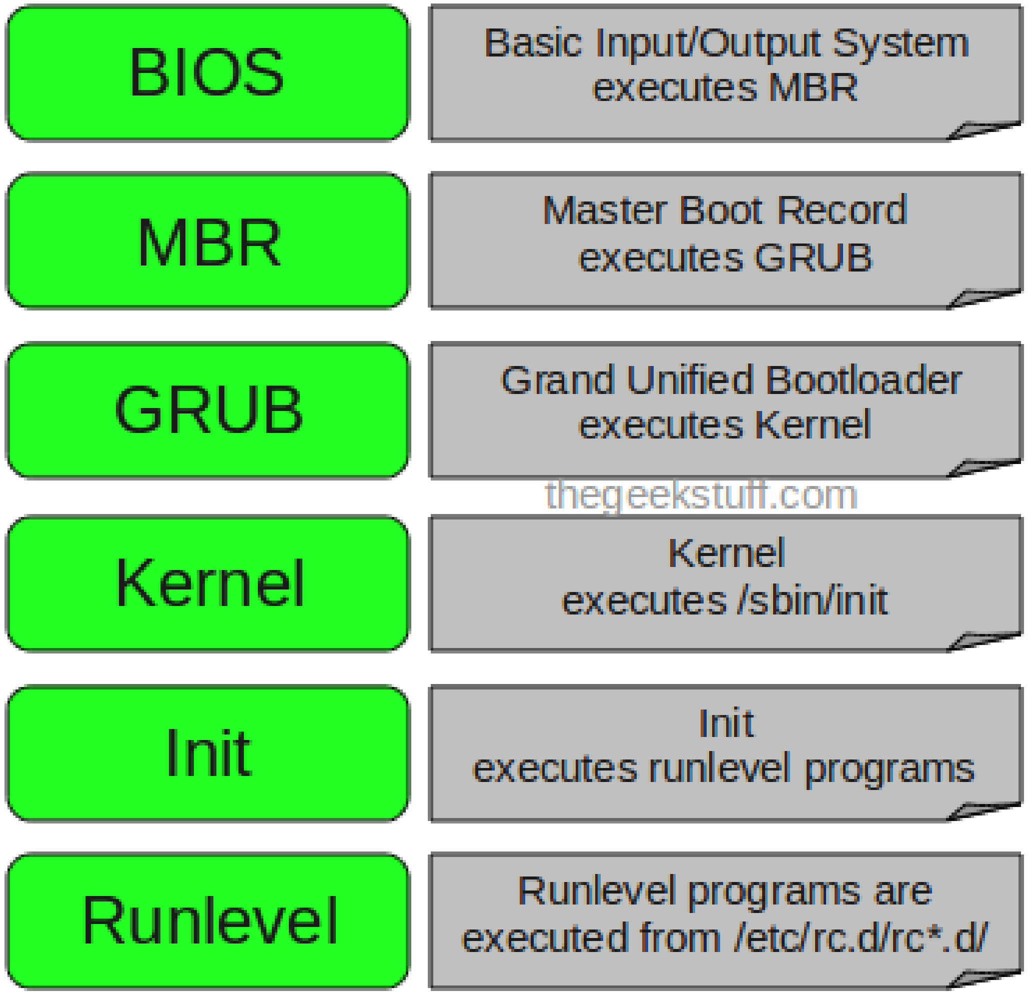
GRUB

- GRUB: Grand Unified Bootloader used in Linux
- GRUB has the knowledge of the filesystem unlike older LILO (LInux LOader)
- Grub config file is at PATH: /boot/grub/grub.conf (or menu.lst)
- GRUB just loads & executes Kernel and initrd images

```
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
hiddenmenu
title CentOS (2.6.18-194.el5PAE)
          root (hd0,0)
          kernel /boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/
          initrd /boot/initrd-2.6.18-194.el5PAE.img
```

- initrd stands for Initial RAM Disk which is used by kernel as temp file system which has essential drivers inside to access disk and other hardware
- Kernel mounts root filesystem /
- Kernel executes /sbin/init user-space program (PID=1)

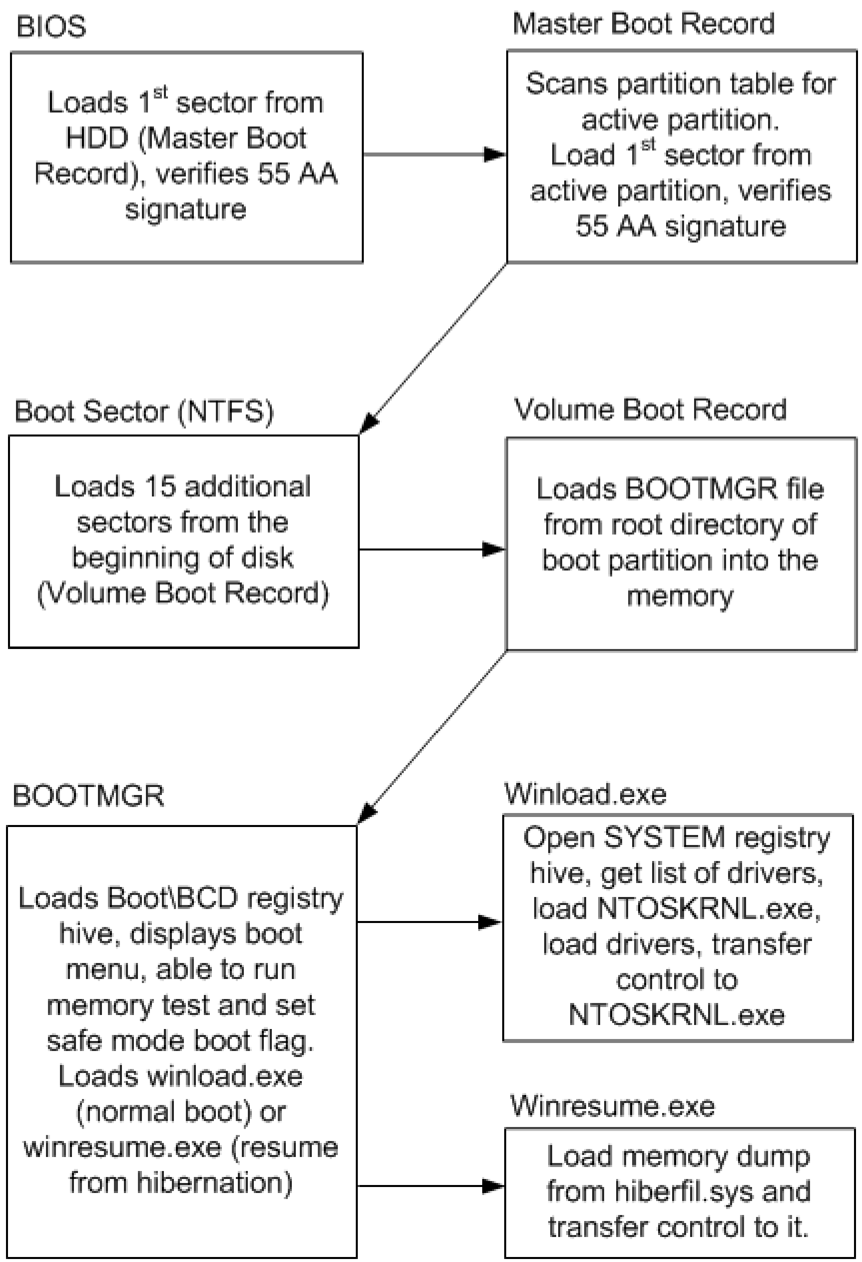
How do you start Linux OS?



man update-rc.d

Source: http://www.thegeekstuff.com/2011/02/linux-boot-process/
Video: http://www.youtube.com/watch?v=mHB0Z-HUauo

How do you start Windows OS?



Comparison of Boot Loaders

NTLDR	BOOTMGR	GRUB(2)
• NTLDR is the default bootloader for Windows NT, 2000, and XP. • BOOT.INI on the active partition contains the list of operating systems and their locations. • NTDETECT.COM is a helper program that runs to detect hardware and identify devices.	• BOOTMGR is the new Windows and is used on Windows Vista, 7, 8, and 10. • The list of operating systems is now read from the BCD file in the BOOT directory on the active partition. • BOOTMGR is self-contained, and does not need any helper programs or routines.	• GRUB is the most-popular bootloader for Linux, though it can boot numerous other OSes as well. • Its boot settings are stored in a file usually called grub.cfg (GRUB2) or menu.lst (GRUB). • GRUB is a modular bootloader, that can load additional modules from disk.

Credits: NeoSmart Tech

Troubleshooting Bootloaders

- EasyBCD: An easy-to-use utility that allows you to set up and configure a dual-boot or multi-boot between Windows, Linux, Mac, FreeBSD, etc

- Super GRUB2 Disk: A bootable GRUB2 disk that can be used to boot into Linux when your GRUB or GRUB2 is misconfigured or malfunctioning

Some interesting queries?

- How does boot process work in dual-boot m/cs like Windows 10 and Ubuntu?
- How does boot process work in Android/iOS?
- Why you need to typically install Windows first and then Linux?
- How about Mac and Linux dual-boot system?
- Why kernel is kept in compressed form in HDD/SSD?
- What is the use of Live-CD, Live USB?
- Secure boot, (Unified Extensible Firmware Interface) UEFI/GPT (GUID Partition Table) boot process in place of BIOS/MBR boot process
- Many many many more …
  - Refer Reading List at the end to find answers!

Administration

- Quiz-1 in next class (Nov 12th)
- Timing: 2PM to 2:45PM
- Syllabus: L1-L4

- Discussion slots:
  - Moving Wednesday slot (3-4pm) or Thursday slot (10-11am) to Tuesday (4-5pm) or Friday (3-4pm)
  - Attendance is mandatory to one of the slots
  - If miss the allotted slot, make it up by attending any other slot
  - Carry your laptop with Linux OS on a VM
  - Get your doubts clarified
  - Complete tasks given by TAs and help others if finished early!

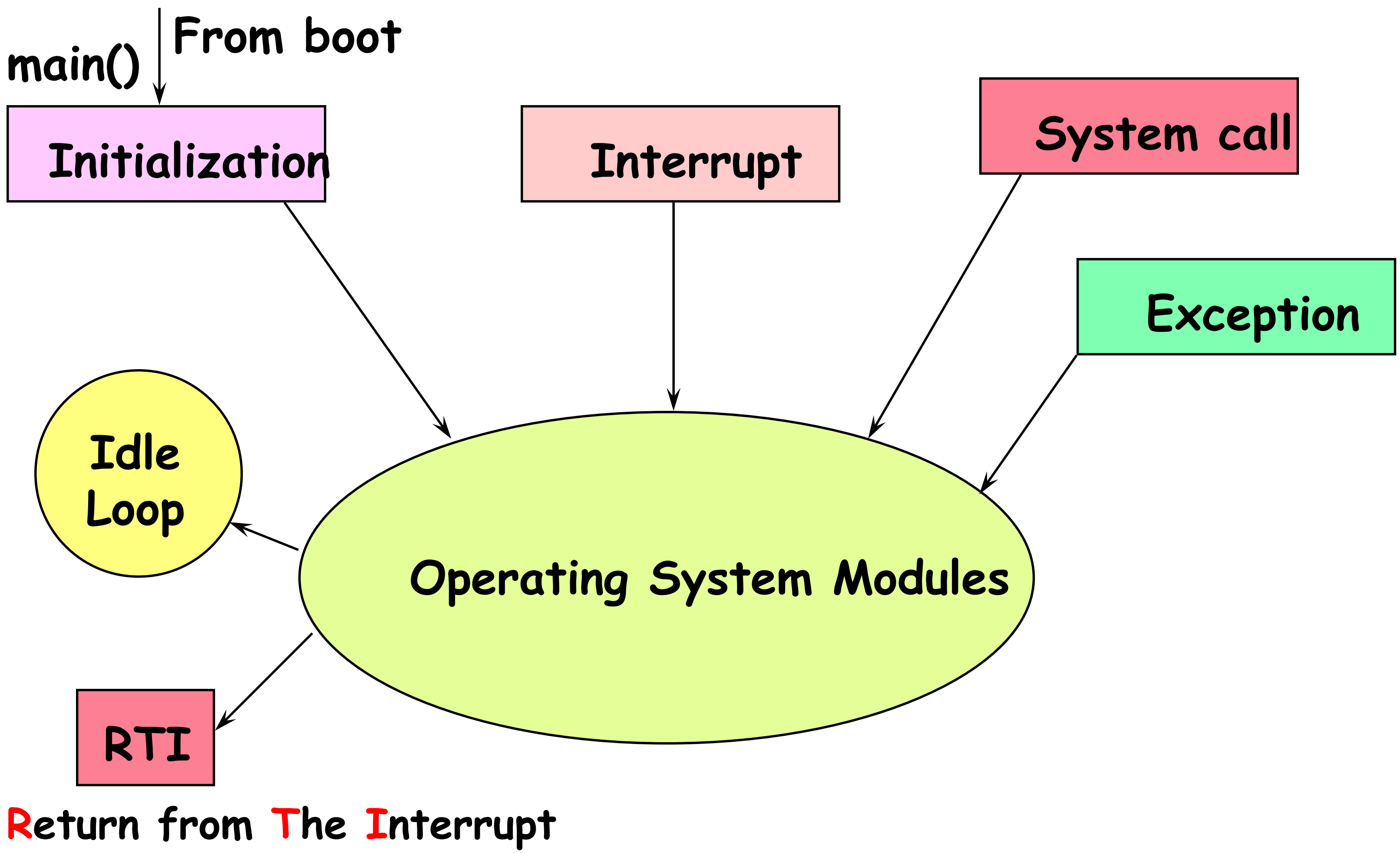
Operating System: A Process

- An OS is just another kind of program running on the CPU – a process:
  - It has main() function that gets called only once (during boot)
  - Like any program, it consumes resources (such as memory)
  - Can do silly things (like generating an exception), etc.
- But it is a very sophisticated program:
  - "Entered" from different locations in response to external events
  - Does not have a single thread of control
    - can be invoked simultaneously by two different events
    - e.g. sys call & a h/w interrupt
  - It is not supposed to terminate
  - It can execute any instruction in the machine

How does the OS do?

- OS runs user programs, if available, else enters idle loop
- In the idle loop:
  - OS executes an infinite loop (UNIX)
  - OS performs some system management & profiling
  - OS halts the processor and enter in low-power mode (laptops)
  - OS computes some function (DEC's VMS on VAX computed Pi)
- OS wakes up on:
  - interrupts from hardware devices
  - System calls from user programs
  - exceptions from user programs

OS Control Flow



Implementation Issues
(How is the OS implemented?)

- Policy vs. Mechanism
  - Policy: What do you want to do?
  - Mechanism: How are you going to do it?
  - Should be separated, since both change
  - Example: timer construct to implement mechanism, policy: different time slices
  - Example: Solaris: Scheduling thru loadable tables
  - Windows&Mac: tie policy & mechanism tightly to get unified look and feel
  - Microkernels: Policy & Mechanisms are decoupled
- High-level lang vs assembly lang tradeoff
  - MS-DOS in 8088, Linux/Unix/Windows in C
- Algorithms used: Linear, Tree-based, Log Structured, etc...
- Backward compatability issues
  - Very important for Windows
- System generation/configuration
  - How to make generic OS fit on specific hardware

Conclusion

- POSIX API ←→ System calls
- OS Boot Process
- Policy vs Mechanism
  - Crucial division: not always properly separated!

Reading and Viewing Assignments

- Appendix A from Understanding Linux Kernel by Bovet et al
- http://www.gnu.org/software/libc/
- http://www.ibm.com/developerworks/library/l-linuxboot/
- https://www.linuxbabe.com/desktop-linux/legacy-bios-vs-uefi-bios
- http://thestarman.narod.ru/asm/mbr/
- http://en.wikipedia.org/wiki/GNU_GRUB & https://www.gnu.org/software/grub/manual/grub/
- http://www.dedoimedo.com/computers/grub-2.html
- http://ubuntuguide.org/wiki/Multiple_OS_Installation

http://en.wikipedia.org/wiki/Master_boot_record

- http://www.gnu.org/software/libc/documentation.html
- Man syscall, syscalls, intro (man –a intro), libc, etc
- https://www.kernel.org/doc/man-pages/
- https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface

* Professor Messer's Linux+ Training:

http://www.youtube.com/playlist?list=PLCDA423AB5CEC8FDB

http://www.youtube.com/watch?v=6eTi2qu4Fb0&feature=c4-overview&list=UUkefXKtInZ9PLsoGRtml2FQ