Linux Kernel

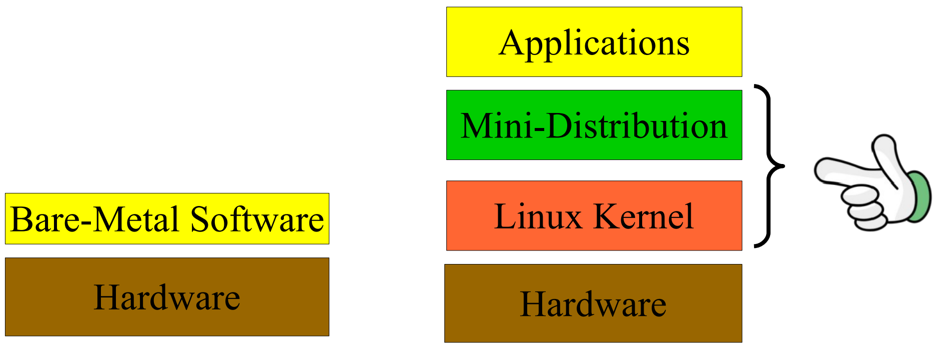
Pr. Olivier Gruber
(olivier.gruber@imag.fr)

Laboratoire d'Informatique de Grenoble
Université de Grenoble-Alpes

This Course Summary

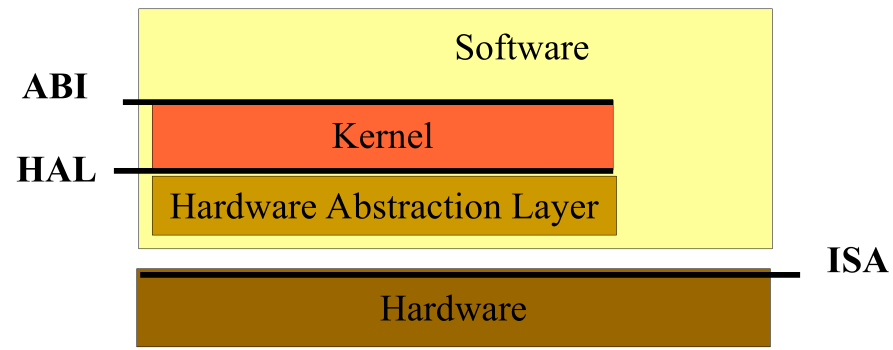
Part-I Bare-metal programming
- Bare-metal programming
- **Making your own minimal distribution**

Part-II RTOS programming
- RTOS programming
- IoT Infrastructure



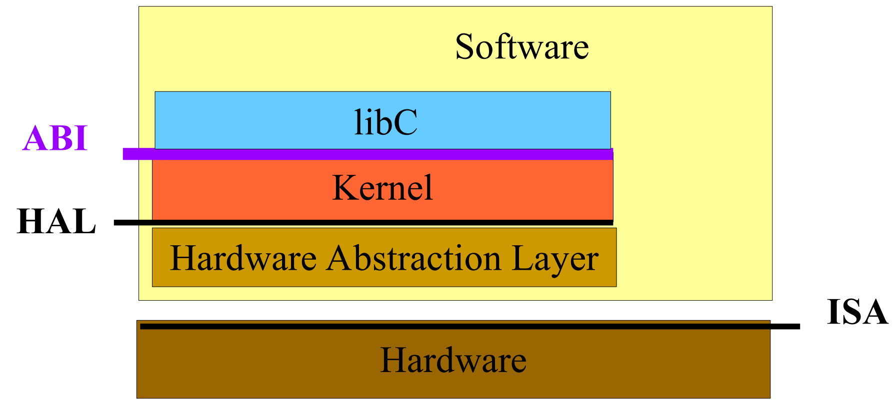
Linux Kernel

- Major Interfaces
  - Instruction Set Architecture (ISA)
  - Hardware Abstraction Layer (HAL)
  - Application Binary Interface (ABI)
- Discussion Points
  - Processes
  - Configurable kernel
  - Minimal file-system layout
  - Kernel boot process



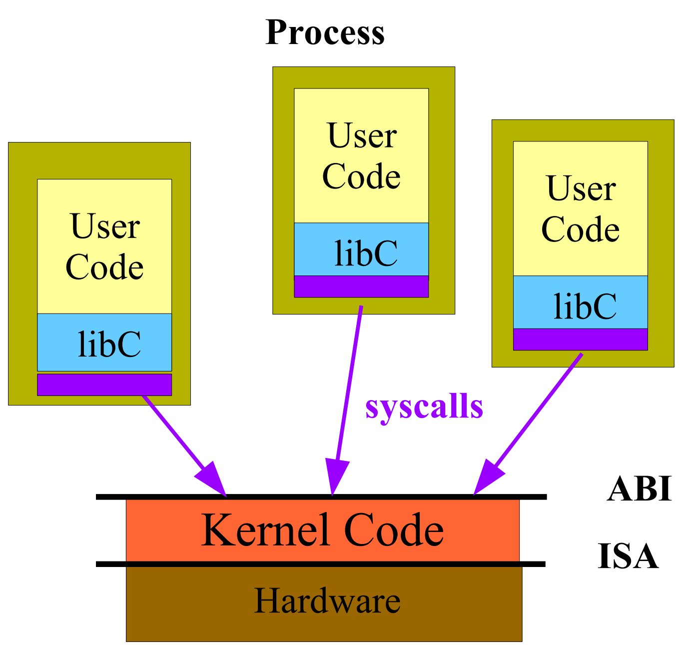
Linux Kernel

- Major Interfaces
  - Instruction Set Architecture (ISA)
  - Hardware Abstraction Layer (HAL)
  - Application Binary Interface (ABI)
- Major concepts
  - Processes and Files



Syscalls: the frontier between user and kernel lands

The kernel acts as a coordinator and controller

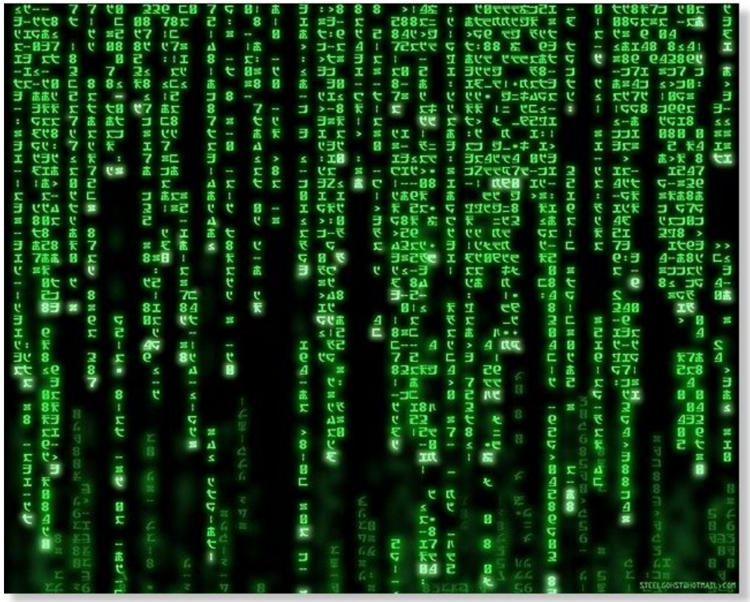


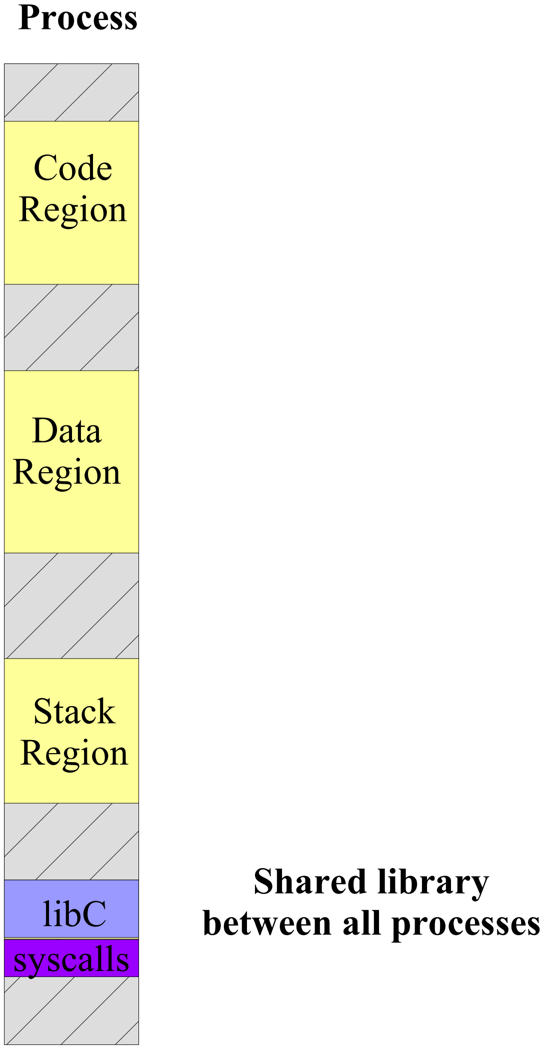
Process Illusion

- The *"Process Instruction Set Architecture"*
  - Memory map based on regions
  - Entry point for the execution flow
  - Regular assembly instruction set
  - *Kernel services* via syscalls

Morpheus: "A prison that you cannot see, touch, or smell…"

Syscalls are the only door to the outside of the illusion...

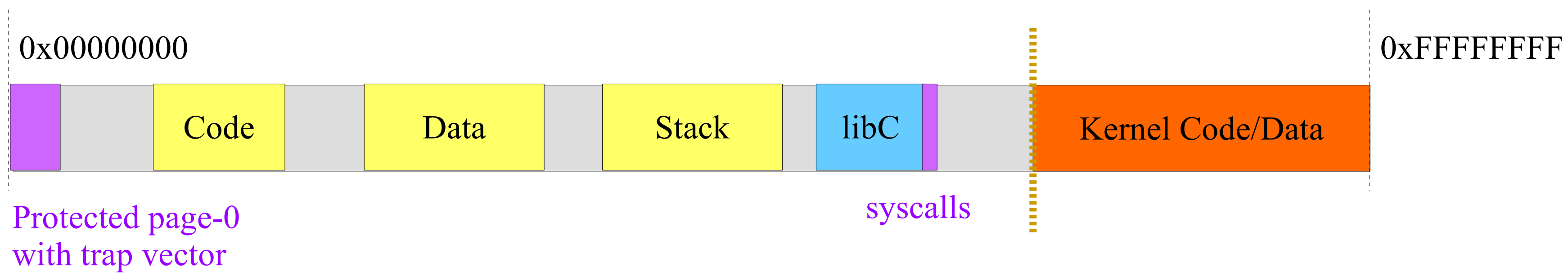




System Calls – Gates to the Outside

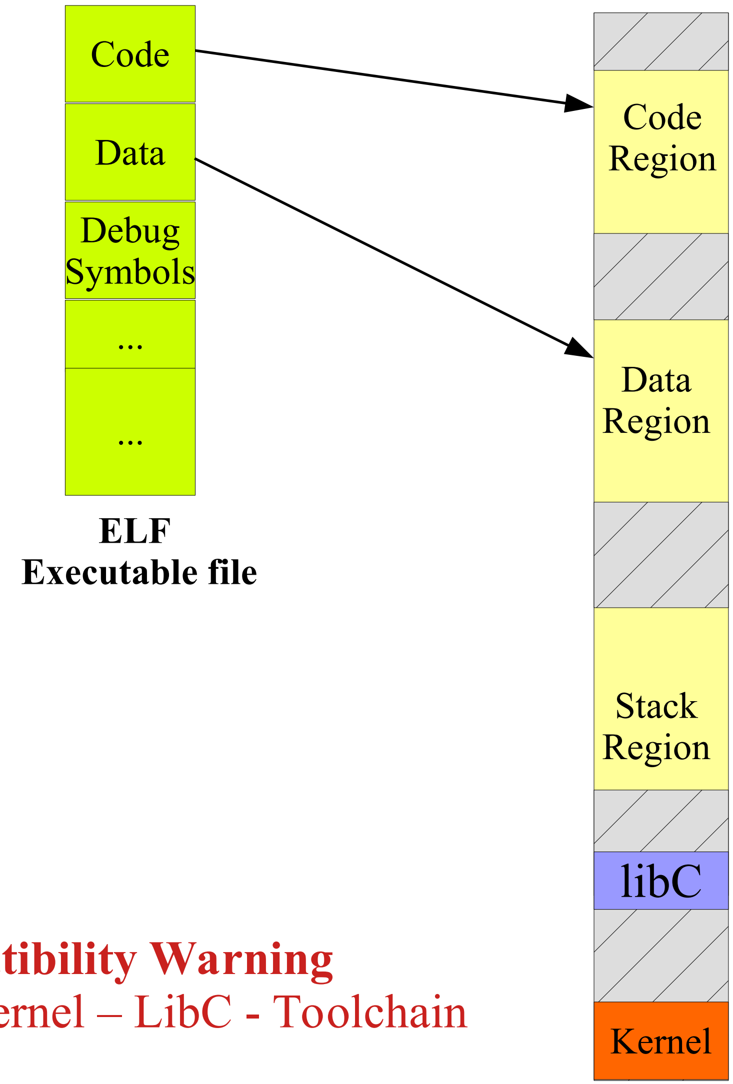
- Requires a way to pass arguments and return values
  - That is, requires a calling convention
  - Using processor registers
  - Using memory allocated buffers
- Requires the kernel to be mapped within the process
  - Syscall is a trap → goes through the trap vector
  - The processor mode changes to *kernel mode*
  - Requires a special instruction to return to user mode

```
/*
 * void close(int fd);
 */
.close
        push %ebp
        mov %ebp,%esp
        sub %esp,#0x10

        mov %ebx, %eax
        mov %eax, #0x05
        syscall

        mov %esp,%ebp
        pop %ebp
        ret
```



Process Creation

- Process creation
  - Syscalls: fork / execv(filename)
  - Executable file: **ELF format** in recent Linux
  - **ELF loader in the kernel**
- GNU Toolchain
  - GNU C **compiler**
    - Source to object files (ELF format)
  - GNU C **linker**
    - Linker script governs the layout of the process regions
    - Entry point[(1)]: *_start* in **crt0.S** → calls *main*
  - GNU standard C library (**libC**)
    - Wraps the Linux kernel syscalls



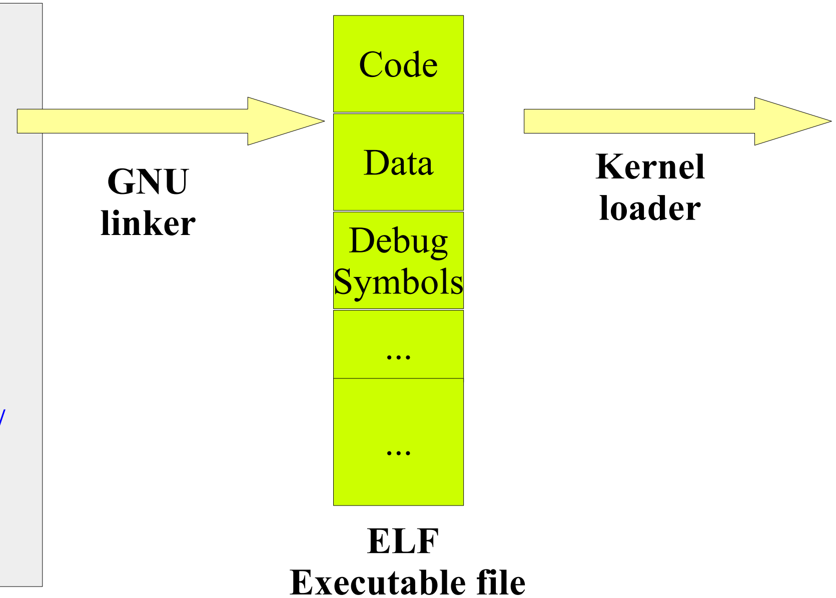
Compatibility Warning
Hardware – Kernel – LibC - Toolchain

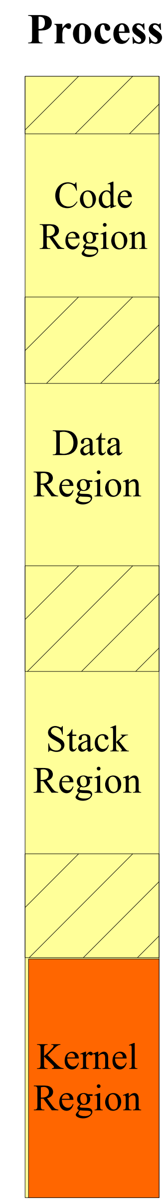
(1) https://en.wikipedia.org/wiki/Crt0

Process Layout – Linker/Loader Cooperation

- Kernel Loader – ELF executable
  - ELF regions know where they should be loaded in memory
- Linker Script[1]
  - The hidden glue: tells where each section goes in memory
  - One default script (tool-chain configuration)
  - Can provide your own when invoking the linker

```
ENTRY(_start)
SECTIONS {
    . = 0x10000;
    .startup . : { startup.o(.text) }
    .text : { *(.text) }
    .data : { *(.data) }
    .bss : { *(.bss) }
    . = ALIGN(8);
    . = . + 0x1000; /* 4kB of stack memory */
    stack_top = .;
}
```

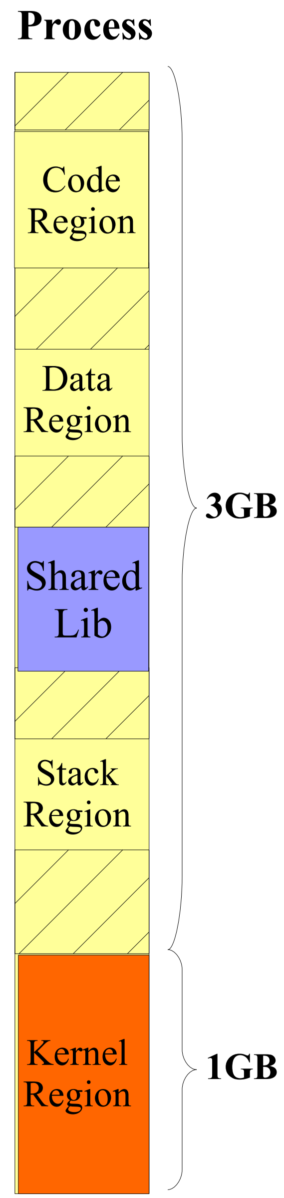




(1) https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_node/ld_6.html

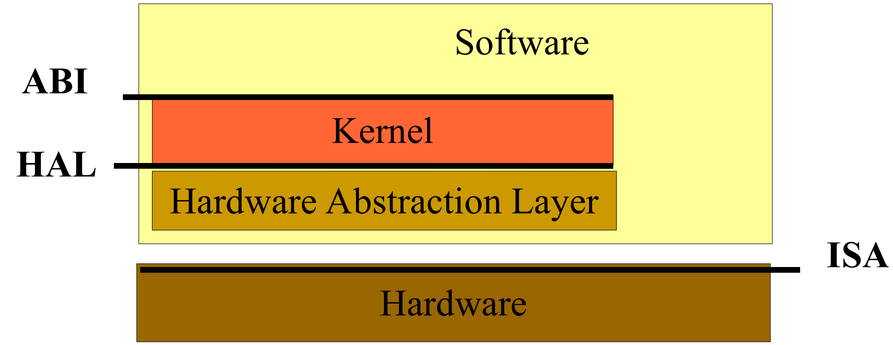
Shared Libraries

- When linking libraries
  - Libraries can be statically linked or shared
  - Note: to be shared, a library must be compiled specifically
- How do we know the shared libraries needed by an executable?
  - *$ **ldd** /bin/ls* (*only for trusted executable*)
  - *$ **objdump -p** /bin/ls | grep NEEDED*
- Two kinds of libraries
  - Those that will be loaded dynamically
    - They are known to be somewhere in the root partition
    - E.g.: **libc.so.6** or **ld-linux.so.2** somewhere on disk
  - Those provided by the kernel
    - They have no path, rather a load address
    - Example: **linux-gates.so.1 or linux-vdso.so.1**



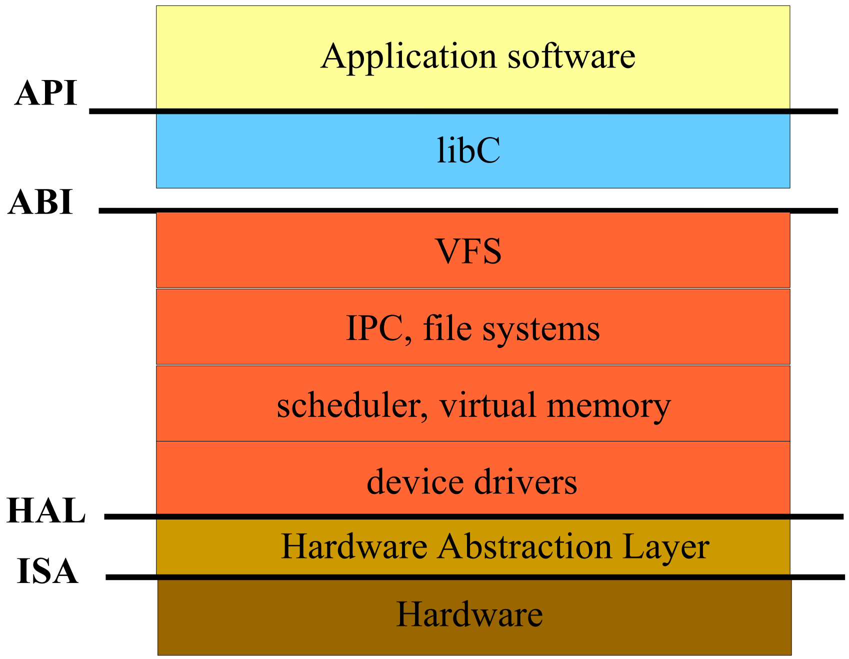
Linux Kernel

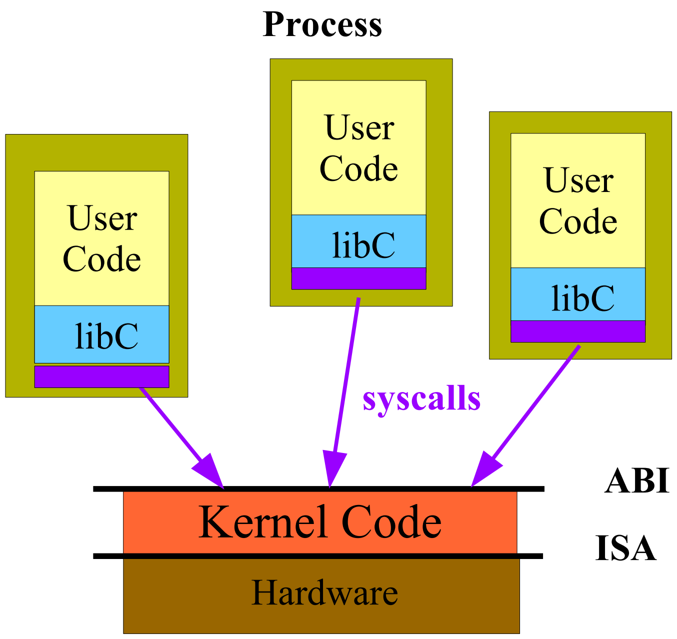
- Major Interfaces
  - Instruction Set Architecture (ISA)
  - Hardware Abstraction Layer (HAL)
  - Application Binary Interface (ABI)
- Discussion Points
  - Processes
  - Configurable kernel
  - Minimal file-system layout
  - Kernel boot process



Configurable Kernel

- Core concepts
  - Processes, threads, files, inter-process communications
- Configurable kernel at build time[1]
  - Configuration "*à la carte*" – certain features are included/excluded/modules





(1) https://wiki.archlinux.org/index.php/Kernel/Traditional_compilation

Linux Kernel Compiling

- Which version of the linux kernel?
  - Version 2.6 is still used in embedded systems
    - Much simpler to configure and much smaller
    - But no longer compatible with GCC 4.8 and above (ABI, see ELF header)
  - Version 4.4 and up are compatible with GCC 5.0
    - Easier for us to use this version
    - We can use the tool-chain of your host Linux

- Linux Kernel Compilation
  - For a given architecture "i386"
  - (a) configure
  - (b) compile the kernel
  - (c) compile the modules

```
$ make ARCH=i386 menuconfig
...

$ make ARCH=i386  bzImage
...
Kernel is arch/x86/boot/bzImage

$ make ARCH=i386  modules
```

Linux Kernel Installation

- Regular installation steps
  - Set the proper environment variables
  - Set INSTALL_PATH and INSTALL_MOD_PATH
  - **OTHERWISE WILL INSTALL ON /**
  - Kernel-2.6.32.65 install → the following files in /*your*/*full*/*path*/**boot**
    - vmlinuz-2.6.32.65
    - config-2.6.32.65
    - System.map-2.6.32.65
  - Module install
    - Modules under /*your*/*full*/*path*/**lib/modules/2.6.32.65/**

```
$ export ARCH=i386
$ export INSTALL_PATH=/your/full/path/boot
$ export INSTALL_MOD_PATH=/your/full/path

$ make install
$ make modules_install
```

Configurable Kernel – Modules

- Dynamic loading/unloading (garbage collection)
  - Look at **modprobe**, **insmod**, **rmmod**, and **lsmod**
  - Tools require **/proc** to work...
- Under **/lib/modules/*kernel-version***
  - Module "*database*", with dependencies
  - Created by running the tool "depmod"

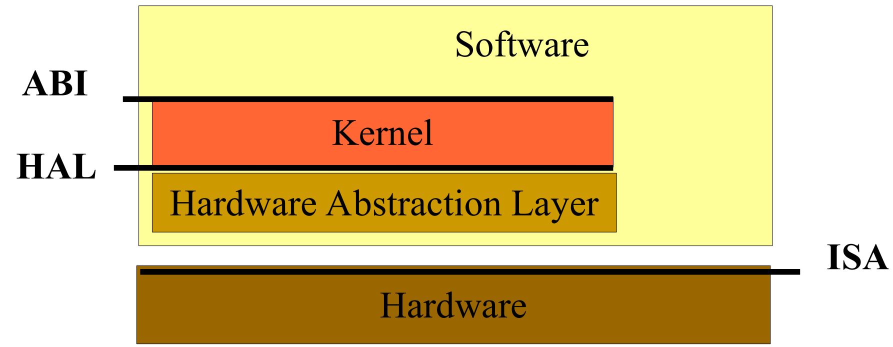
$ ls /lib/modules/3.13.0-105/

build/	modules.builtin.bin	modules.symbols
initrd/	**modules.dep**	modules.symbols.bin
kernel/	modules.dep.bin	updates/
modules.alias	modules.devname	vdso/
modules.alias.bin	modules.order	
modules.builtin	modules.softdep	

$ **depmod** -ae -F /boot/**System.map**-3.13.0-105 -b / -r 3.13.0-105

Linux Kernel

- Major Interfaces
  - Instruction Set Architecture (ISA)
  - Hardware Abstraction Layer (HAL)
  - Application Binary Interface (ABI)
- Discussion Points
  - Processes
  - Configurable kernel
  - Minimal file-system layout
  - Kernel boot process



Linux Kernel Layout

- Kernel files[1]
  - Compressed kernel image: /boot/**vmlinuz**-2.6.32.65
  - Kernel build configuration: /boot/**config**-2.6.32.65
  - Kernel debug symbols: /boot/**System.map**-2.6.32.65
- Initial ram-disk[2]
  - Optional: /boot/initrd-2.6.32.65
- Modules
  - Optional: /lib/modules/2.6.32.65/
- Shared libraries
  - Optional: ***LD_LIBRARY_PATH***=*/lib:/usr/lib*

(1) *Always deploy your* ***kernel*** *and its* ***system map*** *under /boot,*
along with the ***kernel configuration*** *that was used to build that kernel*

(2) *Optional, you can boot directly from a hard disk, see kernel option "root="*

GNU C Library "*Quirk*"

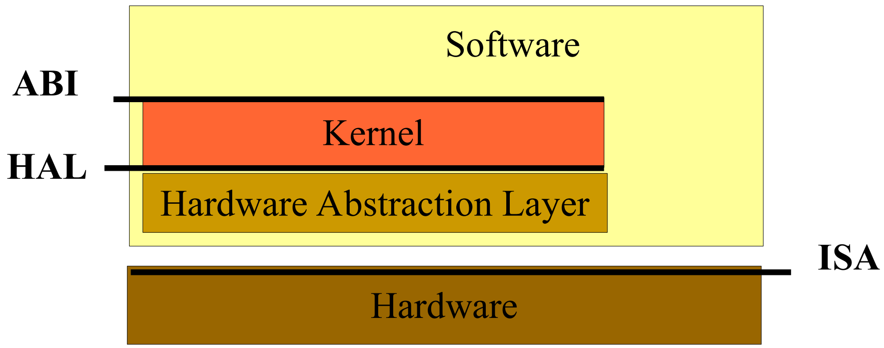
- One would expect
  - That the libC only relies on syscalls and does not depend on any other libraries
  - And if the libC did rely on other libraries that it would be known dependencies
- Unfortunately
  - Not everything has kernel support directly built-in as syscalls
  - The libC **may** need other libraries, but they are not visible as dependencies(1)
  - **May** means it can run without, but some functions will fail, more or less silently...
- Example: **Name Service Switch**
  - Managing naming services such as host names, user names, or group names
  - Configuration file: **/etc/nsswitch.conf**
  - Will load shared library: libnss_dns.so.2 (2)

```
# /etc/nsswitch.conf
# Name Service Switch configuration file.

passwd:     files
group:      files
hosts:      files dns
```

(1) Not listed by ldd. How do you know the dependencies? You don‘t...
(2) current interface version is 2

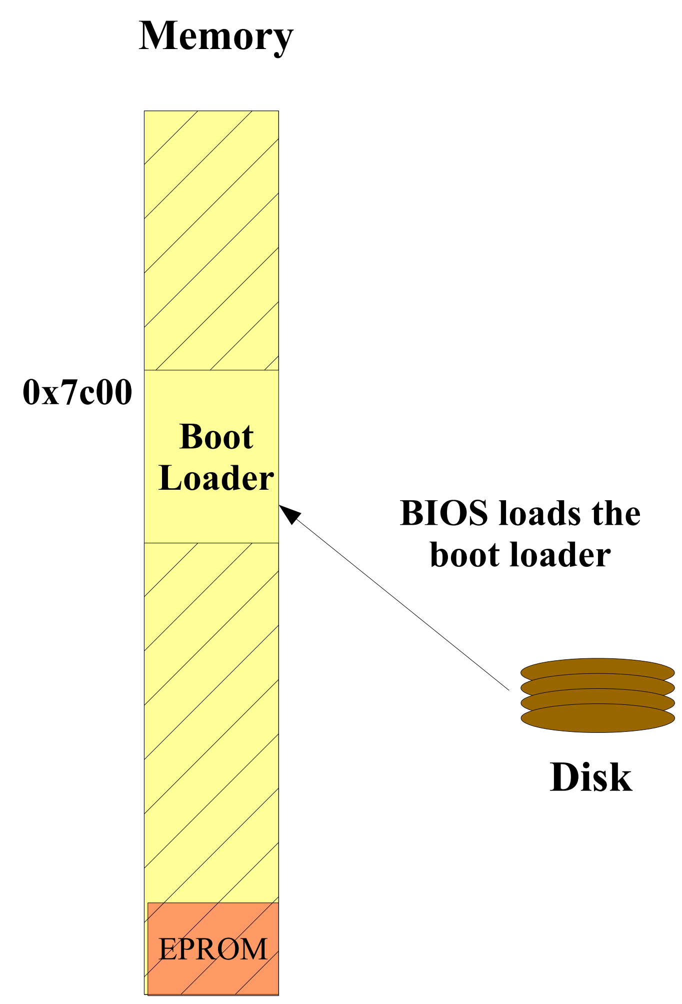
Linux Kernel

- Major Interfaces
  - Instruction Set Architecture (ISA)
  - Hardware Abstraction Layer (HAL)
  - Application Binary Interface (ABI)
- Discussion Points
  - Processes
  - Configurable kernel
  - Minimal file-system layout
  - Kernel boot process
    - BIOS
    - Boot loader
    - Kernel



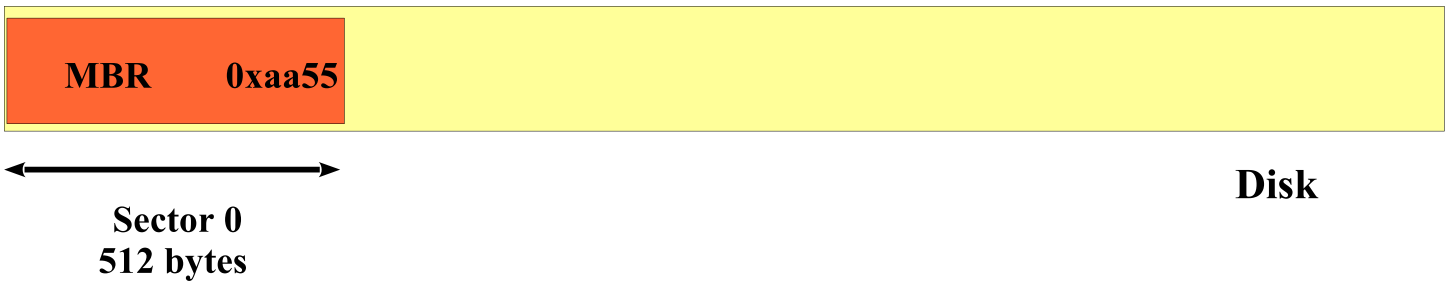
Old Intel Boot Process (not UEFI)

- Hardware wakes up after reset
  - Execute from EPROM/Flash, often called firmware
  - In the Intel world, it is called the BIOS
- BIOS is resident in memory
  - Initializes enough devices (SD card, hard disks, etc.)
  - Provides Basic Input/Output Services (BIOS)
  - Loads the boot loader from one pheripheral device
- Boot loader is the next stage in the boot process
  - Usually means more functionalities than the BIOS
  - It is also the first code that is more portable
  - Relies on the hardware initialization done by the BIO



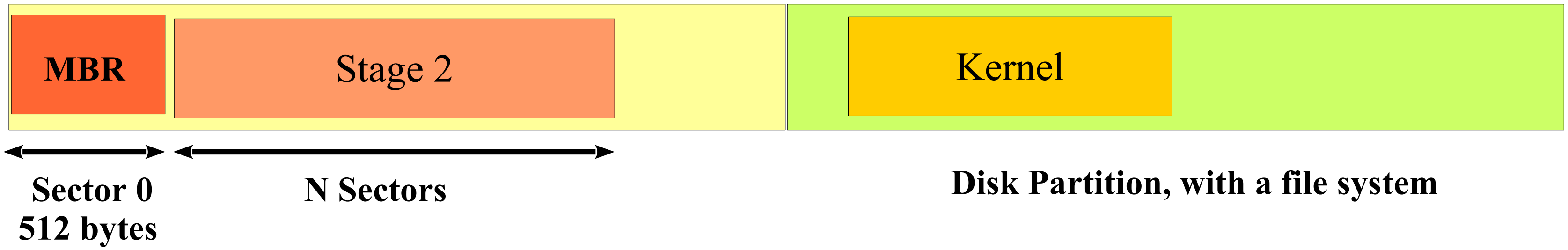
Master Boot Record

- Master Boot Record (first disk sector)
  - Describes the partitions, if any
  - If bootable disk, the MBR ends with 0xaa55 (at offset 0x1fe)
  - If so, it contains a small program, loaded by the BIOS at 0x7c00
  - BIOS transfers the execution at 0x7c00, but in 16bit legacy mode
  - The code usually loads a larger kernel...



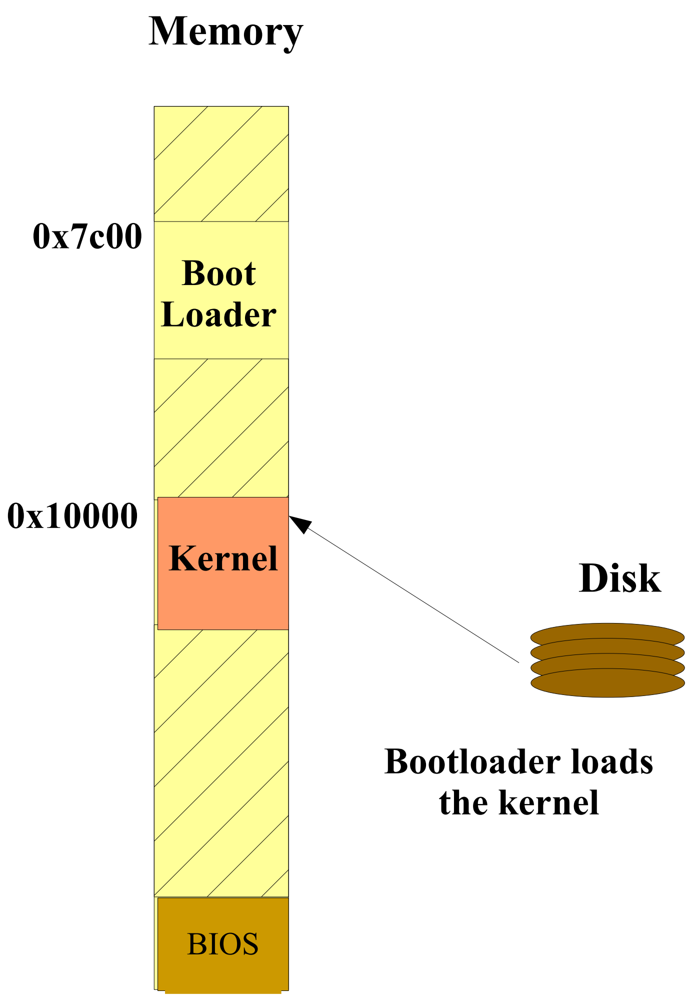
GRUB Boot Loader Details

- Uses multiple stages, because the MBR is really small
  - Stage1 is the MBR, stage2 is the rest of the boot loader, sometimes there is a stage1.5
- Ultimate goal
  - Loads the kernel from a partition with a file system
  - Must therefore have the code to understand that file system structure on disk
- Possible Layout on disk



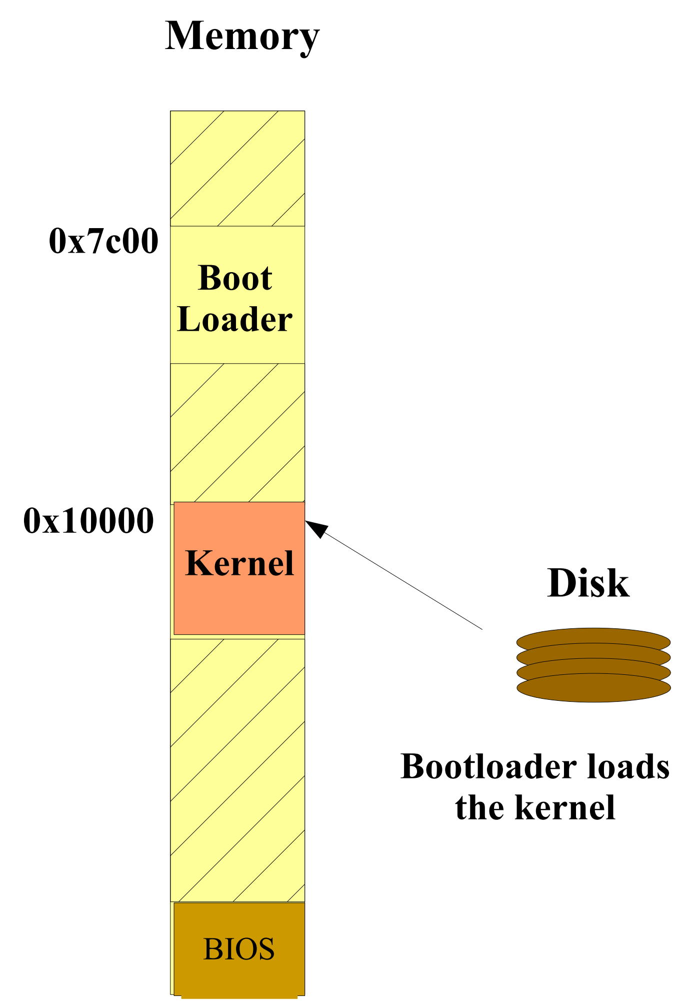
Kernel Loading

- The kernel is loaded at 0x10000 (convention)
  - The kernel takes over the machine
  - The boot loader is no longer needed
  - But the BIOS remains, it may be used or not
- Kernel boot options (given by the boot loader)
  - "**root**" option: where the root file system is
  - "**init**" option: the *ELF* executable file for the first process



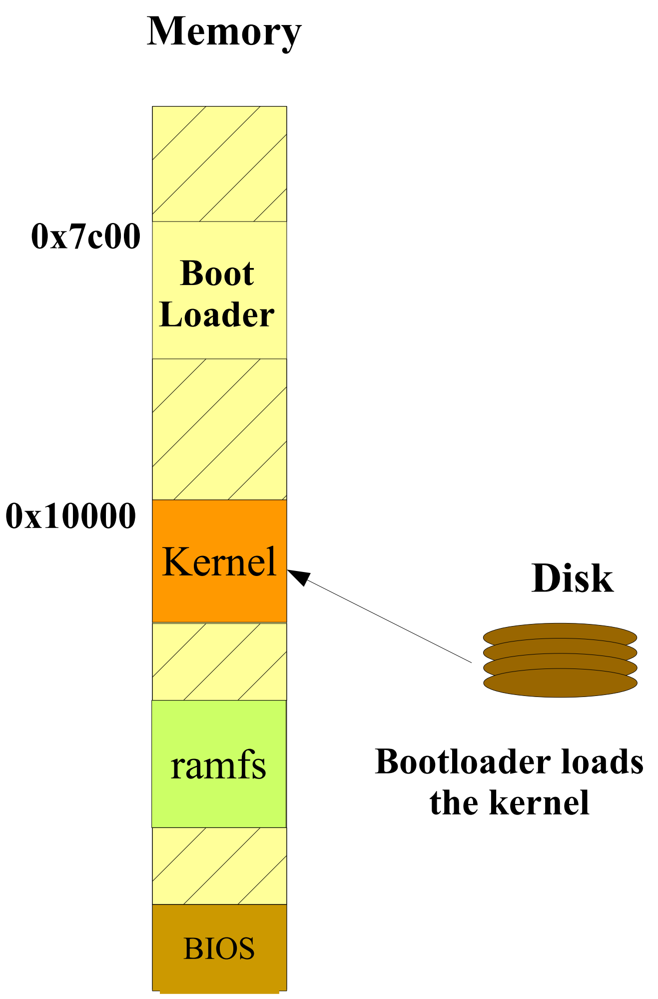
Kernel Loading

- Root as a file system in a disk partition
  - Requires the kernel to have the right support
    - The corresponding file-system type (FAT,EXT2,NTFS)
    - The support for the right bus (IDE,SATA)
    - The support for the right device driver for the hard-disk
  - Ideal for building a specific kernel for a specific hardware
- Traditional setup
  - One disk, with one or more partitions
  - One of the partition is used as the root partition
  - The root partion is mounted read-only by the kernel



Kernel Loading

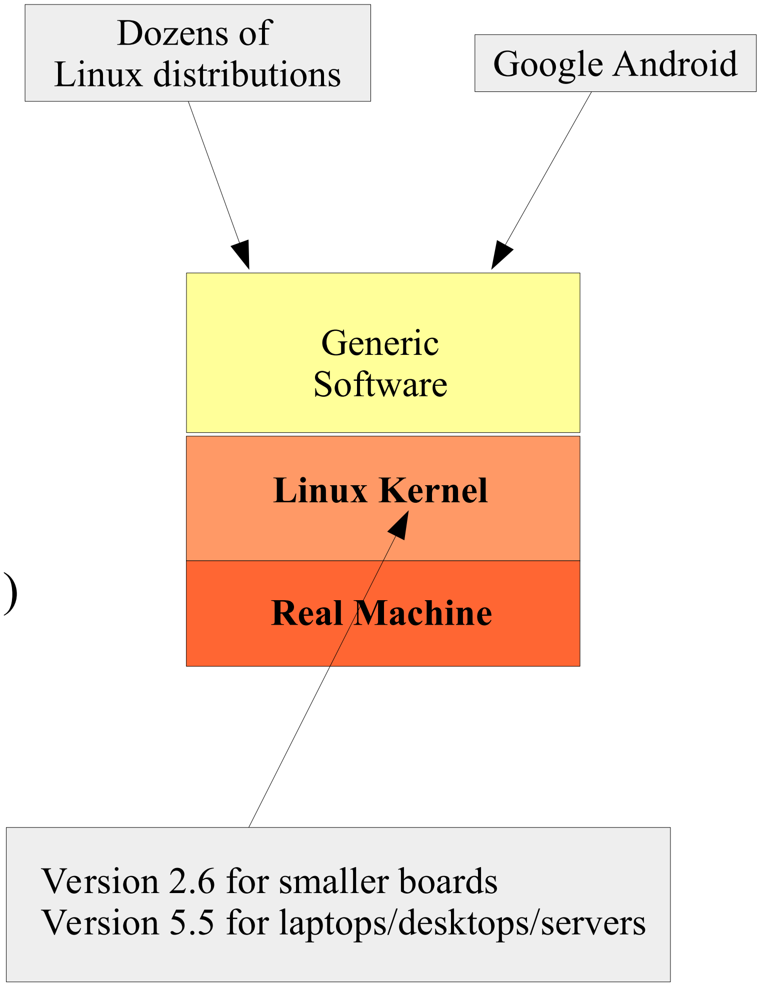
- Root as an in-memory file system (ram-fs)
  - Requires only a small run-everywhere support in the kernel
  - The root partion is mounted read-only by the kernel as a ram-fs
- Initial Ram-Disk (initrd)
  - The initrd is a file somewhere on in a disk partition
    - The file is loaded in memory by the boot loader
    - The disk partition and file system must be readable by the boot loader



The "init" Process

- The frontier – Kernel vs Distribution
  - When the kernel has finished booting
  - It creates a first process, from an ELF executable
  - That is the start of the distribution…
- One Linux Kernel versus many Linux distributions
  - Ubuntu, Debian, Mint, ArchLinux, etc…
  - But still, many kernel versions...
- Google Android
  - Linux kernel with a few C libraries (webkit,codecs,etc.)
  - Java environment for services and tools

But that is another story...



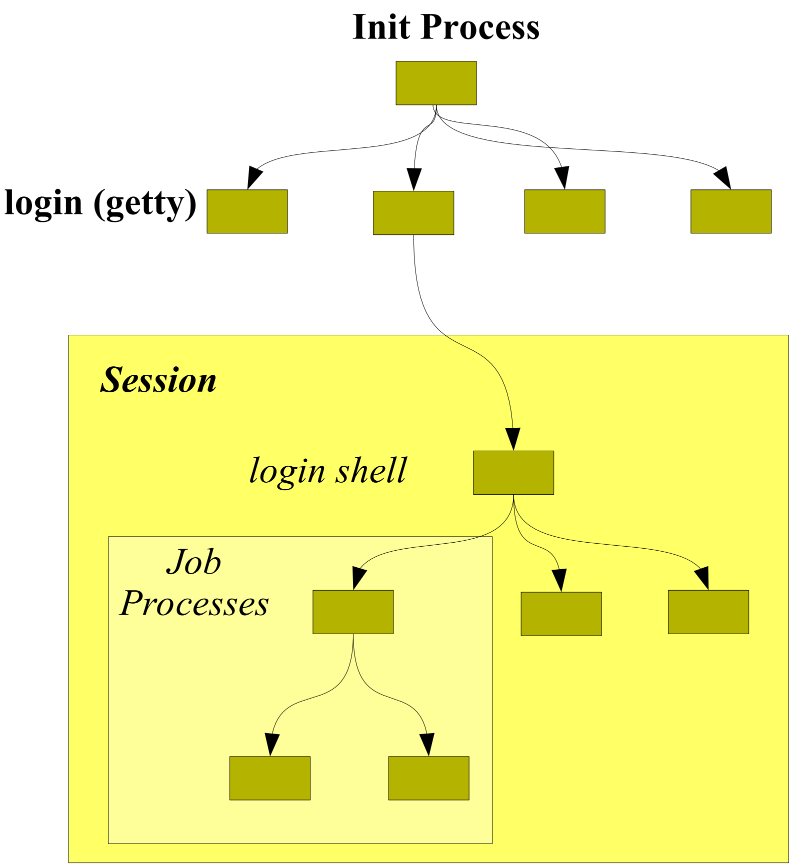
Linux Distribution

- Linux: almost everything is a **file** or a **process**
  - *Process:* executing threads[(1)]
  - *File:* a stream of bytes
- File system
  - Hierarchical name space – directories and files
- Processes
  - Tree of processes, the root is the “init” process

(1) Linux does not have a concept of threads within processes,
threads are processes, sharing the same virtual address space

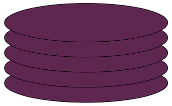
Linux Process Tree

- Kernel creates the first process
  - Right after the kernel is done booting..
- User logging per *terminal* for a *session*
  - Credentials: logins and passwords
  - Once authenticated → create a session
  - Running the first login *shell* to submit *jobs*
- Tree of processes
  - *Root process (called "init")*
    - *The first process created at boot time*
  - *Process attributes*
    - *SessionId (associated to a user), GroupId (job)*
  - *A job*
    - *A set of related processes as a group of processes*



File System

- File system
  - Hierarchical name space – directories and files
  - Resides in a disk partition that is seen as a block devices
- Mounted block devices
  - Block devices – a sequence of blocks
  - Mounted block devices
- Special file system: /proc



/dev/sda : complete disk
/dev/sda1 : partition 1
/dev/sda2 : partition 2

Partitions are described in the MBR

```
$ more /etc/fstab
/dev/sda1    /       ext4    rw   0 1
/dev/sda2    none    swap    sw   0 0

$ mount
/dev/sda1  on  /       type  ext4  (rw,…)
proc       on  /proc   type  proc  (rw,...)
```

Device Files (/dev)

- Special files under **/dev** (called nodes)
  - Created nodes with mknod
  - Specifying if it is a char or block device
  - Identified by a major and minor number

```
$ mknod console c 5 1
$ mknod sda b 8 0
```

- Associated device driver
  - Major and minor are used to identify a device driver in the kernel
  - Char or Block devices

$ ls -al /dev

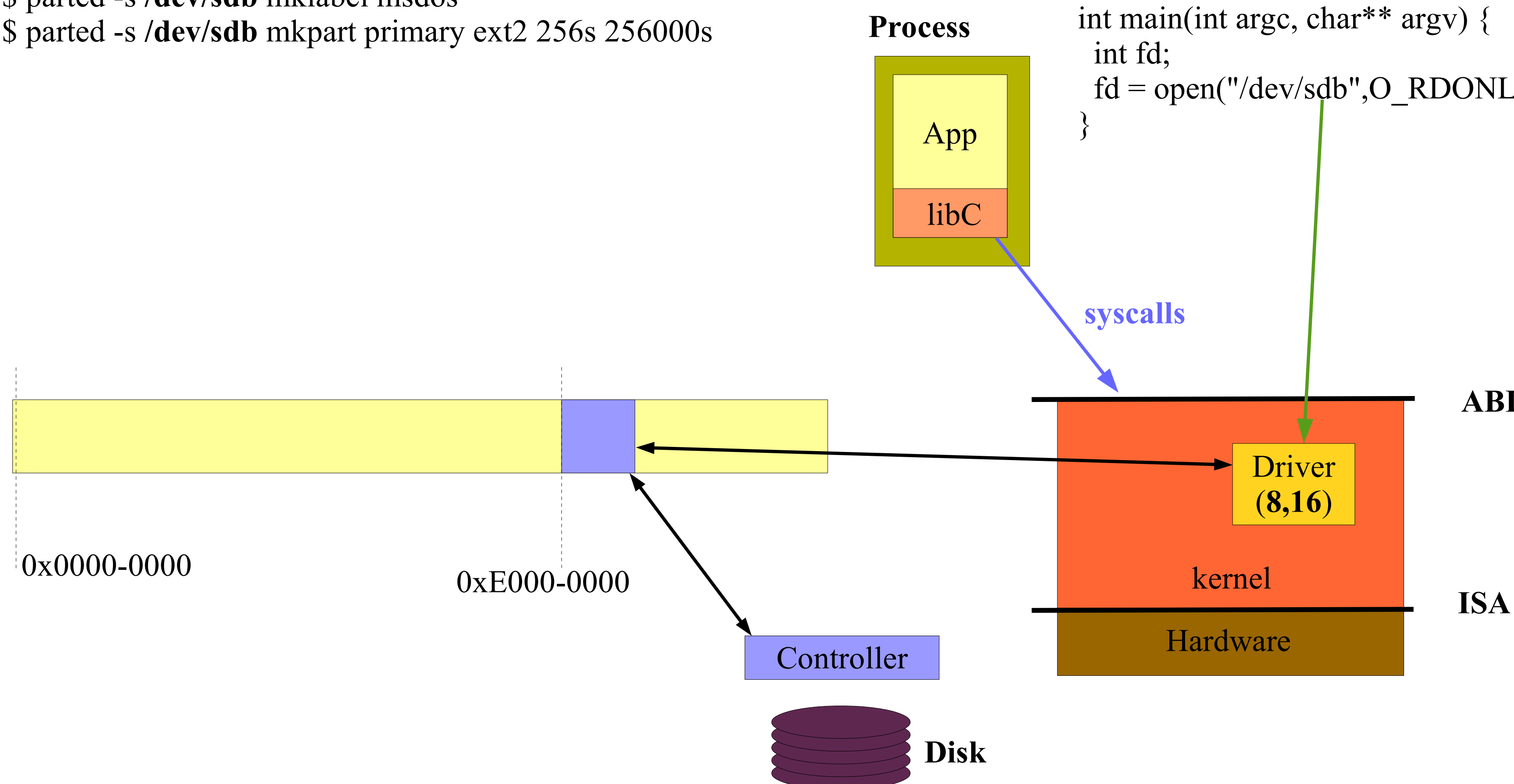
crw-rw-rw-	1 root tty	**5, 0**	**tty**
crw-------	1 root root	**5, 1**	**console**
crw-r-----	1 root kmem	**1, 1**	mem
crw-rw-rw-	1 root root	**1, 3**	null
crw-------	1 root root	**10, 1**	psaux
crw-------	1 root root	**251, 0**	rtc0
brw-rw----	1 root disk	**8, 0**	**sda**
brw-rw----	1 root disk	**8, 1**	**sda1**
brw-rw----	1 root disk	**8, 2**	sda2
brw-rw----	1 root disk	**8, 16**	**sdb**
brw-rw----	1 root disk	**8, 17**	**sdb1**
crw-rw----+	1 root video	**81, 0**	video0
crw-rw-rw-	1 root root	**1, 5**	zero
crw--w----	1 root tty	**4, 0**	**tty0**
crw--w----	1 root tty	**4, 1**	**tty1**

Devices and Drivers

$ ls -al /dev/sd*
brw-rw---- 1 root disk 8, 0 oct. 11 15:37 sda
brw-rw---- 1 root disk 8, 1 oct. 11 15:37 sda1
brw-rw---- 1 root disk 8, 2 oct. 11 15:37 sda2
brw-rw---- 1 root disk **8, 16** oct. 11 15:37 **sdb**
brw-rw---- 1 root disk 8, 17 oct. 11 15:37 sdb1

$ parted -s **/dev/sdb** mklabel msdos
$ parted -s **/dev/sdb** mkpart primary ext2 256s 256000s

```
#include "stdint.h"
#include <fcntl.h>

int main(int argc, char** argv) {
  int fd;
  fd = open("/dev/sdb",O_RDONLY);
}
```



Device Files

- Files under /dev appear as regular files
  - With the usual open, read, write, and close operations
    - But the interaction is with a loaded module
  - Can be seen as message-passing interface
    - Writing bytes is about sending a message to the module, requesting something
    - Reading bytes is reading the reply of the module.
- May not have anything to do with I/Os
  - The written bytes may be entirely interpreted by the module
  - The read bytes may be entirely generated by the module
- If related to a real device
  - The module is a called a device driver
  - It reads and writes mmio registers
  - It reacts to interrupts from its device

Udev – Userspace Devices

- The problem
  - When 20,000 devices were under **/dev**
  - Linux community felts a dynamic solution was necessary
  - To create only the device files for the devices actually present on the machine
- The solution
  - The kernel detects the hardware device → device id, vendor id, release id
  - The kernel upcalls **udev** (userland process)
  - The udev process uses a database to find the device
  - The udev process creates the node under **/dev** using mknod
  - The udev process loads (modprobe) the corresponding module

Note: we will not use udev in our minimal distribution

https://www.linux.com/news/udev-introduction-device-management-modern-linux-system
https://en.wikipedia.org/wiki/Udev

Special File System – /proc

- **/proc**
  - A special file system, backed up by the kernel
    - Exposes information about running processes
    - Exposes *almost* all the internal data structures of the kernel
    - Can be read or modified, changing the kernel's behavior at runtime
  - Many commands read or write the /proc file system
    - Avoids the proliferation of syscalls to interact with the kernel
- Many shell commands read /proc
  - You can write a program that reads /proc files and display information

WARNING – WARNING – WARNING
if you are modifying files in /proc, you are impacting your currently running kernel!
directly changing its data structures in memory...
only for the **bravest** or the **foolish**...

Going further...

http://www.linuxfromscratch.org/

But only if you really need to configure your own Linux from scratch…
this is not easy...