Boot Sequence in Linux

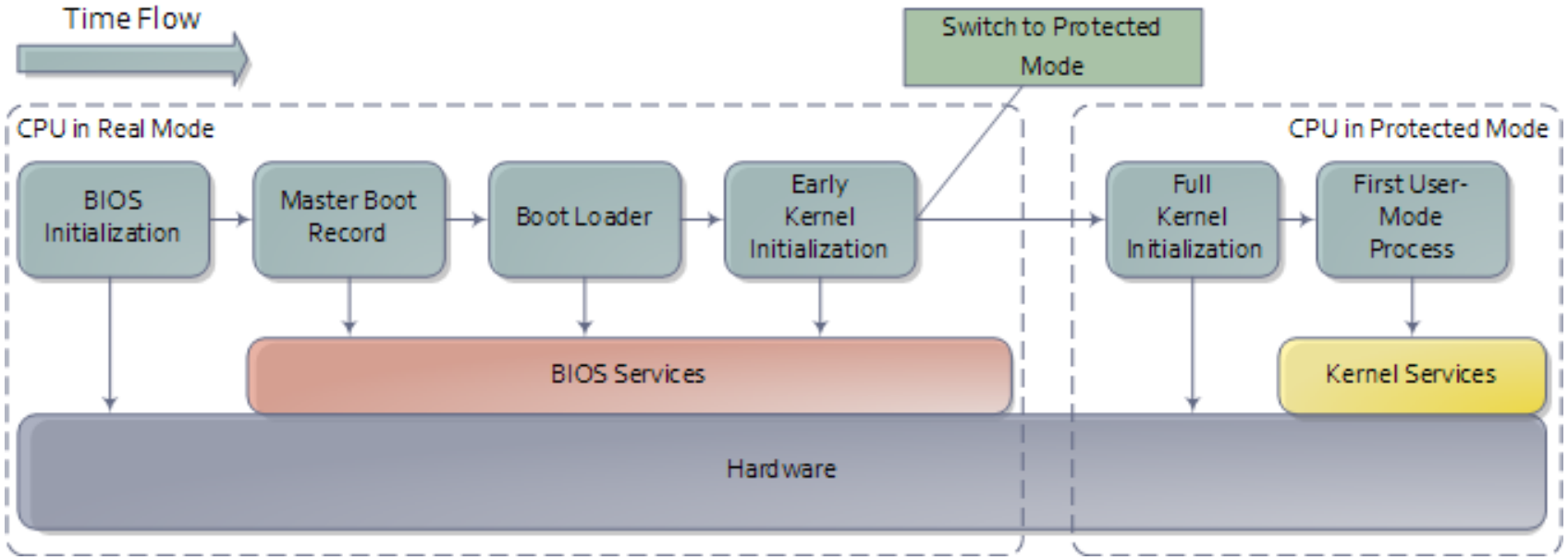
Mayur Sadavarte

Furquan Shaikh

Overview

- BIOS
- Boot Loaders
- Kernel Initialization

Outline of Boot Sequence



BIOS

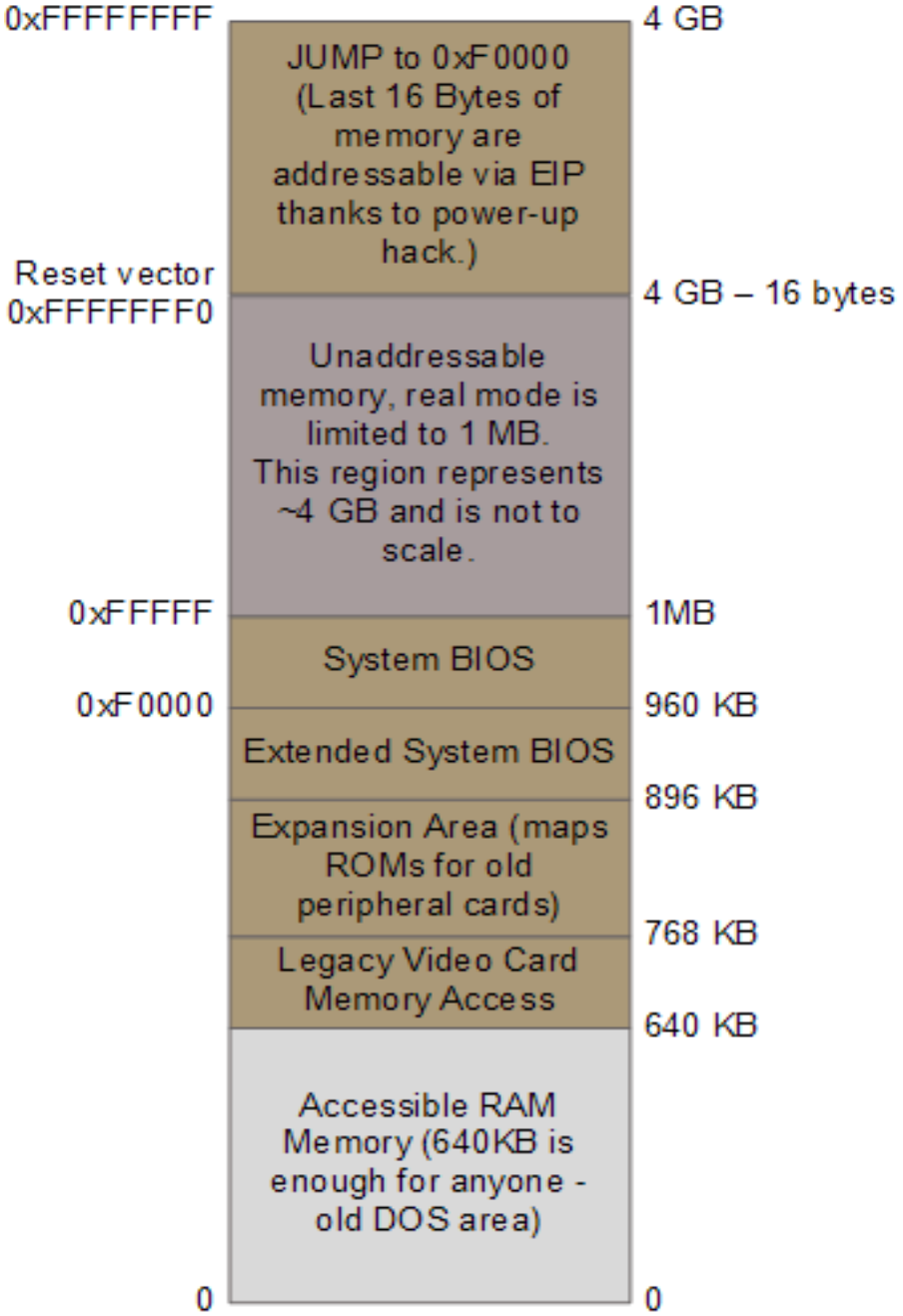
BIOS

- 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 Components

- 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

Memory Layout for the first 4GB in x86



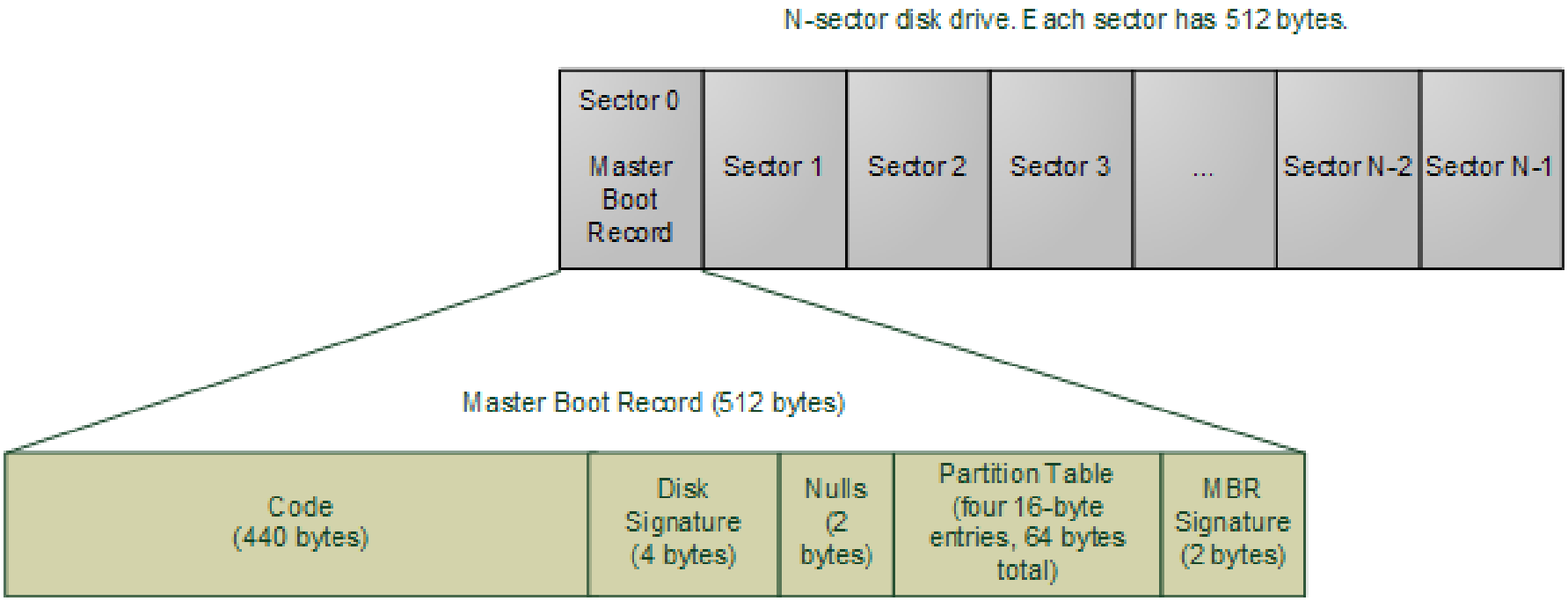
BIOS Tasks

- 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

Bootable devices

- To boot an operating system, BIOS runtime searches devices that are both active and bootable in the order of preference defined in CMOS settings
- Bootable device can be:
  - Floppy Drive
  - CD-ROM
  - Partition on HDD
  - Device on network
  - USB flash memory stick

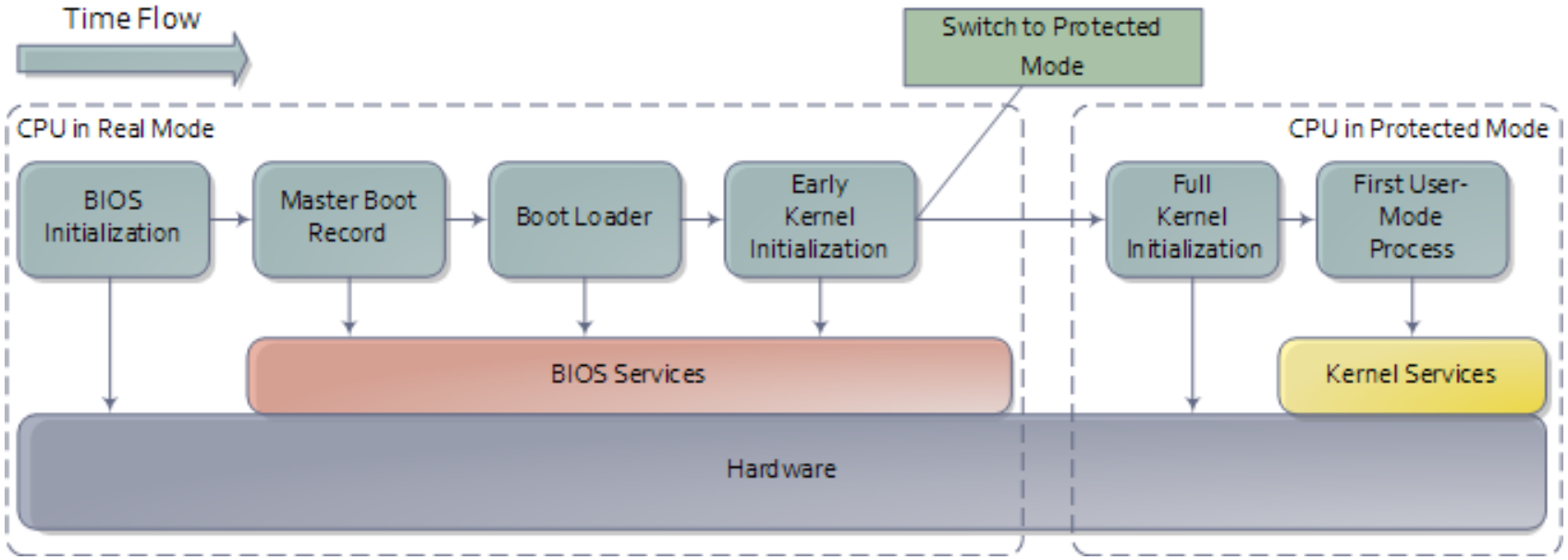
Master Boot Record



MBR (Master Boot Record)

- BIOS reads first 512-byte sector of the hard disk
- Contains two important components:
  - OS-specific bootstrapping program
  - Partition table for the disk
- Loaded at location 0x7c00 in RAM and control is given to this code
- MBR could be
  - Windows specific
  - Linux specific
  - Some virus (favorite spot for hackers to get control right at the beginning)

Outline of Boot Sequence

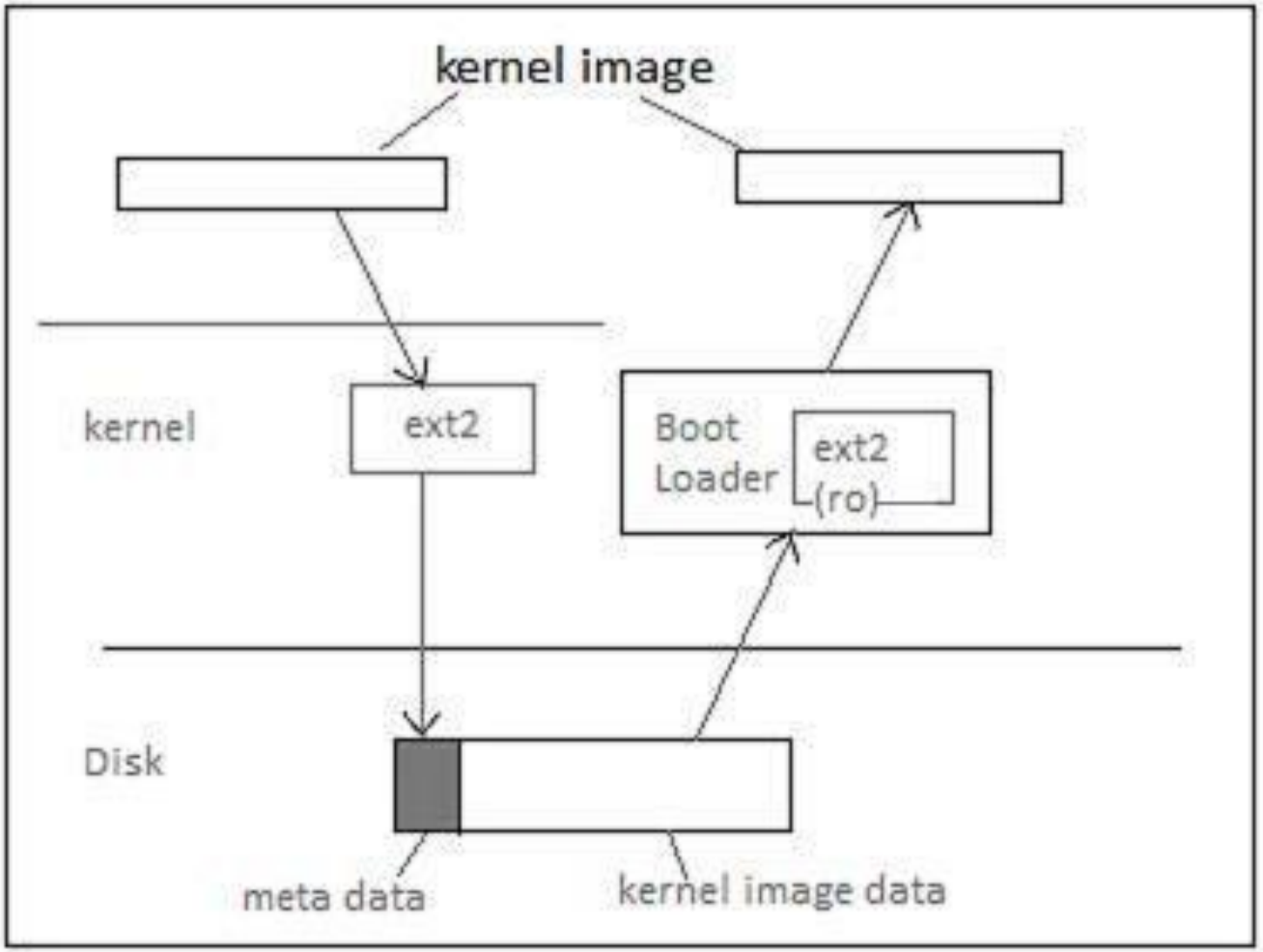


Boot Loaders

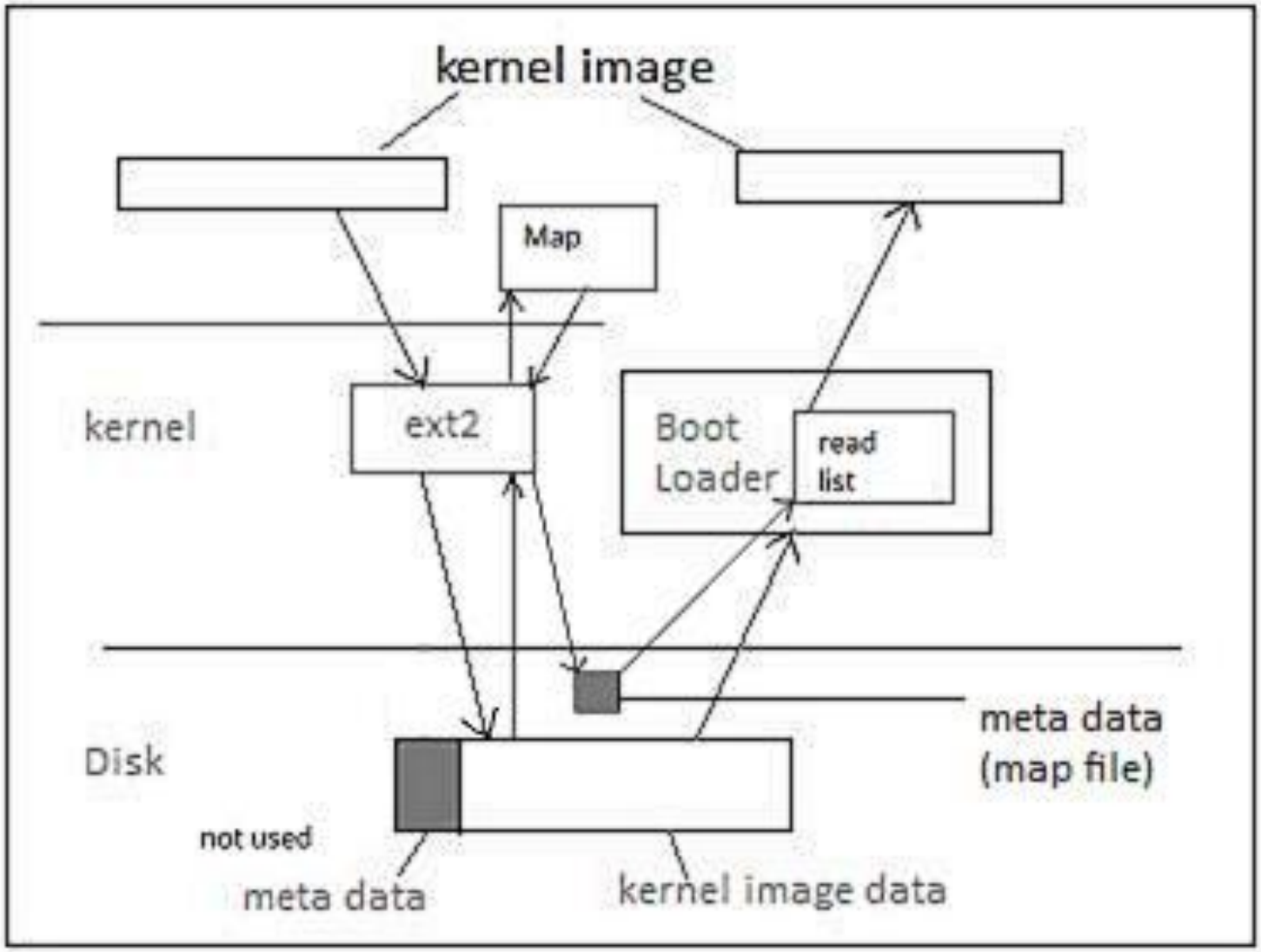
Boot Loaders

- Specialized loaders e.g the floppy boot sector
  - compatible with specific storage medium
- General loaders running under another operating system e.g LOADLIN
  - Use facilities given by host OS to load guest kernel
- File system aware general loaders running on firmware e.g GRUB
  - Almost little Operating Systems by themselves
  - Conversant with one or more file systems
  - Use facilities of firmware and sometimes have their own drivers
- File system unaware general loaders running on the firmware e.g LILO
  - Depends on third party software (/sbin/lilo) to create mapping
  - Mapping stored at some predefined location

File System Awareness



File System Unawareness



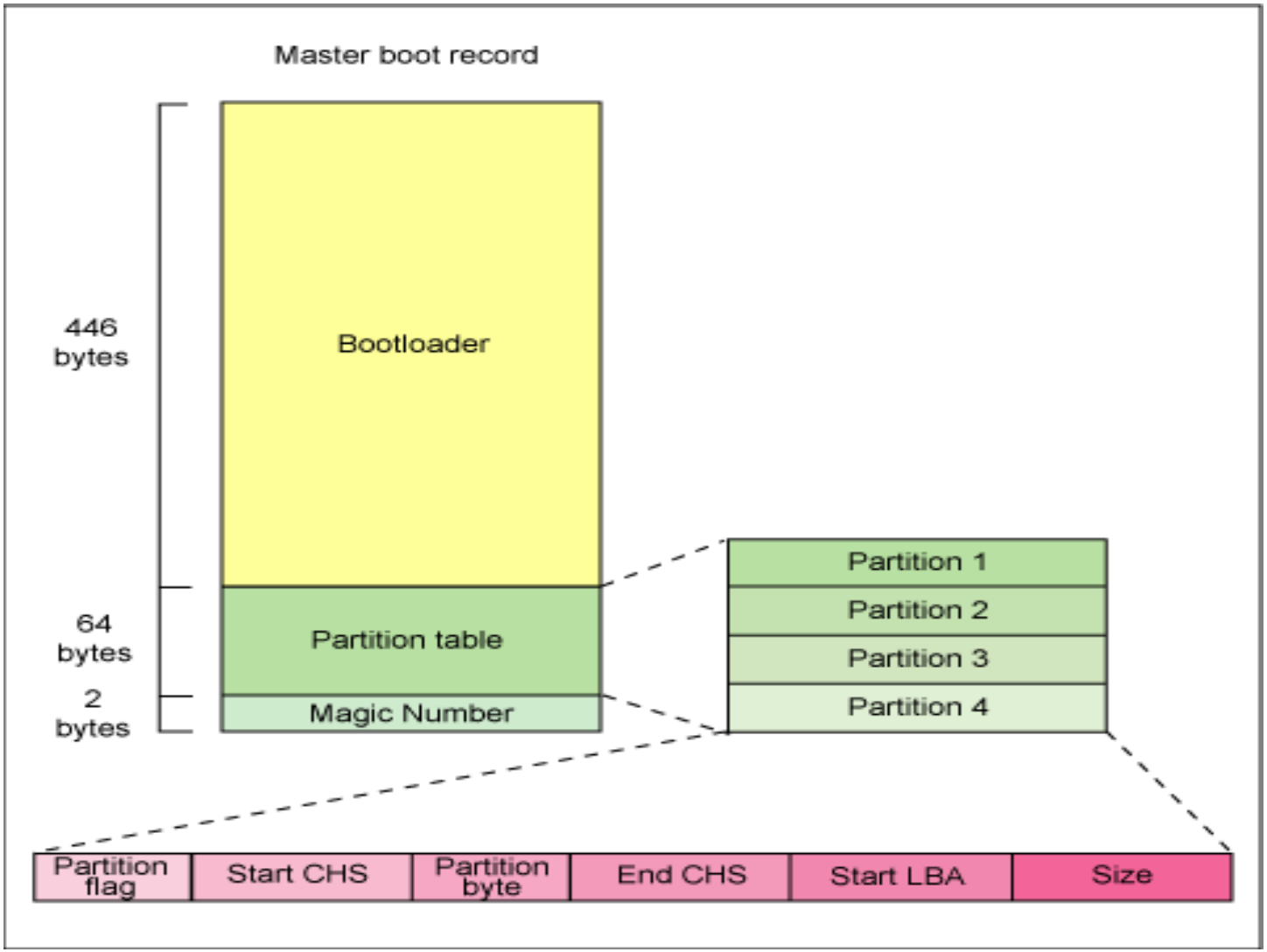
File Unaware Loaders

- Advantage:
  - No changes required in bootloader or map installer if file system of a new device is supported by the linux kernel
- Disadvantage:
  - Map installer has to run after adding new kernel image
  - Or moving the kernel image to a new path

Boot Loaders - Linux

- A multi-stage program which eventually loads the kernel image and initial RAM Disk(initrd)
- Stage-1 Boot Loader is less than 512 bytes (why?)
- Just does enough to load next stage
- Next stage can reside in boot sector or the partition or area in the disk which is hardcoded in MBR
- What is this next stage we are talking about?

Where does Stage-1 BootLoader Reside?



Next stage

- Stage-1 of GRUB mostly resides in MBR (as we just saw)
- Stage-1.5 is a crucial feature (which makes grub file-system aware)
- GRUB Stage 1.5 is located in the first 30 kilobytes of hard disk immediately following the MBR or in the Linux partition
- Stage 1.5 loads Stage 2 from /boot/grub
- What happens when this partition is corrupted?
- The /boot/grub directory contains the stage1, stage1.5, and stage2 boot loaders, as well as a number of alternate loaders (for example, CD-ROMs use the iso9660_stage_1_5)

GRUB Stage-2

- Being powerful and file-system aware, it can display the boot options to user from -

  /boot/grub/grub.cfg

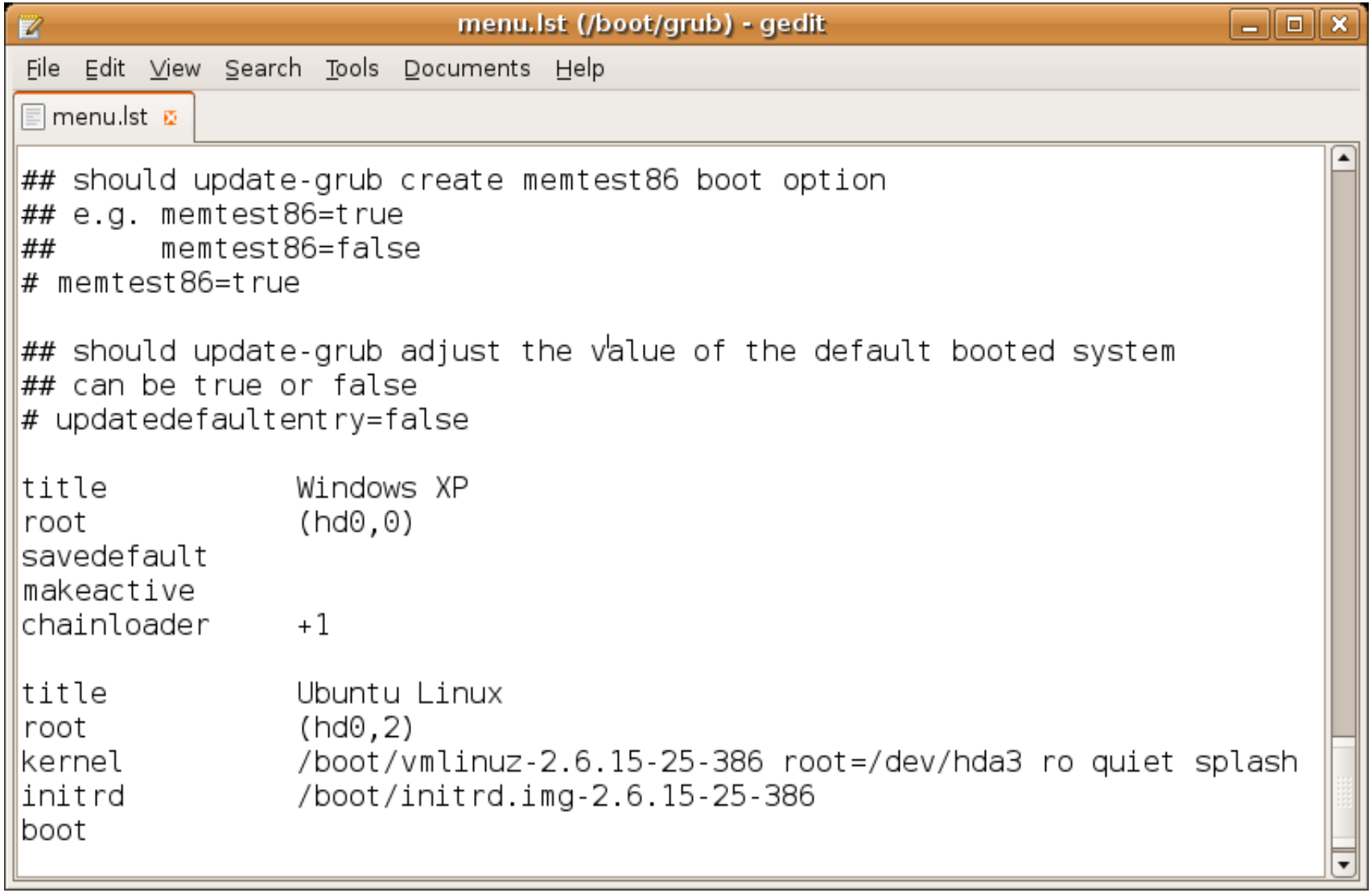
- GRUB command-line - you can boot a specific kernel with a named initrd image as follows:
  - *grub> kernel /bzImage-2.6.14.2*

    [Linux-bzImage, setup=0x1400, size=0x29672e]

  - *grub> initrd /initrd-2.6.14.2.img*

    [Linux-initrd @ 0x5f13000, 0xcc199 bytes]
- Now it's the time to fire the kernel!! But where are we going to place kernel image in the memory??

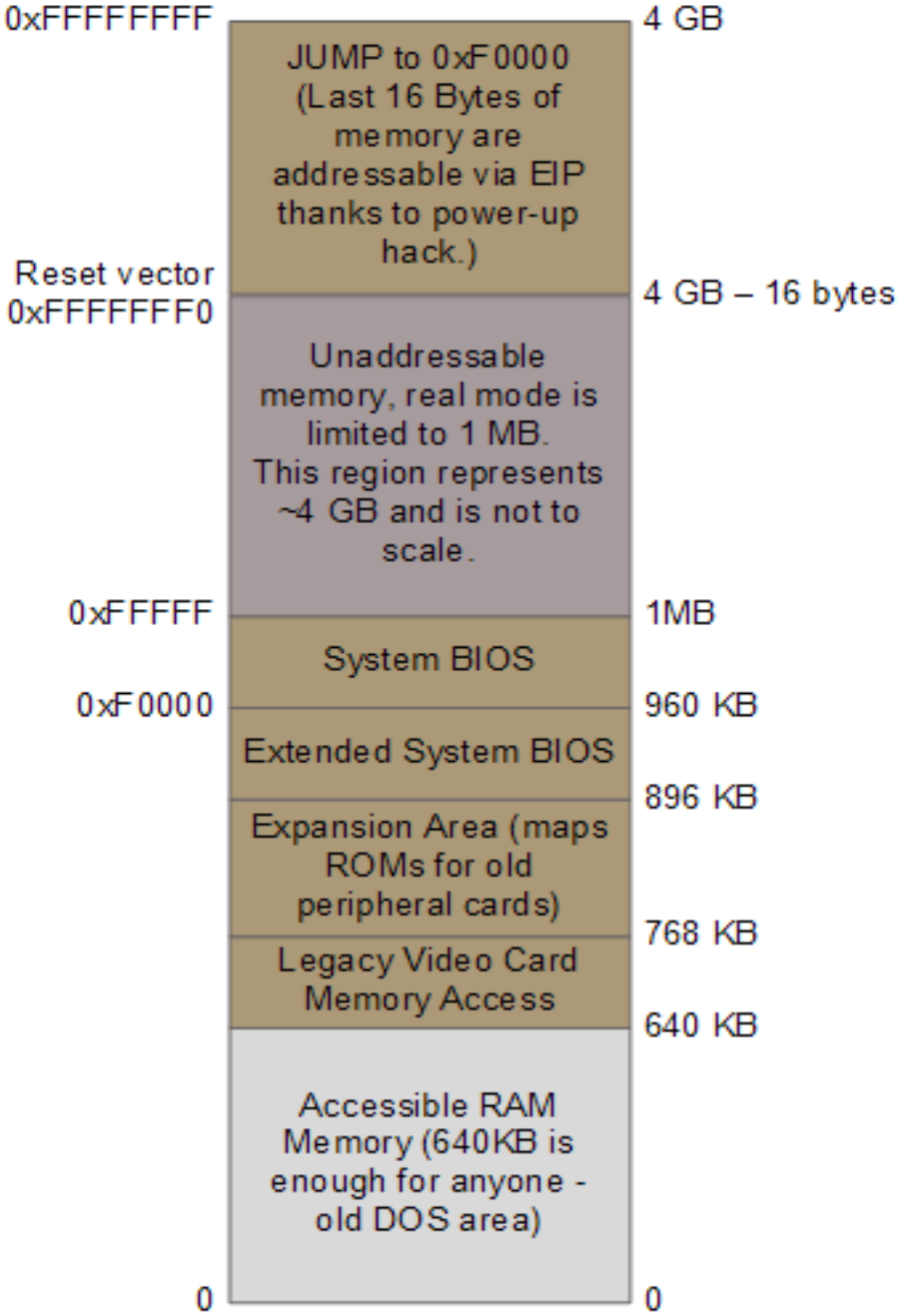
Sample GRUB Conf file



Memory Limitations

- i386 can access 1MB in real mode
- Some space required for bootloader, bios ROMs
- Kernel used to come in *zImage*
- can have maximum size as 512 KB (does seem like a constraint!)
- Hence comes *bzImage* – loaded above 1MB
- how can bootloader access memory more than 1MB in real mode??
  - Switch the CPU mode back and forth (unreal mode)
  - Still uses BIOS Functionalities

Memory Layout for the first 4GB in x86



Root File System

- For mounting root fs kernel requires two things –
  - Media on which root fs resides
  - Driver to access this media
- E.g. ext2 partition on an IDE disk
- Number of root device passed as a parameter
- Kernel normally has IDE driver inbuilt

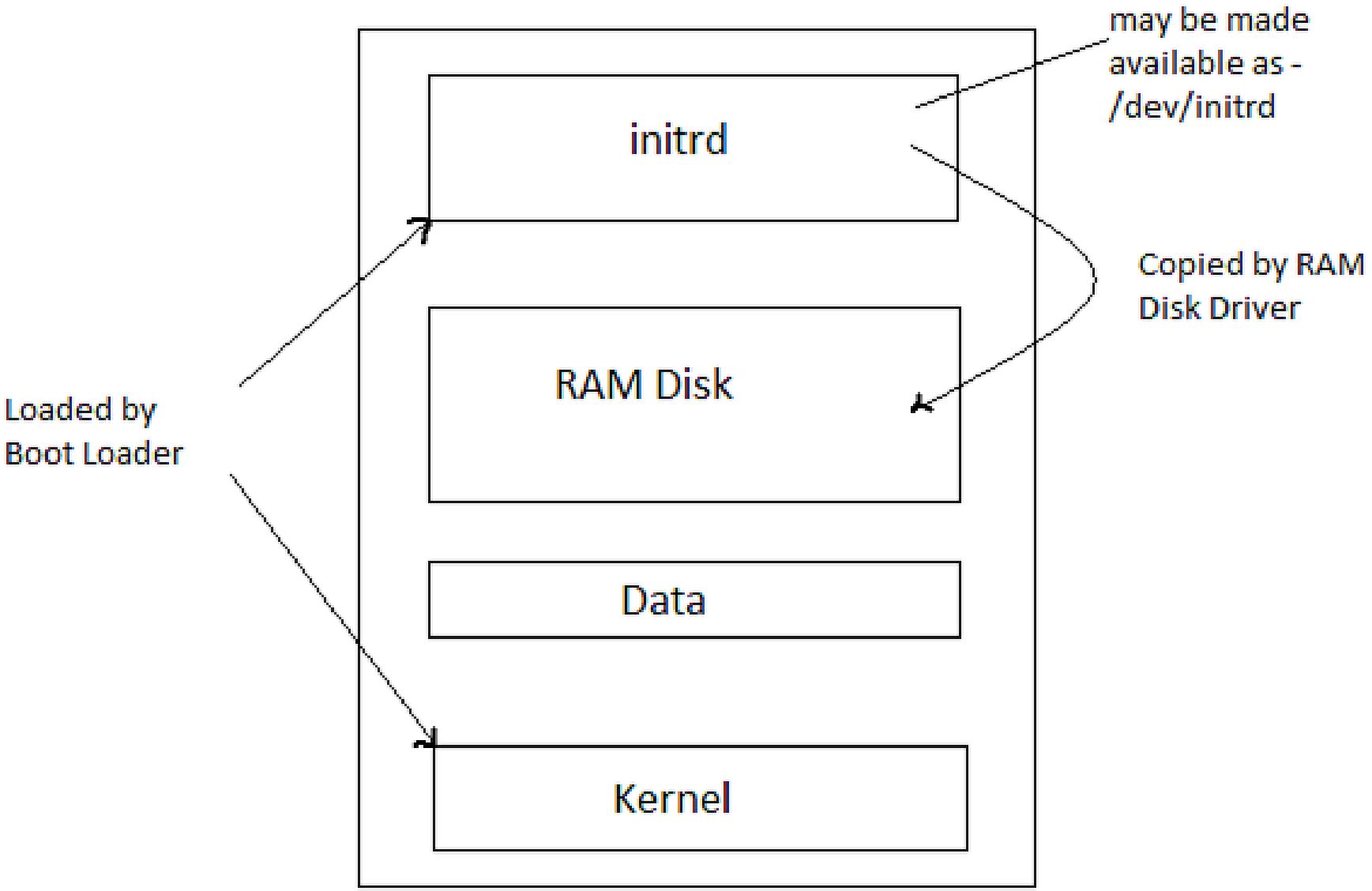
Here come the Complications!!

- What if kernel has no device driver
- Can this happen?
- What will be the size of the kernel if we compile it with all available device drivers?
- Some drivers might upset other hardware while probing for their own devices

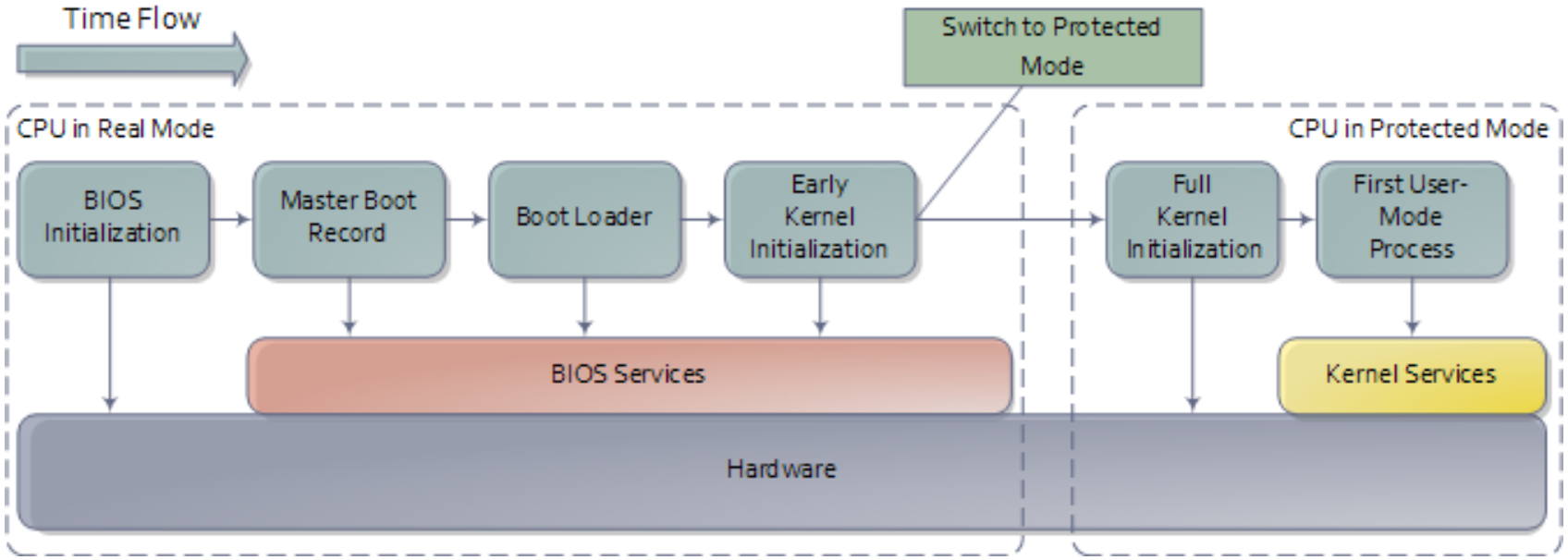
What options do we have?

- Can’t include all possible drivers as part of kernel
- Having multiple pre-compiled kernels
- Compiling the customized kernel
- Linking the pre-compiled kernel with required modules
- Detached modules - *initrd*

initrd


may be made
available as -
/dev/initrd
initrd
Copied by RAM
Disk Driver
RAM Disk
Loaded by
Boot Loader
Data
Kernel

Outline of Boot Sequence

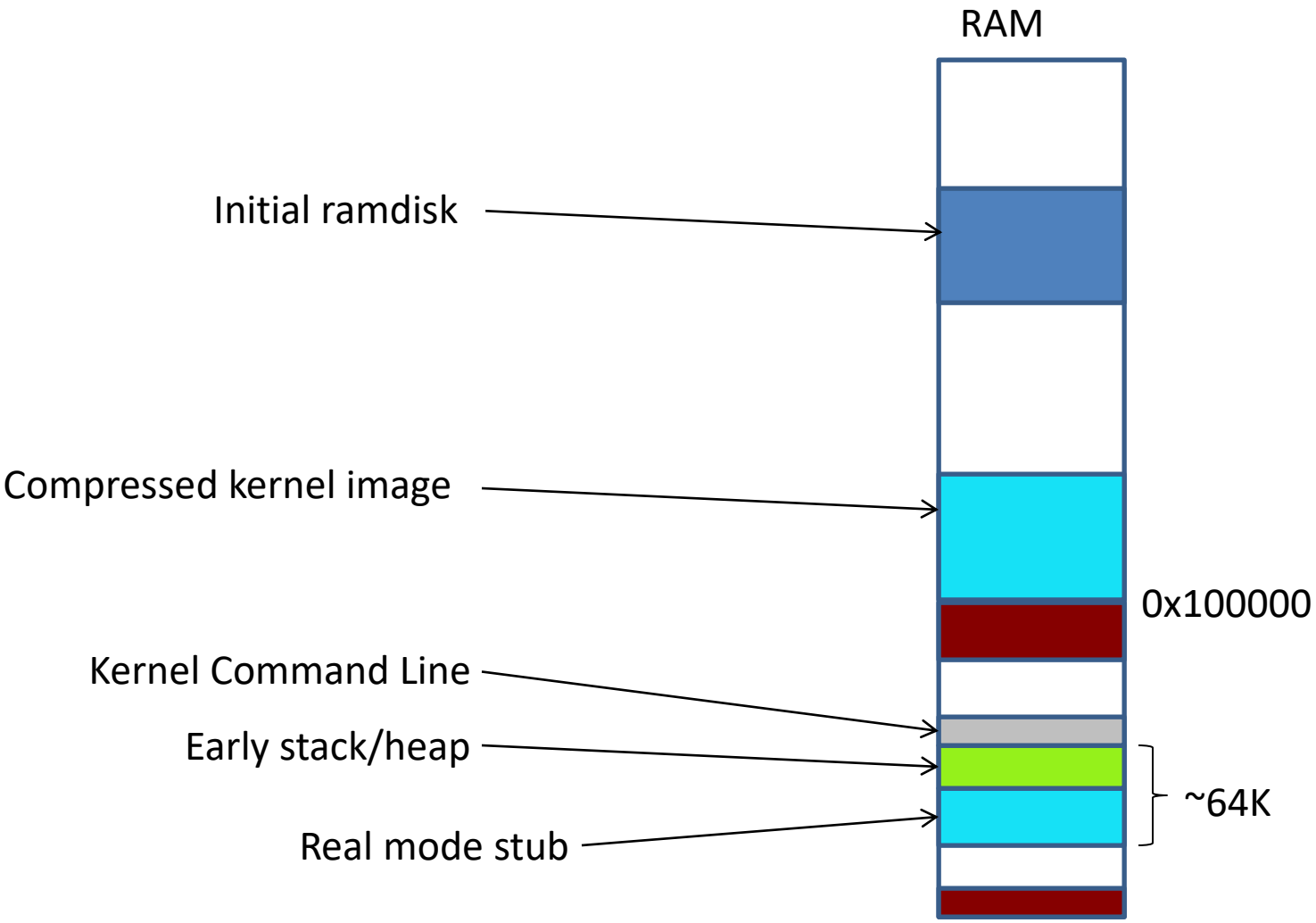


Kernel Initialization

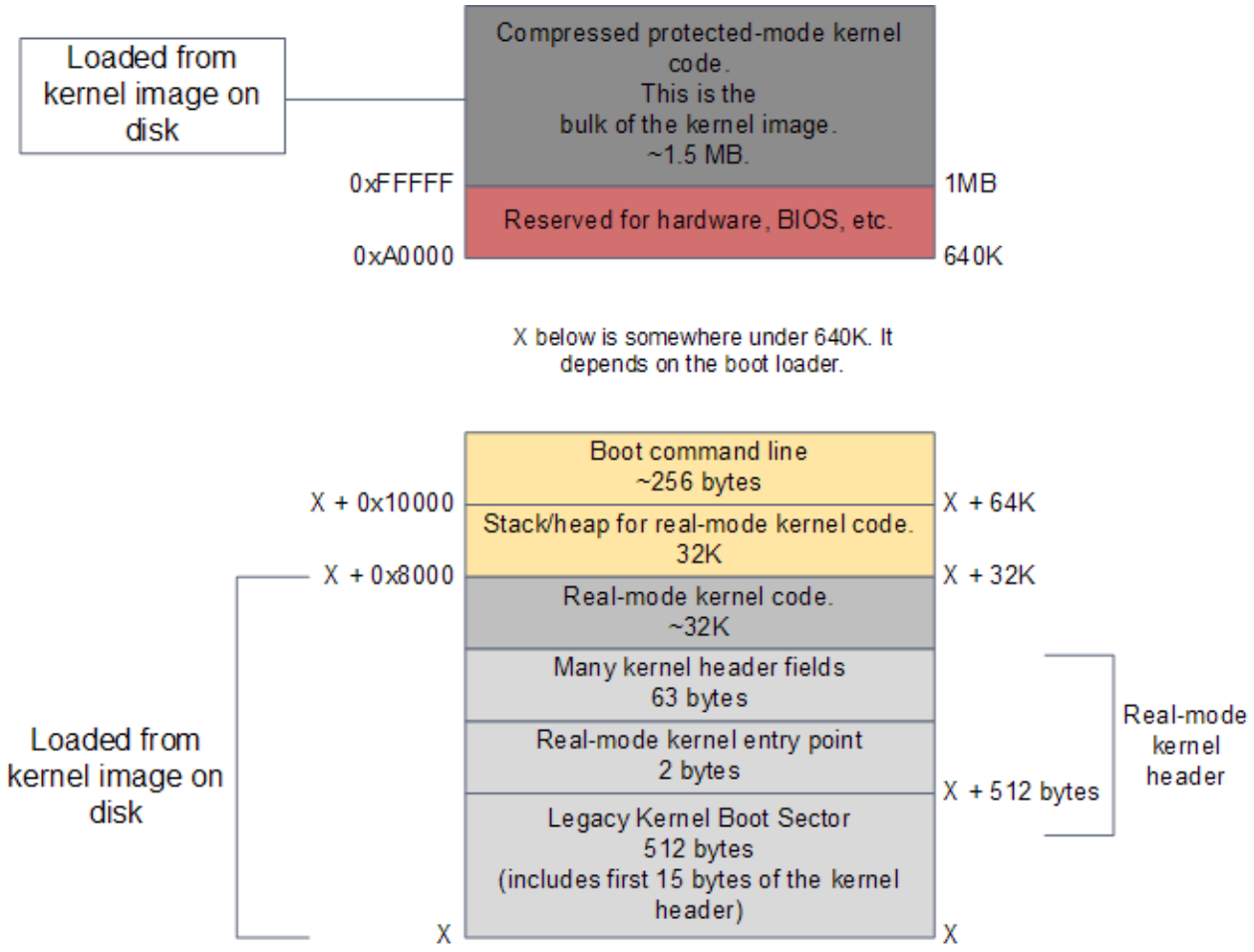
How Operating System Starts Life?

- At this point, the processor is running in real mode.
- Kernel image is loaded into memory by the boot loader using BIOS services
- The image is an exact copy of the image on hard drive */boot/vmlinuz-2.6.38*
- Two major components of this image:
  - Small part containing real-mode kernel code loaded below the 640K barrier
  - Bulk of the kernel code which runs in protected mode, loaded after the first megabyte of memory

In the beginning…



RAM contents after boot loader is done



Major Steps in Kernel Initialization

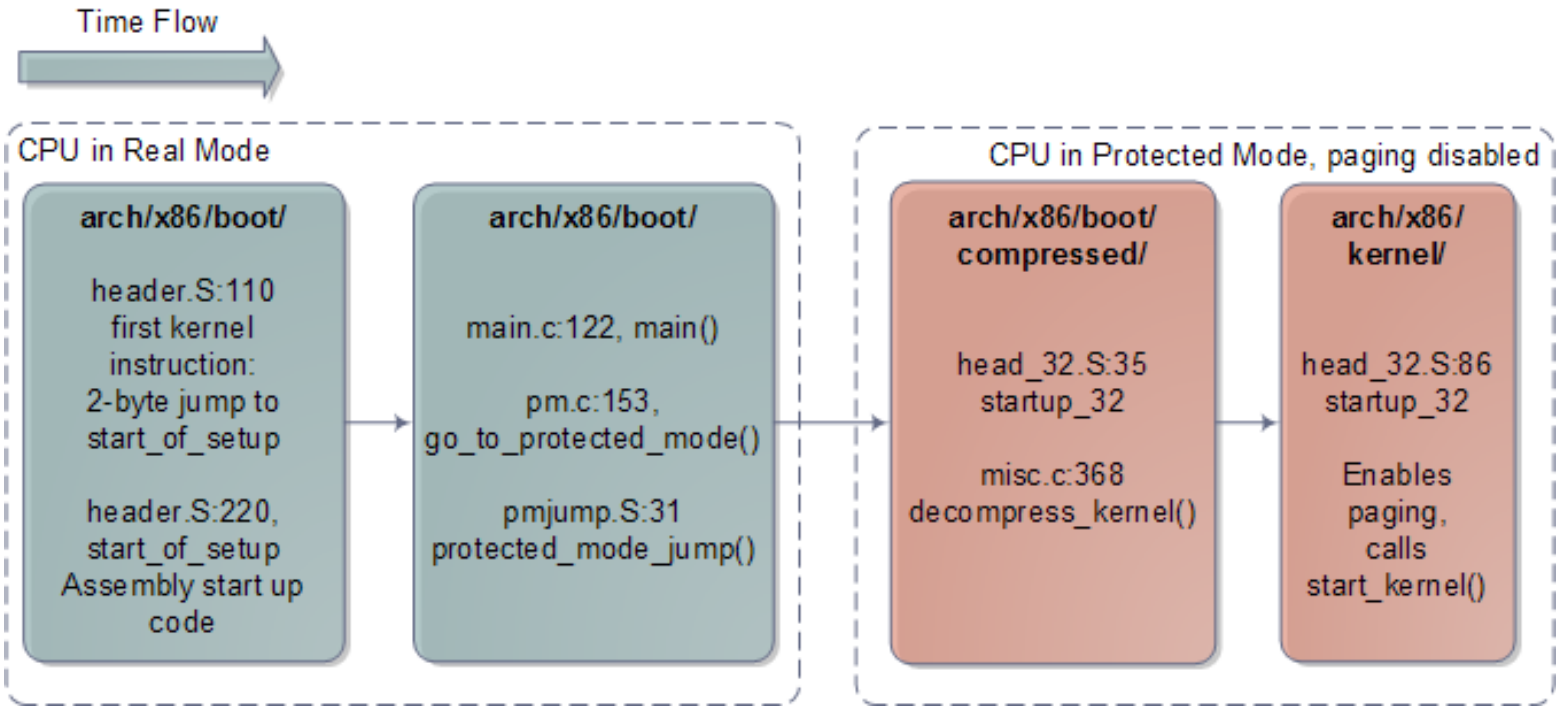
1) Platform-specific initialization

   (In assembly language)

2) Platform-independent initialization

   (In high-level language C)

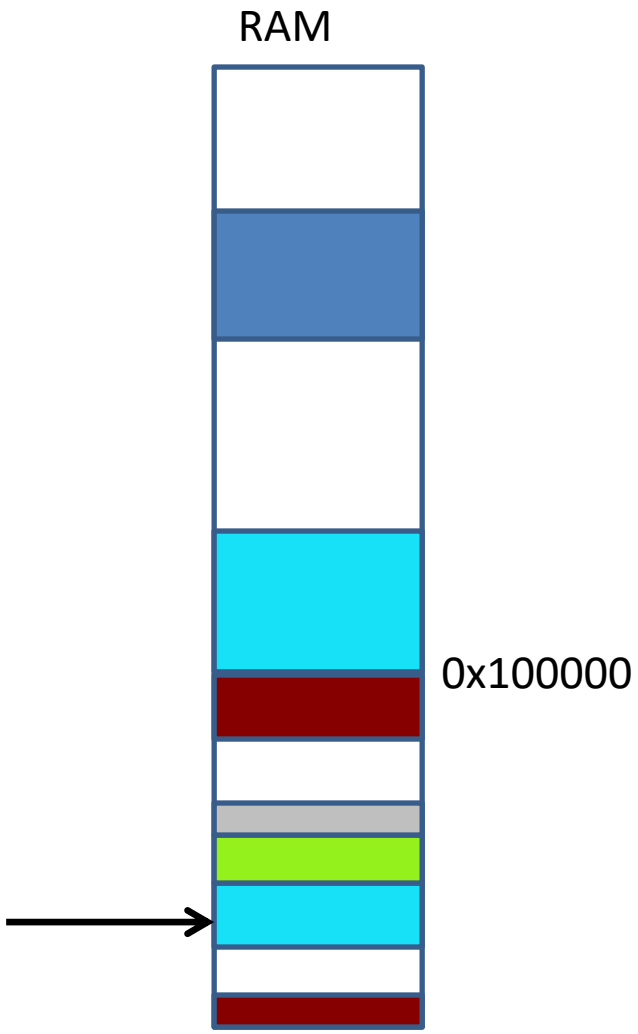
Kernel Initialization Timeline (Arch-dependent)



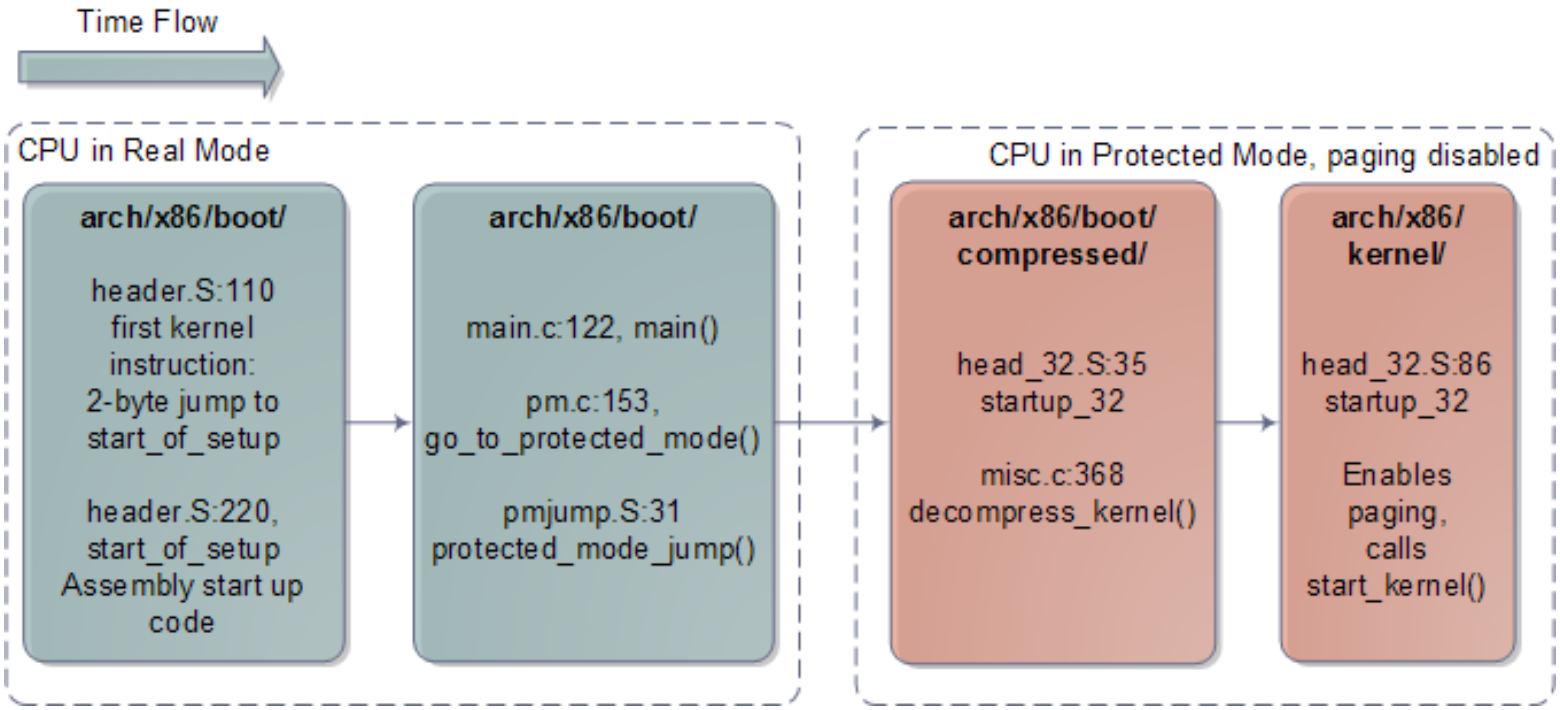
Architecture Specific Setup on IA-32

A) setup assembler function (*arch/x86/boot/header.S)*

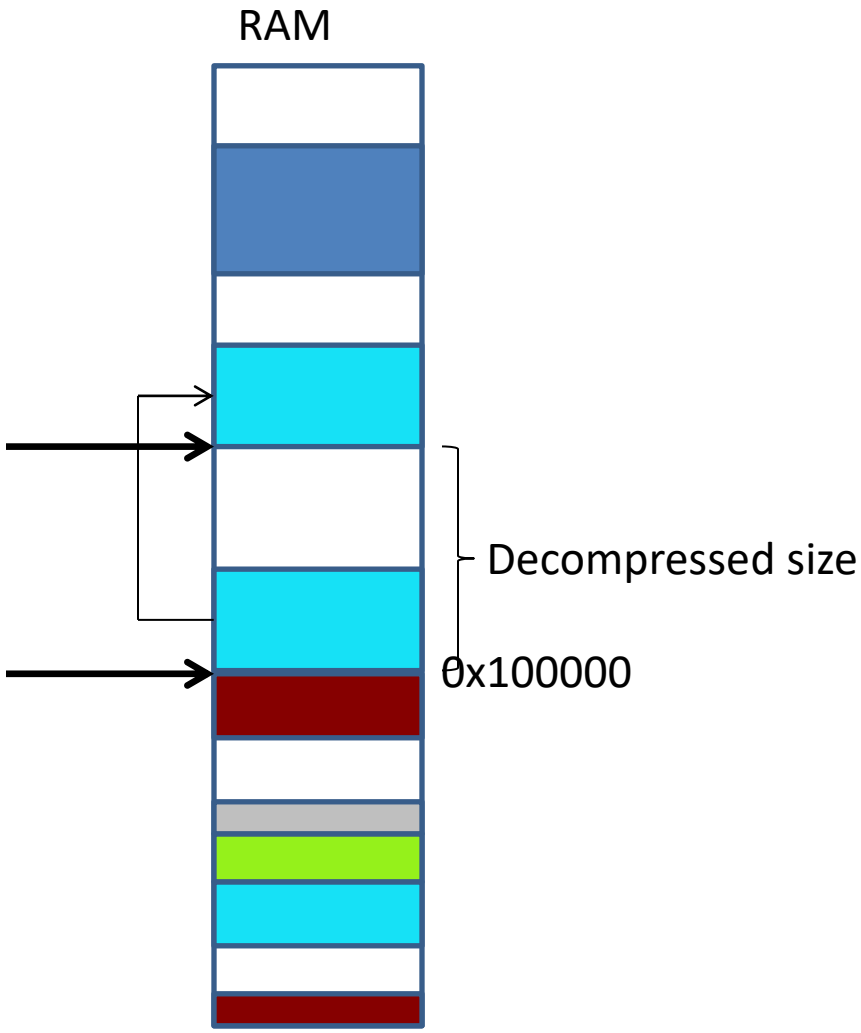
- Checks if kernel was loaded to correct position
- Probes hardware via BIOS
- Determines size of physical memory
- Initializes graphic card
- Switches CPU to protected mode by setting PE bit in *cr0* register



Kernel Initialization Timeline (Arch-dependent)



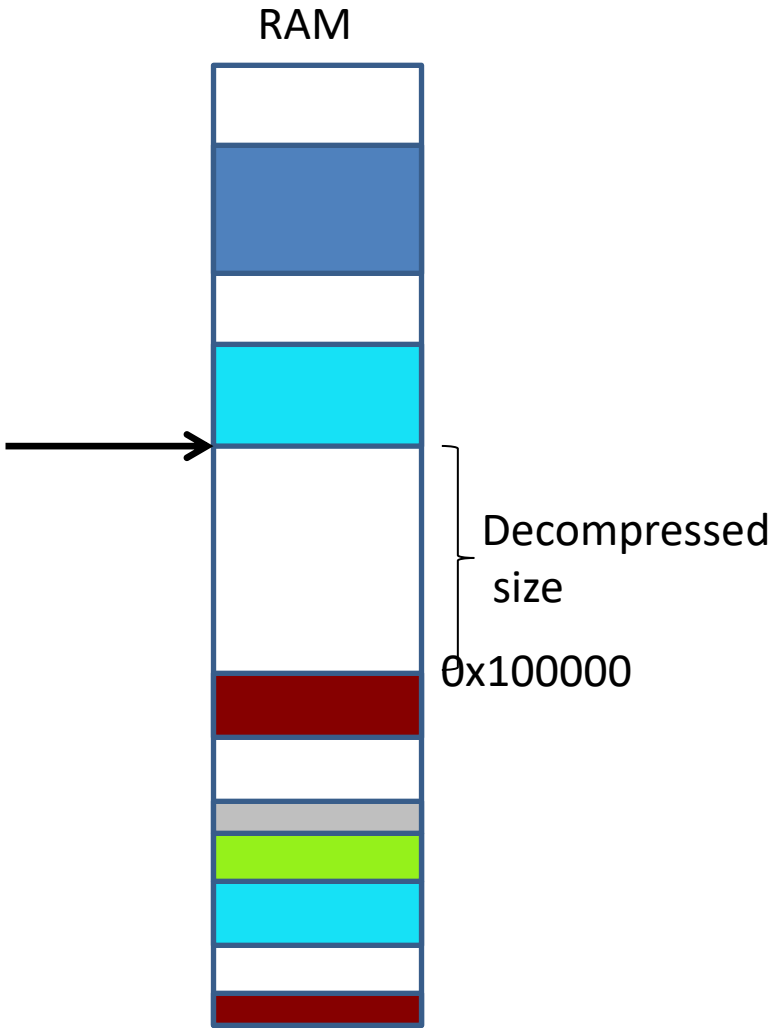
Prepare for decompression



Architecture Specific Setup on IA-32

B) startup_32 assembler function (*arch/i386/boot/compressed/head.S*)

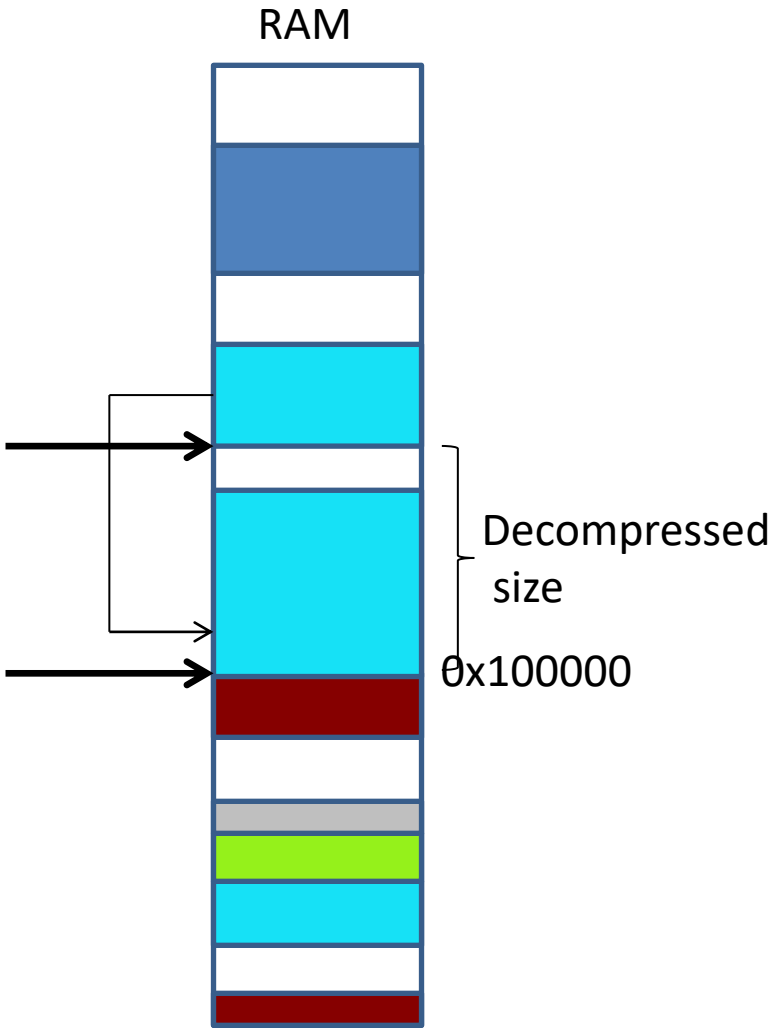
- Creates a provisional kernel stack
- Fills uninitialized kernel data with null bytes (data between _edata and _end constants)
- Calls the C routine decompress_kernel



Architecture Specific Setup on IA-32

C) decompress_kernel() C routine (*arch/x86/boot/compressed/misc_32.c*)

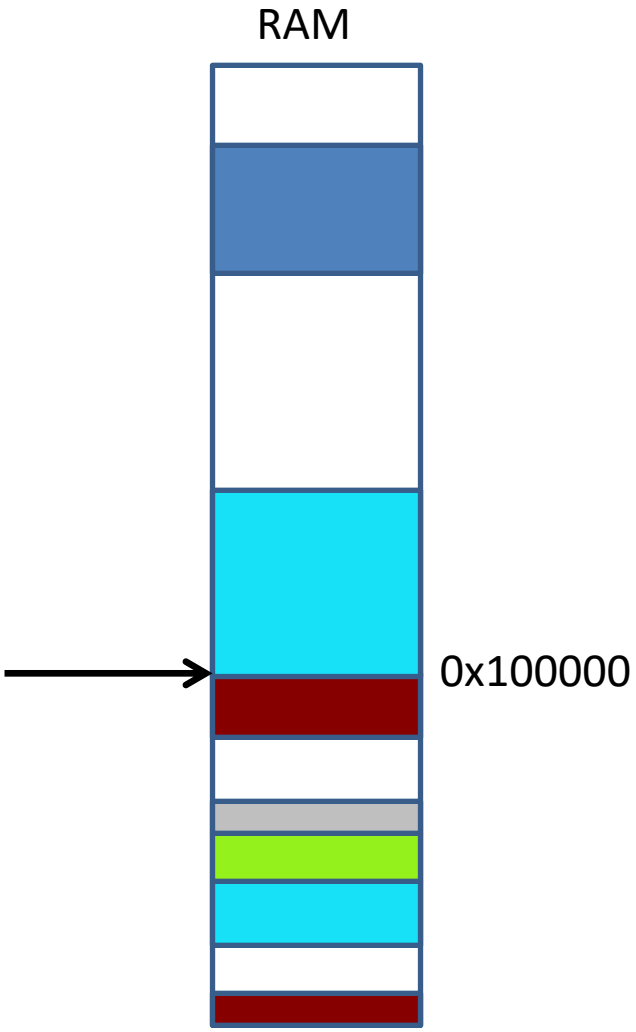
- Decompresses kernel
- Writes uncompressed kernel code to position 0x100000 directly after the first MiB of memory
- Uncompressing is the first operation performed by kernel
- Screen Message *"Uncompressing Linux…"* and *"Ok, booting the kernel"*



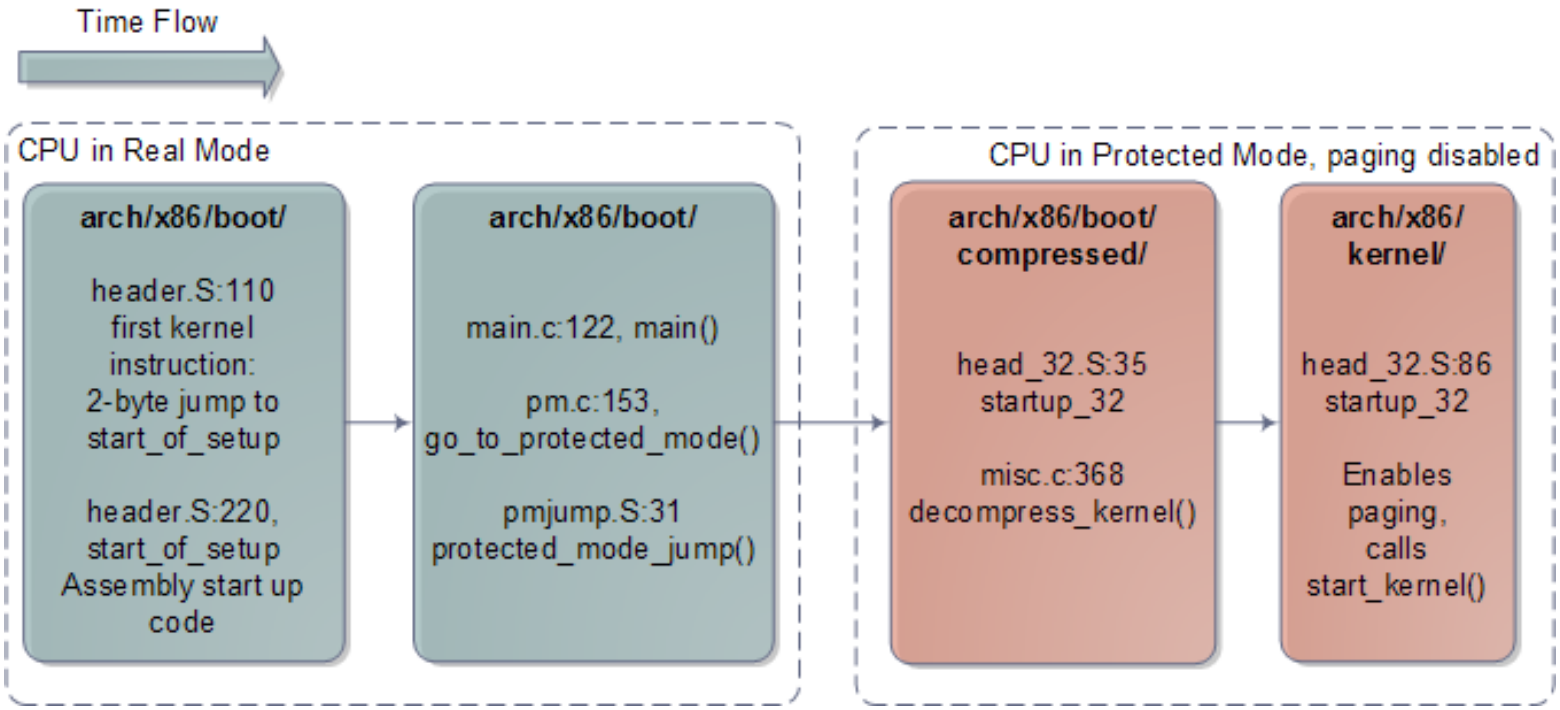
Architecture Specific Setup on IA-32

D) startup_32 assembler function (arch/x86/kernel/head_32.S)

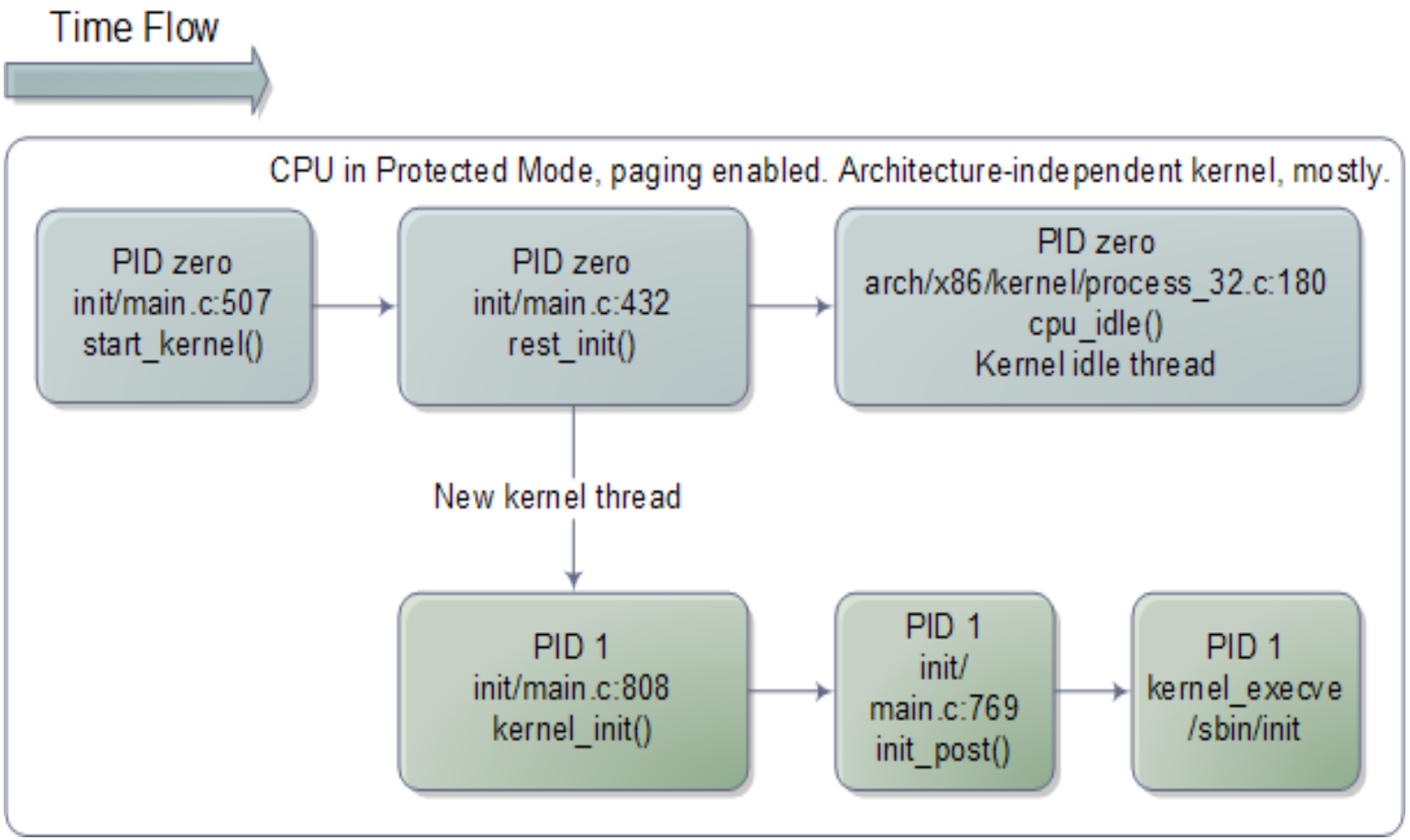
- Fills bss segments of kernel with zeroes
- Initializes provisional kernel Page Tables to identically map the linear addresses to the same physical addresses
- Stores the address of Global Page Directory in *cr3* register
- Enables paging by setting PG bit in *cr0* register
- Puts the parameters obtained from BIOS and parameters passed to the OS into the first page frame
- Jumps to *start_kernel* function



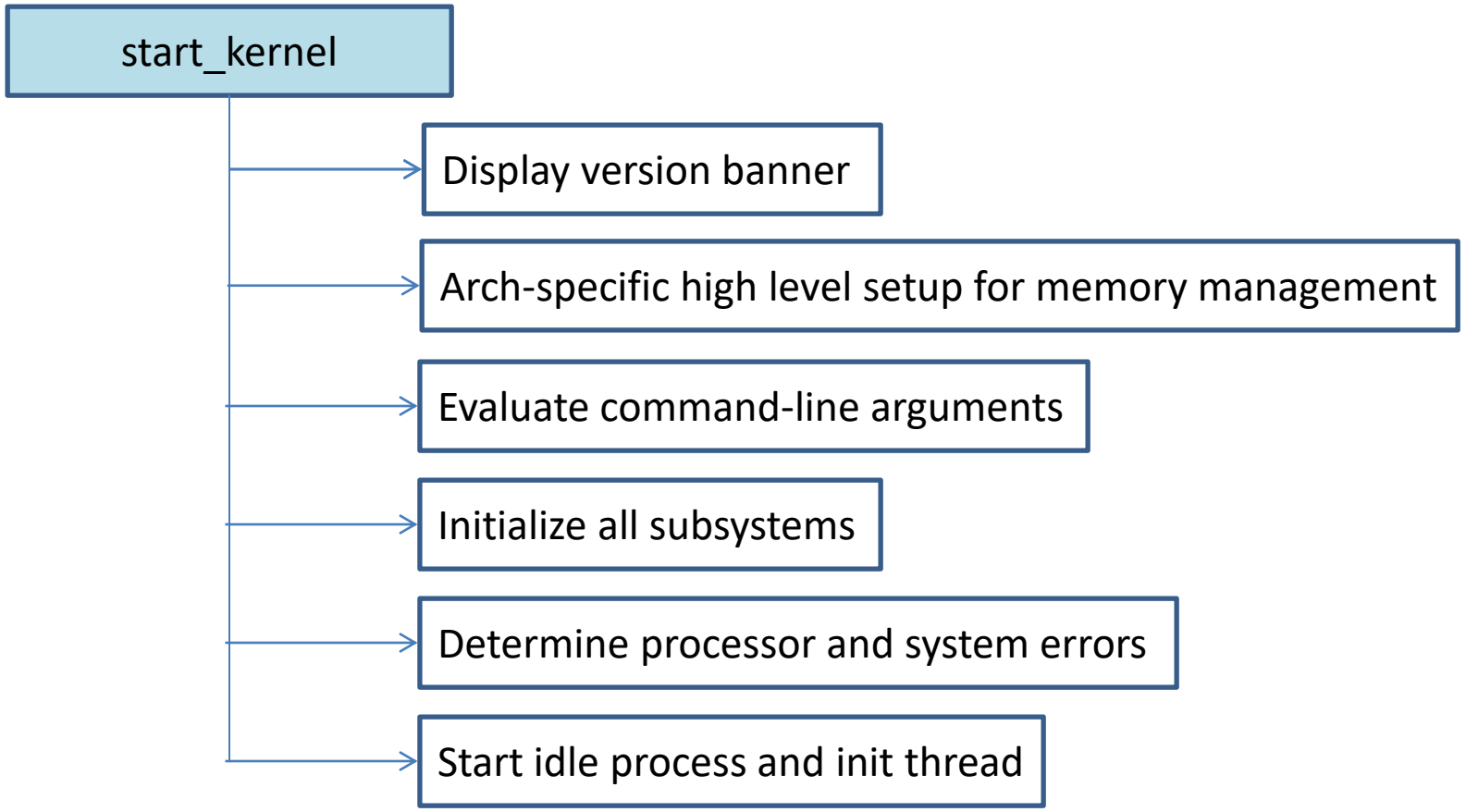
Kernel Initialization Timeline (Arch-dependent)



Kernel Initialization Timeline (mostly Arch-independent)



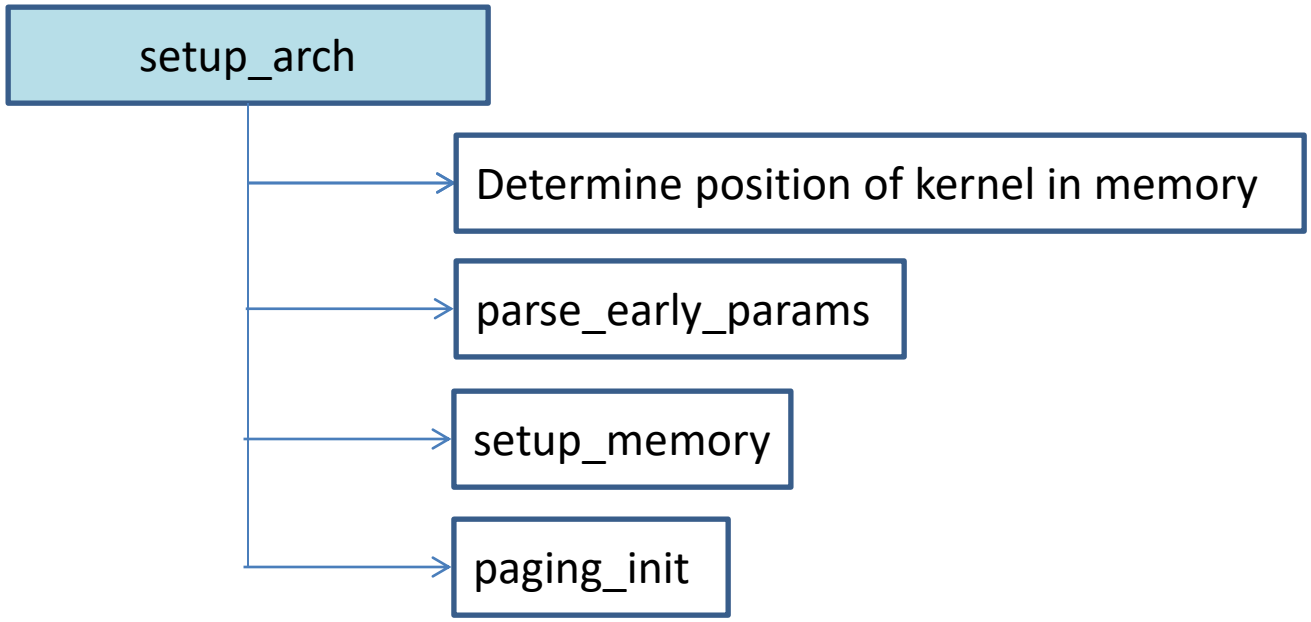
High-Level Initialization



start_kernel: a) Version banner

- First step is to output the version message
- linux_banner global variable in init/version.c

start_kernel: b) setup_arch



start_kernel: c) Interpret command-line arguments

- Interprets command line arguments passed to kernel at boot time
- Arguments passed in the form key=value pairs

start_kernel: d) Initialize various subsystems

- Invoke subroutines to initialize almost all important kernel subsystems
- *Code Snippet:*

```
asmlinkage void __init start_kernel(void)
{
…
 trap_init();
 mm_init();
 sched_init();
 rcu_init();
 init_IRQ();
 init_timers();
 hrtimers_init();
 softirq_init();
 timekeeping_init();
 time_init();
 }
```

Sample Initialization

```
void __init trap_init(void)
{
 set_trap_gate(0,&divide_error);
 set_intr_gate(1,&debug);
 set_intr_gate(2,&nmi);
 set_system_gate(4,&overflow);
…
 set_intr_gate(14,&page_fault);
…
 set_system_gate(SYSCALL_VECTOR,&system_call);
…
}
```

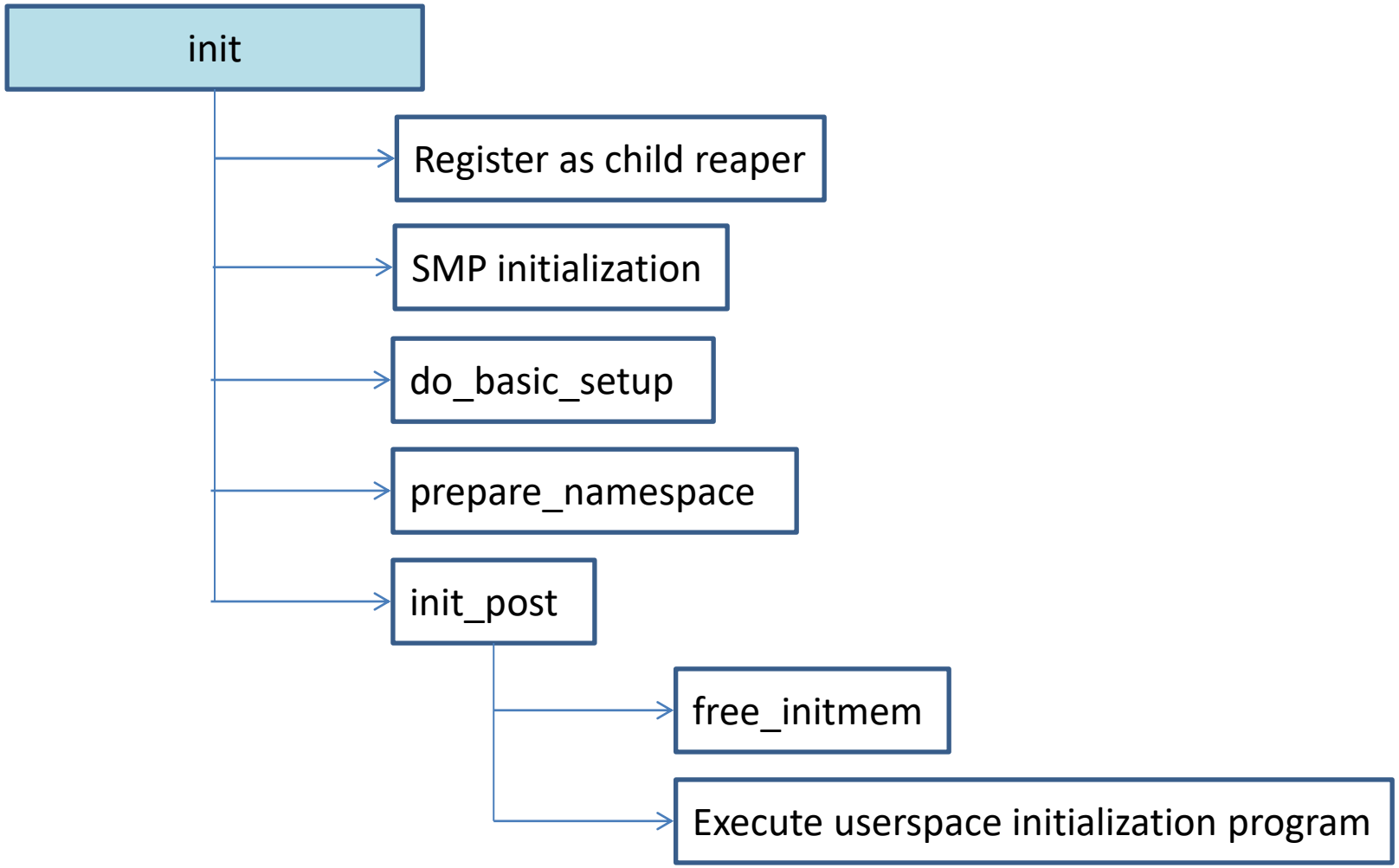
start_kernel: e) Search for known system errors

- Checks for bugs in architecture (check_bugs)
- Replaces certain assembler instructions – depending on processor type – with faster, modern alternatives.

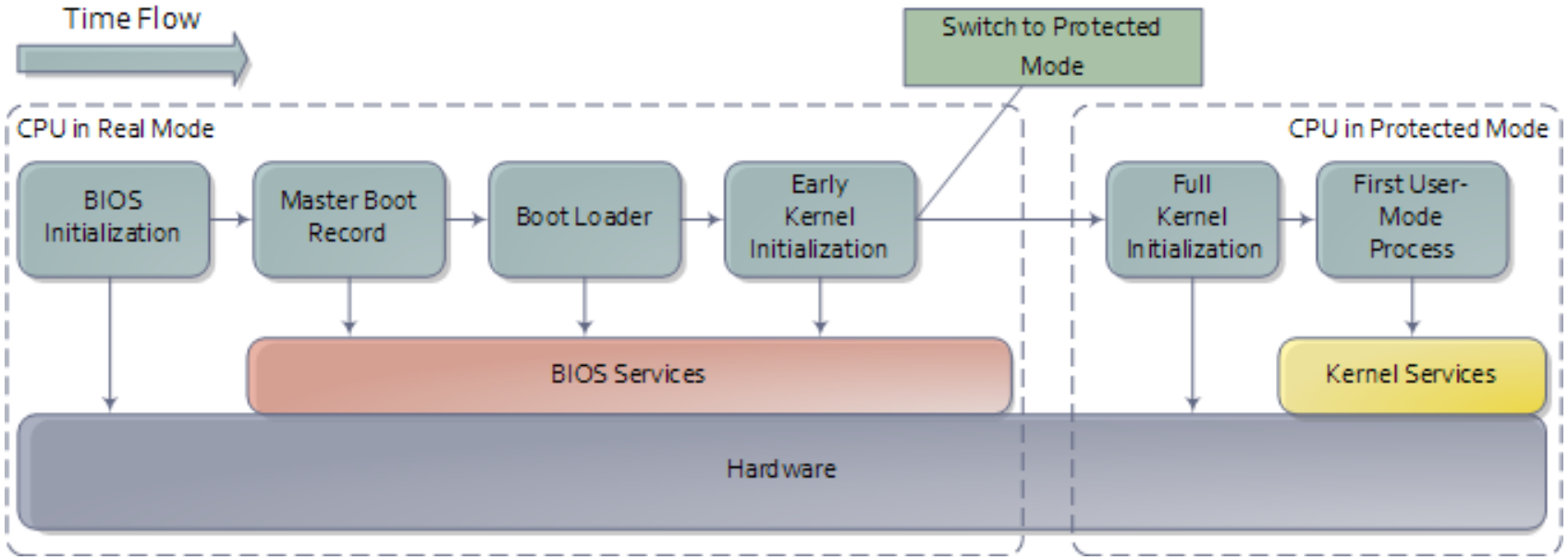
start_kernel: f) init

- Last two actions of start kernel:
  - rest_init : New thread that performs some more initializations and starts the first user-space program /sbin/init
  - The original kernel thread becomes the idle thread that is called when system has nothing else to do

Code flow diagram for init



Outline of Boot Sequence



Questions?

References

- http://duartes.org/gustavo/blog/post/how-computers-boot-up
- Booting Linux: The History and the Future, Werner Almesberger
- Understanding The Linux Kernel, 3rd Edition
- http://www.ibm.com/developerworks/linux/library/l-linuxboot/
- Kernel Walkthrough: The Boot Process by Bart Trojanowski
- The Linux Boot Process by Daniel Eriksen
- **Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide, Parts 1 and 2**