Program Formats

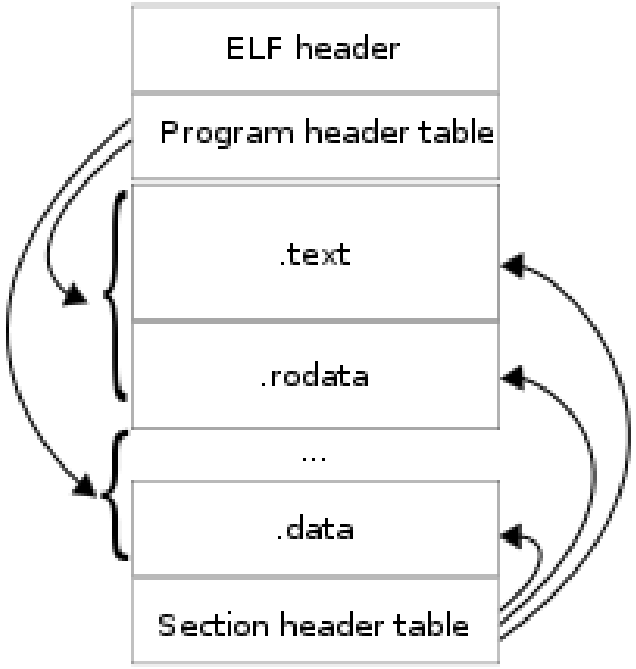
- Programs obey specific file formats
  - CP/M and DOS: COM executables (*.com)
  - DOS: MZ executables (*.exe)
    - Named after Mark Zbikowski, a DOS developer
  - Windows Portable Executable (PE, PE32+) (*.exe)
    - Modified version of Unix COFF executable format
    - PE files start with an MZ header. Why?
  - Unix/Linux: Executable and Linkable Format (ELF)
  - Mac OSX: Mach object file format (Mach-O)

test.c

```c
#include <stdio.h>

int big_big_array[10 * 1024 * 1024];
char *a_string = "Hello, World!";
int a_var_with_value = 100;

int main(void) {
    big_big_array[0] = 100;
    printf("%s\n", a_string);
    a_var_with_value += 20;

    printf("main is : %p\n", &main);
    return 0;
}
```

ELF File Format

- ELF Header
  - Contains compatibility info
  - Entry point of the executable code
- Program header table
  - Lists all the segments in the file
  - Used to load and execute the program
- Section header table
  - Used by the linker



ELF Header Format

```
typedef struct {
        unsigned char e_iden
        Elf32_Half e_type;
        Elf32_Half e_machine
        Elf32_Word e_version
        Elf32_Addr e_entry;
        Elf32_Off e_phoff;
        Elf32_Off e_shoff;
        Elf32_Word e_flags;
        Elf32_Half e_ehsize;
        Elf32_Half e_phentsiz
        Elf32_Half e_phnum;
        Elf32_Half e_shentsiz
        Elf32_Half e_shnum;
        Elf32_Half e_shstrndx
} Elf32_Ehdr;
```

```
$ gcc –g –o test test.c
$ readelf --header test
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:               0x400460
  Start of program headers:          64 (bytes into file)
  Start of section headers:          5216 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         36
  Section header string table index: 33
```

Investigating the Entry Point

```
int main(void) {
        printf("main is : %p\n", &main);
        return 0;
}
```

- Most compilers insert extra code into compiled programs
- This code typically runs before and after main()

```
(base) kolin@mosaic:~/col7001$ readelf --headers hello |grep Entry
  Entry point address:               0x1060
(base) kolin@mosaic:~/col7001$ ./hello
main is : 0x555555555149
(base) kolin@mosaic:~/col7001$
```

```
(base) kolin@mosaic:~/col7001$ readelf --headers hello |grep Entry
  Entry point address:               0x1060
(base) kolin@mosaic:~/col7001$
```

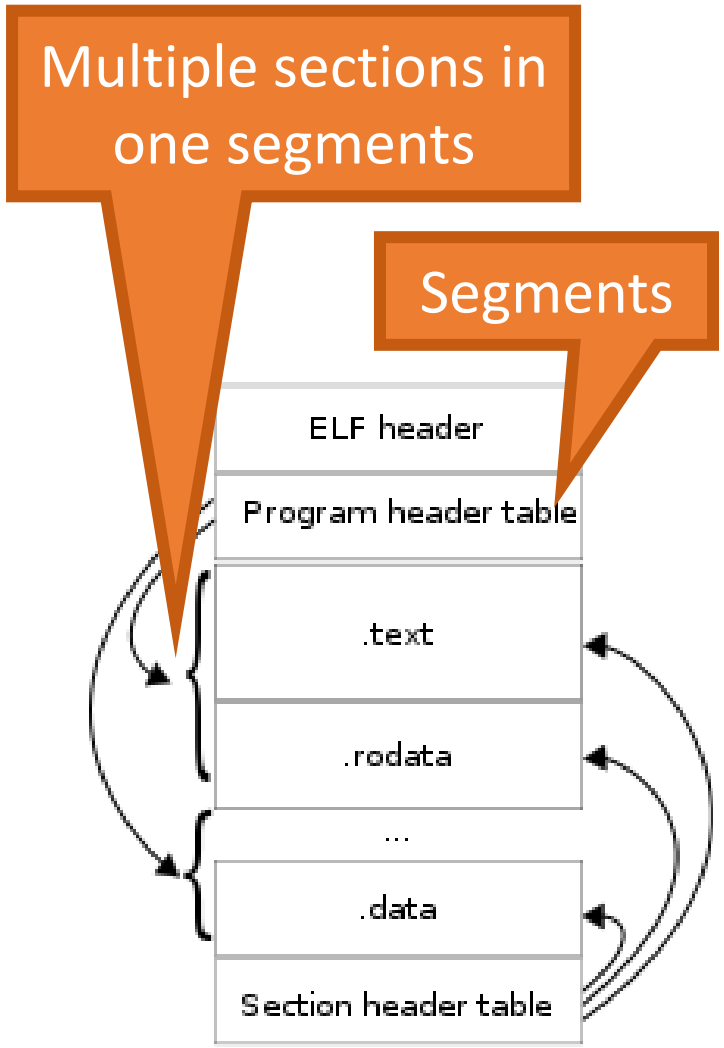
```
(base) kolin@mosaic:~/col7001$ objdump --disassemble -M intel hello |less
```

```
0000000000001060 <_start>:
    1060:       f3 0f 1e fa             endbr64
    1064:       31 ed                   xor    ebp,ebp
```

```
(base) kolin@mosaic:~/col7001$ objdump --disassemble -M intel hello

hello:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    rsp,0x8
    1008:       48 8b 05 d9 2f 00 00    mov    rax,QWORD PTR [rip+0x2fd9]        # 3fe8 <__gmon_start__@Base>
    100f:       48 85 c0                test   rax,rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   rax
    1016:       48 83 c4 08             add    rsp,0x8
    101a:       c3                      ret

Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 9a 2f 00 00       push   QWORD PTR [rip+0x2f9a]        # 3fc0 <_GLOBAL_OFFSET_TABLE_+0x8>
    1026:       f2 ff 25 9b 2f 00 00    bnd jmp QWORD PTR [rip+0x2f9b]        # 3fc8 <_GLOBAL_OFFSET_TABLE_+0x10>
    102d:       0f 1f 00                nop    DWORD PTR [rax]
    1030:       f3 0f 1e fa             endbr64
    1034:       68 00 00 00 00          push   0x0
    1039:       f2 e9 e1 ff ff ff       bnd jmp 1020 <_init+0x20>
    103f:       90                      nop

Disassembly of section .plt.got:

                                                                t>:
                                                                    endbr64
                                                                0   bnd jmp QWORD PTR [rip+0x2fad]        # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
                                                                    nop    DWORD PTR [rax+rax*1+0x0]

                                                                    endbr64
                                                                0   bnd jmp QWORD PTR [rip+0x2f75]        # 3fd0 <printf@GLIBC_2.2.5>
                                                                    nop    DWORD PTR [rax+rax*1+0x0]

                                                                    endbr64
                                                                    xor    ebp,ebp
                                                                    mov    r9,rdx
```

Sections and Segments

- Sections are the various pieces of code and data that get linked together by the compiler
- Each segment contains one or more sections
  - Each segment contains sections that are related
    - E.g. all code sections
- Segments are the basic units for the loader
- Key sections:
  - .text – Executable code
  - .bss – Global variables initialized to zero
  - .data, .rodata – Initialized data and strings
  - .strtab – Names of functions and variables
  - .symtab – Debug symbols



Section Example

String variable → .data

```
int big_big_array[10*1024*1024];
char *a_string = "Hello, World!";
int a_var_with_value = 0x100;

int main(void) {
        big_big_array[0] = 100;
        printf("%s\n", a_string);
        a_var_with_value += 20;
        ...
}
```

Empty 10 MB array → .bss

Initialized global variable → .data

String constant → .rodata

Code → .text

```
$ readelf --headers ./test
...
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r
.rela.dyn .rela.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .ctors .dtors .jcr .dynamic .got
...
There are 36 section headers, starting at offset 0x1460:
Section Headers:
[Nr] Name              Type            Address           Offset
       Size              ES Flags Link Info Align
[ 0]                   NULL            00000000  00000000
       00000000  00       0    0    0
[ 1] .interp           PROGBITS        00400238  00000238
       0000001c  00   A  0    0    1
[ 2] .note.ABI-tag     NOTE            00400254  00000254
       00000020  00   A  0    0    4
[ 3] .note.gnu.build-I NOTE            00400274  00000274
       00000024  00   A  0    0    4
[ 4] .gnu.hash         GNU_HASH        00400298  00000298
       0000001c  00   A  5    0    8
[ 5] .dynsym           DYNSYM          004002b8  000002b8
       00000078  18   A  6    1    8
[ 6] .dynstr           STRTAB          00400330  00000330
       00000044  00   A  0    0    1
[ 7] .gnu.version      VERSYM          00400374  00000374
       0000000a  02   A  5    0    2
...
```

.text Example Header

```
typedef struct {
        Elf32_Word p_type;
        Elf32_Off  p_offset;
        Elf32_Addr p_vaddr;
        Elf32_Addr p_paddr;
        Elf32_Word p_filesz;
        Elf32_Word p_memsz;
        Elf32_Word p_flags;
        Elf32_Word p_align;
```

Address to load section in memory

Offset of data in the file

How many bytes (in hex) are in the section

Data for the program

Executable

```
ons ./test
...
Section He ers:
...
[Nr]  Name      Type        Address     Offset      Size        ES  Flags  Link  Info  Align
[13] .text      PROGBITS    00400460    00000460    00000218    00  AX     0     0     16
...
```

.bss Example Header

```
int big_big_array[10*1024*1024];
```

```
typedef struct {
        Elf32_Word p_type;
        Elf32_Off  p_offset;
        Elf32_Addr p_vaddr;
        Elf32_Addr p_paddr;
        Elf32_Word p_filesz;
        Elf32_Word p_memsz;
        Elf32_Word p_flags;
        Elf32_Word p_align;
```

Offset of data in the file (Notice the length = 0)

Address to load section in memory

Contains no data

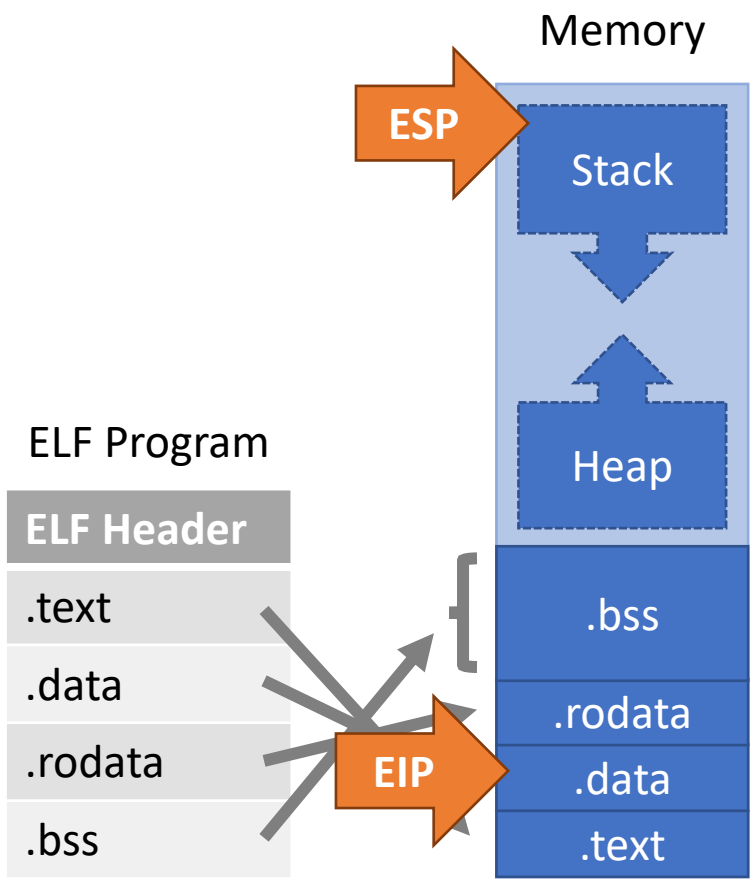
hex(4*10*1024*1024) = 0x2800020

Writable

```
$ readelf -S ./
...
Section Headers:
...
[Nr]  Name        Type        Address     Offset      Size        ES  Flags  Link  Info  Align
[25] .bss         NOBITS      00601040    00001034    02800020    00  WA     0     0     32
[26] .comment     PROGBITS    00000000    00001034    000002a     01  MS     0     0     1
...
```

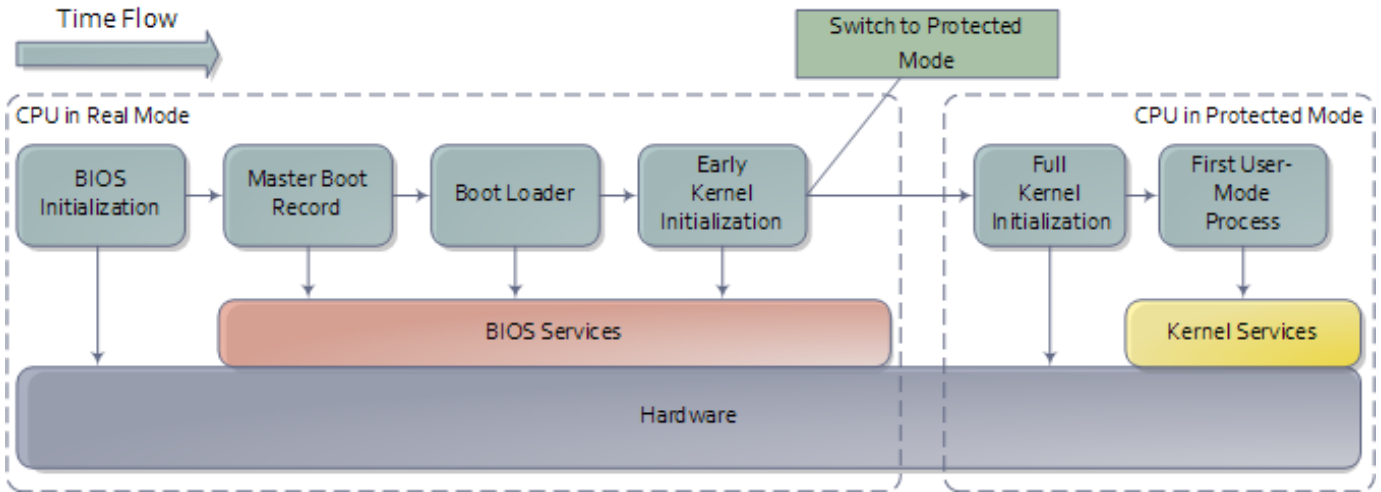
The Program Loader

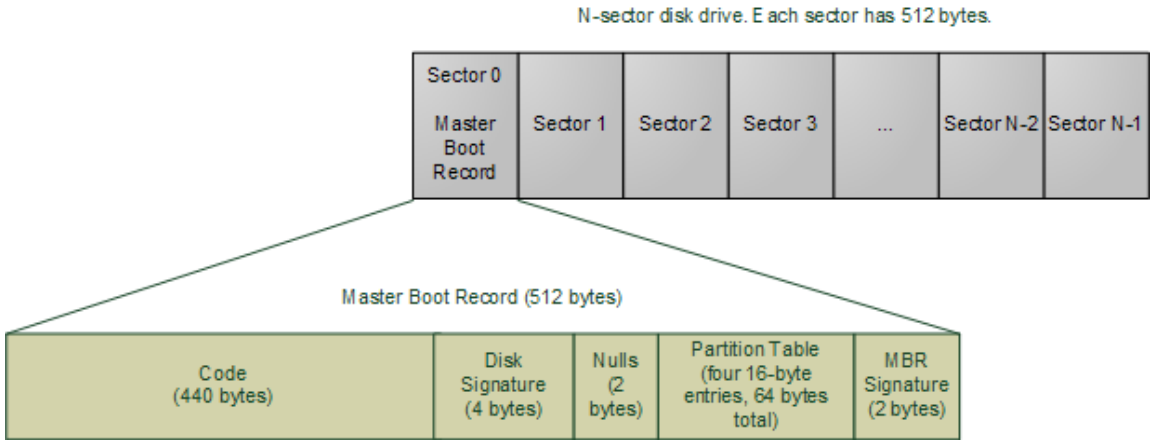
- OS functionality that loads programs into memory, creates processes
  - Places segments into memory
    - Expands segments like .bss
  - Loads necessary dynamic libraries
  - Performs relocation
  - Allocated the initial stack frame
  - Sets EIP to the programs entry point

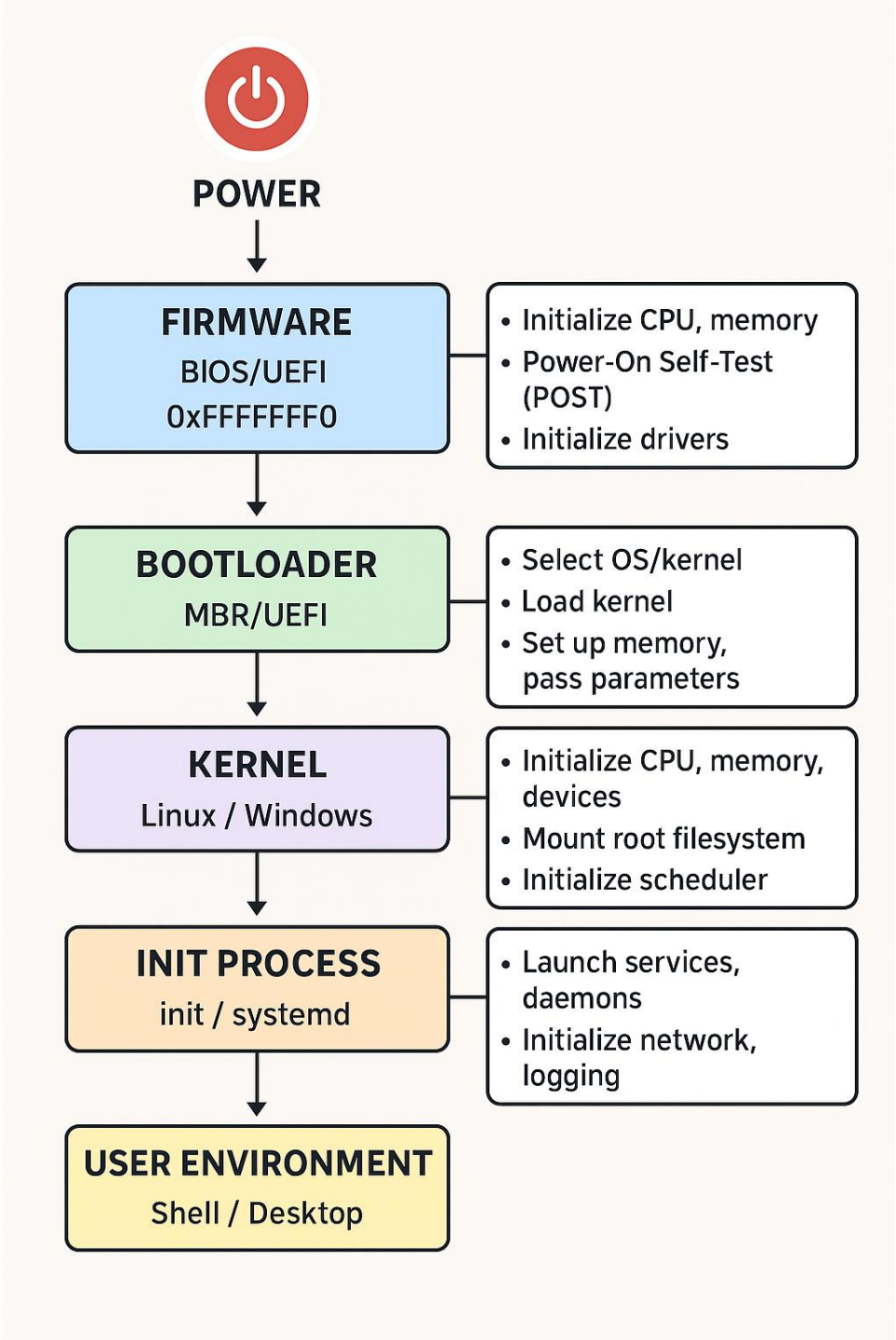


Boot Sequence

- The BIOS
  - Basic I/O System
  - Initial interface between the hardware and the operating system
  - Responsible for allowing you to control your computer's hardware settings for booting up
- Bootable devices
  - MBR
  - Loaded at location 0x7c00 in RAM and control is given to this code

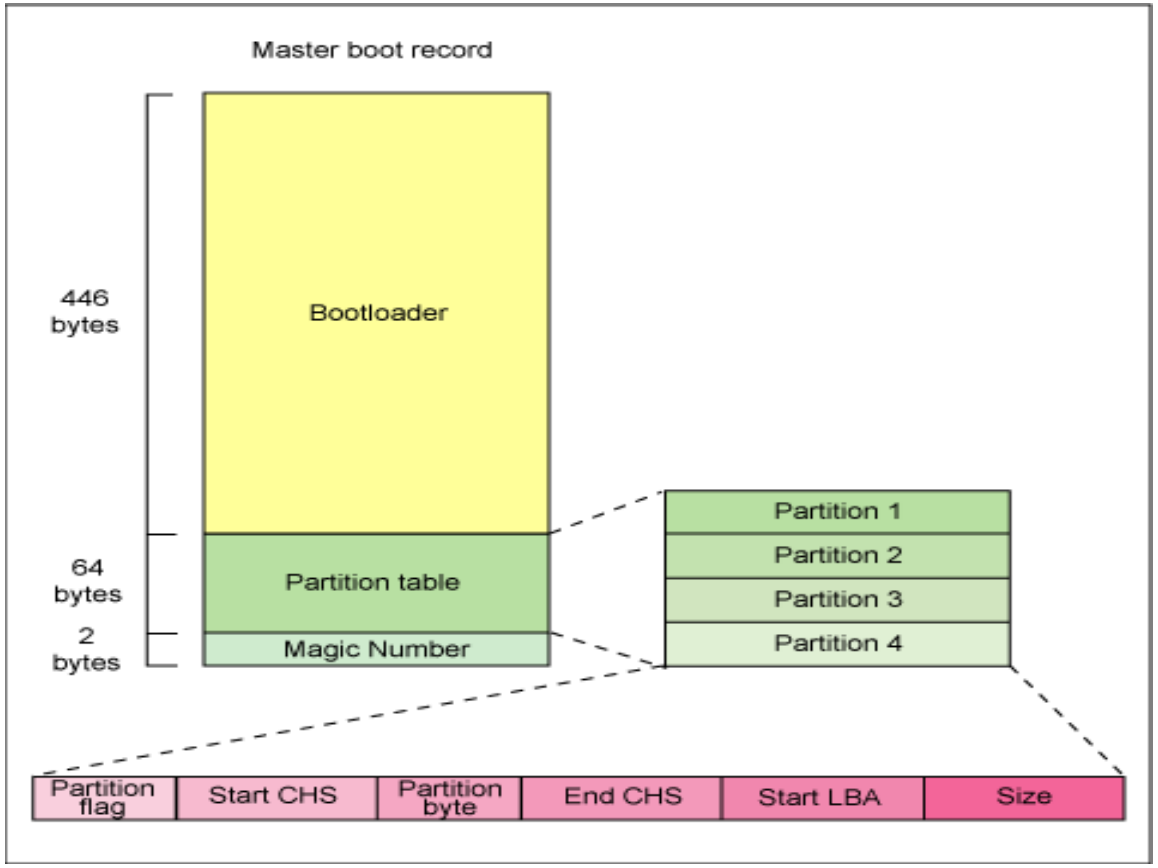





POWER
FIRMWARE
BIOS/UEFI
0xFFFFFFF0
• Initialize CPU, memory
• Power-On Self-Test (POST)
• Initialize drivers
BOOTLOADER
MBR/UEFI
• Select OS/kernel
• Load kernel
• Set up memory, pass parameters
KERNEL
Linux / Windows
• Initialize CPU, memory, devices
• Mount root filesystem
• Initialize scheduler
INIT PROCESS
init / systemd
• Launch services, daemons
• Initialize network, logging
USER ENVIRONMENT
Shell / Desktop

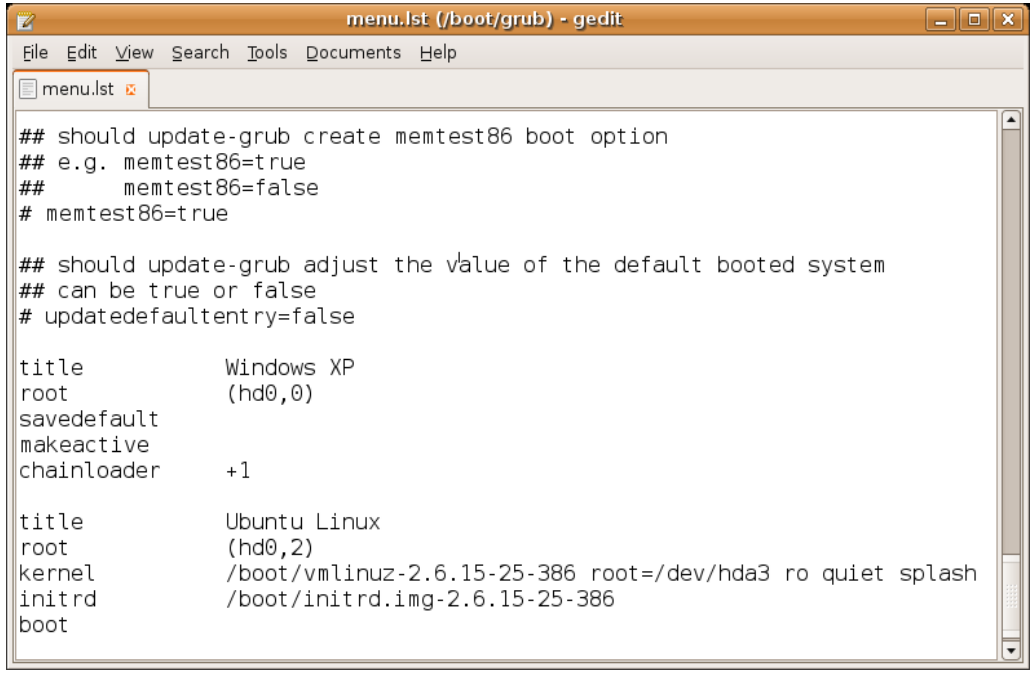
Boot Loaders - Linux

- 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
  - Stage-1.5 is a crucial feature
    - Makes grub file-system aware



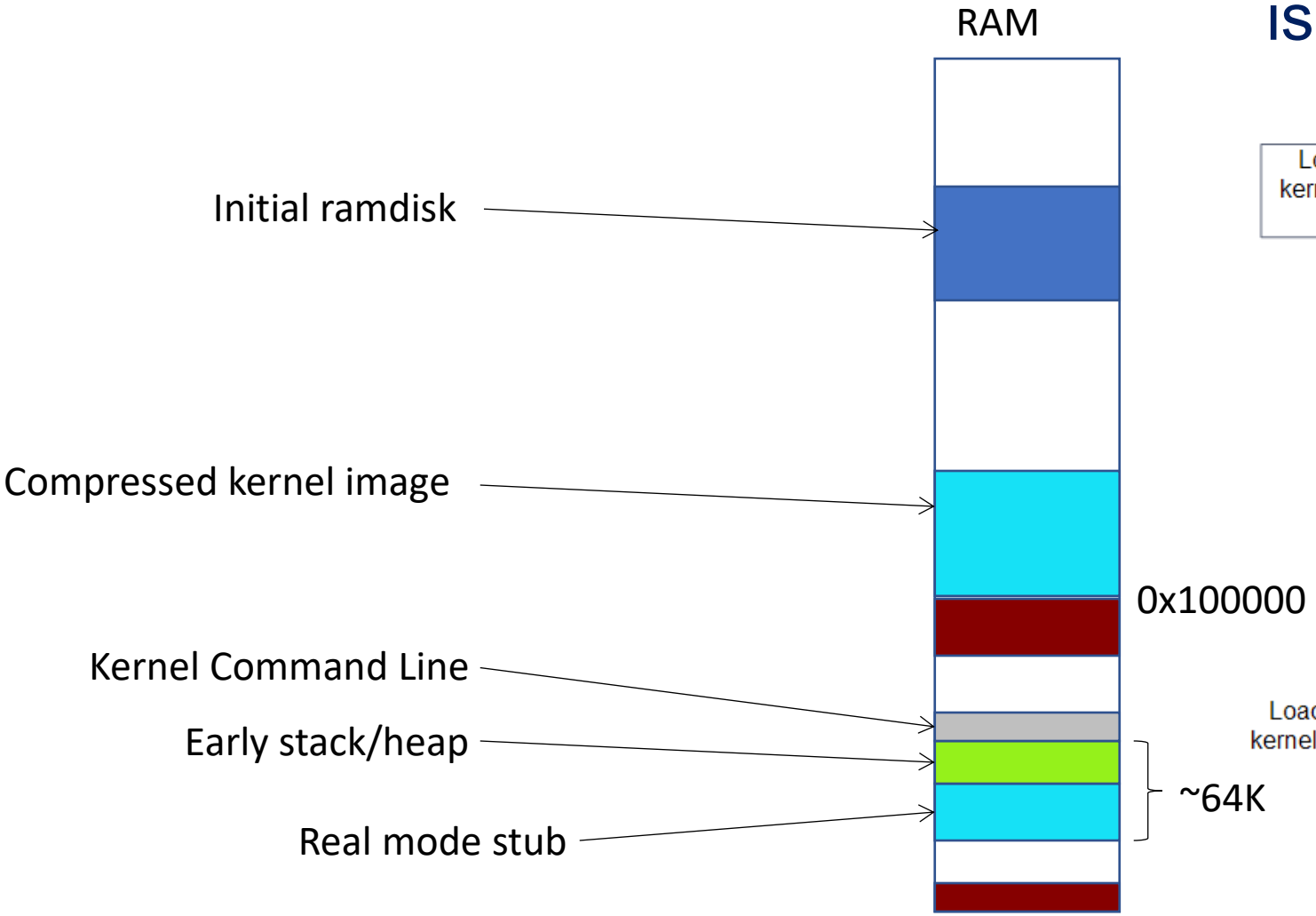
GRUB Stage-2

- Since file-system aware, it can display the boot options to user from
  - */boot/grub/grub.cfg*
- GRUB command-line
  - *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]***
- So now kernel time
  - But where are we going to place kernel image in the memory??

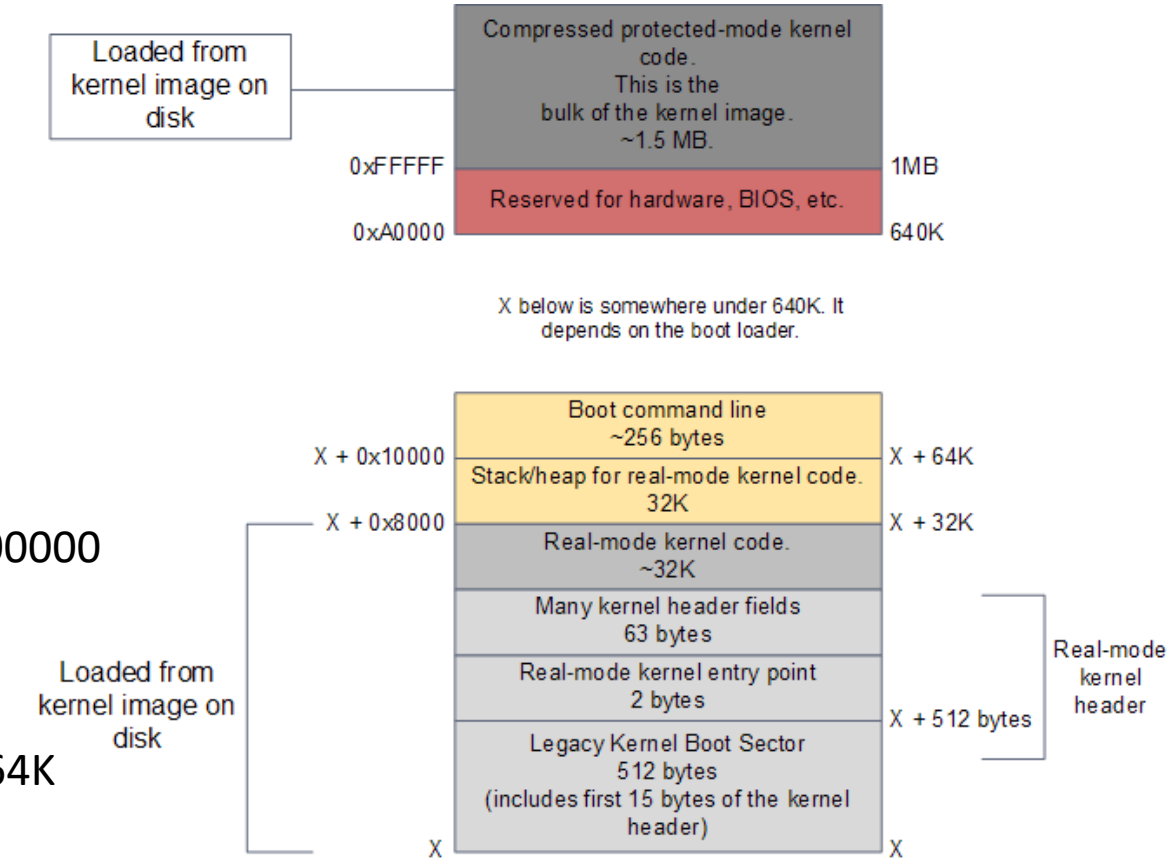


So Kernel Time

- In the beginning…

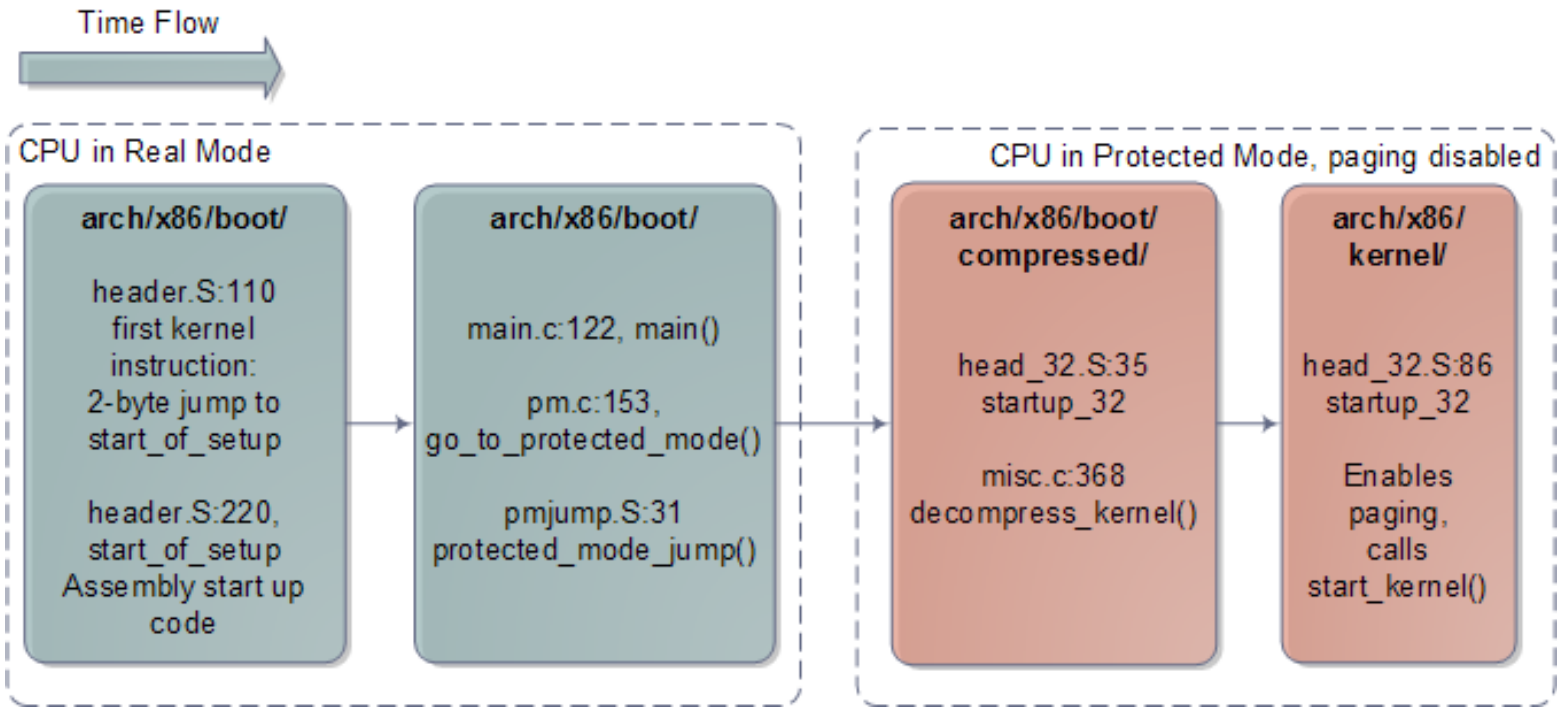


- RAM contents after boot loader is done



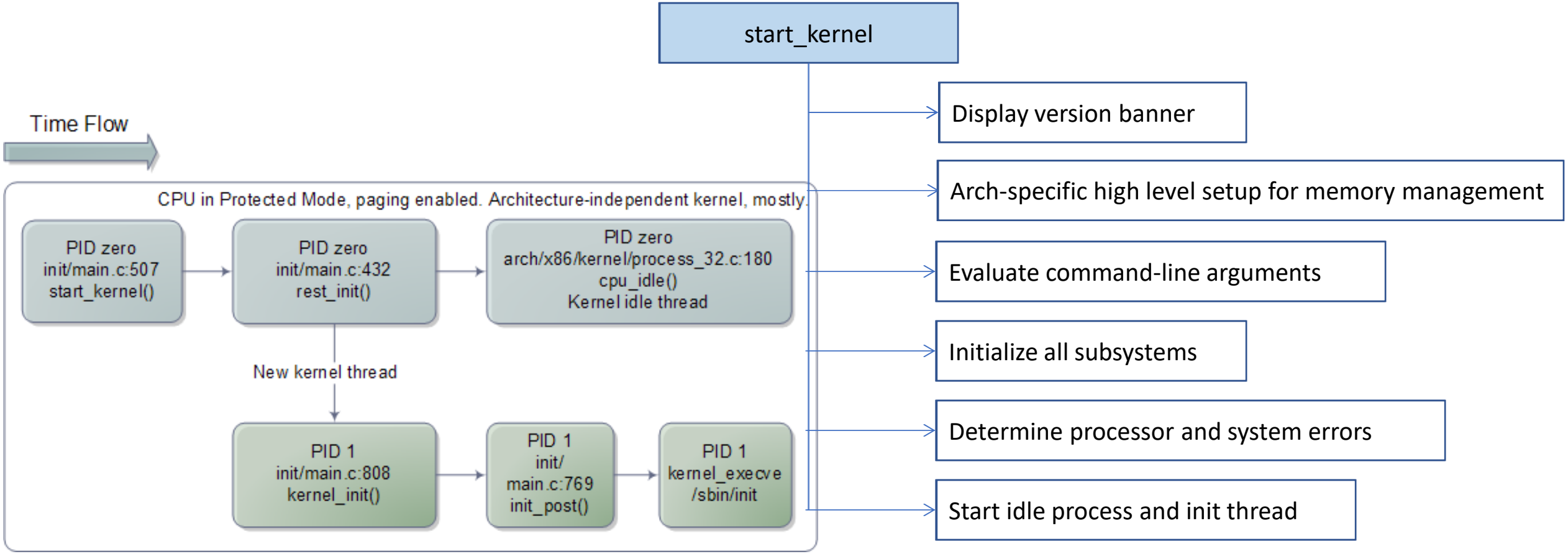
Major Steps in Kernel Initialization

1) Platform-specific initialization
   (In assembly language)
2) Platform-independent initialization
   (In high-level language C)

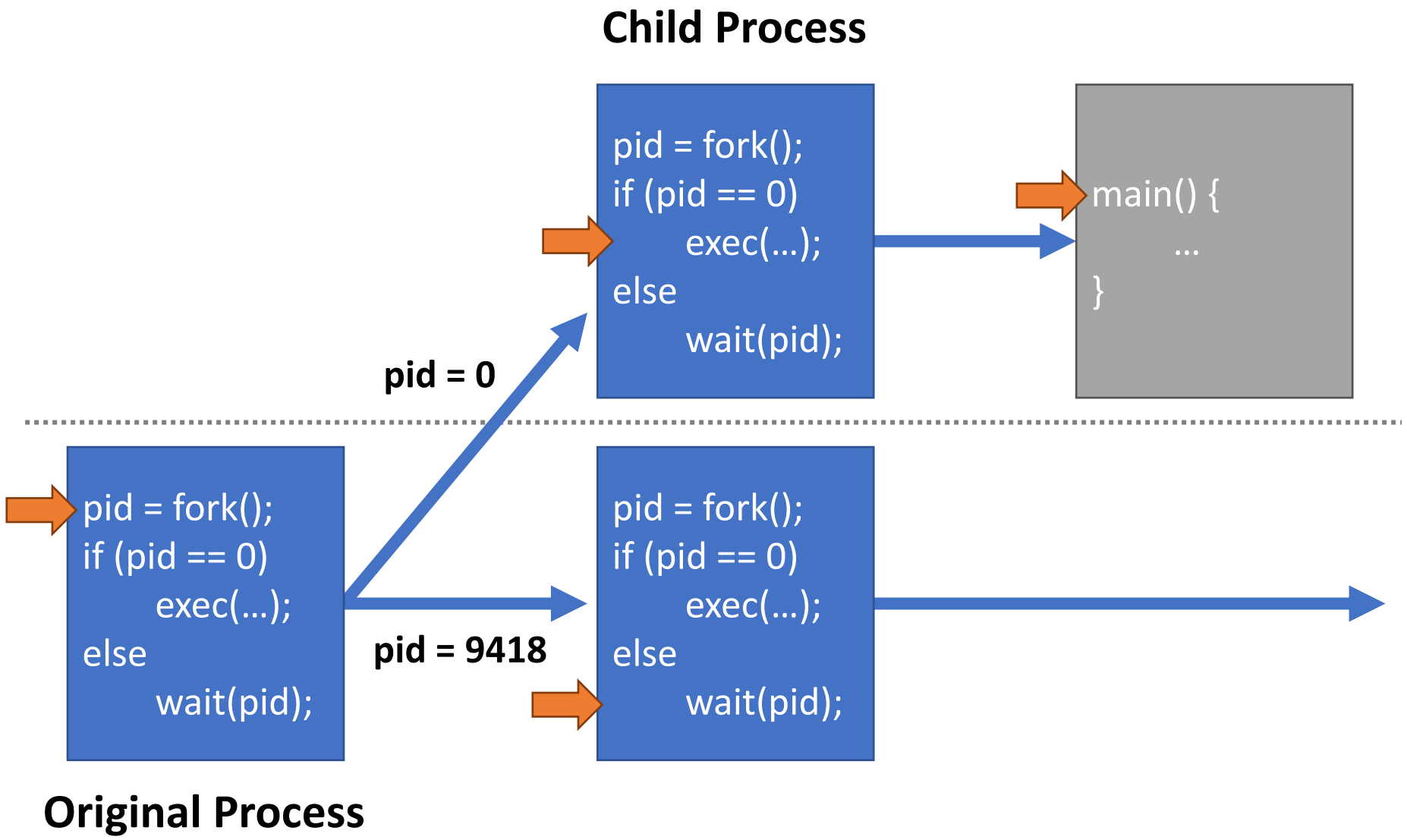


Major Steps in Kernel Initialization

- Platform-independent initialization
  (In high-level language C)



UNIX Process Management



Question: What does this code print?

```
int child_pid = fork();
if (child_pid == 0) {          // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
} else {                       // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```